Lecture Notes for

Logic and Computability

Course Number: IND04033UF

Contact

Bettina Könighofer
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
bettina.koenighofer@iaik.tugraz.at

Table of Contents

9 Satisfiability Modulo Theories **3**
9.1 Definitions and Notations 4
9.1.1 Theory of Equality and Uninterpreted Functions 5
9.2 Eager Encoding 6
9.3 Lazy Encoding 10
9.3.1 Theory Solvers and Congruence Closure 10

9

Satisfiability Modulo Theories

In computer science, the satisfiability modulo theories (SMT) problem refers to the problem of determining whether a formula in predicate logic is satisfiable with respect to some theory. *A theory fixes the interpretation/meaning of certain predicate and function symbols.* Checking whether a formula in predicate logic is satisfiable with respect to a theory means that we are not interested in arbitrary models but in models that interpret the functions and predicates contained in the theory as defined by the axioms in the theory. Consider the following formula that uses arithmetic:

$$\varphi \coloneqq \neg(a \geq b) \wedge (a + 1 > b).$$

We are not interested in models that use a nonstandard interpretation of the symbols $<$, $+$, and 1. We only want to consider models that use the correct interpretation of those symbols.

There exists several commonly used theories in computer science. For example, Presburger arithmetic is the theory of natural numbers with addition, more complex theories include the theory of integers or real numbers with arithmetic, and the theories of data structures such as lists, arrays, or bit vectors.

The two most used approaches for implementing SMT solvers are called *eager encoding* and *lazy encoding*. In eager encoding, all axioms of the theory are explicitly incorporated into the input formula. Eager encoding is not always possible and even if it is, the performance of solvers using eager encoding is often unacceptable. To avoid the explicit encoding of axioms, solvers that use lazy encoding use specialized theory solvers in combination with SAT solvers to decide the satisfiability of formulas within a given theory. In this chapter, we will have a brief glimpse into SMT, eager and lazy encoding.

9.1 Definitions and Notations

A theory in predicate logic is typically defined by its *signature* Σ, which is a set of constants, function and predicate symbols such as $\{0, 1, ..., +, -, ..., \leq\}$ and a set of *axioms* $\mathcal{A}$ that gives meaning to the predicate and function symbols. In SMT the *interpretation* defined by $\mathcal{A}$ is fixed and corresponds to the usual semantic of the operators. The equality symbol $=$ is assumed to be included in every signature.

Definition - Theory. A theory is as a pair $(\Sigma; \mathcal{A})$ where Σ is a signature which defines a set of constant, function, and predicate symbols. The set of axioms $\mathcal{A}$ is a set of closed predicate logic formulas in which only constant, function, and predicate symbols of Σ appear.

Linear Arithmetic Theories

The functions of the theory are $+$ and $-$ and the predicates are $=, \neq, <, >, \leq,$ and $\geq$. Variables can be of real sort ($\mathbb{R}$) or integer sort ($\mathbb{Z}$). If the variables are in $\mathbb{R}$, we have a problem of Linear Real Arithmetic (LRA). If the variables are in $\mathbb{Z}$, we have a problem of Linear Integer Arithmetic (LIA).

Therefore, for the theory of Linear Integer Arithmetic $\mathcal{T}_{LIA}$ we have:

- $\Sigma = \{....., -3, -2, -1, 0, 1, 2, 3..., =, +, -, \neq, <, >, \leq, \geq\}$
- $\mathcal{A}$ defines the usual meaning to all symbols. (Constant number symbols are mapped to the corresponding value in $\mathbb{Z}$, $+$ is interpreted as the function $0 + 0 \to 0$, $0 + 1 \to 1$, etc.).

Example for a formula in $\mathcal{T}_{LIA}$:

$$\varphi := x \geq 0 \wedge (x + y \leq 2 \vee x + y \geq 6) \wedge (x + y \geq 1 \vee x - y \geq 4).$$

$\mathcal{T}$-Terms, $\mathcal{T}$-Atoms and $\mathcal{T}$-Literals

- *Constants* in Σ are $\mathcal{T}$-terms well as *variables*. An applications of a function symbol in Σ where all inputs are $\mathcal{T}$-terms is a $\mathcal{T}$-term. Examples for $\mathcal{T}$-terms in $\mathcal{T}_{LIA}$ are: $x + 2$, 5, $x - y$.
- A $\mathcal{T}$-atom is the application of a predicate symbol in Σ where all inputs are $\mathcal{T}$-terms. Examples for $\mathcal{T}$-atoms in $\mathcal{T}_{LIA}$ are: $x + 2 > 0$, $5 \leq 2$, $x - y > 10$.
- A $\mathcal{T}$-literal is a $\mathcal{T}$-atoms or its negation. Examples for $\mathcal{T}$-literals are: $x+2 > 0$, $\neg(x + 2 > 0)$, $x - y > 10$, $\neg(x - y > 10)$.

