# The theories of Linear Arithmetic and Equality Logic and Uninterpreted Functions in the context of Satisfiablity Modulo Theories*

**Martin Köhler**[1]

1 **Department of Computer Science, University of Kaiserslautern Kaiserslautern, Germany**
m_koehler10@informatik.uni-kl.de

**Abstract**

Satisfiablity of first-order formulas is in general not decidable. For certain theories it is however decidable whether a formula is satisfied within the given theory. This decision problem is called "Satisfiablity Modulo Theories". This work will introduce to basic ways how to solve instances of this problem. Several decision procedures for this class of problems invoke so called "Theory Solvers". This document will give a short overview on the internals of theory solvers. This will be done by examining several solvers for Linear Arithmetic, which is the theory that allows inequalities over sums of constant multiples of variables. As a first glimpse on theory solvers, this work will present a theory solver for the much simpler theory of difference arithmetic, whose predicates talk about the maximum and minimum difference between two arithmetic variables. This document will in a final step discuss a solver for the theory of Equality over Uninterpreted functions, whose only assumption is the fact the equal function input have the same output.

**Keywords and phrases** SMT, Satisfiability, Theories, Theory Solver

## 1 Overview

A well-known decision problem in theoretical computer science is the "Satisfiability Problem" (SAT). It asks whether there exists a valuation such that a given logic formula evaluates to `true`. For Boolean logic, this problem is $\mathcal{NP}$-complete. However, it is known to be undecidable in first-order logic. Although the general problem is undecidable, a modification of the problem is decidable: Setting requirements towards the considered interpretations allows to decide whether there is an interpretation that satisfies a given formula. We will later introduce so called theories as a way to formalize the restriction of considered interpretations.

Among other procedures, satisfiability modulo theories can be decided in three steps: Replacing all occurrences of predicate logic predicates by Boolean variables, applying a well-known (Boolean) SAT-solver and checking whether the chosen Boolean valuation is feasible with respect to the given theory. The last step is done by a so called Theory Solver. This is an algorithm that checks whether a given conjunction is in the given theory. Here, the conjunction consists of all those predicates that are satisfied by the valuation and the negation of all those predicates that are not satisfied by the valuation. The procedure will continue finding SAT-solutions and invoking the theory solver.

This work will present background on the theory of linear arithmetic and the theory of equality logic over uninterpreted functions. It will give an insight how satisfiability for these logic fragments can be decided.

* This work has been created as a seminar thesis at the Concurrency Theory group



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1.1 Background Knowledge

A *valuation satisfies* a Boolean formula if and only if it evaluates the formula to `true`. A formula is said to be *satisfiable* if and only if at least one valuation satisfies the formula.

A *theory* in predicate logic is a set of closed formulas (i.e. those without free variables) that is closed against conclusions. This means: If a formula is satisfied by all models (i.e. satisfying interpretations) of the whole set, then the formula is already element of the set.

We will consider satisfiability and conclusions with respect to theories: When examining which structures satisfy the formulas, we only consider those are models of the theory. This is the origin of the name *Satisfiability Modulo Theories*.

# 2 A Simple Theory Solver

Most algorithms that solve satisfiability modulo theories utilize a SAT-solver for the so called Boolean or propositional core. A SAT-solver computes a solution for a Boolean abstraction of the formula that treats the predicate logic atoms as propositional variables. If this Boolean abstraction of the given formula is not satisfiable, the original formula is not satisfiable either. Satisfiability of the Boolean abstraction however does not imply satisfiability with respect to the theory.

A procedure that decides whether a given conjunct of literal is satisfiable with respect to a theory is called *Theory Solver*. In order to give an impression how a theory solver decides whether the given conjunction is satisfiable with respect to the theory, we will first focus on that problem for the theory of Difference Arithmetic.

Its theory solver is much simpler than the theory solvers for the theory of Linear Arithmetic that we will examine later.

## 2.1 Difference Arithmetic

*Difference Arithmetic* is a theory over integer variables where each atom expresses the maximal difference between two given variables. Consider the numeric variables $x, y$. The predicate $(x - y) \leq 7$ is a possible atom in this theory. The given atom is satisfiable with respect to the theory of Difference Arithmetic: It is satisfied by all structures (over that theory) that assign $x$ and $y$ number such that the value of $y$ can be reached by adding at most 7 to the value of $x$.

Consider the following logic formula: $A \equiv (x - y \leq 7) \wedge (\neg(x - y \leq 7) \vee \neg(x - y \leq 8))$. A propositional logic abstraction of this formula is given by $A' \equiv p \wedge (\neg p \vee \neg q)$. The propositional logic abstraction $A'$ of the predicate logic formula $A$ is satisfied if and only if both $p$ and $\neg q$ are satisfied. In order to check the satisfiability with respect to the theory of difference arithmetic, one needs to consider the formula $(x - y \leq 7) \wedge \neg(x - y \leq 8))$. This conjunction is not satisfiable with respect to the theory of difference arithmetic.

