Satisfiability of Quantitative Contact Logics

Martin Stoev

2019-9-28

Contents

1 Tableaux Method

Properties of a logical formula are best proven when the formula is decomposed into subformulas. Namely, the formula is defined in such a way that it can be described with a structural recursive approach. The idea is to test a property only for some subformula and then to derive that property for the given formula. Being able from a subformula to conclude properties for the formula itself leads to well-structured and performant algorithms.

The Tableau method is a formal proof procedure and can be found in many variants. Formally, this procedure is used to refute the validity of formulas. Specifically, if a formula X is given, and we want to prove that X is valid, then we construct a new formula that will represent the invalidity of X. This new formula is usually created with additional syntactic expressions. Proving that the newly created formula is not valid, it implies proof of the validity of X. The proofing process uses steps to break down the given formula into simpler formulas. If these steps are applied multiple times, the proof of the initial complex formula comes down to a proof using syntactic comparisons of basic formulas, where a basic formula is a formula which can not be decomposed further.

For our purposes, we will use a slightly different version of the method described above, namely with the help of the decomposition steps we will try to find a model with certain properties in which the given formula will not be valid.

1.1 Propositional logic tableau

The tableau method for propositional logic comes down to defining the decomposition of the propositional logic operations $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$. To reach this goal, we need to improve our syntactical arsenal, namely to add the ability for asserting the validity of a formula. In order to accomplish this, we shall introduce two signs. These signs when combined with a formula will assert whether the formula is true or false. Formally, the two signs are { T, F }. Let φ be a formula, then

- Tφ - asserts the validity of the formula φ
- Fφ - asserts the invalidity of the formula φ

The tableau of a formula φ begins with Fφ. The intuition behind it is that we start with an assertion that φ is not valid in some model, and by decomposing the formula during the tableau process we seek for contradictions in the subformals. If such contradiction exists then the initial guess that the formula is not valid is not true meaning that the formula is valid. On the other hand, if a contradiction does not exist then the formula is not valid, additionally a model can be constructed in which the formula φ is not valid.

The decomposition process is a set of tableau steps, which decompose the initial formula into basic formulas. A tableau step applies one of the tableau rules to slightly simplify the given formula. There are two rules for each operation in the logic. In the case of propositional logic with the following operations $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$ there are ten rules. Each rule decomposes the formula in at most two subformulas which depends

on the arity of the operation. The decomposition process forms a branch. A branch contains all signed subformulas which have been asserted in the decomposition process.

A tableau rule which decomposes the formula in two subformulas may require the assertion of only one of the subformulas to be satisfied. In this case, we will say that the branch is split into two branches. However, in some terminologies this is noted as a new branch spawned from the original branch, or that two new branches are created. For the sake of simplicity, we will use the terminology of branch splitting. In a scenario where the branch is split, all asserted formulas in the branch are copied in the split branch together with the newly asserted subformula from the tableau rule. The newly asserted subformulas depend on the tableau rule.

1.2 Rules

Negation The rules for negations are straightforward. When a negation operation is encountered, the sign of the captured formula changes.

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

Conjunction This is a binary operation, which means that the output of these rules is more complex than the negation rules. The conjunction rule for a formula signed as valid decomposes the formula in two subformulas and assert the validity of both of them. On the other hand, the conjunction rule for a formula signed as invalid splits the branch into two branches and asserts the invalidity of the subformulas in each branch, respectively.

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

Disjunction This is a binary operation, which means that the output of these rules is more complex than the negation rules. The disjunction rule for a formula signed as valid splits the branch into two branches and asserts the validity of the subformulas in each branch, respectively. On the other hand, the disjunction rule for a formula signed as invalid decomposes the formula in two subformulas and assert the invalidity of both of them.

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

Implication The implication as well is a binary operation and depending on the asserting sign it decomposes the formula in two subformulas or splits into two separate branches.

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

Equivalence Both of the equivalence rules split the branch into separate branches, the details can be observed in the following equations.

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

Closed branch A branch is said to be closed if and only if it contains the same formula signed as valid and invalid.

Closed tableau A tableau is said to be closed if, after the decomposition process, all branches are closed.

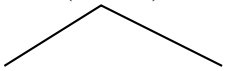
Tautology formula The process to determine if a formula is tautology is as follows: Let φ be a formula

1. Sign the formula as invalid, namely let $\mathbb{F}\varphi$ be the initial formula in the tableau process
2. Execute the tableau process until the tableau contains only basic formulas in each branch
3. If all branches are closed, then the formula is a tautology, otherwise it is not.

The tableau method is best explained through examples. First, let us see an example of a formula which is a tautology.

Consider the following formula $((\neg X \wedge (Y \rightarrow X)) \rightarrow \neg Y$

1. $\mathbb{F}(((\neg X \wedge (Y \rightarrow X)) \rightarrow \neg Y)$
2. $\mathbb{T}((\neg X \wedge (Y \rightarrow X)), \mathbb{F}\neg Y$
3. $\mathbb{T}((\neg X \wedge (Y \rightarrow X)), \mathbb{T}Y$
4. $\mathbb{T}\neg X, \mathbb{T}(Y \rightarrow X), \mathbb{T}Y$
5. $\mathbb{F}X, \mathbb{T}(Y \rightarrow X), \mathbb{T}Y$

