

# Satisfiability Of Modal Logic Formulas

Martin Stoev, Anton Dudov

2019-9-28

## Contents

<b>1</b>	<b>Contact Logic Syntax Representation</b>	<b>4</b>
1.1	Term inductive definition . . . . .	6
1.2	Formula Inductive Definition . . . . .	9
1.3	Model Point Representation . . . . .	12
1.4	Contact Matrix . . . . .	12
1.5	Bitsets in Contact . . . . .	12
1.6	Model Representation . . . . .	13
1.7	Satisfiability Check Idea . . . . .	14
<b>2</b>	<b>Input Formula Analyzation</b>	<b>15</b>
2.1	Symbols Representation . . . . .	15
2.2	Tokenizer . . . . .	16
2.3	Abstract Syntax Tree . . . . .	20
2.3.1	Operation types in the formula/term nodes . . . . .	20
2.4	Visitors Implementation . . . . .	21
2.5	Formula Refiners . . . . .	23
<b>3</b>	<b>Tableaux</b>	<b>27</b>
3.1	Definition: Tableaux Step . . . . .	27
3.1.1	Rules . . . . .	27
3.2	Tableaux implementation . . . . .	28
3.2.1	Handy methods . . . . .	31
3.2.2	Def: Tableaux branch output . . . . .	33
<b>4</b>	<b>Model of formula</b>	<b>38</b>
4.1	Fast Model Algorithm . . . . .	42
4.2	Connected Model . . . . .	46
4.3	System of Inequalities . . . . .	49
4.4	System Construction . . . . .	49
4.5	Measured Less Operator Representation . . . . .	49
4.6	Measured Model . . . . .	51
<b>5</b>	<b>Rest Server</b>	<b>54</b>

TODO: Theoretical intro should be added about the region based modal logics.  
Definitions of Contact,  $\leq$ ,  $\leq_m$ ...  
TODO #REF should be replaced with the real reference of a definition.

# 1 Contact Logic Syntax Representation

Few paragraphs about the modal logic and basic story of how and what we are trying to accomplish

## Def: Variable

A sequence of characters and numbers compose a variable.

Example variables:

- x0
- x1
- foo
- obj111

Let  $\varphi$  be a formula. Then the set of all used variables in  $\varphi$  will be denoted as  $\mathbb{V}$ . From now on we shall use the enumeration of the set  $\mathbb{V}$  for performance optimizations. Namely instead of comparing and storing strings as variables we shall work with their integer enumerations.

## Def: Bitset

A sequence of 1s and 0s compose a bitset. If the sequence has size k, then it will be called k-bitset.

Let x be a bitset, namely  $x = a_0, a_1, \dots$ , then the i-th element of the bitset x is  $a_i$ . To get the i-th element the following notation will be used:

$$x[i] \tag{1}$$

which returns the  $a_i$  element.

## Operations over bitsets

The standard component-wise boolean operations shall be used. The standart operations include AND, OR and NEGATION.

**bitset-PlusOne** Let A be a bitset. Then the operation plus one will be denoted as  $A + 1$ . The bitset  $A + 1$  is simply a binary addition of A and 1. The followings are examples for the plus one operation:

$$\begin{aligned} A &= 0001 \\ A + 1 &= 0010 \end{aligned}$$

$$A = 1010$$

$$A + 1 = 1011$$

$$A = 1011$$

$$A + 1 = 1100$$

**bitset-PlusOne over marked bits** Let  $A$  be a bitset.  $A$  represents the bitset from which the next bitset will be produced. Let  $M$  be a bitset denoting the marked bits. The PlusOne over marked bitset is denoted as  $A \oplus_M 1$ . The  $A \oplus_M 1$  is constructed as  $A + 1$  operation but only on the marked by  $M$  bits:

The followings are examples for the plus one operation:

$$A = 0011$$

$$M = 1111$$

$$A \oplus_M 1 = 0100$$

$$A = 0001$$

$$M = 0101$$

$$A \oplus_M 1 = 0100$$

$$A = 0011$$

$$M = 1011$$

$$A \oplus_M 1 = 1000$$

### Def: Variable bitset substitution

Let  $\mathbb{B}_{set}$  be a set of bitsets, then the Variable bitset substitution is a total mapping  $\rho : \mathbb{V} \mapsto \mathbb{B}_{set}$  and the following notation will be used:

$$x \mapsto b \tag{2}$$

where  $x$  is a variable and  $b$  is a bitset.

### Def: Variable boolean substitution

The Variable boolean substitution is a total mapping  $\rho : \mathbb{V} \mapsto \{0, 1\}$  and the following notation will be used:

$$x \mapsto b \tag{3}$$

where  $x$  is a variable and  $b \in \{1, 0\}$ .

## Special types of bitsets

Let  $A$  be a  $k$ -bitset, then  $A$  is called

- the universe  $k$ -bitset, if  $\forall 0 \leq i < k : A[i] = 1$ , denoted with 1
- the empty  $k$ -bitset, if  $\forall 0 \leq i < k : A[i] = 0$ , denoted with 0

### 1.1 Term inductive definition

- 1 is a constat term
- 0 is a constant term
- $p \in \mathbb{V}ar$  is a term
- If  $x$  is a term, then  $\bar{x}$  is a term as well
- If  $x$  and  $y$  are terms, then  $x\sigma y$  is a term as well,  
where  $\sigma \in \{\sqcap, \sqcup\}$

## Term Implementation Representation

The term is represented inductively by analogy to the definition REF-DEF-TERM. Namely the term is represented as a binary tree. A inner node represent a term operation( union, intersection, complement). A leaf node represent a variable or a constant.

```
class Term
{
    ...
    enum class operation_type : char
    {
        constant_true ,
        constant_false ,

        union_ ,
        intersection ,
        complement ,
        variable ,

        invalid ,
    };
    using operation_t = operation_type;
    ...

    operation_t op_;
    std::size_t hash_;

    struct childs
    {
        term* left;
        term* right;
    }
}
```

```

union
{
    childs childs_;
    std::size_t variable_id_;
}

...
}

```

**Term hash variable** Conducting a test whether two terms are equal in a lexical way is an important procedure for the sake of performant satisfiability algorithms. Such equality checks are required in various situations, such as:

- testing if two terms are equal
- checking if a term exists in a set of terms

Have in mind that the naive solution to do an equality check has complexity of  $O(n)$ , where  $n$  is the size of the term. This solution is to compare the whole structure of the terms. To reduce this complexity a precalculated hash value shall be used. For each term one additional variable is stored. This variable is the hashed term. The hash is computed recursively through the term structure, which means that the hash of a term depends on the hashes of its subterms ( if any ). Let  $\tau_1$  and  $\tau_2$  be two terms. Let  $h_1$  be the precalculated hash of  $\tau_1$  and  $h_2$  be the precalculated hash of  $\tau_2$ . 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 the full structure of the terms is compared.

The following is the implemented hash construction procedure:

```

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_ = ((childs_.left->get_hash() & 0xFFFFFFFF) * 2654435761) +
                    (childs_.right->get_hash() & 0xFFFFFFFF) * 2654435741;
            break;
        case operation_t::complement:
            hash_ = (childs_.left->get_hash() & 0xFFFFFFFF) * 2654435761;
            break;
        case operation_t::variable:
            hash_ = (variable_id_ & 0xFFFFFFFF) * 2654435761;
            break;
        default:
            assert(false && "Unrecognized.");
    }
}

```

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

## Term Set-Evaluation

Let  $X$  be a set of variable Set-Substitutions, namely  $X = \{x_0 \mapsto s_0, x_1 \mapsto s_1, \dots, x_k \mapsto s_k\}$ , then the Term Set-Evaluation is a function which produces a bitset for a given term by applying a set of substitutions. The Term Set-Evaluation is applicable if and only if all variables from the term have a suitable substitution in  $X$ .