A formula is any Boolean combination of $\mathcal{T}$-atoms and Boolean variables.

Models and $\mathcal{T}$-Satisfiability

In SMT, the interpretation of the predicate and function symbols is fixed. The only unspecified entities are free variables, for which a model has to define an assignment. A *model* $\mathcal{M}$ within a theory $\mathcal{T}$ is therefore an assignment of all free variables to a constant in Σ.

For example, lets consider the formula φ in $\mathcal{T}_{LIA}$:

$$\varphi \coloneqq (x + y > 0) \wedge (x = 0).$$

Under the model $\mathcal{M}_1 = \{x \to 5, y \to 1\}$ the formula φ evaluates to *false*. Under the model $\mathcal{M}_1 = \{x \to 0, y \to 0\}$ the formula φ evaluates to *true*.

Definition - $\mathcal{T}$-satisfiablility. We say that a *formula φ is satisfiable in a theory* $\mathcal{T}$, or $\mathcal{T}$-satisfiable, if and only if there is a model $\mathcal{M}$ within $\mathcal{T}$ that satisfies φ (i.e., $\mathcal{M} \vDash A$ for every $A \in \mathcal{A}$ and $\mathcal{M} \vDash \varphi$).

Definition - $\mathcal{T}$-valid. We say that a *formula φ is valid in a theory* $\mathcal{T}$, or $\mathcal{T}$-valid, if and only if all models within $\mathcal{T}$ satisfy φ.

Definition - $\mathcal{T}$-entailment. A set of formulas $\varphi_1, \dots \varphi_n$ $\mathcal{T}$-entails a formula ψ, written as $\varphi_1, \dots \varphi_n \vDash_{\mathcal{T}} \psi$, if every model of $\mathcal{T}$ that satisfies all formulas $\varphi_1, \dots \varphi_n$ satisfies ψ as well.

Definition $\mathcal{T}$-decidable. A theory $\mathcal{T}$ is decidable if there exists an algorithm that always terminates with "yes" if φ is $\mathcal{T}$-valid or with "no" if φ is $\mathcal{T}$-invalid.

9.1.1 Theory of Equality and Uninterpreted Functions

We discuss the theory of equality and uninterpreted functions $\mathcal{T}_{EUF}$ in detail. $\mathcal{T}_{EUF}$ is the simplest first-order theory and therefore we will discuss lazy encoding and eager encoding for this theory. *Uninterpreted functions* are often used as an abstraction technique to remove unnecessarily complex or irrelevant details of a system being modeled. An uninterpreted function or function symbol is one that has no other property than its name and the function congruence property: a function always returns the same output for the same input.

Definition - Theory of Equality and Uninterpreted Functions. The theory of equality and uninterpreted functions $\mathcal{T}_{EUF}$ has the signature

$$\Sigma_{EUF} \coloneqq \{=, a, b, c, \dots, f, g, h, \dots, P, Q, R, \dots\}.$$

The axioms $\mathcal{A}_{EUF}$ are the following:

1. $\forall x.\ \ x = x$ (reflexivity)
2. $\forall x, y.\ \ x = y \to y = x$ (symmetry)
3. $\forall x, y, z.\ \ x = y \wedge y = z \to x = z$ (transitivity)
4. $\forall \overline{x}, \overline{y}.\ \ (\bigwedge_{i=1}^{n} x_i = y_i) \to f(\overline{x}) = f(\overline{y})$ (congruence)
5. $\forall \overline{x}, \overline{y}.\ \ (\bigwedge_{i=1}^{n} x_i = y_i) \to (P(\overline{x}) \leftrightarrow P(\overline{y}))$ (equivalence)

The signature Σ_{EUF} consists the binary predicate $=$ (equality) and all constant symbols such as $a, b, c, \dots$, function symbols such as $f, g, h, \dots$ and predicate symbols such as $P, Q, R, \dots$. The axioms of the theory of equality include the axioms of reflexivity, symmetry, transitivity, function congruence, and equivalence.