The example above shows that satisfiability of the Boolean abstraction does not imply satisfiability with respect to the theory. We will now explain how a theory solver can decide satisfiability with respect to this theory.

A theory solver can be implemented by regarding the problem as a problem on a weighted graph. This requires a certain normal form:

In Difference Arithmetic over integers, the theory solver can assume that all literals in the given conjunction do not contain any negation because it is possible to replace any negated atom by a pure atom. This is done in the following way:

$$\neg(x - y \leq c) \iff (x - y > c) \iff (y - x < -c) \iff (y - x \leq -c - 1)$$

It is now possible to construct any conjunction of those atoms as a directed, weighted graph: Each variable is a vertex. Each atom is a direct edge with the following semantics: An edge $y \xrightarrow{c} x$ leads from $y$ to $x$ with the weight $c$ if and only if $x$ can be reached by adding at most $c$ to $y$. This represents the atom $(x - y \leq c)$.

We will now present a procedure to conclude satisfiability from this weighted graph. To this end, we will consider the semantics of implicit transitive edges and self-loops: Transitive edges represent implicit difference constraints: Assume there are two edges $z \xrightarrow{c_1} y$ and $y \xrightarrow{c_2} x$ that correspond to the atoms $(y - z \leq c_1)$ and $(x - y \leq c_2)$.

*Claim:* If we assume these atoms to be satisfied by a model with respect to the theory of Difference Arithmetic, then we conclude that this model satisfies the formula $(x - z \leq c_1 + c_2)$ as well. Towards a contradiction, we fix a model that satisfies $\neg(x - z \leq c_1 + c_2)$. It follows that one has to increase $z$ by strictly more than $c_1 + c_2$ in order to reach $x$. If this model satisfies one of the first two atoms, it cannot satisfy the other atom since either the difference from $z$ to $y$ will exceed $c_1$ or the difference from $y$ will exceed $c_2$. We conclude for any pair of edges $z \xrightarrow{c_1} y$ and $y \xrightarrow{c_2} x$ that adding the transitive edge $z \xrightarrow{c_1+c_2} x$, whose weight sums up the weights of it, will not alter the satisfiability.

Self-loops represent conditions that are satisfied if and only if the weight is at least 0. An edge $x \xrightarrow{c} x$ represents the condition that it needs to be possible to reach $x$ by increasing $x$ by adding at most $c$. The only value one can add to $x$ in order to reach $x$ is 0. Therefore, the condition is satisfied if and only if the weight $c$ is at least 0. This means in particular that this condition does not depend on any other condition. Thus, we can determine the satisfiability of a self-loop by this single condition.

Having those insights on implicit transitive edges and self-loops, we derive the following graph algorithm to determine satisfiability of the represented formula:

1. Add all implied transitive edges.
2. Return `unsatisfiable` if there is a self-loop with a negative weight. Return `satisfiable` otherwise.

*Claim (correctness):* The weighted graph contains cycles with a negative weight if and only if the corresponding conjunction is unsatisfiable.

$\Rightarrow$ Assume the weighted graph contains a cycle with negative weight. As shown above, one can add implied transitive edges without altering the satisfiability of the conjunction. Since the graph contains a negative weighted cycle, this will lead to at least one negative weighted self-loop. As shown above, the conjunction is unsatisfiable.

$\Leftarrow$ Assume the weighted graph does not contain cycles with negative weight. Positive weighted cycles do not affect satisfiability since they are always satisfied. Pseudo-transitive structures like $z \xrightarrow{c_1} y$, $y \xrightarrow{c_2} x$, $z \xrightarrow{c_3} x$ with $c_3 \neq c_1 + c_2$ cause two concurrent edges. These can can be resolved by only considering the one with the smaller weight.

We have now seen a basic theory solver for Difference Arithmetic. It decides satisfiability of a conjunction of literals by checking the absence of negative cycles in the induced weighted graph. There is an application for this logic fragment: In real-time-scheduling, the time that is requested by a task can be modelled as the minimal difference between this task and the next one: Each variable corresponds to a machine and a task and represents the time when this task is executed on this machine. If the durations of each execution are fixed, one can use Difference Arithmetic to express constraints like earliest start and end, order of machine switches and sequentially of the executions. However, there are applications where a restriction to integers and predicates on differences is not possible.

# 3 Linear Arithmetic

The theory of *Linear Arithmetic* is a logic theory that allows inequalities and equations over real numbers.