The Term Set-Evaluation will be notated as `termSEval` is defined recursively as follows:  
Let  $\tau$  be a term, then:

- if  $\tau \equiv 1$ , then  
return the world bitset
- if  $\tau \equiv 0$ , then  
return the empty bitset
- Let  $v \in X$  and  $\tau \equiv v$ , then  
return  $X[v]$
- Let  $\rho$  be a term and  $\tau \equiv \bar{\rho}$ , then  
return  $inverse(termSEval(\rho))$
- Let  $\tau_1, \tau_2$  be two terms and  $\tau \equiv \tau_1 \sqcap \tau_2$ , then  
return  $termSEval(\tau_1) \& termSEval(\tau_2)$
- Let  $\tau_1, \tau_2$  be two terms and  $\tau \equiv \tau_1 \sqcup \tau_2$ , then  
return  $termSEval(\tau_1) | termSEval(\tau_2)$

## Term Binary-Evaluation

Let  $X$  be a set of variable Boolean-Substitutions, namely  $X = \{x_0 \mapsto b_0, x_1 \mapsto b_1, \dots, x_k \mapsto b_k\}$ , where  $b_i$  is 0 or 1. Thus the boolean variable  $X[i]$  is  $b_i$ . The Term Binary-Evaluation is a function. For a given term and a set of variable Boolean-Substitutions it produces 0 or 1.

The Term Binary-Evaluation will be notated as `termBinEval` is defined recursively as follows:

Let  $\tau$  be a term, then:

- if  $\tau \equiv 1$ , then  
return 1
- if  $\tau \equiv 0$ , then  
return 0



- Let  $v \in X$  and  $\tau \equiv v$ , then  
return  $X[v]$
- Let  $\rho$  be a term and  $\tau \equiv \bar{\rho}$ , then  
return  $\sim (termBinEval(\rho))$
- Let  $\tau_1, \tau_2$  be two terms and  $\tau \equiv \tau_1 \sqcap \tau_2$ , then  
return  $termBinEval(\tau_1) \ \& \ termBinEval(\tau_2)$
- Let  $\tau_1, \tau_2$  be two terms and  $\tau \equiv \tau_1 \sqcup \tau_2$ , then  
return  $termBinEval(\tau_1) \ | \ termBinEval(\tau_2)$

## 1.2 Formula Inductive Definition

A formula is defined by the following inductive definition:

- $T$  is a formula, which represents  $\top$
- $F$  is a formula, which represents  $\perp$
- $C(a, b)$  is a formula, where  $a$  and  $b$  are terms
- $a \leq b$  is a formula, where  $a$  and  $b$  are terms
- If  $\varphi$  is a formula, then  $\bar{\varphi}$  is a formula as well
- If  $\varphi$  and  $\psi$  are formulas, then  $\varphi \sigma \psi$  is a formula as well,  
where  $\sigma \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

### Def: Zero term formula

Let  $a$  and  $b$  be two terms, then

$$a \leq b \iff a \sqcap \neg b = 0 \quad (4)$$

$a \sqcap \neg b = 0$  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 easily grasped in proofs and definitions.

### Def Atomic Formulas

A formula  $\varphi$  is called atomic, if and only if it is one of the followings:

- $T$
- $F$
- $C(a, b)$ , where  $a$  and  $b$  are terms
- $a \leq b$ , where  $a$  and  $b$  are terms
- $a = 0$ , where  $a$  is term

**Less or Equal Formula** The less or equal formula describes the subregion operation and is notated as  $\leq$ . Let  $a$  and  $b$  be two regions, then  $a$  is subregion of  $b$  if all elements from  $a$  are in  $b$ . Regions are described with the usage of terms, and since the terms are represented with  $k$ -bitsets it means that if at some position  $i$ ,  $a[i] = 1$  then  $b[i] = 1$  as well, namely

$$a \leq b := \forall i < k : a[i] = 1 \implies b[i] = 1 \quad (5)$$

The check if  $a$  is a subregion of  $b$  can be easily implemented with the following sequence of bitset operations

$$a \& b = a \quad (6)$$

**Contact Formula** The contact formula describes the relation between regions notated with  $C(a, b)$ , where  $a$  and  $b$  are two terms. A region is in a contact with another region if there exists at least one element from both regions which are in a relation, let us denote the relation with  $R$ .

$$C(a, b) := \exists x \in a \exists y \in b x R y \quad (7)$$

## Formula Implementation Representation

The formula can be represented in memory inductively by analogy to the definition REF-DEF-FORMULA.

```
class Formula
{
    ..
    enum class operation_type : char
    {
        constant_true ,
        constant_false ,

        conjunction ,
        disjunction ,
        negation ,

        measured_less_eq ,
        eq_zero ,
        c ,

        invalid ,
    };
    using operation_t = operation_type;
    ...

    operation_t op_;
    std::size_t hash_;

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

```

    term* left;
    term* right;
};

