Satisfiability Of Modal Logic Formulas

Martin Stoev, Anton Dudov

2019-9-28

Contents

TODO: Theoretical intro should be added about the region based modal logics. Definitions of Contact, <=, <=m...
TODO #REF should be replaced with the real reference of a definiton.

1 Formula 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

Def: Index Variable Function

Let ivf: $\mathbb{V} \longmapsto \mathbb{N}$ be a bijective function which maps a variable to an index integer number. This function is used to map a variable into a computer memory, since integers are easily stored and manipulated.

Def: Optimal Variables indexes set

Let S be a set of unique variables with size k, then there exists a variable index function ivf such that:

$$\forall i < k \; \exists v \in S : ivf(v) = i. \tag{1}$$

Proof

Arrange the variables from S in a sequence $x_0, x_1, ...x_k$, then take the index of the variable to be the value of the index variable function for that variable, namely:

$$\forall i < k \;\; : ivf(x_i) = i. \tag{2}$$

Def: Bitset

A sequence of zeros and ones compose a bitset.

If the sequence has size k, then it will be called k-bitset.

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

$$x[i] \tag{3}$$

which returns the a_i element.

Operations over bitsets

Let x and y be two k-bitsets, where $k \in \mathbb{N}$, then:

bitset-AND operation is defined as:

$$a\&b = a[0]\&b[0], a[1]\&b[1], .., a[k]\&b[k] \tag{4}$$

where the truth table for two basic variables x and y is:

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

bitset-OR operation is defined as:

$$a|b = a[0]|b[0], a[1]|b[1], .., a[k]|b[k] \tag{5}$$

where the truth table for two basic variables x and y is:

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

bitset-NEG operation(inverse function) is defined as:

$$a = a\bar{[}0], .., a\bar{[}k] \tag{6}$$

where the truth table for basic variable x is:

x	$\bar{x}$
0	1
1	0

bitset-PlusOne Deoted a + 1 represents the next bitset after a, which is constructed as follows:

- if $\forall i < k : a[i] \neq 0$, then there are no more k-bitsets, meaning that the next bitset has size k+1 and has only one one at position zero and all others are zeroes.
- Otherwise let i be the index of the last zero position, then the next bitset is constructed as:

  $$a[0]^0, ..., a[i-1]^{i-1}, 1^i, 0^{i+1}, ..., 0^k \tag{7}$$

  The upper right index denotes the position in the bitset sequence.

Def: Variable bitset substitution

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

$$x \longmapsto b \tag{8}$$

where x is a variable and b is a bitset.

Def: Variable boolean substitution

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

$$x \longmapsto b \tag{9}$$

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

Special types of bitsets

Let x be a k-bitset, then x is called

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

Term recursive definition

- 1 is a term
- 0 is a 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 can be represented in memory recursively by analogy to the definition REF-DEF-TERM.

```
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_;
  }

  ...
}
```

The term is represented in a tree structural way, where each node is a an operation over terms and the leafs of the tree are variables. In the above implementation the variables are represented by their index, which can be easily found with the Index Variable Function.

Term hash variable Used for faster comaprison of terms, namely if the hash value of two terms differs then those terms have different structure. However if the hash variables of two terms are equal then the strustures of both terms are comapred. The following method is used to calculate the hash of a term.

```
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;
}
```

Include example for one term building

Term Set-Evaluation

Let X be a set of variable Set-Substituitions, namely $X = \{x_0 \longmapsto s_0, x_1 \longmapsto s_1, ..., x_k \longmapsto s_k\}$, then the Term Set-Evaluation is a function which produces a bitset for a given term by appling 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 τ 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 ρ be a term and $\tau \equiv \bar{\rho}$, then
  return $inverse(termSEval(\rho))$
- Let τ_1, τ_2 be two terms and $\tau \equiv \tau_1 \sqcap \tau_2$, then
  return $termSEval(\tau_1)\&termSEval(\tau_2)$
- Let τ_1, τ_2 be two terms and $\tau \equiv \tau_1 \sqcup \tau_2$, then
  return $termSEval(\tau_1)|termSEval(\tau_2)$

1.1 Term Partial-Evaluation

Let X be a set of variable Boolean-Substituitions, namely $X = \{x_0 \longmapsto b_0, x_1 \longmapsto b_1, ..., x_k \longmapsto b_k\}$, then the Term Partial-Evaluation is a function which for a given term produces new term by appling a set of substituations. After the substitutions are applied the returned term can be a constant term, namely true/false or can be a term with a simpler structure.