Other than the theory of Difference Arithmetic, the atoms in the theory of Linear Arithmetic are not difference inequalities over integer numbers but equalities and inequalities on a fragment of real number arithmetic. This fragment of real-arithmetic allows sums and variables with constant coefficients. One possible real-world application can be found in the field of compiler-optimization: In some cases it is wise to perform reads only once rather than repeating those reads within a loop. Thus, the reads from the memory should be saved in a register before executing the loop. This rearranging, however, requires neither the location nor the value to change during the execution of the loop. It is possible to generate formulas over Linear Arithmetic that phrase these requirements. This is done by building a formula that expresses the inequality between the initial state of the variables and the variables after the operations in the body of the loop.

Basic algorithms that solve Satisfiability Modulo Theories for Linear Arithmetic can be obtained similarly as we did it for Difference Arithmetic: We assume an existentially quantified formula. A SAT-solver for propositional logic will enumerate satisfying assignments for the atoms. The generated assignments will be interpreted as conjunctions over literals in Linear Arithmetic. A theory solver will analyze whether the suggested assignments of the atoms are actually possible in Linear Arithmetic.

## 3.1 Overview on theory solvers for Linear Arithmetic

In the following, we will introduce two procedures to implement a theory solver for Linear Arithmetic. These procedures are based on techniques from linear programming and variable elimination. We will also introduce procedures that adapt those methods for integer numbers.

### Linear Programming

It is notable that a conjunct of literals in Linear Arithmetic has similarities to class of problem that *Linear Programming* addresses: In their general form, linear programs find the maximum[1], positive vector that satisfies a set of linear constraints: Each of the constraints requires a weighted sum of the components not to exceed a certain constant value. Therefore, Linear Programming optimizes solutions for formulas of the form:

$$\bigwedge_{1\leq i\leq m} \sum_{1\leq j\leq n} c_{i,j}\cdot x_j \leq b_i \text{ with } x_1,\ldots,x_n \text{ variables and } m \text{ constraints}$$

The main difference between Linear Programming and the intended theory solver is the optimization: In order to solve satisfiability, only the existence of a certain vector is of importance. Linear programming however maximizes the vector if it exists. A procedure that only solves the existence of a solution for a linear program is the so called *General Simplex Method*. It is based on the well-known *Simplex Method* and will be considered in this work.

[1] In Linear Programming, a vector is considered to be maximal if its value of the *objective function* is maximal. The canonical function is the scalar product with a given constant vector.

### Variable Elimination

Another procedure that can be used for theory solvers and exists in practical implementations is the *Fourier-Motzkin*-method. It is based on a procedure for variable elimination and can be used as a theory solver for the theory of Linear Arithmetic over real numbers.

### Linear Arithmetic over Integers

There are however use-cases where the theory of Linear Arithmetic over integers is required. This theory can be solved by procedures that are similar to the mentioned ones: The field of *Integer Linear Programming* is a variation of linear programming where the constants and the solution vector consist of integers. The *Branch and Bound* procedure is an algorithmic way to solve integer linear programs.

The basic idea behind the Fourier-Motzkin-method, the elimination of variables, can be transferred to Linear Arithmetic over integers. A procedure that follows this concept is the so called *Omega Test*.

## 3.2 The General Simplex Method

We will introduce the *General Simplex Method* as a way to solve conjuncts of literals in Linear Arithmetic.

The general simplex method has its origins in the field of linear programming where it is applied for finding solution vectors to linear programs. In opposition to the simplex method, the general simplex does not optimize the solution vector. A notable difference is the input and the output format: Unlike its optimizing counterpart the general simplex does not restrict the solution vector to non-negative[2] values but allows optional, constant bounds for each of the components. The problem constraints are not phrased as upper-bounds for scalar products with a constant vector but as scalar products that need to be equal to zero.

### Converting the conjunction into the general form

The described input format for the general simplex method is known as the *general form*. Any conjunct of literals in linear arithmetic can be transformed into this general form: Given an atom like $x - y - 1 \leq 1$, one has to isolate the constants which yields a inequality like $x - y \leq 2$. The general form however requires the constraints to be phrased as zero-checks. We will formulate the constraint as an existence-problem: Can we reach 0 by subtracting at most 2 from $x - y$? This yields the constraint $x - y - s = 0$ that introduces the new *additional variable* that is bounded by $s \leq 2$. This leads to a procedure that introduces new variables for each constant.

### Applying the general simplex method

Based on this transformation, we will assume that our conjunction of literals is now given as an input for the general simplex algorithm in general form. We will give a basic overview how the general simplex works. For this explanation, we will fix the variables to be $\mathcal{N} := \{x_1, \ldots, x_n\}$, the additional variables to be $\mathcal{B} := \{s_1, \ldots, s_m\}$. The bounds for each additional variable $s_i$ will be the lower $l_i$ and the upper bound $u_i$. If there is no lower (upper) bound, we set

[2] The *standard form* requires the result to be non-negative. Problem formulation in forms that allow negative solutions can be rewritten.