union {
    child_formulas child_f_;
    child_terms child_t_;
};
}

```

The formula analogous to the term implementation is represented in a tree structural way, where each node is an operation over formulas and the leafs of the tree are atomic formulas.

**Formula hash variable** The importance of the formula hash variable is analogous to the Term hash variable. Conducting a test whether two formulas are equal in lexical way is an important procedure for the sake of performant satisfiability algorithms. Such equality checks are required in various situations, such as:

- testing if two formulas are equal
- checking if a formula exists in a set of terms

Have in mind that the naive solution to do an equality check has complexity of  $O(n)$ , where  $n$  is the size of the formula. This solution is to compare the whole structure of the formulas. With the use of precalculated hash value this complexity can be reduced to an amortized complexity of  $O(1)$ . For each formula one additional variable is stored. This variable is the hashed formula. The hash is computed recursively through the formula structure, which means that the hash of a formula depends on the hashes of its subformulas or subterms ( if any ). Let  $\varphi_1$  and  $\varphi_2$  be two formulas. Let  $h_1$  be the precalculated hash of  $\varphi_1$  and  $h_2$  be the precalculated hash of  $\varphi_2$ . The equality check procedure follows the following steps:

- if  $h_1 \neq h_2$ , then the formulas are not equal
- if  $h_1 = h_2$ , then the structures of both formulas are compared.

The contact operation requires a special check because of the commutativity of Contacts is supported, namely

Let  $a, b$  be two terms, then

$$C(a, b) = C(b, a) \quad (8)$$

Checking whether two atomic contact formulas are lexically equal is done as follows:

$$C(a, b) = C(c, d) \rightarrow (a = c \wedge b = d) \vee (a = d \wedge b = c). \quad (9)$$

The implementation of the formula hash construction procedure is analogous to the term hash construction defined in Term Implementation Representation.

### 1.3 Model Point Representation

A model point  $p_0$  is represented as an integer number. The first constructed model point is represented with 0, the second with 1 and so on.

### 1.4 Contact Matrix

A matrix of  $n \times n$  size, where the rows and columns are model points and has only boolean values is called Contact Matrix. Contact matrices usually will be noted as  $\mathbb{C}^m$ . The following property is always true for Contact Matrices.

$$\text{There is a contact between model points } i \text{ and } j \iff \mathbb{C}^m[i][j] = 1 \quad (10)$$

The Contact Matrix of size  $n$  is implemented with  $n$  bitsets and  $\mathbb{C}^m[i]$  represents the  $i^{th}$  bitset.

### 1.5 Bitsets in Contact

Let  $a, b$  be two bitsets

Let  $\mathbb{C}^m$  be a Contact Matrix

then

$$C(a, b) \iff (a \& b \neq 0) \text{ or } \exists i < n : a[i] = 1 \text{ and } (\mathbb{C}^m[i] \& b \neq 0)$$

1.  $an(a \& b)$  is true if and only if  $a$  and  $b$  have a common point
2.  $\exists i < n : a[i] = 1 \text{ and } any(\mathbb{C}^m[i] \& b)$  is true if and only if  $a$  contains a model point which is in contact with a model point from  $b$  according to the Contact Matrix.

### Formula Set-Evaluation

Let  $X$  be a set of variable Set-Substitutions, namely  $X = \{x_0 \mapsto b_0, x_1 \mapsto b_1, \dots, x_k \mapsto b_k\}$ , Let  $\mathbb{C}^m[i]$  be a contact matrix, then the Formula Set-Evaluation is a function which determines the validity of a formula, namely for a given term produces a boolean result by applying a set of substitutions. The Formula Set-Evaluation is applicable if and only if all variables from the term have a suitable substitution in  $X$ .

The Formula Set-Evaluation will be notated as formulaSEval is defined recursively as follows:

Let  $\varphi$  be a formula, then:

- if  $\varphi \equiv T$ , then  
return true
- if  $\varphi \equiv F$ , then  
return false

- Let  $\tau$  be a term and  $\varphi \equiv \tau = 0$ , then  
return  $termSEval(\tau) = 0$
- Let  $\tau_1, \tau_2$  be two terms and  $\varphi \equiv C(\tau_1, \tau_2)$ , then  
let  $left = termSEval(\tau_1)$ ,  
let  $right = termSEval(\tau_2)$ ,

$$return \begin{cases} 1, & \text{left and right bitsets are in contact according to } \mathbb{C}^m[i] \\ 0, & \text{otherwise} \end{cases}$$

- Let  $\psi$  be a formula and  $\varphi \equiv \neg\psi$ , then  
return  $!formulaSEval(\psi)$
- Let  $\psi_1, \psi_2$  be two formulas and  $\varphi \equiv \psi_1 \wedge \psi_2$ , then  
return  $formulaSEval(\psi_1) \& formulaSEval(\psi_2)$
- Let  $\psi_1, \psi_2$  be two formulas and  $\varphi \equiv \psi_1 \vee \psi_2$ , then  
return  $formulaSEval(\psi_1) | formulaSEval(\psi_2)$

## 1.6 Model Representation

A model  $\mathcal{M}$  can be decomposed to its key components and each of these components can be programmatically represented, namely a model is represented from the following components:

- $W$  - is the set model points, where  $W$  is the whole world.
- $R$  - is the reflexive and symmetric relation between model points.
- $\nu$  - is an evaluation function which maps a variable to a set of points subset of  $W$

In order to construct a model programmatically each of these components has to be represented in a programmable way.

**Model Points** Each model point is represented as an integer value, counting from zero to the size of the points. Each model point's integer value is unique and is less than the number of model points. This allows compact storage and the representation is done with a simple integer value, which represents the number of the model point with highest integer value. Referencing a model point is simply done with the integer value which represents that model point.

**Model Points Relation** The relation might be between all points, namely one point might be in a relation with all other points, including itself. To represent such conditions an adjacency matrix can be used, where the rows and columns of the adjacency matrix represent the integer representation of the model points and the adjacency matrix values are simple boolean values indicating whether two model points are in relation. Such adjacency matrix has been defined in the Contact Matrix section 1.4.

Maybe add an example matrix.

**Evaluation function** This function determines the mapping from formula variables to sets of model points which are subset of  $W$ . The representation of such mapping is done with an associative array. Since the formula variables are numbered from zero to the size of all different formula variables, the keys are expressed with integer values less than the size of all different formula variables. The values of this associative array represent regions of model points, for which a detailed definition has been given in [REF TO BISET REGIONS]

Add example as well, or maybe an example for a model.

## 1.7 Satisfiability Check Idea

Let  $\varphi$  be a modal formula.

Problem: Does there exists a model  $\mathcal{M}$ , such that:

$$\mathcal{M} \models \varphi \quad (11)$$

This problem, namely to construct a valid model  $\mathcal{M}$  in which the formula  $\varphi$  is satisfiable is with NP-complete problem. In any case of algorithm selection there will be exponentially complex case. The idea of the algorithmic schema is to first determine the maximum model points  $W$ , needed to be able to satisfy a modal formula. Try to construct the relations matrix  $R$  from the modal formula context, and then to construct an evaluation function  $\sqsubseteq$  such that:

$$\langle W, R, \nu \rangle \models \varphi \quad (12)$$

This idea is used in several algorithms defined and explained in details in the following sections.

## 2 Input Formula Analyzation

The idea behind the overall solution is to have a formula as an input string and to output whether the given formula is statisfiable maybe a more detailed output will contain a graph of the model or a proof why the formula is not statisfiable. The first step in implementing such solution is to:

- choose a set of symbols for the representation of formula components and the operations over them.
- translate the input formula string into a more readable format where the language symbols are determined. In this new format it should be easy to manipulate with the formula and even to have the ability to do transformation over the original formula.

### 2.1 Symbols Representation

#### Atomic terms

- 0 is the term constant False
- 1 is the term constant True
- Identifier containing only letters and numbers represents a variable. Lowercase and uppercase letter are allowed.

**Terms operations** Let  $t_1, t_2$  be terms, then the followings are the representations for term operations

- $\neg t_1$  is used to represent the complement term operation, namely  $\overline{t_1}$
- $t_1 * t_2$  is used to represent the intersection term operation, namely  $t_1 \sqcap t_2$
- $t_1 + t_2$  is used to represent the union term operation, namely  $t_1 \sqcup t_2$

**Atomic formulas** Let  $t_1, t_2$  be terms, then the followings are representation of atomic formulas

- T is the formula constant True
- F is the formula constant False
- $C(t_1, t_2)$  is the contact operation
- $\leq (t_1, t_2)$  is the less or equal operation
- $\leq m(t_1, t_2)$  is the measured less or equal operation
- $(t_1) = 0$  is the equal to zero operation

**Formula operations** Let  $\varphi_1, \varphi_2$  be terms, then the followings are the representations for term operations

- $\sim\varphi_1$  is used to represent  $\overline{\varphi_1}$
- $\varphi_1 \& \varphi_2$  is used to represent  $\varphi_1 \wedge \varphi_2$
- $\varphi_1 | \varphi_2$  is used to represent  $\varphi_1 \vee \varphi_2$

Flex [1] and Bison [2] are used to generate a parser with the modal logic grammar. Flex is used as a tokenizer. Bison is used as the parser. There are two types of building elements in the modal logic formula:

- terms, defined in the Term Inductive Definition section 1.1
- formulas, defined in the Formula Inductive Definition section 1.2

## 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 whitespaces are not significant and a tokens are not created for them. We shall use Flex as a tokenizer [1].

**Grammar** The tokenizer's grammar is composed from two types of symbols

- Single character tokens
- Multi character tokens

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];



In the above table the ‘do nothing’ token is marked as ; , which means to ignore all whitespaces. The yytext[0] is the matched character. All single character tokens are passed as their ASCII code for an easier use in Bison. The last matched symbol in the table represents everything else, if nothing has been matched then just return the text itself. Later on the parser will use it to prompt where the unrecognized symbol was found and the symbol itself can be printed out.

**Parser literals** 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 '(' ')'

**Parser grammar** With the usage of the Parser literals, the input formula can be parsed and an Abstract Syntax Tree(AST) can be built from it. 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 and will produce better performance when a model is sought in the satisfiability algorithms. For example the following formula

$$X \rightarrow Y \quad (13)$$

can be transformed to

$$\sim X|Y \quad (14)$$

after the AST is built, and the implication operation can even be removed from all the calculations in the satisfiability algorithms, since all implications can be transformed when the AST is built.

The following construction methods will be used in the following parser algorithm:

- create\_term\_node
- create\_formula\_node

Both methods construct AST nodes. The create\_term\_node method creates a term AST node from an operation and one or two other terms depending on the operation arity. The create\_formula\_node analogous to the create\_term\_node method, creates a formula AST node from an operation and depending on the operation the followings can be arguments of create\_formula\_node

- one or two terms
- one or two formulas

Few special symbols to define beforehand:

**Returned Value** \$\$ is the return value to the 'parent' who can use the matched subsequence

**Matched Symbol at Position** \$i, where i is an integer value represents the i-th matched symbol

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

```
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;
}
;

```

## 2.3 Abstract Syntax Tree

The resulting AST can be easily modified and optimized, this is best achieved with the visitor pattern [3]. The visitor pattern uses double virtual dispatching.

### 2.3.1 Operation types in the formula/term nodes

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

```
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
};
```

**Term and formula types** These two types are defined with a separate class one for term and one for formula.

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

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

    void accept(Visitor& v) override { v.visit(*this); }

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

```

};

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

    void accept(Visitor& v) override { v.visit(*this); }

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

```

**Visitor Structure** The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. The visitor pattern allows for a new visitors to be added in a simple manner. The new visitor should override the two visit methods and can read or modify the AST nodes.