With the theory of uninterpreted function and equality it is often possible to show properties of systems that use complex functions by abstracting away the complexity of the functions, e.g., in order to analyse properties of encryption schemas. For example, suppose we want to prove that the following set of theory literals is unsatisfiable:

$$\{a \cdot (f(b) + f(c)) = d, \quad b \cdot (f(a) + f(c)) \neq d, \quad a = b\}.$$

At first, it may appear that this requires reasoning in the theory of arithmetic. However, if we replace $+$ and $\cdot$ with uninterpreted functions p and m respectively, we get a new set of literals:

$$\{m(a, p(f(b), f(c))) = d, \quad m(b, p(f(a), f(c))) \neq d, \quad a = b\}.$$

We can prove that the conjunction of these literals is unsatisfiable without any arithmetic, just by using the definition uninterpreted functions.

9.2 Eager Encoding

The eager approach to SMT solving involves *translating the original formula to an equisatisfiable Boolean formula in a single step.* The translation is done by encoding enough relevant consequences of the theory $\mathcal{T}$ into the Boolean formula. The eager approach applies in principle to any theory with a decidable satisfiability problem, possibly however at the cost of a significant blow-up in the translation.

The main idea of eager encoding is that the input formula is translated into a propositional formula with all relevant theory-specific information encoded into the formula. The resulting large propositional formula can then be given to any off-the-shelf SAT solver. Given a formula φ, algorithms based on eager encoding (i.e., direct encoding of axioms) operate in three steps:

1. Replace any unique $\mathcal{T}$-atom in the original formula φ with a fresh Boolean variable to get a Boolean formula $\hat{\varphi}$.
2. Generate a Boolean formula φ_{cons} that constrains the values of the introduced Boolean variables to preserve the information of the theory.
3. Invoke a SAT solver on the Boolean formula $\varphi_{prop} \coloneqq \hat{\varphi} \wedge \varphi_{cons}$ that corresponds to an equisatisfiable propositional formula to φ.

The translations used in the eager approach are of course theory specific. In the following Sections, we will discuss the translation of a formula within $\mathcal{T}_{EUF}$ to an equisatisfiable propositional formula. The translation consists of two separate steps to construct the propositional formula φ_{prop}. First, we remove all function instances using the Ackermann algorithm. In a second step, we remove all equality instances by a graph-based algorithm.

Elimination of Function Applications - via Ackermann Algorithm

The first step in the transformation to a Boolean formula is to eliminate applications of function and predicate symbols of non-zero arity. These applications are replaced by new propositional symbols and additional constraints are added on this fresh variables to maintain functional consistency (the congruence property).

In detail, given an input formula φ_{EUF} in $\mathcal{T}_{EUF}$, the Ackermann construction algorithm works as follows:

- It generates the formula $\hat{\varphi}_{EUF}$ by replacing every function application in φ_{EUF} with a fresh variables.
- It generates the formula φ_{FC} which encodes all required functional congruence constraints.
- $\varphi_E = \hat{\varphi}_{EUF} \wedge \varphi_{FC}$ is equisatisfiable with φ_{EUF} and contains no uninterpreted function symbols. Therefore, φ_E is in $\mathcal{T}_E$.

Example. Given the formula

$$\varphi_{UEF} \coloneqq (f(a) = f(b)) \wedge \neg(f(b) = f(c)).$$

Apply the Ackermann construction algorithm to compute an equisatisfiable formula in $\mathcal{T}_E$.

Solution. In φ_{UEF} we have three instances of the function f. Therefore, we need three fresh variables f_a, f_b, and f_c, one for each instance. Replacing the function instances with the fresh variables yields to:

$$\hat{\varphi}_{EUF} \coloneqq (f_a = f_b) \wedge \neg(f_b = f_c).$$

Second, we encode the functional consistency constraints for f:

$$\varphi_{FC} \coloneqq ((a = b) \rightarrow f_a = f_b) \wedge ((b = c) \rightarrow f_b = f_c) \wedge ((a = c) \rightarrow f_a = f_c).$$

The resulting equisatisfiable formula in $\mathcal{T}_E$ is $\varphi_E \coloneqq \hat{\varphi}_{EUF} \wedge \varphi_{FC}$.