$l_i := -\infty$ ($u_i := +\infty$). The zero-checks will be denoted as a matrix multiplication with the matrix $A$. [3]

Basically speaking, the general simplex method initializes all variables with 0, which satisfies at least the zero-check. Then the algorithm tries to adjust the assigned values so that the bounds of the additional variables are satisfied. As an invariant, the algorithm will not violate the zero-checks.

#### Adjusting the variable assignments

Let $s$ be an additional variable that violates its bounds. Just increasing or decreasing this variable will violate the zero-check in the corresponding row. Therefore, one needs to determine a variable $x$ in the same row that can be adapted accordingly. This means that we increase or decrease $x$ just enough to be able to bring $s$ within its bounds. If there is no variable that can be adapted, the system is determined `unsatisfiable`. If we want to adjust the variable $x$, we have to make sure that this adjustment does not violate any zero-checks.

**Pivoting** In order to adjust a variable $x$, one needs to change the other assignments as well. To this end, a technique called *pivoting* is applied. Basically speaking, pivoting can be seen as swapping the additional variable $s$ with the variable $x$. This is done by solving the row containing $s$ for $x$ and substituting occurrences of $x$ in other rows accordingly. Note that after this substitution only one equation depends on $x$.

The formerly additional variable $s$ will now be seen as a non-additional variable, $x$ will be seen as an additional variable. The assignments for the remaining additional variables can (and have to) be adjusted now, since all zero-checks contain this variable $s$ now. This might violate bounds of additional variables. However, the bounds for $s$ are satisfied after pivoting as this technique allowed to change $s$ and $x$ while still satisfying the zero-check. Since only $s$ and the additional variables (including $x$) have been reassigned, still only additional variables violate the bounds.

The general simplex method keeps calling the pivot operation to bring additional variables in their bounds until either a satisfying assignment is found or no suitable variable for pivoting is available[4].

## 3.3 The Fourier-Motzkin method

The Fourier-Motzkin method is a popular procedure when it comes to implementing a theory solver for Linear Arithmetic over real numbers. Unlike the general simplex method it is not based on Linear Programming but uses variable elimination by projecting to lower-dimensional spaces. One basic characteristic of this procedure is that we iteratively neglect all variables that are only bound in one direction.

---

[3] This matrix has $n + m$ columns due to the number of variables and it has $m$ rows since there is one row for each of the zero-checks. Note: The rightmost $m \times m$-submatrix consists only of a diagonal of $-1$ since each row has precisely one additional variable.

[4] As mentioned above, a variable is not suitable for pivoting if the desired adjustment violates the bounds of the variable. However, this could still lead to a cyclic behavior. Thus the appearance of such cycles is another criteria to mark a variable unsuitable for pivoting. In practical implementation this is done by introducing an order on the variables.

### Eliminating equations

Linear Arithmetic allows equations. As mentioned above, the Fourier-Motzkin considers only inequalities. Thus equations have to be eliminated. This is done by solving each equation for one variable and substituting all occurrences of this variable accordingly.

We will assume that the inequalities are of the form $\vec{a} \cdot \vec{x} \leq b$ where $\vec{x}$ is a vector of variables, $\vec{a}$ is a constant vector and $b$ is a constant. Any inequality in Linear Arithmetic can be transformed in an equivalent inequality of that form.

**Upper and lower bounds** Consider an inequality like $\vec{a} \cdot \vec{x} \leq b$ where a variable $x_j$ has the coefficient $a_j$. Is this constraint an upper or lower bound for $x_j$? Isolating $x_j$ the inequality yields to an inequality of the form $a_j \cdot x_j \leq \sigma$. [5] Dividing by $a_j$ on both sides[6], this is clearly an upper bound for positive values of $a_j$ and a lower bound for negative values of $a_j$ since the minus sign of $a_j$ flips the direction of the inequality when dividing.

**Bounded and unbounded variables** One can now decide whether a variable does not have both upper and lower bounds. If this is the case, it is said to be *unbounded*. If a variable has both a lower and an upper bound, we derive a constraint that does not contain this variable: Let $x_1, x_2, x_3$ be variables with $x_1 - x_2 \leq 0$ and $-x_1 - x_2 + x_3 \leq 0$. The first constraint is equivalent to $x_1 \leq x_2$ and thus an upper bound for $x_1$ and a lower bound for $x_2$. The second constraint is equivalent to $-x_2 + x_3 \leq x_1$ and thus an lower bound for both $x_1$ and $x_2$ and an upper bound for $x_3$. The variables $x_2$ and $x_3$ are therefore called unbounded since $x_2$ has only two lower bounds and $x_3$ has only an upper bound. The variable $x_1$ is bounded since it has bounds in both directions. It is now possible to combine the lower and the upper bound for $x_1$ to the new constraint $-x_2 + x_3 \leq x_2$, which is equivalent to $-2x_2 + x_3 \leq 0$.