```

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

class VPrinter : public Visitor
{
public:
    void visit(NFormula& f) override
    {
        // Here it prints the formula node information
    }

    void visit(NTerm& t) override
    {
        // Here it prints the term node information
    }
};

```

## 2.4 Visitors Implementation

### VReduceConstants

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

---



---

• $\sim T \equiv F$	• $C(0,0) \equiv F$	• $\sim F \equiv T$	• $C(1,1) \equiv T$
• $(T \ \& \ T) \equiv T$	• $C(a,0) \equiv F$	• $(F \mid F) \equiv F$	• $C(0,a) \equiv F$
• $(g \ \& \ T) \equiv g$	• $-1 \equiv 0$	• $(g \mid T) \equiv T$	• $-0 \equiv 1$
• $(T \ \& \ g) \equiv g$	• $(1 * 1) \equiv 1$	• $(T \mid g) \equiv T$	• $(0 + 0) \equiv 0$
• $(g \ \& \ F) \equiv F$	• $(t * 1) \equiv t$	• $(g \mid F) \equiv g$	• $(t + 1) \equiv 1$
• $(F \ \& \ g) \equiv F$	• $(1 * t) \equiv t$	• $(F \mid g) \equiv g$	• $(1 + t) \equiv 1$
• $0=0 \equiv T$	• $(t * 0) \equiv 0$	• $1=0 \equiv F$	• $(t + 0) \equiv t$
• $<=(0,a) \equiv T$	• $(0 * t) \equiv 0$	• $<=(a,1) \equiv T$	• $(0 + t) \equiv t$

---

### VConvertContactsWithConstantTerms

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

- $C(a,1) \equiv \sim(a=0)$
- $C(1,a) \equiv \sim(a=0)$

### VConvertLessEqContactWithEqualTerms

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

- $<=(a,a) \equiv T$ ,  
since  $(a * -a = 0)$
- $C(a,a) \equiv \sim(a=0)$

### VReduceDoubleNegation

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

- $\sim(\sim g) \equiv g$
- $\sim(\sim t) \equiv t$

### 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.

- $(f \rightarrow g) \equiv (\sim f \mid g)$
- $(f \leftrightarrow g) \equiv ((f \ \& \ g) \mid (\sim f \ \& \ \sim g))$

### **VConvertLessEqToEqZero**

Converts a  $\leq$  formula to an equals to zero atomic formula

- $\leq(a,b) \equiv (a * -b) = 0$

### **VSplitDisjInLessEqAndContacts**

Divides C and  $\leq$  atomic formulas with a disjunction term into two simpler formulas

- $C(a + b, c) \equiv C(a, c) \mid C(b, c)$
- $C(a, b + c) \equiv C(a, b) \mid C(a, c)$
- $\leq(a + b, c) \equiv \leq(a,c) \ \& \ \leq(b,c)$

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

## **2.5 Formula Refiners**

The formula refiners are an optional parameter when building a formula node. They are a set of optional filters which may optimize the formula by reducing redundant parts of the formula or by transforming some nodes in a way that they will be easier to be evaluated later on.

Most important formula refiners:

- Convert C and  $\leq$  atomic formulas which have identical terms
- Convert disjunction in C and  $\leq$  atomic formulas
- Reduce constants
- Reduce C with constants
- Remove double negation

These formula refiners are done when the formula is built, meaning that after the AST tree is constructed each of these refiners enables or disables a visitor. The formula refiners are mapped in a one to one manner to a subset of all the visitors defined in the Visitors section 2.4.

- Convert Contacts and less or equal atomic formula which have same terms  
is mapped to VConvertLessEqContactWithEqualTerms
- Convert disjunction in contacts and less or equal atomic formulas  
is mapped to VSplitDisjInLessEqAndContacts
- Reduce constants  
is mapped to VReduceConstants

- Reduce contacts with constants  
is mapped to `VConvertContactsWithConstantTerms`
- Remove double negation  
is mapped to `VReduceDoubleNegation`

## Implementation

The formula refiners are just a flag denoting if a formula refiner should be applied.

[illegible]



```

// For not just convert them after the splitting.
// It's just a small optimization.
if(has_flag(refiners_flags ,
    formula_refiners::convert_contact_less_eq_with_same_terms))
{
    VConvertLessEqContactWithEqualTerms
        convertor_lessEq_contact_with_equal_terms;
    formula_AST->accept(convertor_lessEq_contact_with_equal_terms);
    info_buff << "Converted  $C(a,a); \leq(a,a)_{\square\square}:\square$ ";
    formula_AST->accept(printer);
}

if(has_flag(refiners_flags ,
    formula_refiners::convert_disjunction_in_contact_less_eq))
{
    VSplitDisjInLessEqAndContacts disj_in_contact_splitter;
    formula_AST->accept(disj_in_contact_splitter);
    info_buff << "C(a+b,c)->C(a,c)|C(b,c) $_{\square}$ ;\n";
    info_buff << "  $\leq(a+b,c) \rightarrow \leq(a,c) \& \leq(b,c) : \square$ ";
    formula_AST->accept(printer);
}

if(has_flag(refiners_flags ,
    formula_refiners::convert_contact_less_eq_with_same_terms))
{
    VConvertLessEqContactWithEqualTerms
        convertor_lessEq_contact_with_equal_terms;
    formula_AST->accept(convertor_lessEq_contact_with_equal_terms);
    info_buff << "Converted  $C(a,a); \leq(a,a)_{\square\square}:\square$ ";
    formula_AST->accept(printer);
}

VConvertLessEqToEqZero eq_zero_convertor;
formula_AST->accept(eq_zero_convertor);
info_buff << "Converted  $\square(\leq\square=0)_{\square}$  formula  $\square:\square$ ";
formula_AST->accept(printer);

if(has_flag(refiners_flags ,
    formula_refiners::reduce_constants))
{
    VReduceConstants trivial_reducer;
    formula_AST->accept(trivial_reducer);
    info_buff << "Reduced  $\square$  constants  $\square\square\square\square\square\square\square\square:\square$ ";
    formula_AST->accept(printer);
}

if(has_flag(refiners_flags ,
    formula_refiners::reduce_contacts_with_constants))
{
    VConvertContactsWithConstantTerms
        contacts_with_constant_as_term_convertor;
    formula_AST->accept(contacts_with_constant_as_term_convertor);
    info_buff << "Converted  $C(a,1) \rightarrow \sim(a=0)_{\square\square}:\square$ ";
    formula_AST->accept(printer);
}

if(has_flag(refiners_flags ,

```

```

        formula_refiners::remove_double_negation))
    {
        VReduceDoubleNegation double_negation_reducer;
        formula_AST->accept(double_negation_reducer);
        info_buff << "Reduced_double_negation::";
        formula_AST->accept(printer);
    }

    info() << info_buff.str();

    // Will cash all variables and when building the formula
    // tree we will use their ids instead of the heavy strings
    VVariablesGetter::variables_set_t variables;
    VVariablesGetter variables_getter(variables);
    formula_AST->accept(variables_getter);

    variables_.reserve(variables.size());
    variable_to_id_.reserve(variables.size());
    for(const auto& variable : variables)
    {
        variable_to_id_[variable] = variables_.size();
        variables_.emplace_back(variable);
    }