Example. Given the formula $\varphi_{EUF} \coloneqq$

$$(z = f(x, z) \leftrightarrow f(x, y) = x) \wedge (y \neq x \vee f(y, z) = f(x, y) \vee x = z) \rightarrow f(x, z) = z.$$

Apply the Ackermann construction algorithm to compute an equisatisfiable formula in $\mathcal{T}_E$.

Solution.

$$\begin{aligned}
\varphi_{FC} \equiv & (x = x \wedge y = z) \rightarrow (f_{xy} = f_{xz}) \wedge \\
& (x = y \wedge y = z) \rightarrow (f_{xy} = f_{yz}) \wedge \\
& (x = y \wedge z = z) \rightarrow (f_{xz} = f_{yz}) \\
\hat{\varphi}_{EUF} \equiv & (z = f_{xz} \leftrightarrow f_{xy} = x) \wedge \\
& (y \neq x \vee f_{yz} = f_{xy} \vee x = z) \rightarrow f_{xz} = z
\end{aligned}$$

$$\textit{Formula in Theory of Equality} : \varphi_E \equiv \varphi_{FC} \wedge \hat{\varphi}_{EUF}$$

Example. Given the formula $\varphi_{EUF} \coloneqq f(x) = f(g(y)) \wedge f(y) \neq y \vee f(x) = f(y) \wedge f(y) = y \vee f(y) = f(x) \wedge y \neq f(g(y)) \vee f(y) = g(x) \wedge f(y) = y$.

Apply the Ackermann construction algorithm to compute an equisatisfiable formula in $\mathcal{T}_E$.

Solution.

$$\begin{aligned}\varphi_{FC} \equiv (x = y) &\rightarrow (f_x = f_y) \wedge \\ (x = g_y) &\rightarrow (f_x = f_{g_y}) \wedge \\ (y = g_y) &\rightarrow (f_y = f_{g_y}) \wedge \\ (x = y) &\rightarrow (g_x = g_y)\end{aligned}$$

$$\begin{aligned}\hat{\varphi}_{EUF} \equiv f_x = f_{g_y} \wedge f_y \neq y \vee \\ f_x = f_y \wedge f_y = y \vee \\ f_y = f_x \wedge y \neq f_{g_y} \vee \\ f_y = g_x \wedge f_y = y\end{aligned}$$

$$\varphi_E \equiv \varphi_{FC} \wedge \hat{\varphi}_{EUF}$$

Elimination of Equalities - Graph-based Reduction

For an input formula φ_E in $\mathcal{T}_E$, the graph based reduction algorithm, introduced by Bryant and Velev, computes an equisatisfiable propositional formula. The algorithm computes $\hat{\varphi}_E$ to preserve the logical structure, reflexivity and symmetry properties, and φ_{TC} to preserve transitivity.

The formula $\hat{\varphi}_E$ is computed via the following steps:

1. Every *reflexivity* instance $a = a$ in φ_E is replaced by *true*.
2. Every equality atom is rewritten such that the first term precedes the second term with respect to some total order.
3. Every equality atom $a = b$ is replaced by a fresh propositional variable $e_{a=b}$. This results in the formula $\hat{\varphi}_E$.

To compute φ_{TC} , we construct a so-called *non-polar equality graph*: This graph has a node for every term and an edge for every equality and disequality in the formula (there is no difference between equality and disequality in the graph). This graph is then made *chordal*.

Definition - Chords, Chord-free Cycles, and Chordal Graphs. In a graph G, let n_1 and n_2 be two non-adjacent nodes in a cycle. An edge between n_1 and n_2 is called a chord. A cycle is said to be *chord-free*, if in the cycle there

exist no non-adjacent nodes that are connected by an edge. A graph is called *chordal*, if it contains no chord-free cycles with size greater than 3.