**Eliminating variables** We will use these observations on bounded and unbounded variables to eliminate these variables: If a variable is unbounded, its constraints will just be removed. This is possible since the unbounded variable can be arbitrarily far away from its only bound without violating it: Given any valuation for the other variables, we are able to just assign the unbounded variable the concrete (constant) value of its only bound. If it is bounded, the constraints will be replaced by the implied (transitive) inequality. This will not affect satisfiability:

Deleting constraints will only allow more satisfying assignments. A satisfiable system of constraints stays thus satisfiable.

Can an unsatisfiable system of constraints turn satisfiable by deleting the inequalities containing an unbounded variable? It cannot. Assume after deleting the inequalities containing an unbounded variable $x$, the remaining constraints without this variable have a satisfying assignment for the remaining variables. Since all of the deleted constraints are upper (lower) bounds, the constraints containing $x$ can now just be satisfied by choosing $x$ small (big) enough. Hence, the full system of constraints was therefore already satisfiable.

Do the derived constraints for variables affect satisfiability? They do not: After adding those constraints to a satisfiable system, the system will stay satisfiable since these new inequalities are implied by the old ones. If there is a satisfying assignment for the system

[5] The variable $\sigma$ represents the difference between $b$ and the remaining scalar product $\vec{a} \cdot \vec{x}$ except for the components $a_j, x_j$

[6] Assuming that $x_j$ actually occurs in the inequality which means that $a_j$ is not zero

with the derived constraints, then one only needs to pick an assignment that satisfies the bounds of the bounded variable. This is possible since the derived constraints ensure that the upper and lower bounds are not contradicting.

**The decision procedure** The Fourier-Motzkin method eliminates variables as described above. Unsatisfiable inequalities[7] will directly lead to an output of `unsatisfiable`. Once only one variable remains, the set of inequalities is `unsatisfiable` if and only if the last variable is bounded by contradicting constants. The order of elimination is chosen heuristically. This is the crucial part: Each elimination of a bounded variable can square the number of inequalities. This leads to a double exponential number of inequalities. Nevertheless, for small input sizes the Fourier-Motzkin method is preferred in practice.

## 3.4 The Branch and Bound Method

The introduced principles are possible methods for theory solvers for Linear Arithmetic over real numbers. One can easily come up with examples that do have real solutions but no solutions in integer numbers.[8] The *Branch and Bound* method is used in Integer Linear Programming. This section will present an overview on *Feasibility-Based Branch and Bound*, a procedure that finds solutions without considering optimization.

**Invoking the simplex method** As explained earlier, the simplex method find solutions for linear programs but allows non-integer solutions. However, it might find integer solutions by coincidence. They key principle of Branch and Bound is invoking an algorithm for the non-integer case and retrying the search by adding constraints that prohibit found non-integer values.

**Branch and bound works recursively** The method starts with the input constraints and tries to find a solution using the general simplex method. If there is no solution, this recursive call stops. The components of a solution are checked for integrity. A integer solution vector will immediately be returned as a valid solution. A non-integer solution will be used to derive two constraints: For the (heuristically picked) non-integer component $x_j := c$, the constraints $x_j \leq \lfloor c \rfloor, \lceil c \rceil \leq x_j$ are derived. Each of those two constraints leads to a separate recursive call with the original constraints and one of the added ones [9]. Once all recursive calls have terminated without returning a integer solution, the set of inequalities is found unsatisfiable.

**Termination** The presented version of the algorithm might not terminate: There cases where a variable never gets an integer value while increases stepwise to meet the freshly added lower bound. Termination can be guaranteed by making use of the *Small Model Property*, which means that there is a function that computes the size of a search space for a given input of the problem. The search space has the property that it contains a solution if and only if the problem has a solution. This finite search space has shown to be determinable for integer linear systems: If there is a solution for an integer linear system, then one of the extreme points of the convex hull is a solution as well. In this solution, none

[7] such as $23.42 \leq 13.37$

[8] The inequalities $y \leq 2 - 2x$, $y \leq 2x$ and $2y \geq x - 2$ define the triangle $\left(\left(\frac{1}{2}, 1\right), \left(\frac{1}{3}, \frac{2}{3},\right), \left(\frac{3}{5}, \frac{4}{5}\right)\right)$, whose enclosed points do not have integer $x$-coordinates.