    // The building of the formula from it's AST
    return f_.build(*formula_AST);
}

```

### Node evaluation

TODO: this should be explained very well, and not just here but with multiple definitions on theory level

### 3 Tableaux

The Tableaux process is a decision procedure, which recursively breaks down a given formula into basic components based on which a decision can be concluded. The recursive step which breaks down a formula creates one or two new formulas, which in terms of their structure are simpler than the initial formula. The recursive step can create at most two new formulas. This means that at most two branches can be created. Continuously applying the recursive step produces a binary tree, where the nodes are the formulas and the links represent the recursive step. In each branch a set of formula is obtained which are considered to be in conjunction. The set of formulas from different branches are considered to be in disjunction.

Contradiction may arise when in the same branch, on some step there exists a formula and its negation. If in some branch there exists a contradiction, then that branch closes. If all branches close then the proof is complete.

The main principle of the tableaux is to break complex formula into smaller ones until complementary pairs of literals are produced or no further expansion is possible.

#### 3.1 Definition: Tableaux Step

##### Definition: Signed Formula

Let  $\varphi$  be a formula, then:

- $\mathbb{T}\varphi$  signs  $\varphi$  as true
- $\mathbb{F}\varphi$  signs  $\varphi$  as false

##### Definition: Signed formulas set

The signed formulas set consists only of signed formulas and the letter X will be usually used for its representation.

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.

##### 3.1.1 Rules

###### Negation

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

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

###### And

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

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

**Or**

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

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

**Implication**

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

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

**Equivalence**

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

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

The final output of the Tableaux process is "False" when all branches are closed or a set of atomic formulas, when there exists a branch which is not closed.

For our usecases the functionality of the tableaux process shall be extended to achieve better results, since 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 which 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.

## 3.2 Tableaux implementation

The program implementation of the tableaux method follows the standard tableaux process. For detailed explanation see TODO:THEORY.

First interesting design decision is to keep all true signed formulas in one data set, and all false signed formulas in another data set. This enables fast searches whether a formula has been signed as true or false.

### Definition: Signed Formula Collection

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

This collection is implemented with `unordered_map` (hashmap), which stores the formulas by pointers to them, uses their precalculated hash and `operator==` to compare them. The average complexity for search, insert and erase in this collection is  $O(1)$ .

There exist 8 important 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  $\varphi$  is inserted into the formula\_F
- contacts\_T - contains only contact formulas signed as true
- contacts\_F - contains only 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

### Definition: Formula Contradiction

Let  $\varphi$  be a signed formula, then  $\varphi$  is causing a contradiction if any of the followings is true:

- $\varphi$  is a non-atomic signed as true and  $\varphi$  belongs to formulas\_F
- $\varphi$  is a non-atomic signed as false and  $\varphi$  belongs to formulas\_T
- $\varphi$  is a contact formula signed as true and  $\varphi$  belongs to contacts\_F
- $\varphi$  is a contact formula signed as false and  $\varphi$  belongs to contacts\_T
- $\varphi$  is a zero terms formula signed as true and  $\varphi$  belongs to zero\_terms\_F
- $\varphi$  is a zero terms formula signed as false and  $\varphi$  belongs to zero\_terms\_T
- $\varphi$  is a measured less formula signed as true and  $\varphi$  belongs to measured\_less\_eq\_F
- $\varphi$  is a measured less formula signed as false and  $\varphi$  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 of the resulting components causes a contradiction.

### Example

Let's assume that  $\text{contacts\_T} = \{ C(a, b) \}$  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  $\text{contacts\_T}$  formulas

### Tableaux Algorithm

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

As a first step if the formula  $\varphi$  is the constant F, then false is returned directly, otherwise the whole formula  $\varphi$  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.

### Lemma: A

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

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

for any term  $y$ .

### Lemma: A-inverse

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

### Time Complexity A and A-inverse

The algorithmic complexity to check whether a new formula leads to contradiction by Lemma A and Lemma A-inverse is done effectively, namely in constant time with the usage of one new collection `contact_T_terms_` which 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.

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

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

### 3.2.1 Handy methods

#### Find formula

##### Find formula signed as true

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

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

- $C(x, y)$ , then return true  $\iff \varphi$  belongs to contacts\_T
- $x \leq y$ , then return true  $\iff \varphi$  belongs to zero\_terms\_T
- $x \leq_m y$ , then return true  $\iff \varphi$  belongs to measured\_less\_eq\_T
- $\neg\psi$ , then return true  $\iff \varphi$  belongs to formulas\_T
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then return true  $\iff \varphi$  belongs to formulas\_T

##### Find formula signed as false

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

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

- $C(x, y)$ , then return true  $\iff \varphi$  belongs to contacts\_F
- $x \leq y$ , then return true  $\iff \varphi$  belongs to zero\_terms\_F
- $x \leq_m y$ , then return true  $\iff \varphi$  belongs to measured\_less\_eq\_F
- $\neg\psi$ , then return true  $\iff \varphi$  belongs to formulas\_F
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then return true  $\iff \varphi$  belongs to formulas\_F

#### Add formula

##### Mark formula as true

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

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

- $C(x, y)$ , then  $\varphi$  is added to contacts\_T, and the terms  $x$  and  $y$  are added to the contact\_T\_terms\_collection.
- $x = 0$ , then  $x$  is added in zero\_terms\_T
- $x \leq_m y$ , then  $\varphi$  is added to measured\_less\_eq\_T
- $\neg\psi$ , then  $\varphi$  is added to formulas\_T
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then  $\varphi$  is added to formulas\_T

### Mark formula as false

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

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

- $C(x, y)$ , then  $\varphi$  is added to `contacts_F`.
- $x = 0$ , then  $x$  is added in `zero_terms_F`
- $x \leq_m y$ , then  $\varphi$  is added to `measured_less_eq_F`
- $\neg\psi$ , then  $\varphi$  is added to `formulas_F`
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then  $\varphi$  is added to `formulas_F`

### Remove formula

#### Remove formula signed as true

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

Removes the formula  $\varphi$  from the respective positive collection. Namely if  $\varphi$  is of type:

- $C(x, y)$ , then  $\varphi$  is removed from `contacts_T`, and the terms  $x$  and  $y$  are removed from the `contact_T_terms_` collection.
- $x = 0$ , then  $x$  is removed from `zero_terms_T`
- $x \leq_m y$ , then  $\varphi$  is removed from `measured_less_eq_T`
- $\neg\psi$ , then  $\varphi$  is removed from `formulas_T`
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then  $\varphi$  is removed from `formulas_T`

#### Remove formula signed as true

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

Removes the formula  $\varphi$  from the respective negative collection. Namely if  $\varphi$  is of type:

- $C(x, y)$ , then  $\varphi$  is removed from `contacts_F`
- $x = 0$ , then  $x$  is removed from `zero_terms_F`
- $x \leq_m y$ , then  $\varphi$  is removed from `measured_less_eq_F`
- $\neg\psi$ , then  $\varphi$  is removed from `formulas_F`
- $\psi_1\sigma\psi_2$ , where  $\sigma \in \{\wedge, \vee\}$ , then  $\varphi$  is removed from `formulas_F`



### 3.2.2 Def: Tableaux branch output

As stated above the output of a branch in the tableaux process is a set of atomic formulas. These atomic formulas can be grouped in six groups:

- Contacts
- Non Contacts
- Equal to Zero Terms
- Not Equal to Zero Terms
- Measured Equal to Zero Terms
- Measured Not Equal to Zero Terms

These groups can be represented with the following formula:

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

### Tableaux Satisfiable Step Implementation

```

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())
    {
        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 it multiple times
{
    return satisfiable_step ();
}

add_formula_to_F(X);
auto res = satisfiable_step ();
remove_formula_from_F(X);
return res;
}

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

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

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

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

    return res;
}

assert(op == op_t::disjunction);

//  $T(X \vee Y) \rightarrow T(X) \vee T(Y)$ 
auto X = f->get_left_child_formula();
auto Y = f->get_right_child_formula();
trace() << "Will split into 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))
        {
            return true; // there was no contradiction in the left path,
                        // so there is no need to continue with
                        // the right path
        }

        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)
{
    //  $F(X \vee Y) \rightarrow F(X) \& F(Y)$ 
    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);

```

```

//  $F(X \& Y) \rightarrow F(X) \vee F(Y)$ 
auto X = f->get_left_child_formula();
auto Y = f->get_right_child_formula();

trace() << "Will split into 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))
{
    return true; // there was no contradiction in left path,
                // so there is no need to continue with the
                // right path
}