1.1.1 Term Construction Functions

A new term can be constructed in two ways:

1. From a variable index
2. From operation and the terms involved in the operation

create_variable_node will denote the function which from a given variable index constructs a new term.

create_node will denote the function which constructs a new term from an operation σ and the involved terms, namely

- If σ is the negation operation, and τ is a term, then the constructed term is $\bar{\tau}$
- If $\sigma \in \{\sqcup, \sqcap\}$, and τ_1, τ_2 are terms, then the constrcuted term is $\tau_1 \sigma \tau_2$

The Term Partial-Evaluation will be notated as termParEval is defined recursively as follows:
Let τ be a term, then:

- if $\tau \equiv 1$, then
  return constant_true
- if $\tau \equiv 0$, then
  return constant_false
- Let $v \in X$ and $\tau \equiv v$, then
  return X[v]
- Let $v \notin X$ and $\tau \equiv v$, then
  return create_variable_node(v)
- Let ρ be a term and $\tau \equiv \bar{\rho}$, then
  return eval_negation(ρ)
- Let τ_1, τ_2 be two terms and $\tau \equiv \tau_1 \sqcap \tau_2$, then
  return eval_intersection(τ_1, τ_2)
- Let τ_1, τ_2 be two terms and $\tau \equiv \tau_1 \sqcup \tau_2$, then
  return eval_union(τ_1, τ_2)

where eval_negation, eval_intersection and eval_union are defined as follows:

eval_negation(τ_1) Let

$$res = termParEval(\tau) \tag{10}$$

And based on the evaluated result the following is returned.

$$return \begin{cases} 0, & res = 1 \\ 1, & res = 0 \\ \bar{\tau}, & otherwise \end{cases}$$

eval_intersection(τ_1, τ_2) Let

$$\begin{aligned} left &= termParEval(\tau_1) \\ right &= termParEval(\tau_2) \end{aligned}$$

And based on the evaluated left and right values, the following is returned.

$$return \begin{cases} 0, & left = 0 \; or \; right = 0 \\ 1, & left = 1 \; and \; right = 1 \\ \kappa, & left = 1 \; and \; right \equiv \kappa \\ \kappa, & left \equiv \kappa \; and \; right = 1 \\ \tau_1 \sqcap \tau_2, & otherwise \end{cases}$$

eval_union(τ_1, τ_2) Let

$$left = termParEval(\tau_1)$$
$$right = termParEval(\tau_2)$$

And based on the evaluated left and right values, the following is returned.

$$return \begin{cases} 1, & left = 1 \; or \; right = 1 \\ 0, & left = 0 \; and \; right = 0 \\ \kappa, & left = 0 \; and \; right \equiv \kappa \\ \kappa, & left \equiv \kappa \; and \; right = 0 \\ \tau_1 \sqcup \tau_2, & otherwise \end{cases}$$

Def Formula definiton

A formula is defined by the following recursive definition:

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

Def: Zero term formula

Let a and b be two terms, then

$$a \leq b \iff a \sqcap \neg b = 0 \tag{11}$$

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

Def Atomic Formulas

A formula φ 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 <= 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 <=. 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

$$a <= b := \forall i < k : a[i] = 1 \implies b[i] = 1 \tag{12}$$

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

$$a\&b = a \tag{13}$$

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.

$$C(a, b) := \exists x \in a \exists y \in b x R y \tag{14}$$

Formula Implementation Representation

The formula can be represented in memory recursively 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 Used for faster comaprison of formulas, namely if the hash value of two formulas differs then those formulas have different structure. However if the hash variables of two formulas are equal then the strustures of both formulas are comapred. The method which calculates the hash of a formula is similar to the one defined for the calculation of terms in REF-TERM-HASH-CALC, with the two differences that the formula operations are more than the operations over term and that for atomic formulas the hash from the term is used in order to construct the formula hash.