[9] If there is an integer solution for the original set of constraints, then it satisfies one of the new constraints as well, since the since there is not integer number between the new bounds.

of the components exceeds a certain bound that depends on the input size and the maximum constant in the constraints. This allows computing a maximum bound that none of the components of the solution may exceed. Adding this constraint ensures termination after finite time since there are only finitely many integer numbers within this finite space.

**Improving the runtime** Consider the problem as a visual problem: The integers form a grid on the coordinate plane, the integer linear system describes a region and our goal is to find a point of the grid within this region. One way to improve the whole procedure is tightening this region by cutting out corners of the area that certainly do not contain a point of the interger grid. In practice, this is done by a procedure called *cutting planes* that adds additional constraints to tighten the solution space without forfeiting integer solutions.

## 3.5 Omega-Test

The *Omega-Test* extends the principle of variable elimination as applied in the Fourier-Motzkin-method to Integer Linear Arithmetic.

**Normalizing constraints** The Omega-Test requires the input to be given as a set of scalar products of variable vectors and integer coefficient vector together with either upper bounds or equality constraints. Rational coefficients of equations and inequalities can be transformed to integers by multiplying them with the least common multiple of their denominators. However, the runtime of this procedure also depends on the size of the coefficients. Thus, the equations and inequalities are brought to a normalized form by dividing the whole equation or inequality by the greatest common divisor. This is already a first chance to discover unsatisfiability of equations:

Suppose the equation is of the form $a_1 \cdot x_1 + \cdots + a_n \cdot x_n = b$ where $a_1 \dots a_n$ share the common divisor $d$ and can thus be written as $a_i = c_i \cdot d$ (with $1 \leq i \leq n$). This yields the equation $d \cdot (c_1 \cdot x_1 + \cdots + c_n \cdot x_n) = b$. Since an integer solution is required, $b$ has to be a multiple of $d$. Otherwise this equation cannot be satisfied by an integer solution.

If an upper bound is not divisible by the greatest common divisor of the coefficients, the bound is just rounded down to the nearest lower integer since no integer solution can directly hit a non-integer bound.

**Eliminating equations** Similar to the Fourier-Motzkin procedure, the Omega test eliminates equations. For the coefficients $-1$ and 1, this can be done in the same way as it is done by the Fourier-Motzkin procedure. If an equations does not contain such coefficients, dividing by a coefficient is not possible since it none of coefficients divides any other coefficient in the normalized form. Thus, it is necessary to find a method other than division that transforms all constraints such that at least one of the coefficients in the relevant equations turns 1 or $-1$.

This is done by replacing a variable $x_n$ by a term that contains a fresh variable $\sigma$. This will decrease the absolute value of the coefficient in the equation. In order to define this term, we introduce a new binary operator that is applied on the coefficients:

▶ **Definition 1** (Symmetric Modulo)**.** The binary operator $\widehat{mod}$ is a modulo operation whose output might be negative: If $a \ mod \ b$ exceeds $\frac{b}{2}$, the output is a negative modulo:

$$a \ \widehat{mod} \ b := a - b \cdot \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor = \begin{cases} a \ mod \ b & \text{if } (a \ mod \ b) < \frac{b}{2} \\ (a \ mod \ b) - b & \text{otherwise} \end{cases}$$

Assume we want to replace the variable $x_n$ with the coefficient $a_n$, we add a new equation. This additional equation is obtained from the equation where $a_n$ and $x_n$ appear: We use the symmetric modulo to bring all constants into the bounds $-\frac{a_n+1}{2}$ and $+\frac{a_n+1}{2}$. This will add or subtract a multiple of $a_n + 1$ in total. We compensate this by adding a multiple of $a_n + 1$, namely $\sigma \cdot (a_n + 1)$:

original equation: $$\sum_{i=1}^{n} a_i \cdot x_i = b$$

additional equation: $$\sum_{i=1}^{n} (a_i \mathbin{\widehat{mod}} (a_n + 1)) \cdot x_i = m \cdot \sigma + b \mathbin{\widehat{mod}} (a_n + 1)$$

We can easily solve the additional equation for $x_n$ since its coefficient is $a_n \mathbin{\widehat{mod}} (a_n+1) = -1$. One can prove two claims about the constants after inserting this solution for $x_n$: They will be dividable by $(a_n + 1)$ and their absolute value will strictly decrease. This means that we can iterate this introduction of variables until one coefficient becomes 1 or $-1$.