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

```

## 4 Model of formula

Let  $\mathcal{M}$  be a model, defined as in 1.6. Let  $\varphi$  be a formula. Then  $\mathcal{M}$  is a model for  $\varphi$ , if and only if

$$\mathcal{M} \models \varphi \quad (15)$$

A brief explanation on how a model can be represented has been given in 1.6. In this section the implementation details will be discussed in depth. All algorithms in this paper are seeking for a model which model the input formula, thus a common representation of the model representation can be done. This common model representation shall be used for all algorithms defined in the following sections.

**Model Data** As per 1.6 to represent a model the three components of the model need to be represented, namely the:

- Set of model points
- Evaluation Function
- Contact Matrix

**Points** The points are numbered sequentially starting from zero. This means that having the number of the points in the universe is sufficient to uniquely determine each model point. A region in the described Kripke style semantics is a subset of the universe. So, the regions are represented as bitsets with size equal to the size of the model (the number of the points).

**Evaluation Function** The evaluation function is represented with a `variable_evaluations_` variable which type is as defined in 1.6, namely an associative array of size the number of variables in the formula, each element is a set of points represented as a bitset. Keeps the variable evaluations. i.e.  $v(p)$ . Model evaluation  $\mathbf{v}$  which for a given term returns a subset of the model points:

- $v(p) = \{ i \mid (i) (xxx...x)[p] == 1, \text{ i.e. in point 'i' the evaluation of the variable 'p' is 1 } \}$ ,  
where 'p' is a variable (for optimizations it is an id of the string representation of that variable)
- $v(a * b) = v(a) \& v(b)$
- $v(a + b) = v(a) \mid v(b)$
- $v(-a) = \tilde{v}(a)$

For example, let us have the following formula

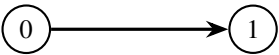
$$C(a * b, c) \& c! = 0 \& -a! = 0 \quad (16)$$


Then the formula model could be the following:


(110) (a\*b) 0 — 1 (c) (001)

(011) (c) 2

(000) (-a) 3

(110) (a\*b)  (c) (001)

(011) (c) 

(000) (-a) 

Let us assume that the variable ids are as:

- ‘a’ has variable id 0
- ‘b’ has variable id 1
- ‘c’ has variable id 2

The bit matrix will be the following:

	0	1	2	3
0	1	0	0	0
1	1	0	1	0
2	0	1	1	0

where the rows represent the variable ids, the columns represent the model points. The followings are the evaluations of a, b and described in the above matrix, namely the evaluation is composed from the positions where there is a one in the column:

- $v(a) = \{ 0 \}$
- $v(b) = \{ 0, 2 \}$
- $v(c) = \{ 1, 2 \}$

**Contact Matrix** The contact matrix is represented with a `contact_relations_` variable which type is defined in 1.4. The `contact_relations_` variable is a symmetric square matrix. The matrix row `contact_relations_[ i ]` gives a bitset of all points which are in contact with the point **i**.

For example, let us have a model with 6 points:

0 — 1 , contact between 0 and 1

2 — 3 , contact between 2 and 3

4 , point created from  $\neq 0$  term

5 , point created from  $\neq 0$  term

The contact\_relations\_ will be the following:

	0	1	2	3	4	5
0	1	1	0	0	0	0
1	1	1	0	0	0	0
2	0	0	1	1	0	0
3	0	0	1	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	0	1

**Model Create** All models can be created(or at least an attempt can be done for their creation) with the Tableaux branch output 3.2.2 as its input arguments. This is achieved via the 'create' method.