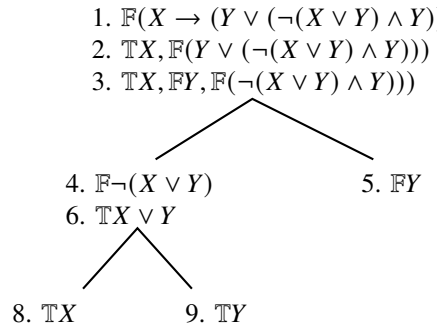


6. $\mathbb{F}Y$ 7. $\mathbb{T}X$

The first step of our tableau method for proving if a formula is a tautology is to sign it as not valid and analyze it. This means that at 1. This signed formula is added to the root of the tableau method. In the next step, there is only one possibility, namely to decompose the only formula currently in the tableau process. This formula is decomposed to two formulas present in 2. At this point there are 2 possible outcomes, the first one is to decompose the formula $\mathbb{T}((\neg X \wedge (Y \rightarrow X))$. The second one is to decompose the formula $\mathbb{F}\neg Y$. In this example we have chosen the second approach and the result is present in step 3, namely $\mathbb{F}\neg Y$ after a tableau step is $\mathbb{T}Y$. From step 3 to step 4 there is only one possibility. The $\mathbb{T}\neg X$ formula is decomposed to $\mathbb{F}X$. in step 5. Now, the decomposition of the formula $\mathbb{T}(Y \rightarrow X)$ will split the tableau branch into two branches. $\mathbb{F}Y$ will be added in the one of the branches, which in turn will contradict

with $\mathbb{T}Y$ which can be observed from step 3. This means that the branch closes. In the other branch, $\mathbb{T}X$ is added, which contradicts with $\mathbb{F}X$ from the previous step. This as well means that the second branch closes. Having a tableau with all branches closed proves that the initial assumption to sign the formula as invalid was wrong, hence the formula is a tautology.

The following tableau method is not closed, and after the tableau process reaches all basic formulas a model in which the formula is not valid can be constructed. The formula used is $X \rightarrow (Y \vee (\neg(X \vee Y) \wedge Y))$.



In step 1 the input formula is signed as not valid and added to the tableau. This signed formula is decomposed into two signed formulas in step 2. Where one of them is already a basic formula and can not be decomposed in simpler signed formulas. The second one is decomposed, and the results can be seen in step 3. Again, one of the new signed formulas is a basic formula. The only formula that can be decomposed is $\mathbb{F}(\neg(X \vee Y) \wedge Y)))$ which splits the tableau branch. In the split branch $\mathbb{F}Y$ is added, this is step 5. This formula does not cause a contradiction in the branch, and since there are no more possible decompositions, it follows that the branch is not closed. At this point, we know that the tableau is not closed. However, for the purposes of this example, let us examine the rest of the tableau process. In step 4 we can observe the additional signed formula which was added to the main branch. This signed formula is decomposed with the usage of one of the negation rules, as seen in step 6. Applying the disjunction rule for a formula signed as true on the formula in step 6 cases a branch split. Only one of the branches closes, namely the formula added in step 8. From this follows that the tableau is not closed and that the initial assumption was true, namely the formula is not a tautology.

2 Region-based Contact Logics

The region-based theory of space has the notion of region and relations between regions as one of the basic primitive notions of the theory. A region is defined as a set of elements in some space. Union and intersection are used as primary operations over regions. Two relations are introduced, part-of relation and contact relation. The part-of relation constructs the structural dependencies between regions, namely the part-of relations defines the mereology of regions. The contact relation defines the topological relation of connection. We will abstract ourselves from the inner relation between two connected regions, namely this relation might be defined as "region A and region B are neighbors". What we will use for this inner relation is its reflexivity and symmetry.

2.1 Syntax

Variable represents a region. Two predefined variables shall be used in order to simplify the notation.

- W represents the whole world, namely the biggest region which contains all elements.
- $\emptyset$ represents the empty region

Let $\mathbb{V}ar$ be a countable set of all variables.

Boolean constants Boolean constants are defined for W and $\emptyset$, namely 1 represents the world, while 0 represents the empty region.

Boolean operations are operations over regions. The followings are boolean operations:

- $\sqcap$, denotes boolean intersection
- $\sqcup$, denotes boolean union
- * , denotes boolean complement

Term is defined with the following inductive definitions:

- Boolean constant is a term
- $p \in \mathbb{V}ar$ is a term
- If x is a term, then *x is a term
- If x and y are terms, then $x \ \sigma \ y$ is a term,
  where $\sigma \ \in \ \{\sqcap, \sqcup\}$

Propositional constants $\top$ and $\bot$

Propositional connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$

Atomic Formulas A formula φ is called atomic, if it has one of the following forms:

- C(a, b), where a and b are terms
- $a \leq b$, where a and b are terms
- $a \leq_\mu b$, where a and b are terms

Formula is defined by the following inductive definition:

- Each propositional Constant is a formula
- Each atomic formula is a formula
- If φ is a formula, then $\neg\varphi$ is a formula as well
- If φ and ψ are formulas, then $\varphi\ \sigma\ \psi$ is a formula as well, where $\sigma \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

2.2 Semantics

The goal is to define the semantics of a formula and to answer the question when a given formula is satisfiable. When trying to define the satisfiability of a formula, a couple of questions arise, such as:

- How is a region represented
- What does it mean for two regions to be in contact
- In which scope is the formula satisfiable

Let us first define the semantics of a formula, which will lead us to its valuation and satisfiability.
Let W be a countable set and $W \neq \emptyset$. This set will represent all elements in the world. A region will be a subset of W. Namely, a set of elements represents a region. Let $R \subseteq W^2$ be a binary relation over regions. The relation R defines the contacts between regions. The relation R is a reflexive and symmetric relation.
For example, let us take:

$$W = \text{"the set of all countries in the world"}$$

With the above W, we can define the relation R, for example let $x, y \in W$, then:

$$R(x, y) = \text{"x has similar population count with y"}$$

Having (W, R) as defined above, we conclude the followings:

$$R(\text{ Germany, Turkey })$$

$$R(\text{ Bulgaria } \sqcup \text{ Macedonia , Austria })$$

Relational System A relational system is defined as $\mathcal{F} = (W, R)$, where $W \neq \emptyset$. $\mathcal{F}$ is usually called a frame.

