# 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 $\varphi$ is a formula, then $\neg\varphi$ is a formula as well
- If $\varphi$ and $\psi$ are formulas, then $\varphi\ \sigma\ \psi$ is a formula as well, where $\sigma \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

### 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 $t\bar{1}$
- 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 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:

| | | |
|---|---|---|
| [ \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 |

## 2 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.

## 2.1 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 $\varphi$ be a formula and X be the set of accumulated formulae, then $\varphi$ 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.

### 2.1.1 Rules

**Negation**

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

**And**

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

**Or**

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

**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. 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.

## 2.2 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 $\varphi$ 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 $\varphi$ be a marked formula, then $\varphi$ is causing a contradiction if any of the followings is true:

- $\varphi$ is marked as true and $\varphi \in$ formulas_F_
- $\varphi$ is marked as false and $\varphi \in$ formulas_T_
- $\varphi$ is a contact formula marked as true and $\varphi \in$ contacts_F_
- $\varphi$ is a contact formula marked as false and $\varphi \in$ contacts_T_
- $\varphi$ is a zero terms formula marked as true and $\varphi \in$ zero_terms_F_
- $\varphi$ is a zero terms formula marked as false and $\varphi \in$ zero_terms_T_
- $\varphi$ is a measured less formula marked as true and $\varphi \in$ measured_less_eq_F_
- $\varphi$ 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 $\varphi$, the following algorithm determines final atomic formulae in all branches of the tableaux process.

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

#### Remarks

- true boolean value is used to represent the formula constant T
- false boolean value is used to represent the formula constant F
- 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;
```

### 2.2.1 Handy methods

#### Find formula

#### Find formula marked as true

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

Checks if the formula $\varphi$ exists in any positive collection depending on the type of $\varphi$, namely if $\varphi$ 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 $\varphi$ exists in any negative collection depending on the type of $\varphi$, namely if $\varphi$ 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 $\varphi$ as true in in the respective positive collection, namely if $\varphi$ is of type:

- C(x, y), then $\varphi$ is added to contacts_T_, and the terms x and y are added to the contact_T_terms_ collection.
- $x = 0$, then x is added in zero_terms_T_
- $x \leq_m y$, then $\varphi$ is added to measured_less_eq_T_
- $\neg\psi$, then $\varphi$ is added to formulas_T_
- $\psi_1 \sigma \psi_2$, where $\sigma \in \{\wedge, \vee\}$, then $\varphi$ is added to formulas_T_

### Mark formula as false

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

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

- C(x, y), then $\varphi$ is added to contacts_F_.
- $x = 0$, then x is added in zero_terms_F_
- $x \leq_m y$, then $\varphi$ is added to measured_less_eq_F_
- $\neg\psi$, then $\varphi$ is added to formulas_F_
- $\psi_1\sigma\psi_2$, where $\sigma \in \{\wedge, \vee\}$, then $\varphi$ is added to formulas_F_

### Remove formula

### Remove formula marked as true

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

Removes the formula $\varphi$ from the respective positive collection, namely if $\varphi$ is of type:

- C(x, y), then $\varphi$ is removed from contacts_T_, and the terms x and y are removed from the contact_T_terms_ collection.
- $x = 0$, then x is removed from zero_terms_T_
- $x \leq_m y$, then $\varphi$ is removed from measured_less_eq_T_
- $\neg\psi$, then $\varphi$ is removed from formulas_T_
- $\psi_1\sigma\psi_2$, where $\sigma \in \{\wedge, \vee\}$, then $\varphi$ is removed from formulas_T_

### Remove formula marked as true

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

Removes the formula $\varphi$ from the respective negative collection, namely if $\varphi$ is of type:

- C(x, y), then $\varphi$ is removed from contacts_F_
- $x = 0$, then x is removed from zero_terms_F_
- $x \leq_m y$, then $\varphi$ is removed from measured_less_eq_F_
- $\neg\psi$, then $\varphi$ is removed from formulas_F_
- $\psi_1\sigma\psi_2$, where $\sigma \in \{\wedge, \vee\}$, then $\varphi$ is removed from formulas_F_

### 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);
}
```