```
virtual auto 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 = 0;
```

For simpler explanations in the following algorithms for formula model construction the following constants are introduced. They represent the sizes for the formulas collections which are passed as an argument to the create method of each model construction algorithm.

- N - number of variables in the atomic formulas
- M - number of atomic formulas
- I - number of Contact atomic formulas
- J - number of Non Contact atomic formulas
- K - number of Not Equal to Zero atomic formulas
- L - number of Equal to Zero atomic formulas
- S - number of Measured Not Equal to Zero atomic formulas
- U - number of Measured Equal to Zero atomic formulas

The creation of a model can be done with various algorithms. In this paper the following algorithms will be defined and described in details:

- Fast Model



- Connected Model
- Measured Model

## 4.1 Fast Model Algorithm

A model is simply a set of points and contacts between them which satisfies the Contacts and Less or equal atomic formulas.

This model will be easily explained with an example first:

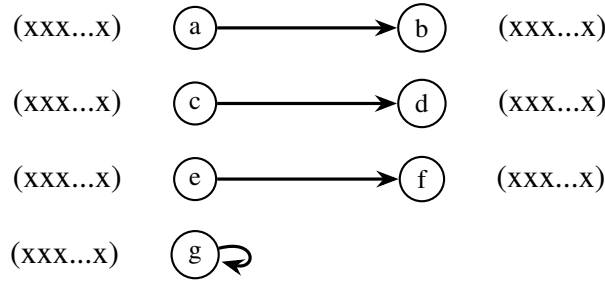
### Example 1.1

Let  $\varphi$  be the following formula:

$$\varphi = C(a, b) \& C(c, d) \& C(e, f) \& g! = 0 \& x = 0 \& C(y, z)$$

This formula is constructed from 3 contacts, one not equal to zero term, one equal to zero term and one non contact atomic formula.

The model for this formula is constructed from seven modal points, namely two modal points for each contact and one point for the not equal to zero term.



In the above graph the (xxx...x) is a bitset, for example (010...1) which gives zero/one evaluation for the variables in the formula. Its size is the number of different variables in the formula. The model points are the nodes in the graph ranging from 0 to 6. Each node connected to its self, this way the reflexivity of the contacts is represented, and the edges represent the contact relations.

One note can be made that in the above graph the Zero terms and Non Contact atomic formulas are not represented since they do not require existence of model points.

### Definition Model Evaluation

Model evaluation  $v$  which for a given term returns a subset of the model points. The model evaluation  $v$  is defined as:

$$v(p) = \{i | (i)(xxx...x)[p] == 1\}$$

which means that for a given variable  $p$ ,  $v(p)$  returns a set of all points in which the variable  $p$  is evaluated to true.

### Properties

- $v(a * b) = v(a) \& v(b)$

- $v(a + b) = v(a) \mid v(b)$
- $v(-a) = v(a)$

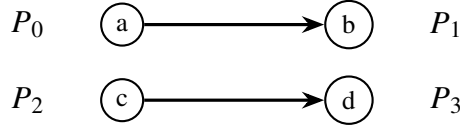
Note that each point's evaluation evaluates it's term to the constant true in order to that point to be in the MODEL evaluation of the term, i.e. for the point 0, the term 'a' and it's evaluation (xxx..x):  $a \rightarrow \text{evaluate}(\text{xxx}..x) = \text{constant\_true}$  in order to 0 belongs to  $v(a)$

### Example 1.2

Let  $\varphi$  be the following formula:

$$\varphi = C(a, b) \& C(c, d) \& e = 0 \& f = 0 \& C(g, h) \& C(i, j)$$

Then the model should be of the following type:



NOTE that if  $P_i$  evaluates the term 't' to constant true then the model evaluation  $v(t)$  contains the point  $(P_i \_)$  (and vice versa)

Where:

$P_0$  evaluates(binary) a to constant true && e and f to constant false

$P_1$  evaluates(binary) b to constant true && e and f to constant false

$P_2$  evaluates(binary) c to constant true && e and f to constant false

$P_3$  evaluates(binary) d to constant true && e and f to constant false

Also, to satisfy the requirement for C: (we need to check each contact relation):

$[(P_0 \text{ evaluates } g \text{ to constant true} \&\& P_1 \text{ evaluates } h \text{ to constant true}) \mid \mid$

$(P_1 \text{ evaluates } g \text{ to constant true} \&\& P_0 \text{ evaluates } h \text{ to constant true})] \&\&$

$[(P_0 \text{ evaluates } i \text{ to constant true} \&\& P_1 \text{ evaluates } j \text{ to constant true}) \mid \mid$

$(P_1 \text{ evaluates } i \text{ to constant true} \&\& P_0 \text{ evaluates } j \text{ to constant true})] \&\& \dots$  analogous for  $P_2$  and  $P_3$ .

The contact relation is reflexive, so for each point we also need to check also:

$(P_i \text{ evaluates } g \text{ to constant true} \&\& P_i \text{ evaluates } h \text{ to constant true}) \&\&$

$(P_i \text{ evaluates } i \text{ to constant true} \&\& P_i \text{ evaluates } j \text{ to constant true})$

**Algorithm** Let us have  $\varphi$  in the format defined in the definition of the tableaux branch output. Let us define few constants which will ease the annotation:

- N - number of variables in the atomic formulas
- M - number of atomic formulas
- I - number of Contact atomic formulas
- J - number of Non Contact atomic formulas

- K - number of Not Equal to Zero atomic formulas
- L - number of Equal to Zero atomic formulas

Note: The term evaluation is with an accumulated complexity of  $O(L)$  which is not included in the  $O$  notation below.

The algorithm which creates such points in an iterative manner is as follows:

1) Creates contact points, i.e. for each  $C(a,b)$  we create two points  $P_a$  and  $P_b$ .

Note that a point is just an variable evaluation.

They satisfy the following rules:

- $P_a[a]=1$  and  $P_b[b]=1$ .
- does not break any  $t=0$  atomic formula,  
i.e.  $P_a[t]=0$  and  $P_b[t]=0$ .
- does not break the reflexivity part of all  $C(e,f)$  atomic formulas,  
i.e.  $P_a[e]=0 \ \& \ P_a[f]=0 \ \& \ P_b[e]=0 \ \& \ P_b[f]=0$ .
- does not break any  $C(e,f)$  atomic formulas,  
i.e.  $[(P_a[e]=1 \ \& \ P_b[f]=1) \mid (P_b[e]=1 \ \& \ P_a[f]=1)]$

If any of the rules above is not satisfied we change  $P_b$ 's evaluation, if not possible to generate a new evaluation for  $P_b$  then we change  $P_a$ 's evaluation and reset the  $P_b$ 's evaluation to the 'first' one.

If again some rules is broken we change  $P_b$ 's evaluation, etc. if we can't generate new evaluation for  $P_a$  then there is no satisfiable model.

Complexity for this step:  $O(2^{\hat{N}} * 2^{\hat{N}} * (G+M))$

2) Creates a point for each  $a!=0$  atomic formula -  $P_a$ , which satisfy the following rules:

- $P_a[a] = 1$
- does not break any  $t=0$  atomic formula,  
i.e.  $P_a[t]=0$
- does not break the reflexivity part of all  $C(e,f)$  atomic formulas,  
i.e.  $P_a[e]=0 \ \& \ P_a[f]=0$

If any of the rules above is not satisfied with  $P_a$ , then we generate next  $P_a$ 's evaluation and check them again. If not possible to generate next evaluation then there is no satisfiable model.

Complexity for this step:  $O(2^N * (G + M))$

3) We create the contact connectivity matrix -  $P \times P$  in a simple way because we know that only first  $2K$  points are in contact.  $0 \leq i < K$ : set a contact relation between point

i and i+1, i.e. we set the bit  $[i][i+1]$  to 1 and  $[i+1][i]$  to 1.

Complexity for this step:  $O(P^2)$

5) We calculate the  $v(X)$  for each variable. We create a vector of N bitsets. Each bitset will hold all points in the evaluation of that variable, i.e. bitset at position 'j' is the set  $v(j)$ , where 'j' is some variable id.

For each point  $P_i$ , we iterate over the set bits in its evaluation and if we have a set bit at position j we add that point to the evaluation of variable 'j', i.e. we set the bit 'i' in the variable's evaluation bitset.

Complexity for this step:  $O(P * N)$

Overall Complexity:  $O(2^N * 2^N * (G + M))$

## 4.2 Connected Model

A Connected model is called a formula model for which the following lemma is true.

### Connectivity Lemma

$$\forall a \in R : a \neq \emptyset \vee a \neq \mathbb{W} \leftarrow C(a, \bar{a}) \quad (17)$$

**Connected Model Algorithm** Let  $\varphi$  be the input formula, on which the tableaux process is applied, thus returning the tableaux branch output 3.2.2. 1) Let  $\mathbb{M}$  denote the model of all possible unique model points, where one model point corresponds to one variable evaluation  $P_i$ .

Since there are  $N$  variables, the number of unique model points is  $2^N$ .

1.1) Filter out the model points which interfere with the zero terms, which means that if  $P_i$  evaluates some term  $G_l$  to constant true then  $P_i$  is filtered out.

1.2) Filter out the model points which interfere with the Non contacts, which means that if  $P_i$  evaluates some terms  $E_j$  and  $F_j$  to true, then  $P_i$  is filtered out.

NOTE: that if  $P_i$  evaluates the term  $\tau$  to constant true then the model evaluation  $v(\tau)$  contains the point  $P_i$  (and vice versa).

So far, the model  $\mathbb{M}$  satisfies all Equal to Zero atomic formulas and the reflexivity property of the Non Contacts.

TODO: explain reflexivity of contacts in the theoretical part.

2) Verification is done for the Not Equal to Zero atomic formulas in  $\mathbb{M}$ , namely:

$$\text{If } \exists k : v(D_k) \neq \emptyset, \text{ then there does not exist a connected model.} \quad (18)$$

2.1) Verification is done for the Contact atomic formulas in  $\mathbb{M}$ , namely:

$$\text{If } \exists i : v(A_i) = \emptyset \vee v(B_i) = \emptyset, \text{ then there does not exist a connected model.} \quad (19)$$

So far,  $O(2^N * N)$  bits of memory to hold all points and their evaluations and  $O(2^N * M)$  time to process them.

3) Let  $\mathbb{C}$  be a bit matrix of size  $2^N \times 2^N$ .

This matrix will be called connectivity matrix and will represent the connected points in  $\mathbb{M}$ .

$$\text{Model points } i \text{ and } j \text{ are in contact if } \mathbb{C}[i][j] \text{ is set.} \quad (20)$$

Initially all points are in contact, which means that the initial value of each bit is set:

$$\text{Let } \forall i, j < 2^N : \mathbb{C}[i][j] = 1 \quad (21)$$

3.1) Remove all connections which interfere with Non Contact atomic formulas.