**Eliminating variables** Similar to the Fourier-Motzkin method, the Omega-Test eliminates variables. The difference is that there are different variants how to modify the set of inequalities to eliminate a variable. The Omega-Test will sequentially try those variants, apply itself recursively on the new set of inequalities and conclude satisfiability or unsatisfiability from the result. These three variants are called *Real Shadow*, *Dark Shadow* and *Gray Shadow*. The Omega-Test uses them as follows:

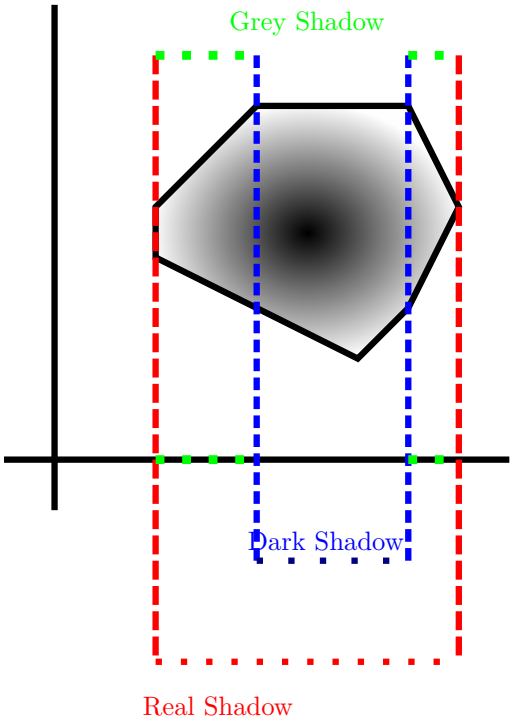


**Figure 1** Visualizing the phases of the Omega-Test as shadows in the two-dimensional case

- A set of inequalities with only one variable will trivially be checked for satisfiability. For other sets, an occurring variable is chosen. The following methods are tried on this system with that variable:
- *Real Shadow*: Similar to the Fourier-Motzkin method, it removes all constraints containing this variable if it is unbounded. For bounded variables, all possible pairs of bounds are combined.[10]
  - *recursive call returns* `satisfiable`: There is an (integer) solution for the derived system without the eliminated variable. However, this assignment might restrict the bounded variable to a non-integer value. In this case the solution has been over-approximated, continue with the under-approximation by *Dark Shadow*.
  - *recursive call returns* `unsatisfiable`: The derived system without the eliminated variable has no (integer) solution, therefore, the original system had no solution either. Return `unsatisfiable`
- *Dark Shadow*: The partial assignment might not be a suitable solution since there might not be an integer within the borders (of the eliminated variable). Thus, we add constraints that ensure that the range between these borders is wide enough to contain an integer for sure.
  - *recursive call returns* `satisfiable`: If we still find a solution after widening the gap between the borders, we know for sure that there is a possible integer value for the eliminated variable. Return `satisfiable`.
  - *recursive call returns* `unsatisfiable`: We have not found a solution for the non-eliminated variables in the part where the eliminated variable has wide enough borders to enforce the existence of an integer. We still need to check whether there are solutions for the non-eliminated variables where the bounds for the eliminated variable are narrow but still hit an integer.
- *Gray Shadow*: We have found a solution in the over-approximation and no solution in the under-approximation. By combining the constraints for the bounds of the eliminated variable and the negation of the constraint that widened the bounds, we obtain bounds that restrict a constant multiple of the eliminated variable to be within a finite range above a variable (but division-free) term. Now, we know that this multiple of our variable is in a finite rage. Thus, we can produce finitely many equations that express this finite multiple as a sum of the variable division-free term and a constant. We try adding each of these constraints and perform a recursive call on each derived set of inequalities.
  - *at least one recursive call returns* `satisfiable`: If one the equations leads to an integer solution for the remaining variables, then we have found an integer solution for the eliminates variable as well. Return `satisfiable`.
  - *all recursive calls return* `unsatisfiable`: We had an integer solution in the over-approximation, that was not present in the under-approximation. We tried all possible integer solutions in the region of the over-approximation without the under-approximation. This was not successful. Hence, there is no (integer) solution. Return `unsatisfiable`.

[10] The difference is the way how constants are treated: Let the variable $x$ be bounded by a lower bound $l \leq c \cdot x$ and a upper bound $d \cdot x \leq u$. In integer arithmetic, it is not possible to divide by $c$ or $d$. Thus, one has to multiply crosswise by $c$ and $d$ which yields: $d \cdot l \leq d \cdot c \cdot x \leq c \cdot u$

## 4 Equality Logic and Uninterpreted Functions

In its general form, Equality Logic is a theory that only allows variables and constants. There are no functions and no other predicates besides the equality-predicate $=$. When checking for satisfiability, one can even neglect constants since one can substitute them by fresh variables together with added constraints (e.g $\neg(x_a = x_b)$). These constraints prohibit the equality of each pair of variables (e.g. $x_a, x_b$) that represent two different constants (e.g. $a, b$). Here, we allow *uninterpreted functions*. This means that any model for a formula may interpret the function arbitrarily since the theory does not add any assumptions on the interpretation of the functions. The only requirement is *functional consistency*: Equal inputs lead to the same output. Solving equality logic with uninterpreted functions is a good start for validity and unsatisfiability: Formulas that are unsatisfiable (or valid) when neglecting the interpretation of their functions, keep this property (unsatisfiablity or validity, respectively) when fixing the interpretation of the function. The converse, however, is not the case since there might or might not be other interpretations of the function that allow the formula to be satisfied or not.

