Satisfiability Of Modal Logic Formulas

Martin Stoev, Anton Dudov

2019-9-28

1 Formula Representation

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:

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

1.1 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:

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

1.2 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.

1.3 Definition: Atomic Formula

A formula will be called atomic formula if it is a constant or it is in one of the followings:

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

where x and y are terms.

1.4 Definition/Theorem: Zero Term Formula

Let x and y be two terms, then:

$$x \leq y \iff x \sqcap \bar{y} = 0 \tag{1}$$

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

1.5 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

1.5.1 Symbols Representation

Atomic terms

- 0 is the term constant True
- 1 is the term constant False
- 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 $\bar{t1}$
- 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 formulas

- 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

1.6 Tokenizer

Grammar

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

[\t \n]	;	Returns nothing, ignore all whitespace
[,TF01()C&| *+-]	return yytext[0];	all single charecter tokens will be passed as their ASCIIs for an easier use in bison
"<="	return T_LESS_EQ;	Returns a special literal which maps the '<=' sequence
"<=m"	return T_MEASURED_LESS_EQ;	
"=0"	return T_EQ_ZERO;	
"= 0"	return T_EQ_ZERO;	
"->"	return T_FORMULA_OP_IMPLICATION;	
"<->"	return T_FORMULA_OP_EQUALITY;	
[a-zA-Z0-9]+	{yylval->T_STRING = create_lexer_string(yytext, yyleng); return 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
.	return yytext[0];	bison will trigger an error if it's unrecognized symbol

Parser literals

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

```
modal_logic_formula // The start non-terminal
    : formula {
        // $1 gives the matched formula and we write
        // it in a 'global' variable parsed_formula
        parsed_formula.reset($1);
    }
  ;
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

1.6.1 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

2 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 in building, this way getting the hash code of a node is done in 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;
};
```

2.1 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

$$C(a,b) = C(b,a) \tag{2}$$

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

$$C(a,b) = C(c,d) \rightarrow (a = c \wedge b = d) \vee (a = d \wedge b = c). \tag{3}$$

2.1.1 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 formas 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

3 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.

3.1 Definition: Tableaux Step

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

Definition: Marked Formula

Let φ be a formula and X be the set of accumulated formulae, then φ is said to be marked as:

- true if and only if $\mathbb{T}\varphi$
- false if and only if $\mathbb{F}\varphi$

Definition: Accumulated Formulae

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

3.1.1 Rules

Negation

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

And

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

Or

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

Implication

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

Equivalence

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

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. This verification can be done in different manners, depending on the algorithm type. The best way to think about it is to have the tableaux process return a list, where each element is the set of atomic formulae found in a specific branch. This way the atomic formulae for each branch are produced, and afterwards can be used in different algorithms. This is just an example, a way of thinking about the problem the real implementation is much more space efficient.

3.2 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 formulae in one data set, and all false formulae in another data set. This enables fast searches wheter a formula has been marked as true or false.

Definition: Marked Formula Collection

Let X be a set of formulae, then X is called marked formula collection if and only if all formulae in X are marked as true or all formulae are marked 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 marked formula collections:

- formulas_T_ - contains only formulae marked as true
- formulas_F_ - contains only formulae marked 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 marked as true
- contacts_F_ - contains only contacts formulae marked as false
- zero_terms_T_ - contains only formulae of type $\varphi \leq \psi$ marked as true
- zero_terms_F_ - contains only formulae of type $\varphi \leq \psi$ marked as false
- measured_less_eq_T_ - contains only formulae of type $\varphi \leq_m \psi$ marked as true
- measured_less_eq_F_ - contains only formulae of type $\varphi \leq_m \psi$ marked as false

Definition: Formula Contradiction

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

- φ is marked as true and $\varphi \in$ formulas_F_
- φ is marked as false and $\varphi \in$ formulas_T_
- φ is a contact formula marked as true and $\varphi \in$ contacts_F_
- φ is a contact formula marked as false and $\varphi \in$ contacts_T_
- φ is a zero terms formula marked as true and $\varphi \in$ zero_terms_F_
- φ is a zero terms formula marked as false and $\varphi \in$ zero_terms_T_
- φ is a measured less formula marked as true and $\varphi \in$ measured_less_eq_F_
- φ is a measured less formula marked as false and $\varphi \in$ measured_less_eq_T_

Invariant

At any time, all formulae in all eight marked 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 will produce $\mathbb{T}T, \mathbb{T}\neg C(a, b)$.

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 final 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
- The commutativity of the contacts: 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 marked 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 marked as true, then marking the formula x = 0 as true will lead to contradiction.

Lemma: 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;
```

3.2.1 Handy methods

Find formula

Find formula marked 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 marked 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 marked 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 marked 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_

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);
}
```

3.3 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 marked 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_;
```