Formula Set-Evaluation

Let X be a set of variable Set-Substituitions, namely $X = \{x_0 \longmapsto b_0, x_1 \longmapsto b_1, ..., x_k \longmapsto b_k\}$, then the Formula Set-Evaluation is a function which determins the validity of a formula, namely for a given term produces a bolean result by appling 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 φ be a formula, then:

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

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

- Let τ be a term and $\varphi \equiv \tau = 0$, then
  return $termSEval(\tau) = 0$

- Let τ_1, τ_2 be two terms and $\varphi \equiv C(\tau_1, \tau_2)$, then
  let $leftl = termSEval(\tau_1)$,
  let $right = termSEval(\tau_2)$,
  return left & right or left is in relation with right

- Let ψ be a formula and $\varphi \equiv \neg\psi$, then
  return $!formulaSEval(\psi)$

- Let ψ_1, ψ_2 be two formulas and $\varphi \equiv \psi_1 \wedge \psi_2$, then
  return $formulaSEval(\psi_1) \& formulaSEval(\psi_2)$

- Let ψ_1, ψ_2 be two formulas and $\varphi \equiv \psi_1 \vee \psi_2$, then
  return $formulaSEval(\psi_1) | formulaSEval(\psi_2)$

2 Formula Representation1

Let $\mathbb{V}ar$ be the set of variables:

$$\mathbb{V}ar = \{p_0, p_1, p_2...\}$$

Let $\mathbb{C}_t$ be the set of Term constants:

$$\mathbb{C}_t = \{0, 1\}$$

2.1 Term recursive definition

- $a \in \mathbb{C}_t$ is a 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 $\sigma \in \{\sqcap, \sqcup\}$

Parentheses are used to define the priority of an operation.
Let $\mathbb{C}_f$ be the set of formula constants:

$$\mathbb{C}_f = \{T, F\}$$

2.2 Formula recursive definition

- $a \in \mathbb{C}_f$ is a formula
- If x and y are terms, then C(x, y) is a formula
- If x and y are terms, then $x \leq y$ is a formula
- If x and y are terms, then $x \leq_m y$ 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 $\sigma \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

Formula Parentheses

Parentheses are used to define the priority of an operation. They are defined for terms and formulae.

2.3 Definition: Atomic Formula

A formula will be called atomic formula if it is one of the followings:

- T
- F
- C(x, y)
- $x \leq y$
- $x \leq_m y$

where x and y are terms.

2.4 Definition/Theorem: Zero Term Formula

Let x and y be two terms, then:

$$x \leq y \iff x \sqcap \bar{y} = 0 \tag{15}$$

A formula in the form $x \sqcap \bar{y} = 0$ will be called Zero Term Formula.

2.5 Parser

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

- terms, defined in #REF Term Recursive Definition
- formulae, defined in #REF Formula Recursive Definition

2.5.1 Symbols Representation

Atomic terms

- 0 is the term constant False
- 1 is the term constant True
- Arbitrary string is used to represent a variable

Terms operations

Let t1, t2 be terms, then the followings are the representations for term operations

- - t1 is used to represent the complement term operation, namely $t\bar{1}$
- t1 * t2 is used to represent the intersection term operation, namely $t1 \sqcap t2$
- t1 + t2 is used to represent the union term operation, namely $t1 \sqcup t2$

Atomic formulae

Let t1, t2 be terms, then the followings are representation of atomic formulae

- T is the formula constant True
- F is the formula constant False
- C(t1, t2) is the contact operation
- <=(t1, t2) is the less or equal operation
- <=m(t1, t2) is the measured less or equal operation
- (t1)=0 is the equal to zero operation

2.6 Tokenizer

Grammar

The tokenizer's grammar is pretty simple and is explaned in details in the following table with Flex syntaxes:

Matched token	Formatted token	Description
[\t \n]	;	Returns nothing, ignore all whitespace
[,TF01()C&| *+-]	yytext[0];	yytext[0] is the matched character. All single charecter tokens will be passed as their ASCIIs for an easier use in Bison
"<="	T_LESS_EQ;	Returns a special literal which maps the '<=' sequence
"<=m"	T_MEASURED_LESS_EQ;	
"=0"	T_EQ_ZERO;	
"= 0"	T_EQ_ZERO;	
"->"	T_FORMULA_OP_IMPLICATION;	
"<->"	T_FORMULA_OP_EQUALITY;	
[a-zA-Z0-9]+	T_STRING;	Returns T_STRING literal type and the string value is written to yylval which can be later accessed from the parser. Note that it uses our simple memory manager to allocate this string in order to be able to safely free all allocated strings even when some syntax error occures
.	yytext[0];	bison will trigger an error if it's unrecognized symbol

Parser literals

One character tokens are passed as their ASCII to Bison. 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 "<->"

Parser literals

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

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

Parser grammar

In the process of parsing the formula the Abstract Syntax Tree(AST) is built as well, which later is used to make formula modifications, for example converting X -> Y to ~X | Y .
create_formula_node and create_term_node are two helper functions which construct AST nodes

```
formula // 'formula' non-terminal
    : 'T' { // trying to match token 'T'
        // $$ is the return value to the 'parent'
        // who can use the matched subsequence
        $$ = create_formula_node(constant_true);
    }
    | 'F' {
        $$ = create_formula_node(constant_false);
    }
    | 'C' '(' term ',' term ')' {
        // $3 and $5 gives the 'values' for the two
        // matched terms (3 and 5 are their positions)
        // and construct a contact node with them
        $$ = 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;
    }
  ;
```

Abstract Syntax Tree

The visitor pattern is used to modify the formula, namely it's AST tree.
The visitor pattern uses double virtual dispatching.

Operation types in the formula/term nodes

```
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
};

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

```
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
    }
};
```

The visitor pattern gives us the ability to add new visitors in a very simple manner. The new visitor should override the two visit methods and do whatever it wants with the ast nodes.

Visitors Implementation

VReduceConstants

Removes all unnecessary childs 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 contacts 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 contacts and <= atomic formulas with same 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 conjuction and disjunction. Main reason for that is to simplify the tableau method - to work only with conjunctions and disjunctions.

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

VConvertLessEqToEqZero

Converts the less equal atomic formula to an equals to zero atomic formula

- <=(a,b) ≡ (a * -b) = 0

VSplitDisjInLessEqAndContacts

Converts the less equal atomic formula to an equals to zero atomic formula

- C(a + b, c) ≡ C(a,c) | C(b,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

3 Formula Structure

The formula structure is similar to the AST structure, but with restricted formula operations and additional information. There are 2 types of nodes:

- formula
- term

Every member variable in the nodes is caluclated dulring its building, this way getting the hash code of a node is done in a constant time.

Formula Node Structure

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

    operation_type op_;
    formula_mgr* formula_mgr_;
    std::size_t hash_;

    // For simplicity raw pointers and union,
    // a better solution would be unique pointers and variants
    struct child_formulas
    {
        formula* left;
        formula* right;
    };
    struct child_terms
    {
        term* left;
        term* right;
    };
    union {
        child_formulas child_f_;
        child_terms child_t_;
    };

    auto build(const NFormula& f) -> bool;

    auto operator==(const formula& rhs) const -> bool;
    auto operator!=(const formula& rhs) const -> bool;
    // Evalates the formula but ignores all measured less or equal
    // atomic formulas, as if there were not existing
    auto evaluate(const variable_id_to_points_t& evals,
                  const contacts_t& contact_relations) const -> bool;
};
```

Term Node Structure

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

    operation_type op_;

    // Used to make a bitmask for the used variables in the
    // term(w.r.t all variables in the whole formula) and to print the
    // term with it's variable names instead of their (optimized) IDs.
    const formula_mgr* formula_mgr_;
    variables_mask_t variables_; // All used variables in the term.