**Solving Equality Logic** Given a conjunction in Equality Logic that does not contain uninterpreted functions. One can represent this conjunction as a graph whose edges are annotated by $\neq$ and $=$. The corresponding conjunction is satisfiable if and only if the graph does not contain a cycle with precisely one $\neq$-edge: Due to transitivity of $=$, a satisfying variable assignment needs to assign the same value to all variables whose nodes are in a $=$-component[11]. A cycle with precisely one $\neq$-edge connects nodes in a $=$-component. Since their variables need to have same value, the conjunction is not satisfiable. If there is no cycle with precisely one $\neq$-edge, then all these $\neq$-edges connect different $=$-components. Thus, these negated equations can be resolved by just assigning different values to the components.

**The Ackermann Method** It is possible to convert a formula in Equality Logic with Uninterpreted Functions into an equisatisfiable one in pure Equality Logic. The Ackermann procedure substitutes terms by variables and adds constraints that ensure that a pair of variables is equal if they represent function calls whose input variables were equal. For example $x_{f(y)}, x_{f(z)}$ represent the function calls $f(y), f(z)$ and need to be equal if $y, z$ are equal: $(y = z) \rightarrow (x_{f(y)} = x_{f(z)})$. However, this procedure produces implications. Thus it not be used with the graph-based theory solver for Equality Logic, which required pure conjunctions.

**Congruence Closure Algorithm** It is possible to build a different theory solver for Equality Logic with Uninterpreted Functions that does not require substituting functions. The Congruence Closure Algorithm uses equality (explicit and transitive) and functional consistency to conclude sets of terms that have the same value in a satisfying assignment. This is done in three steps:

- *Explicit equality*: Each (not negated) equation forms sets consisting of both terms it compares. Terms that occur in the formula but in none of the two-elements sets are added as singletons. For example $x$ in $f(x) = g(x)$ and $y, z$ in $\neg(y = z)$ will be added as singletons.

[11] A connected component in the graph restricted to edges annotated by $=$

- *Transitive equality*: Until fixed point: Unite all pairs of sets sharing at least one term. This catches transitive equality.
- *Functional consistency*: For each equation between two calls of the same function: If each pair of corresponding input variables is in the same set, unite the sets containing the function-terms itself: This means that $x_1, y_1 \in F_1, \dots x_n, y_n \in F_n$ implies that the union $F_{f_x} \cup F_{f_y}$ with $f(x_1, \dots x_n) \in F_{f_x}, f(y_1, \dots y_n) \in F_{f_y}$ will be added. Repeat this until no function-equality adds new information. This step is technically the congruence closure. It catches equivalences that are implied by functional consistency.

These sets describe exactly which terms need to be equal. The original conjunction is satisfiable if and only if this conjunction contains no negated equality whose compared terms are in the same set.

By invoking a SAT-solver and the theory solver Congruence Closure Algorithm, one can solve the satisfiability modulo theories problem for (existentially quantified) CNF-formulas in Equality Logic with uninterpreted functions.

## 5 Conclusion

We have seen several procedures that can decide the satisfiability of a conjunct with respect to some theories. Our start was difference arithmetic where we used a graph theoretic problem. Pivoting, a technique of the simplex method, helped us to solve the problem for linear arithmetic. Variable elimination, as it is done by the Fourier-Motzkin method, was another way to implement a theory solver for linear arithmetic. In the case of integer linear arithmetic, we used the Branch and Bound Method to restrict the solutions to integer numbers. The Omega method transferred the idea of variable elimination to integer linear arithmetic. In the end, we introduced the congruence closure algorithm that implemented a theory solver for quality logic over uninterpreted functions.

## References


1 Kroening, Daniel, and Ofer Strichman. Decision procedures. Vol. 5. Heidelberg: Springer, 2008.

2 Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli. "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)." Journal of the ACM (JACM) 53.6 (2006): 937-977.

3 De Moura, Leonardo, and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications." Communications of the ACM 54.9 (2011): 69-77.

4 Barrett, Clark W., et al. "Satisfiability Modulo Theories." Handbook of satisfiability 185 (2009): 825-885.