Boolean variable valuation Let $\mathcal{F}$ be a frame, then a boolean variable valuation is any function υ assignig to each Boolean variable a subset $v(a) \subseteq W$. The valuation is then extended inductively to all Boolean terms, as follows

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

2.3 Measured Model

Measure function A measure function will denote a function which maps a region to a positive real number.

$$\mu : W \longrightarrow \mathbb{R}^+$$

Measured Model Let μ be a measure function. The tuple $\mathcal{M} = (\mathcal{F}, \mu, \upsilon)$ is called a model. The truth of a formula φ in $\mathcal{M}$ (in symbols $\mathcal{M} \models \varphi$) is defined inductively as follows:

- $\mathcal{M} \not\models \bot$
- $\mathcal{M} \models \top$
- $\mathcal{M} \models aCb$ iff $\upsilon(a)$ iff $(\exists x \in \upsilon(a)), (\exists y \in \upsilon(b))(xRy)$
- $\mathcal{M} \models a \leq b$ iff $\upsilon(a) \subseteq \upsilon(b)$
- $\mathcal{M} \models a \leq_\mu b$ iff $\mu(\upsilon(a)) \leq \mu(\upsilon(b))$
- $\mathcal{M} \models \neg\varphi$ iff $\mathcal{M} \not\models \varphi$
- $\mathcal{M} \models a\ \sigma\ b$ iff $\mathcal{M} \models a\ \sigma\ \mathcal{M} \models b$, where $\sigma \in \{\vee, \wedge\}$

In the previously defined 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.

A model $\mathcal{M}$ is a model of a formula φ if φ is true in $\mathcal{M}$, in such a case we say that $\mathcal{M}$ satisfies φ, in symbols $\mathcal{M} \models \varphi$; $\mathcal{M}$ is a model of a set of formulas A if $\mathcal{M}$ is a model of all formulas from A, in symbols $\mathcal{M} \models A$. A formula φ is true in a frame $\mathcal{F}$, or that $\mathcal{F}$ is a frame for φ, in symbols $\mathcal{F} \models \varphi$, if $\mathcal{M} \models \varphi$ for all models $\mathcal{M}$ based on $\mathcal{F}$, i.e. for all evaluations υ we have $\mathcal{F}, \upsilon \models \varphi$. If Σ is a class of frames, we say that φ is true in Σ, in symbols $\Sigma \models \varphi$, if φ is true in all frames in Σ. We say that a set of formulas A is satisfiable in Σ, or A is Σ-consistent, if there is a model $\mathcal{M} = (\mathcal{F}, \upsilon)$ with $\mathcal{F} \in \Sigma$ such that $\mathcal{M}$ is a model of A.

2.4 Formula Properties

Axiom 2.1 (Contact reflexivity)**.** *Let a be a term, then*

$$a \neq 0 \implies aCa$$

Axiom 2.2 (Contact symmetry)**.** *Let a and b be two terms, then*

$$aCb \iff bCa$$

Lemma 2.3 (Term equality)**.** *Let a and b be two terms and let υ be a valuation, then*

$$a = b \implies \upsilon(a) = \upsilon(b)$$

Lemma 2.4 (Formula equality)**.** *Let φ and ψ be two formulas and let υ be a valuation, then*

$$\varphi = \psi \implies \upsilon(\varphi) = \upsilon(\psi)$$

Lemma 2.5 (Zero term formula)**.** *Let a and b be two terms, then*

$$a \leq b \iff a \sqcap \neg b = \emptyset$$

The formula $a \sqcap \neg b = \emptyset$ will be called zero term formula. Since $a \sqcap \neg b$ is a new term, it can be assigned a variable $s = a \sqcap \neg b$, and now the zero term formula above can be written as $s = 0$ which is with better readability and can be straightforwardly grasped in proofs and definitions.

Lemma 2.6 (Non-zero term)**.** *Let a and b be two terms, then*

$$\neg(a \leq b) \iff a \sqcap \neg b \neq \emptyset$$

Lemma 2.7 (Contact monotonicity)**.** *Let a and b be two terms, then*

$$aCb \wedge a \leq a' \wedge b \leq b' \implies a'Cb'$$

Lemma 2.8 (Contact distributivity)**.** *Let a and b be two terms, then*

$$aC(b \sqcup c) \iff aCb \vee aCc$$

Lemma 2.9 (Measure Formula Addition Property of Equality)**.** *Let a, b and d be three terms, then*

$$(a \sqcap d = 0) \wedge (b \sqcap d = 0) \implies (a \leq_\mu b \iff a \sqcup d \leq_\mu b \sqcup d)$$

Proof. Let $\mathcal{M} =< B, \mu, \upsilon >$ be an arbitrary model. From the implication we can assume the following:

$$\begin{gathered}
\mathcal{M} \models (a \sqcap d = 0) \wedge (b \sqcap d = 0) \\
\mathcal{M} \models a \sqcap d = 0 \text{ and } \mathcal{M} \models (b \sqcap d = 0) \\
\upsilon(a \sqcap d) = \upsilon(0) \text{ and } \upsilon(b \sqcap d) = \upsilon(0) \\
\upsilon(a) \sqcap_B \upsilon(d) = 0_B \qquad \upsilon(b) \sqcap_B \upsilon(d) = 0_B \\
\upsilon(a) \cap \upsilon(d) = \emptyset \qquad \upsilon(b) \cap \upsilon(d) = \emptyset
\end{gathered}$$

$\Longrightarrow$

$$
\begin{aligned}
&\text{Let } \mathcal{M} \models a \leq_\mu b \\
&\mu(v(a)) \leq \mu(v(b)) \\
&\text{We know that } \mu(v(d)) \in \mathbb{R}^+ \\
&\Rightarrow \mu(v(a)) + \mu(v(d)) \leq \mu(v(b)) + \mu(v(d)) \\
&\text{From } v(a) \cap v(d) = \emptyset \Rightarrow \mu(v(a)) + \mu(v(d)) = \mu(v(a \sqcup d)) \\
&\text{From } v(b) \cap v(d) = \emptyset \Rightarrow \mu(v(b)) + \mu(v(d)) = \mu(v(b \sqcup d)) \\
&\Rightarrow \mu(v(a \sqcup d)) \leq \mu(v(b \sqcup d)) \\
&\Rightarrow \mathcal{M} \models a \sqcup d \leq_\mu b \sqcup d
\end{aligned}
$$