    // For simplicity raw pointers and union,
    // a better solution would be unique pointers and variants
    struct childs
    {
        term* left;
        term* right;
    };
    union {
        childs childs_;
        size_t variable_id_;
    };

    std::size_t hash_;

    auto build(const NTerm& t) -> bool;

    auto operator==(const term& rhs) const -> bool;
    auto operator!=(const term& rhs) const -> bool;

    auto evaluate(const variable_id_to_points_t& variable_evaluations,
        const size_t points_count) const -> model_points_set_t;

    // Evaluates the term with the provided variables evaluations but if
    // some of the used variables in the term do not have evaluations
    // it creates a smaller subterm with the unknown variables.
    // Note that it reduces the term's constants, e.g. '1 + X' -> '1'
    // and does not look at the X's evaluation at all.
    // With the latest algorithms, this is not used.
    auto evaluate(const variables_evaluations_block& evaluation_block,
        bool skip_subterm_creation = false) const -> evaluation_result;
};
```

3.1 Building a formula node

The building of the formula structure is trivial, namely based on the AST node's operation types. Few important specifications of a node

- hash code
- variable mask for all used variables in the term(and it's child terms)

Node's hash code

The hash code of a node is mainly used to compare if two nodes are equal. This comparison is used in all maps and sets where the key is a node.

If for two formulas/terms nodes, their hash codes are same, then the content of the nodes is compared in order to verify that they are the same formulas/terms.
The contact operatior requires a special check since the commutativity of Contacts is supported.

Let a, b be two terms, then

$$C(a,b) = C(b,a) \tag{16}$$

therefore the check for equality will be done twice instead of once, namely

$$C(a,b) = C(c,d) \rightarrow (a = c \wedge b = d) \vee (a = d \wedge b = c). \tag{17}$$

TODO Explain item variable mask for all used variables in the term(and it's child terms)

3.1.1 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 Contacts and less or equal atomic formas which have same terms
- Convert disjunction in contacts and less or equal atomic formulas
- Reduce constants
- Reduce contacts 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 disabled a visitor. The formula refiners are mapped in a one to one manner to a subset of all the visitors defined in #REF visitor.

- 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.

```
enum class formula_refiners : int32_t
{
    none                                   = 0,
    convert_contact_less_eq_with_same_terms = 1 << 1,
    convert_disjunction_in_contact_less_eq  = 1 << 2,
    reduce_constants                       = 1 << 3,
    reduce_contacts_with_constants         = 1 << 4,
    remove_double_negation                 = 1 << 5,
    all = convert_contact_less_eq_with_same_terms
        | convert_disjunction_in_contact_less_eq
        | reduce_constants
        | reduce_contacts_with_constants
        | remove_double_negation
};
```

```
auto formula_mgr::build(const std::string& f,
        const formula_refiners& refiners_flags) -> bool
{
    // Parsing the formula
    parser_error_info error_info;
    auto formula_AST = parse_from_input_string(f.c_str(), error_info);
    if(!formula_AST)
    {
        std::stringstream error_msg;
        error_info.printer(f, error_msg);
        info() << "\n" << error_msg.str();
        return false;
    }

    std::stringstream info_buff;
```

```
info_buff << "Parsed␣formula:␣";
VPrinter printer(info_buff);
formula_AST->accept(printer);
info_buff << "\n";

VConvertImplicationEqualityToConjDisj convertor;
formula_AST->accept(convertor);
info_buff << "Converted␣(->␣<->)␣␣␣␣␣␣␣␣:␣";
formula_AST->accept(printer);

// NOTE: Consider making VSplitDisjInLessEqAndContacts
// and VSplitDisjInLessEqAndContacts combined because in some
// intermediate splitting the two terms might match.
// Nevertheless, this will still be not 100\% sufficient because
// the order of spliting might take a big role and skip
// some pontential matches.
// 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);<=(a,a)␣␣:␣";
    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)␣;\n";
    info_buff << "<=(a+b,c)-><=(a,c)&<=(b,c):␣";
    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);<=(a,a)␣␣:␣";
    formula_AST->accept(printer);
}

VConvertLessEqToEqZero eq_zero_convertor;
formula_AST->accept(eq_zero_convertor);
info_buff << "Converted␣(<=␣=0)␣formula␣:␣";
formula_AST->accept(printer);

if(has_flag(refiners_flags,
    formula_refiners::reduce_constants))
{
    VReduceConstants trivial_reducer;
```

```
        formula_AST->accept(trivial_reducer);
        info_buff << "Reduced␣constants␣␣␣␣␣␣␣␣␣:␣";
        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)->~(a=0)␣␣:␣";
        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 multilpe definitions on theory level

4 Tableaux

The Tableaux process is 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 then the initial formula. Since the recursive step can create at most two new formulas, this means the recursive step will create at most two branches or a binary tree, where the nodes are the formulas and the links represent the recursive step. The different branches are considered to be disjuncted while nodes of the same branch are considered in conjunction. The procedure modifies the tableau in such a way that the formula represented by the resulting tableau is equivalent to the original one.