$$\begin{aligned} & \forall a, b < 2^N, \forall j < J : \\ & \text{if } P_a[E_j] = 1 \wedge P_b[F_j] = 1 \vee P_a[F_j] = 1 \wedge P_b[E_j] = 1, \\ & \text{then unset } \mathbb{C}[a][b] \text{ and } \mathbb{C}[b][a]. \end{aligned}$$

Note that the reflexivity has been satisfied in 1).

For this step  $O(2^N * 2^N) = O(2^{2*N})$  bits of memory were used and total of  $O(2^N * 2^N * J)$  time complexity.

4) Check that all Contacts are still satisfied

$$\forall i, j < 2^N : C(A_i, B_i) \text{ is satisfied} \quad (22)$$

The check whether  $C(A_i, B_i)$  is satisfied is done in the following manner:

$$\forall p \in v(A_i) : (\mathbb{C}[p] \text{ bit\& } v(B_i)) \neq 0 \quad (23)$$

Where  $\mathbb{C}[p]$  is a bitset representing all contacts of p and  $v(B_i)$  is a bitset representing the evaluation of  $B_i$ .

Thus  $(\mathbb{C}[p] \text{ bit\& } v(B_i)) \neq 0$  means that there is a common point between the evaluation of  $B_i$  and the contacts of p.

The complexity for this step is:  $O(I * (H + L * G))$ , where:

- H is the complexity for evaluation some term, it's at most the number of operations in the term, something small.
- L is the number of points in  $v(A_i)$  which is at most  $2^N$ .
- G is the complexity for "bit operator &" which is  $2^N / 2^6$ .

5) The previous step produces the maximal unique satisfiable model, which is a graph  $\mathbb{G}$ . If there exists a satisfiable connected subgraph in the  $\mathbb{G}$ , then this is the connected model. Finding all connected subgraphs has time complexity of  $O(2^N + 2^N * 2^N)$ . Checking if a subset of points satisfies the formula can be done in a similar fashion (as the one above).

Note that only the check for Not Equal to Zero and Contacts atomic formulas has to be done, since when a point is removed the Equal to Zero and Non Contacts atomic formulas can not be broken.

Let Y be the number of points in the subset, then at most  $O(Y * Y * M)$  time complexity. Note that this must be done for each connected subgraph and the subgraphs are not overlapping! Upper bound for Y is  $2^N$ , so the upper bound and on the time complexity is:

$$O(2^N * 2^N * M),$$

since

$$x * x * M + y * y * M \ll (x + y) * (x + y) * M$$

Overall memory complexity:  $O(2^N * 2^N)$  bits  
Overall time complexity:  $O(2^N * 2^N * M)$



### 4.3 System of Inequalities

The systems are of the following type:

$$\left\{ \begin{array}{l} \sum_{i^1} X_{i^1} \leq \sum_{j^1} X_{j^1} \\ \dots \\ \sum_{i^n} X_{i^n} \leq \sum_{j^n} X_{j^n} \\ \sum_{k^1} X_{k^1} > \sum_{l^1} X_{l^1} \\ \dots \\ \sum_{k^m} X_{k^m} > \sum_{l^m} X_{l^m} \end{array} \right.$$

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.

### 4.4 System Construction

The system of inequalities is constructed from the potential measured model by evaluating each of the terms in the Measured less or equals and Measured greater atomic formulas. The points in the model are enumerated from 0 to N - 1, where N is the number of model points. The system will have N different variables  $X_0, X_1, \dots, X_n$ , where  $\forall i < N : X_i$  is mapped to point i

TODO: Add beter theoretical explanation

1) For each  $\leq m(x, y)$  calculate  $v(x)$  and  $v(y)$ , then an inequality of the following type is created:

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

If some X is on both sides of an inequality, then both of them are removed.

2) For each  $\sim \leq m(x, y)$  calculate  $v(x)$  and  $v(y)$ , then an inequality of the following type is created:

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

If some X is on both sides of an inequality, then both of them are removed.

3) For each variable in the model a new greater than constraint is added to the system  
If the system of inequalities constructed in this maner has solution, then this system of inequalities satisfies the measured model.

### 4.5 Measured Less Operator Representation

All of the researched third party libraries for solving systems of inequalities work only with Less or Equal inequalities. This means that the greater inequalities must be

simulated with the usage of Less or Equal 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

$$\{\sum_{i^1} X_{i^1} > \sum_{j^1} X_{j^1}\}$$

then this inequality is transformed into:

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

where  $\epsilon$  is a small value.

The inequality  $X > 0$ , which is a variable to be greater than zero is transformed into:

$$\{0 \leq X - \epsilon\}$$

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  $X - \epsilon \geq 0$ . While testing the  $\epsilon$  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 of systems of inequalities defined above.

## 4.6 Measured Model

Let us have  $\varphi$  in the format defined in the definition of the tableaux branch output.

### Points construction

The Measured Model is composed of modal points, the number of modal points is  $I + L + S + U$ .

The following atomic formulas produce an existence of a modal point:

- $C(A_i, B_i)$  produces two modal points, where  $x$  and  $y$  are terms.
- $g_l \neq 0$  produces one modal point, where  $x$  is a term.
- $\leq_m (H_s, O_s)$  produces one modal point
- $\neg(\leq_m (Q_u, R_u))$  produces one modal point

For each contact  $C(A_i, B_i)$  two new model points are created  $A$  and  $B$ , where  $A \in v(A_i)$  and  $B \in v(B_i)$ . In some cases the point created for a contact might be one, but this in turn is simulated by having the same evaluation for two points, which means that the point is duplicated.

Each Point will be represented with  $P(\tau, e)$ , where  $\tau$  is a term and  $e$  is the variable evaluation. The point's evaluation  $P[e]$  positively evaluates the point's term  $P[\tau]$ , meaning that  $P$  belongs to  $v(P[\tau])$ .

Example: Let's have  $C(a + b, c * d)$

then a valid model point for ' $a + b$ ' is  $P('a+b', [1000])$ , where  $[1000]$  is the evaluation for  $(a, b, c, d)$  variables. That evaluation evaluates ' $a+b$ ' to true.

For ' $c*d$ ' a point could be:  $P('c*d', [0011])$ .

All possible combinations of such points are  $2^{N*(I+L+S+U)}$ , since there are  $I + L + S + U$  number of points and each point has  $2^N$  potential evaluations.

The following algorithm determines if there is a satisfiable measured model.

### Measured Model Algorithm

- 1) For each  $C(A_i, B_i)$  two points are created  $P_a$  and  $P_b$ :  $P_a[A_i] = true$  and  $P_b[B_i] = true$ . If not possible - then there is no satisfiable model.
- 2) For each  $g_l \neq 0$  one point is created  $P$ :  $P[g_l] = true$ . If not possible - then there is no satisfiable model.
- 3) For each measured atomic formula a modal point is created which is associated to '1' constant term.
- 4) A connection is made between the contact points, first  $2*I$  points are from Contact terms, i.e. for each  $0 \leq i < I$  a connection is added in the connectivity matrix

between  $2*i$  and  $2*i + 1$  points.

A new combination of points is generated by 9) until the steps from 5 to 8 are satisfied.

5) Calculate the  $v(X)$  for each variable.

6) For each  $G_l \neq 0$  evaluate  $v(G_l)$  and check that it is not the empty set.  $O(L)$

7) For each  $\sim C(E_j, F_j)$  check that  $v(E_j)$  and  $v(F_j)$  does not contain a common point and that  $v(E_j)$  does not contain a point which is in contact with a point  $v(F_j)$ .

Complexity:  $O(J * (I + J) * G)$

- $G$  is the complexity for "bit operator &" which is something like  $(I + J)/2^6$ .

- $I+J$  is the number of points.

8) Check the measured atomics:

For each  $\leq_m (H_s, O_s)$  calculate  $v(H_s)$  and  $v(O_s)$  and create an inequality of the following type:

$$\sum_{i \in v(H_s)} X_i \leq \sum_{j \in v(O_s)} X_j$$

For each  $\neg(\leq_m (Q_u, R_u))$  calculate  $v(Q_u)$  and  $v(R_u)$ , create an inequality of the following type:

$$\sum_{i \in v(Q_u)} X_i > \sum_{j \in v(R_u)} X_j$$

Each inequality is a row in the system of inequalities. If this system has a solution, then the generated model is a satisfiable one.

Finish!

9) The generation of new combination of points:

Similar to binary +1 operation but instead of just 0 and 1 states on each position there are  $2^N$  combinations, from which some will not be valid because they will not evaluate the point's term to true

Here is an example of binary +1 operation:

$$000 + 1 = 001$$

$$001 + 1 = 010$$

$$010 + 1 = 011$$

$$011 + 1 = 100$$

$$100 + 1 = 101$$

$$101 + 1 = \dots$$

If there are two variables and two points and for the sake of simplicity ignore the term for now. At first there will be only 0s.

[00][00] where the first [00] is the evaluation of the first point which evaluates both variables to 0.

The next generations looks as follows:

```
[00][00] – (next combination)– > [00][01]
– > [00][10]
– > [00][11]
– > [01][00]
– > [01][01]
– > [01][11]
– > [10][00]
–..– > [11][11]
```

In the above example the [11][11] is the last combination.

When the condition that the point's evaluation should evaluate the term to true is taken into account, then all evaluation which do not evaluate the term to true are skipped.

Example: if the variables are 'x' and 'y', first point's term is 'x', second point's term is 'y' then the valid sequence of combinations will be:

```
[x : 1y : 0,' x'][x : 0y : 1,' y']– > [10][11]
– > [11][01]
– > [11][11]
```

TODO: THIS complexity should be explained and recalculated:

Complexity:  $O((I + L + S + U) * G)$ , where G is the complexity for finding a next evaluation for a point, namely finding an evaluation which evaluates the point's term to true.

## 5 Rest Server

The Web Server is used to serve:

- Rest APIs
- Resources

The Rest APIs are used to check for satisfiability, to find connected models or measured 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 terminates 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](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](https://en.wikipedia.org/wiki/Visitor_pattern)