$\Longleftarrow$

$$
\begin{aligned}
&\text{Let } \mathcal{M} \models a \sqcup d \leq_\mu b \sqcup d \\
&\Rightarrow \mu(v(a \sqcup d)) \leq \mu(v(b \sqcup d)) \\
&\Rightarrow \mu(v(a)) + \mu(v(d)) \leq \mu(v(b)) + \mu(v(d)) \\
&\Rightarrow \mu(v(a)) \leq \mu(v(b)) \\
&\Rightarrow \mathcal{M} \models a \leq_\mu b
\end{aligned}
$$

□

Lemma 2.10 (Trivial formula implications)**.** *Let a, b, c be three terms and let φ and ψ be two formulas, then*

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

- $aC0 \implies F$
- $1C1 \implies T$
- $aC1 \implies a \neq 0$
- $a \neq 0 \implies aCa$
- $0 \leq_\mu a$
- $a \leq_\mu 1$
- $(a = 0) \iff (a \leq_\mu 0)$
- $(a = 1) \iff (1 \leq_\mu a)$
- $(a \leq_\mu b) \vee (b \leq_\mu a)$
- $a_1 \leq_\mu a_2 \wedge a_2 \leq_\mu a_3 \implies a_1 \leq_\mu a_3$

3 Quantitative Contact Logics

The quantitative contact logics is basically the region based contact logics with the addition of quantitative measures. The quantitative measure is represented with two new atomic formulas:

- $\leq_\mu$
- $<_\mu$

Where μ is a measure function. Detailed definition is given in 2.3.

3.1 Formula Satisfiability

The satisfiability verification for a given formula is done by constructing a model. The formula is satisfiable only if such a model exists. The model construction is done in several steps. Foremost, the tableau method is used to simplify the problem and to detect contradictions earlier in the process, even before the model construction. If all branches of the tableau method are closed, then a model in which the given formula is satisfiable does not exist. If there are open branches in the tableau result, then semantical reasoning shall be done in order to construct a model, if such a model exists. Let us define few definitions before we define the process of formula satisfiability.

Tableau Branch Conjunction A branch from the tableau method consists of signed atomic formulas. For a formula to be satisfiable, it means that all atomic formulas in a tableau branch should be satisfied. This in terms can be represented as a conjunction of the atomic formulas. Let's call it a branch conjunction. For the quantitative contact logics, the conjunction contains the signed contact, zero term and measure atomic formulas.

Deffinition 3.1. *Let φ be a formula and let $\mathcal{T}$ be a tableau from the φ formula. Then a branch of the tableau method is defined with the following notation:*

$$B = \{TC(a_i, b_i) \mid i \in \{1, \ldots, I\}\} \cup \{FC(e_k, f_k) \mid k \in \{1, \ldots, K\}\} \cup$$
$$\{Fd_j = 0 \mid j \in \{1, \ldots, J\}\} \cup \{Tg_l = 0 \mid l \in \{1, \ldots, L\}\} \cup$$
$$\{Tm_p \leq_\mu n_p \mid p \in \{1, \ldots, P\}\} \cup \{Fu_q \leq_\mu v_q \mid q \in \{1, \ldots, Q\}\}$$

The branch conjunction of the tableau method branch is defined as:

$$\bigwedge_{i=1}^{I} C(a_i, b_i) \ \wedge \ \bigwedge_{k=1}^{K} \neg C(e_k, f_k) \ \wedge$$
$$\bigwedge_{j=1}^{J} d_j \neq 0 \ \wedge \ \bigwedge_{l=1}^{L} g_l = 0 \ \wedge$$
$$\bigwedge_{p=1}^{P} m_p \leq_\mu n_p \ \wedge \ \bigwedge_{q=1}^{Q} u_q <_\mu v_q$$

Deffinition 3.2. *Let B be a branch conjunction, then the set of all variables used in all atomic formulas in B is defined as* $\mathbb{V}ar_B$.

Deffinition 3.3. *Let B be a branch conjunction and let* $\mathbb{V}ar_B$ *be the set of all variables in B, then the variable valuation* v_{var} *defines a mapping from* $\mathbb{V}ar_B$ *to a boolean value, namely:*

$$v_{var} : \mathbb{V}ar_B \rightarrow \{\textbf{\textit{false}}, \textbf{\textit{true}}\}$$

Boolean valuation Let e be a variable valuation and let $\mathcal{T}_s$ be the set of all terms, then the function $v\mathcal{B}_e : \mathcal{T}_s \rightarrow \{\textbf{false}, \textbf{true}\}$ is called a boolean valuation and is defined recursively as:

- $v\mathcal{B}_e(0) = \textbf{false}$
- $v\mathcal{B}_e(1) = \textbf{true}$
- $v\mathcal{B}_e(p) = e(p),$ *where* $p \in \mathbb{V}ar_B$
- $v\mathcal{B}_e(a \sqcap b) = v\mathcal{B}_e(a) \wedge v\mathcal{B}_e(b)$
- $v\mathcal{B}_e(a \sqcup b) = v\mathcal{B}_e(a) \vee v\mathcal{B}_e(b)$
- $v\mathcal{B}_e(a*) = \neg v\mathcal{B}_e(a)$

Deffinition 3.4. *A model point is represented by a variable valuation.*

Deffinition 3.5. *Let a be a term, let* e *be a variable valuation which is a representation of a model point p. Then the following holds:*

$$v\mathcal{B}_e(a) = true \rightarrow p \in v(a)$$

Where v *is the model valuation function. In such a case when* $v\mathcal{B}_e(a) = \textbf{\textit{true}}$ *we will say that the point p is* ***valid****.*

Model Point Creation Constructing a valid model for a given branch conjunction of a formula consists of finding W, R and the model valuation function v. To construct W we are going to generate the infimum number of model points in such a way to keep the formula satisfied at all times. If this is not possible, then a model does not exist in the given branch conjunction. Generating only the minimum required model points helps out to avoid unnecessary computations.