Contradiction may arise when in the same branch, on some step there exists a formula and the negation of the same formula. 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 formulae into smaller ones until complementary pairs of literals are produced or no further expansion is possible.

4.1 Definition: Tableaux Step

Definition: Signed Formula

Let φ be a formula, then:

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

Definition: Signed Formulae set

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

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

4.1.1 Rules

Negation

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

And

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

Or

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

Implication

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

Equivalence

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

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

For our usecases the functionality of the tableaux process shall be extended to achive 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 formuas 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.

4.2 Tableaux implementation

The programming implementation of the tableaux method follows the standard tableaux process explaned above.

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

Definition: Signed Formula Collection

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

This collection is implemented with unordered_map (hashmap), which stores the formulae 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 formulae signed as true
- formulas_F - contains only non-atomic formulae 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 contacts formulae signed as true
- contacts_F - contains only contacts formulae signed as false
- zero_terms_T - contains only formulae of type $\varphi \leq \psi$ signed as true
- zero_terms_F - contains only formulae of type $\varphi \leq \psi$ signed as false
- measured_less_eq_T - contains only formulae of type $\varphi \leq_m \psi$ signed as true
- measured_less_eq_F - contains only formulae of type $\varphi \leq_m \psi$ signed as false

Definition: Formula Contradiction

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

- φ is a non-atomic signed as true and $\varphi \in$ formulas_F
- φ is a non-atomic signed as false and $\varphi \in$ formulas_T
- φ is a contact formula signed as true and $\varphi \in$ contacts_F
- φ is a contact formula signed as false and $\varphi \in$ contacts_T
- φ is a zero terms formula signed as true and $\varphi \in$ zero_terms_F
- φ is a zero terms formula signed as false and $\varphi \in$ zero_terms_T
- φ is a measured less formula signed as true and $\varphi \in$ measured_less_eq_F
- φ is a measured less formula signed as false and $\varphi \in$ measured_less_eq_T

Invariant

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

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

Example

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

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

Tableaux Algorithm

Given a formula φ, the following algorithm determines the atomic formulae 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 commutativity, meaning that: C(a,b) == 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 formulae as true will lead to contradiction:

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

for some arbitrary term y.

Lemma: A-inverse

Let x, y and z be terms, suppose that the atomic formulae 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: addd 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 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;
```

4.2.1 Handy methods

Find formula

Find formula signed as true

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

Checks if the formula φ exists in any positive collection depending on the type of φ, namely if φ is of type:

- C(x, y), then return true $\iff \varphi \in contacts\_T$
- $x \leq y$, then return true $\iff \varphi \in zero_terms\_T$
- $x \leq_m y$, then return true $\iff \varphi \in measured_less_eq\_T$
- $\neg\psi$, then return true $\iff \varphi \in formulas\_T$
- $\psi_1 \sigma \psi_2$, where $\sigma \in \{\wedge, \vee\}$, then return true $\iff \varphi \in formulas\_T$

Find formula signed as false

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

Checks if the formula φ exists in any negative collection depending on the type of φ, namely if φ is of type:

- C(x, y), then return true $\iff \varphi \in contacts\_F$
- $x \leq y$, then return true $\iff \varphi \in zero_terms\_F$
- $x \leq_m y$, then return true $\iff \varphi \in measured_less_eq\_F$
- $\neg\psi$, then return true $\iff \varphi \in formulas\_F$
- $\psi_1 \sigma \psi_2$, where $\sigma \in \{\wedge, \vee\}$, then return true $\iff \varphi \in formulas\_F$

Add formula

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), then φ 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 φ is added to measured_less_eq_T
- $\neg\psi$, then φ is added to formulas_T
- $\psi_1 \sigma \psi_2$, where $\sigma \in \{\wedge, \vee\}$, then 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 in the respective negative collection, namely if φ is of type:

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

Remove formula

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), then φ 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 φ is removed from measured_less_eq_T
- $\neg\psi$, then φ is removed from formulas_T
- $\psi_1 \sigma \psi_2$, where $\sigma \in \{\wedge, \vee\}$, then φ is removed from formulas_T

Remove formula signed as true

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

4.2.2 Def: Tableaux branch output

As stated above the ouput 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
- Meassured Equal to Zero Terms
- Meassured Not Equal to Zero Terms

These four groups can be represented with the following formula:

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

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) -> 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))
        {
            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 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))
    {
        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);
}
```