Deffinition 3.6. *Let* $\mathbb{V}ar_B$ *be the set of all variables in a branch conjunction B and let a be a term. Then a model point p is constructed by generating a variable valuation* e*, such that p is valid:*

$$v\mathcal{B}_e(a) = 1$$

Let n be the number of variables in $\mathbb{V}ar_B$*, then all possible variants of a variable valuations are* 2^n.

Deffinition 3.7. *Let B be a branch conjunction as:*

$$\bigwedge_{i=1}^{I} C(a_i, b_i) \ \wedge \ \bigwedge_{k=1}^{K} \neg C(e_k, f_k) \ \wedge$$
$$\bigwedge_{j=1}^{J} d_j \neq 0 \ \wedge \ \bigwedge_{l=1}^{L} g_l = 0 \ \wedge$$
$$\bigwedge_{p=1}^{P} m_p \leq_\mu n_p \ \wedge \ \bigwedge_{q=1}^{Q} u_q <_\mu v_q$$

The following atomic formulas require model point existence:

- $C(a_i, b_i)$, *for* $i < I$
- $d_j \neq 0$, *for* $j < J$
- $m_p \leq_\mu n_p$, *for* $p < P$
- $u_q <_\mu v_q$, *for* $q < Q$

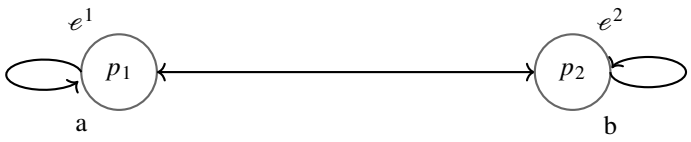
Deffinition 3.8. *Let B be a branch conjunction and* $C(a, b) \in B$*, where a and b are two terms. Then two model points are generated p1 and p2 with their corresponding variable valuations* e^1 *and* e^2 *such that:*

- $v\mathcal{C}_{e^1}(a) =$ ***true***
- $v\mathcal{C}_{e^2}(b) =$ ***true***

Along with the generation of a valid variable valuation, the R relations is extended as well, namely the following relations are added:

- $p_1 R p_1$ *- reflexivity of the modal point* p_1
- $p_2 R p_2$ *- reflexivity of the modal point* p_2
- $p_1 R p_2$ *- symmetric relation between* p_1 *and* p_2
- $p_2 R p_1$ *- symmetric relation between* p_2 *and* p_1

The constructed points and the relation between them is best seen on a diagram



Generated model points and their relations for contact C(a, b)

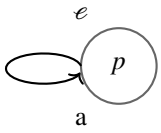
Deffinition 3.9. *Let B be a branch conjunction and $a \neq 0 \in B$, where a is a term. Then one model point p is generated and its variable valuations e such that:*

- $v\mathcal{C}_e(a) =$ ***true***

The R relations is extended as follows:

- pRp *- reflexivity of the modal point p*

The constructed point diagram:



Generated model points and its relation for not equal term $a \neq 0$

Deffinition 3.10. *Let B be a branch conjunction and $a \leq_\mu b \in B$, where a and b are two terms. Then one model point p is generated and its variable valuations e is the constant true, namely the constant term 1.*

Deffinition 3.11. *With definitions (1, 2, 3) we are able to construct a model where the contacts and non zero terms are satisfied. We will denote this model as* ***partially constructed model****. However to satisfy a branch conjunction B we need to verify that the non contacts and zero terms are satisfied as well.*

Deffinition 3.12. *Let M = (W, R) be a partially constructed model and let $\mathcal{T}_s$ be the set of all terms. Then the model valuation $v : \mathcal{T}_s \to \mathcal{P}(W)$ is defined recursively as:*

- $v(0) = W$
- $v(1) = \emptyset$
- $v(p) = \{q \mid e(q, p) = 1\}$, *where* $p \in \mathbb{V}ar_B$
- $v(a \sqcap b) = v(a) \cap v(b)$
- $v(a \sqcup b) = v(a) \cup v(b)$
- $v(a*) = W \setminus v(a)$

The tuple (W, R, v) is called a partially constructed frame.

Deffinition 3.13. *Let $\mathcal{F} = (W, R, v)$ be a partially constructed frame. Then we say that the frame $\mathcal{F}$ satisfies an equal to zero term $a = 0$, if the following holds:*

$$\forall p \in W : v(a) \neq \emptyset$$

Deffinition 3.14. *Let $\mathcal{F} = (W, R, \upsilon)$ be a partially constructed frame. Then we say that the frame $\mathcal{F}$ satisfies a non contact $\neg C(a, b)$, if the following holds:*

$$\forall p \in W : \upsilon(a) = \emptyset \vee \upsilon(b) = \emptyset$$

$$\begin{aligned} \forall (p, q) \in R : (\upsilon(p, a) = \emptyset \vee \upsilon(q, b) = \emptyset) \\ \wedge\ (\upsilon(p, b) = \emptyset \vee \upsilon(q, a) = \emptyset) \end{aligned}$$

Satisfied atomic formulas Let M be a partially constructed model from a branch conjunction B. M is said to be a potential model if:

- $\forall\ \neg C(a, b) : M$ *satisfies* $\neg C(a, b)$
- $\forall\ a = 0 : M$ *satisfies* $a = 0$

To validate that the potential model is correct, meaning that it satisfies the formula, we must take into account the measure atomic formulas. Without loss of generality, we can transform each $<_\mu$ atomic formula, to an equivalent formula which uses only the $\leq_\mu$ atomic formula and boolean operators. In a branch conjunction B, there might be more than one atomic measure formula. Validating that the potential model satisfies the branch conjunction together with the measure atomic formulas, means that all atomic measure formulas to be satisfied. Satisfying a set of measure atomic formulas produces a system of inequalities.

3.2 System of Inequalities

The system of inequality has the following structure:

$$\begin{cases} \sum_{i^1} X_{i^1} \leq \sum_{j^1} X_{j^1} \\ \quad \cdots \\ \sum_{i^n} X_{i^n} \leq \sum_{j^n} X_{j^n} \\ \sum_{k^1} X_{k^1} > \sum_{l^1} X_{l^1} \\ \quad \cdots \\ \sum_{k^m} X_{k^m} > \sum_{l^m} X_{l^m} \end{cases}$$

3.3 System Construction

Let $\mathcal{F} = (W, R, \upsilon)$ be a potential frame. The system of inequalities is constructed from the potential frame by evaluating each of the terms in the $\leq_\mu$ and $<_\mu$ atomic formulas. The number of points in the model are $N = \mid W \mid$. Let us enumerate the points $p_0, p_1, ..., p_N$. The system will have N different variables $X_0, X_1, ..., X_N$, where $\forall i < N : X_i$ *is mapped to point* p_i.

Deffinition 3.15. *Let x and y be two terms, then the formula* $\leq_\mu (x, y)$ *is transformed to an inequality as follows:*

$$\sum_{i \in v(x)} X_i <= \sum_{j \in v(y)} X_j$$

Deffinition 3.16. *Let x and y be two terms, then the formula* $<_\mu (x, y)$ *is transformed to an inequality as follows:*

$$\sum_{i \in v(x)} X_i < \sum_{j \in v(y)} X_j$$

Deffinition 3.17. *Let x and y be two terms and let* $\mathcal{I}_q$ *be the inequality produced from* $\leq_\mu (x, y)$*. Then the inequality* $\mathcal{L}_q$ *can be simplified by removing all variables present on both sides of the equation. This can be written as:*

$$\sum_{i \in v(x) \setminus v(y)} X_i <= \sum_{j \in v(y) \setminus v(x)} X_j$$

Deffinition 3.18. *Let x and y be two terms and let* $\mathcal{I}_q$ *be the inequality produced from* $<_\mu (x, y)$*. Then the inequality* $\mathcal{L}_q$ *can be simplified by removing all variables present on both sides of the equation. This can be written as:*

$$\sum_{i \in v(x) \setminus v(y)} X_i < \sum_{j \in v(y) \setminus v(x)} X_j$$

Deffinition 3.19. *Let* $\mathcal{F} = (W, R, v)$ *be a potential frame. Let* $\mathcal{S}$ *be the system of inequalities defined by*

- *inequalities for all* $\leq_\mu$ *fromulas by def: 4.17*
- *inequalities for all* $<_\mu$ *fromulas by def: 4.18*
- *inequality* $0 < X_i$ *for each* $0 \leq i < N$

If the system of inequalities $\mathcal{S}$ *has a solution, then we say that the system is valid.*

Lemma 3.20. *Let* φ *be a formula. If there exists branch conjunction B for the formula* φ *and there exists a potential frame* $\mathcal{F}$ *for the branch conjunction B and there exists a valid system of inequalities defined from the potential frame* $\mathcal{F}$*, then the formula* φ *is satisfiable by the frame* $\mathcal{F}$*.*

4 Implementation Introduction

The main programming language is C++. Flex & Bison are used to parse the input formula. The formula prover 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 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 proving.
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 below describes the formula's syntax:

Terms		
0	0	Boolean constant 0
1	1	Boolean constant 1
-	$*$	Boolean complement
*	$\cap$	Boolean meat
+	$\cup$	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:

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

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

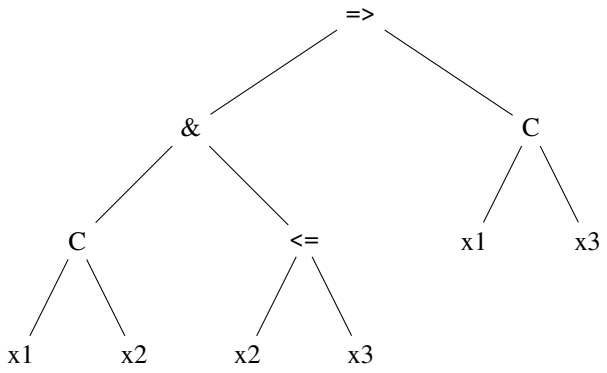
4.2 Formula parsing

The formula is a sequence of characters. These characters do not give us any information for the formula's structure. It should be analyzed. Flex [1] and Bison [2] are used to parse it into an AST (Abstract Syntax Tree). 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 ϕ = ($C(x1, x2)$ & <= $(x2, x3)$) => $C(x1, x3)$. 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 symbols 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 whitespace is not significant, and a token is not created for them. We shall use Flex as a tokenizer [1].

Grammar The tokenizer's grammar is composed of 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 explained 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 identifiers. 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 prompt where the unrecognized 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 high (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 structured way. On the AST additional optimizations can be done which will simplify the initial formula. It will produce better performance when a model is sought 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. Its arguments are an operation and up to two child terms. Depending on the operation arity.

The create_formula_node is analogous 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 Syntax Tree.

parser.y

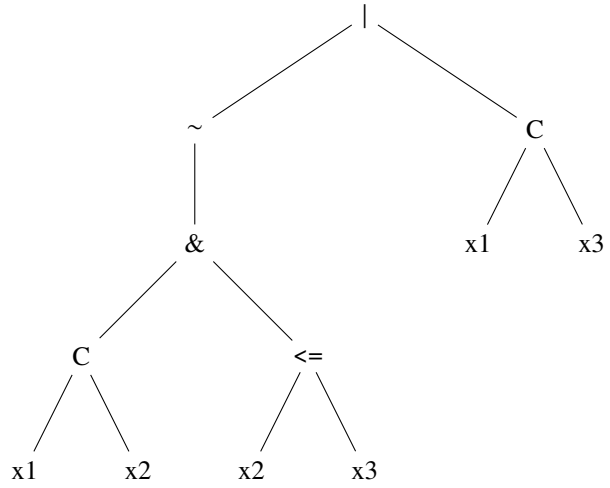
```
formula // 'formula' non-terminal
    : 'T' { // matching token 'T'
        $$ = create_formula_node(constant_true);
    }
    | 'F' {
        $$ = create_formula_node(constant_false);
    }
    | 'C' '(' term ',' term ')' {
        $$ = create_formula_node(contact, $3, $5);
    }
    | "<=" '(' term ',' term ')' {
        $$ = 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 [3]. 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 implement 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 its class. Does not require changes in the AST node classes or other visitor classes.

visitor.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 equal 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 modified. 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 structure. The complexity is O(n), where n is the size of the subformula. To reduce this complexity, a precalculated 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.h

```
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 corresponding formula/term node is constructed. The implementation could be found in *term.cpp* and *formula.cpp* in the repository.