5 Formula Model

Every model type has at least two things in common:

- Adjacency matrix for the contacts between the points
- Vector of bitsets representing the value of v(p), namely the set of points which are in v(p)

All models also have similar collection of points each of which has an evaluation for all used variables in the tableau's path (this evaluation identifies the point).
All models can be created(or at least tried to be created) via the signed formula collections and some meta information for the whole formula, 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;
```

Adjacency Matrix member variable

```
/*
    Symetric square bit matrix. contacts_[i] gives a bit mask
    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 from !=0 term
        5      // point from !=0 term

    The bit matrix will be the following:
        \ 012345
        0 110000    // from 0---1 and reflexivity (0---0)
        1 110000    // from 1---0 and reflexivity (1---1)
        2 001100    // from 2---3 and reflexivity (2---2)
        3 001100    // from 3---2 and reflexivity (3---3)
        4 000010    // just reflexivity (4---4)
        5 000001    // just reflexivity (5---5)
*/
contacts_t contact_relations_;
```

Variable Evaluations member variable

```
/*
    A vector 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 'v' which for a given term returns
    a subset of the model points.
    :
     - v(p) = { i | (i) (xxx...x)[p] == 1, i.e. in point 'i'
         the evaluation of the variable 'p' is 1 }, where 'p'
         is a variable(for optimzations it's an id of the
         string representation of that variable)
     - v(a * b) = v(a) & v(b)
     - v(a + b) = v(a) | v(b)
     - v(-a) = ~v(a)