5 Tableaux Implementation

The Tableaux 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 applying 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 tableaux is to break a complex formula into smaller ones until complementary pairs of atomic formulas are produced.

Deffinition 5.1. *A set of only signed formulas represents a signed formula set. The letter X is usually used for its representation.*

5.1 Tableaux Step

The *Tableaux 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 tableaux steps. The output of the tableaux step depends on the rule applied to the formula.

5.2 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

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

Disjunction

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

For our case, the functionality of the tableaux 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 tableaux process returns a not-closed branch and if there is a model for the set of atomic formulas in this branch, then the formula is satisfiable, otherwise the tableaux process proceeds with the next not-closed branch. If such branch does not exist, then the formula is not satisfiable.

5.3 Implementation

The program implementation of the tableaux method follows the standard tableaux process. The 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.

Deffinition 5.2. *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 the same structure will be treated as identical.

The average complexity for search, insert and erase in this collection is O(1). There is no formula copying. 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 tableaux 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 collections are unordered sets of points to the formulas/terms.

types.h

```
using formulas_t = std::unordered_set<
        const formula*,
        formula_ptr_hasher,
        formula_ptr_comparator>;
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_;
```

Deffinition 5.3. *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 occur if a formula is split and some 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.

Tableaux Algorithm

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

Emptiness Lemma

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.

Inverse Emptiness Lemma

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 Emptiness Lemma and Inverse Emptiness Lemma

The algorithmic complexity to check whether a new formula leads to contradiction by Emptiness Lemma and Inverse Emptiness Lemma 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 Emptiness Lemma or Inverse Emptiness Lemma, the following method is used:

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

5.4 Handy methods

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 $formulas\_F$

5.5 Tableaux Satisfiable Step Implementation

The Tableaux satisfiable step is the whole Tableaux algorithm.

tableau.cpp

```
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_conjuction_child X(*this, f->get_left_child_formula());
            T_conjuction_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

Tableaux branch output

As stated above the ouput of a branch in the tableaux 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 building 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 an identifier 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 boolean elements. The memory for N elements is roughly N bits. The element at position X is the evaluation for the variable with identifier 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 implemented 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 implemented 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 assigns a constant true or false to the term for some variable evaluation. This variable evaluation assigns 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 satisfaction

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 satisfaction

Checks whether a modal point (or a pair of points) does not conflict with the non-contacts. It is split 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;
}
```

6.5 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;
}
```

6.6 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 Quantitative Contact Logics Implementation

Model Valuation of a Term

The implementation of the model valuation **??** is in the *term* class. The details are in the *term.cpp* file. The model valuation assigns a set of points to the term for some variable valuation.

term.h

```
class term {
...
auto term::evaluate(
        const variable_id_to_points_t& variable_evaluations,
        const size_t points_count) const -> model_points_set_t;
```

7.1 System of Inequalities Implementation

The implementation of the system of inequality is located in a separate class, named *system_of_inequalities.h*. To calculate these systems of inequalities, a third party library is used, which is specialized to solve systems of inequalities. The library's name is Kiwi. Since the system of inequalities are of a special type only two operations are introduced:

system_of_inequalities.h

```
    enum class inequality_operation
    {
        LE, // less or equal
        G, // greater
    };
```

An inequality is added to the system with the method:

system_of_inequalities.h

```
     /*
      * Returns true if the system is solvable and the added inequality
      *     does not makes the system unsolvable.
      */
    auto add_constraint(const variables_set& lhs,
        const variables_set& rhs,
        inequality_operation op) -> bool;
```

The check whether the system has a solution is done with:

system_of_inequalities.h

```
    /*
     * Returns true if the system is still solvable.
     */
    auto is_solvable() const -> bool;
```

The final result when the system of inequalities is solvable can be taken with the following method:

system_of_inequalities.h

```
/*
 * If the system is solvable, returns a vector of values,
 *      which satisfy the system.
 * Element at position 'i' is the value of the i-th variable.
 * If the system is not solvable, returns an empty vector.
 */
auto get_variables_values() const -> std::vector<double>;
```

7.2 Measured Less Operator Representation

All t he researched third party libraries for solving systems of inequalities work only with less or equal than inequalities. This means that the greater than inequalities must be simulated with the usage of less or equal than inequalities.
This absence is solved with the addition of a really small variable while converting the greater inequality to less or equal inequality.

Let us have the following inequality

$$\left\{\sum_{i^1} X_{i^1} > \sum_{j^1} X_{j^1}\right.$$

then this inequality is transformed into:

$$\left\{-\sum_{i^1} X_{i^1} + \sum_{j^1} X_{j^1} + \epsilon \leq 0\right.$$

where ϵ is a small value.

The inequality X > 0, where X is a variable is transformed into:

$$\left\{0 \leq X - \epsilon\right.$$

Google's linear solver Glop (OR-Tools) was one of the tested libraries and it has precision around $1 * 10^{-7}$. Kiwi's precision is around $1 * 10^{-8}$.
These results are based on simple empirical testing, namely:
A simple system of only one inequality $X > 0$ which is converted to $X - \epsilon >= 0$. While testing the ϵ value was slowly decreasing and when it gets smaller than $1 * 10^{-7}$ ($1 * 10^{-6}$ for google's Glop (OR-Tools)) the solver gave a wrong answer.

Besides this anomaly, the precision is good enough for the purposes of finding solutions for the special type

Building the System of Inequalities The system construction is done by adding constraints for all measured atomic formulas. This is done by the
has_solvable_system_of_inequalities method:

measured_model.h

```
auto measured_model::has_solvable_system_of_inequalities() -> bool
{
    // For each <=m(a,b) calculate v(a) and v(b), then we will create
    // an inequality of the following type:
    //     SUM_I Xi <= SUM_J Xj, where I is v(a) and J is v(b).
    // For each ~<=m(a,b) calculate v(a) and v(b), then we will create
    // an inequality of the following type:
    //     SUM_I Xi > SUM_J Xj, where I is v(a) and J is v(b).
    // Each inequality is a row in the system of inequalities.
    // If this system has a solution, then we are good.

    const auto points_size = points_.size();
    system_.clear();

    for(const auto& m : measured_less_eq_T_)
    {
        const auto v_a = m->get_left_child_term()->evaluate(
                variable_evaluations_, points_size);
        const auto v_b = m->get_right_child_term()->evaluate(
                variable_evaluations_, points_size);
        if (!system_.add_constraint(v_a, v_b,
                system_of_inequalities::inequality_operation::LE))
        {
            verbose() << "Unable to find a solution for the system"
                "when adding the constraint for " << *m << "\n" <<
                *static_cast<imodel* const>(this);
            return false;
        }
    }

    for(const auto& m : measured_less_eq_F_)
    {
        const auto v_a = m->get_left_child_term()->evaluate(
                variable_evaluations_, points_size);
        const auto v_b = m->get_right_child_term()->evaluate(
                variable_evaluations_, points_size);
        if (!system_.add_constraint(v_a, v_b,
                system_of_inequalities::inequality_operation::G))
        {
            std::stringstream out;
            print_system(out);
            verbose() << "Unable to find a solution for the system"
                "when adding the constraint for ~" << *m << "\n" <<
                *static_cast<imodel* const>(this);
            return false;
        }
    }

    info() << "Found a solution for the system:\n" <<
                *static_cast<imodel* const>(this);
    return true;
}
```

Model Points Valuations Generation The generation of new model point valuations is done as follows:

measured_model.h

```
auto measured_model::generate_next() -> bool
{
    for (auto& point : points_)
    {
        if (point.evaluation.generate_next_evaluation())
        {
            calculate_the_model_evaluation_of_each_variable();
            return true;
        }

        const auto is_reset = point.evaluation.reset();
        assert(is_reset);
        (void)is_reset;
    }
    return false;
}
```

7.3 Building Algorithm

The building algorithm using the described methods. Follows the Lemma 3.20.

measured_model.h/cpp

```
auto measured_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& measured_less_eq_T,
    const formulas_t& measured_less_eq_F,
    const variables_mask_t& used_variables,
    const formula_mgr* mgr)
    -> bool
{
    clear();
    mgr_ = mgr;
    used_variables_ = used_variables;
    number_of_contacts_ = contacts_T.size();

    measured_less_eq_T_ = measured_less_eq_T;
    measured_less_eq_F_ = measured_less_eq_F;

    if (!construct_contact_model_points(contacts_T) ||
        !construct_non_zero_model_points(zero_terms_F))
    {
        trace() << "Unable to construct the model points "
         "with binary variable evaluations which evaluates "
         "the contact & non-zero terms to 1";
        return false;
    }

    auto additional_points = measured_less_eq_T.size() +
                measured_less_eq_F.size();
    if(points_.empty() && additional_points == 0)
```

```
    {
        additional_points = 1;
    }
    if(additional_points > 0)
    {
        info() << "Adding␣additional␣" << additional_points <<
            "␣points␣because␣of␣the␣<=m/~<=m␣atomics␣or␣empty␣model.␣"
            "We␣need␣a␣potential␣one␣model␣point␣for␣each␣side␣"
         "of␣each␣inequality␣in␣the␣system.";
        constant_true_ = std::make_unique<term>(mgr_);
        constant_true_->construct_constant(true);
        points_.insert(points_.end(), additional_points,
            {
                constant_true_.get(),
                variables_evaluations_block_for_positive_term(
                *constant_true_, used_variables_)
            });
    }
    create_contact_relations_first_2k_in_contact(points_.size(),
         contacts_T.size());
    calculate_the_model_evaluation_of_each_variable();

    system_ = system_of_inequalities(points_.size());
    while (!is_zero_term_rule_satisfied(zero_terms_T) ||
           !is_contact_F_rule_satisfied(contacts_F) ||
           !has_solvable_system_of_inequalities())
    {
        TERMINATE_IF_NEEDED();
        if (!generate_next())
        {
            trace() << "Unable␣to␣generate␣a␣new␣combination␣of␣"
                "␣binary␣variable␣evaluations␣for␣the␣model␣points.";
            return false;
        }
    }

    return true;
}
```

8 Rest Server

The Web Server is used to serve:

- Rest APIs
- Resources

The Rest APIs are used to chek for satisfiability, to find connected models or meassured models.
The resources are:

- Html pages
- Images
- Javascript code
- Styles

The Web Interface is easy to use, contains only the needed information and from there can be executed all of the described satisfiability algorithms.

One client can execute only one algorithmic program at a time. This way it is ensured that there are not a lot of simultaneous executing programs by the server.

If the program execution time gets too long the client has the possibility to terminate the execution of the current program by server. This will re-enable the client to execute another program and will remove the execution load from the server.

If the client posts a couple formula for calculation and terminetes the session (for example closes the browser), then the task in the backend will be terminated as well.

Output

The output of the program is separated in three modules:

- Resulting output - indicates what was the final result of the execution.
  For example: The formula is satisfiable.
- Verbose output - This is the output which contains the full proof of the formula execution.
  This output is printed though the whole execution.
- Visualized graph - This is the end result of the model, if such model exists
  The visualization is done with a JavaScript third party library for drawing graphs.

References

[1] Flex Tokenizer,
http://web.mit.edu/gnu/doc/html/flex_1.html

[2] Bison Parser,
https://www.gnu.org/software/bison/

[3] Visitor Pattern,
https://en.wikipedia.org/wiki/Visitor_pattern