    For example, let us have the following
         formula: C(a * b, c) & c != 0 & -a != 0
    Then a model could be the following:
         (110) (a * b) 0---1 (c) (001)
         (011) (c)  2
         (000) (-a) 3
    Let assume that 'a' has a variable id 0,
         'b' has id 1 and 'c' has id 2.
    The bit matrix will be the following:
         \ 0123     // variable ids (rows) \ model points (columns)
         0 1000     // v(a) = { 0 }, i.e. all points
              which have evaluation with bit at positon 0 set to 1
         1 1010     // v(b) = { 0, 2 }, ... at position 1 ...
         2 0110     // v(c) = { 1, 2 }, ... at position 2 ...
*/
variable_id_to_points_t variable_evaluations_;
```

5.1 Connected Model

NOTE Intro on what the connected model really is ! ADD Connectivity lemma. Let us have φ 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 formulae
- J - number of Non Contact atomic formulae
- K - number of Not Equal to Zero atomic formulae
- L - number of Equal to Zero atomic formulae

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 contants, which mens 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 τ to constant true then the model evaluation v(τ) 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:

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

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

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

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 $2^N x 2^N$.

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

$$\text{Model points i and j are in contanct if } \mathbb{C}[i][j] \text{ is set.} \tag{20}$$

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

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

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

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

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

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

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

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

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 $(\mathbb{C}[p]$ 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, wihch 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 ach 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

$$x * x * M + y * y * M <<< (x + y) * (x + y) * M$$

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

5.2 System of Inequalities

The systems are of the following type:

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

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.

5.3 System Construction

The system of inequalities is constructed from the potential meassured model by evaluating each of the terms in the Meassured less or equals and Meassured 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 $X_0, X_1, ..., X_n$, where $\forall i < N : X_i$ is mapped to point i
TODO: Add beter theoretical explanation

1) For each <=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.
2) For each ~<=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 meassured model.

5.4 Meassured 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 inequalites 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

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

then this inequality is transformed into:

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

where ϵ is a small value.
The inequality X > 0, which is a variable to be greater than zero 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 of systems of inequalities defined above.

5.5 Measured Model

Let us have φ 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 formulae
- J - number of Non Contact atomic formulae
- K - number of Not Equal to Zero atomic formulae
- L - number of Equal to Zero atomic formulae
- S - number of Measured Not Equal to Zero atomic formulae
- U - number of Measured Equal to Zero atomic formulae

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 existance 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.
- $<=_m (H_s, O_s)$ produces one modal point
- $\neg(<=_m (Q_u, R_u))$ 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 τ 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 determins 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 <= 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 $<=_m$ (H_s, O_s) calculate $v(H_s)$ and $v(O_s)$ and create an inequality of the following type:

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

For each $\neg(<=_m\ (Q_u, R_u))$ 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:

$$\begin{aligned} 000 + 1 &= 001 \\ 001 + 1 &= 010 \\ 010 + 1 &= 011 \\ 011 + 1 &= 100 \\ 100 + 1 &= 101 \\ 101 + 1 &= \ldots \end{aligned}$$

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:

$$\begin{aligned}
[00][00] - (\text{next combination}) -> & [00][01] \\
& - > [00][10] \\
& - > [00][11] \\
& - > [01][00] \\
& - > [01][01] \\
& - > [01][11] \\
& - > [10][00] \\
& -..- > [11][11]
\end{aligned}$$

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:

$$\begin{aligned}
[x : 1y : 0,' x'][x : 0y : 1,' y'] -> & [10][11] \\
& - > [11][01] \\
& - > [11][11]
\end{aligned}$$

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.

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 easirly explained with an example first:

Example 1.1

Let φ 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.
HERE DRAW SOMEHOW THE MODAL
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 existance 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) | 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->evaluate(xxx..x) = constant_true in order to 0 belongs to v(a)

Example 1.2

Let φ 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:
DRAW THIS (P0 a)—-(P1 b) (P2 c)—-(P3 d)
NOTE that if Pi evaluates the term 't' to constant true then the model evaluation v(t) contains the point (Pi _) (and vice versa)
Where:
P0 evaluates(binary) a to constant true && e and f to constant false
P1 evaluates(binary) b to constant true && e and f to constant false
P2 evaluates(binary) c to constant true && e and f to constant false
P3 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):
[(P0 evaluates g to constant true && P1 evaluates h to constant true) ||
(P1 evaluates g to constant true && P0 evaluates h to constant true)] &&
[(P0 evaluates i to constant true && P1 evaluates j to constant true) ||
(P1 evaluates i to constant true && P0 evaluates j to constant true)] && ... analogous for P2 and P3.

The contact relation is reflexive, so for each point we also need to check also:
(Pi evaluates g to constant true && Pi evaluates h to constant true) &&
(Pi evaluates i to constant true && Pi evaluates j to constant true)

5.5.1 Algorithm

Let us have φ 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 formulae
- J - number of Non Contact atomic formulae
- K - number of Not Equal to Zero atomic formulae
- L - number of Equal to Zero atomic formulae

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

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 Pa and Pb.
Note that a point is just an variable evaluation.
They satsfy the following rules:

- Pa[a]=1 and Pb[b]=1.
- does not break any t=0 atomic formula,
  i.e. Pa[t]=0 and Pb[t]=0.
- does not break the reflexivity part of all C(e,f) atomic formulas,
  i.e. Pa[e]=0 & Pa[f]=0 & Pb[e]=0 & Pb[f]=0.
- does not break any C(e,f) atomic formulas,
  i.e. [(Pa[e]=1 & Pb[f]=1) | (Pb[e]=1 & Pa[f]=1)]

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

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

Complexity for this step: O(2N̂ * 2N̂ * (G+M))

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

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

If any of the rules above is not satsfied with Pa, then we generate next Pa'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 - PxP in a simple way because we know that only first 2K points are in contact. 0 <= 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 Pi, we iterate over the set bits in it's evaluation and if we have a set bit at position j we add that point to the evaluation of varialbe 'j', i.e. we set the bit 'i' in the variable's evaluation bitset.

Complexity for this step: O(P * N)

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

6 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.