A graph can be made chordal by adding additional edges. By only having such triangles in the graph, we avoid an exponential blow-up in the number of transitivity constraints. Based on the chordal graph, we can compute the transitivity constraints. For every triangle (x, y, z) in the graph, we add the following constraints:

$$(e_{x=y} \wedge e_{y=z} \rightarrow e_{x=z}) \wedge (e_{x=y} \wedge e_{x=z} \rightarrow e_{y=z}) \wedge (e_{y=z} \wedge e_{x=z} \rightarrow e_{x=y})$$

We connect the transitivity constraints for all triangles via conjunction to obtain φ_{TC}. The resulting equisatisfiable propositional formula:

$$\varphi_{prop} \coloneqq \hat{\varphi}_E \wedge \varphi_{TC}.$$

Example. Perform the graph-based reduction on the following formula to compute an equisatisfiable formula in propositional logic.

$$f_x = f_a \wedge f_y \neq y \vee f_x = f_y \wedge f_y = y \vee f_y = f_x \wedge y \neq f_a \vee f_y = g_x \wedge f_y = y$$

Solution.

$$\begin{aligned}
\varphi_{TC} \equiv & (e_{f_x=f_a} \wedge e_{y=f_a} \rightarrow e_{f_x=y}) \wedge \\
& (e_{f_x=f_a} \wedge e_{f_x=y} \rightarrow e_{y=f_a}) \wedge \\
& (e_{y=f_a} \wedge e_{f_x=y} \rightarrow e_{f_x=f_a}) \wedge \\
& (e_{f_x=y} \wedge e_{y=f_y} \rightarrow e_{f_x=f_y}) \wedge \\
& (e_{f_x=y} \wedge e_{f_x=f_y} \rightarrow e_{y=f_y}) \wedge \\
& (e_{y=f_y} \wedge e_{f_x=f_y} \rightarrow e_{f_x=f_y}) \\
\hat{\varphi}_E \equiv & e_{f_x=f_a} \wedge \neg e_{f_y=y} \vee \\
& e_{f_x=f_y} \wedge e_{f_y=y} \vee \\
& e_{f_y=f_x} \wedge \neg e_{y=f_a} \vee \\
& e_{f_y=g_x} \wedge e_{f_y=y}
\end{aligned}$$

$$\textit{Boolean Formula} : \varphi_{prop} \equiv \varphi_{TC} \wedge \hat{\varphi}_E$$

Example. Perform graph-based reduction to translate a formula in $\mathfrak{T}_E$ into an equisatisfiable formula in propositional logic.

$$(z = f_{xz} \leftrightarrow f_{xy} = x) \wedge (\neg y = x \vee f_{yz} = f_{xy} \vee x = z) \rightarrow z = f_{xz}$$

Solution. $\hat{\varphi}_E \equiv (e_{z=f_{xz}} \leftrightarrow e_{f_{xy}=x}) \wedge (\neg e_{y=x} \vee e_{f_{yz}=f_{xy}} \vee e_{x=z}) \rightarrow e_{z=f_{xz}}$
$\varphi_{TC} \coloneqq \mathit{true}$
$\varphi_{prop} \equiv \varphi_{TC} \wedge \hat{\varphi}_E$

9.3 Lazy Encoding

Lazy encoding is based on the interaction between a SAT solver and a so-called theory solver. A theory solver is an algorithm that can decide satisfiability of the conjunctive fragment of a theory. In contrast to eager encoding, where a sufficient set of constraints is computed at the beginning, lazy encoding starts with no constraints at all, and lazily adds constraints only when required.

The principle of lazy encoding is shown in Figure 9.1. To decide whether or not a $\mathcal{T}$-formula φ is $\mathcal{T}$-satisfiable, the propositional skeleton skel(φ) is given to a SAT solver. If the SAT solver returns *unsatisfiable*, the procedure is done and we know that φ is not $\mathcal{T}$-satisfiable. If, however, the SAT solver returns *satisfiable*, we obtain a satisfying assignment for the truth values of the theory atoms in φ. This assignment is a formula in the conjunctive fragment of $\mathcal{T}$, which we pass to the theory solver. If the theory solver returns *satisfiable*, we have found an assignment of truth values to the theory atoms that is *consistent* with $\mathcal{T}$. Thus we know that ϕ is $\mathcal{T}$-satisfiable. If, however, the theory solver returns *unsatisfiable*, we have to look for another assignment, as the present one is not consistent with $\mathcal{T}$. To obtain a different assignment, we negate the inconsistent assignment - which conveniently turns it into a clause - and add it as a so-called blocking clause to the CNF of skel(φ). The blocking clause ensures that the next satisfying assignment obtained from the SAT solver (if one exists) is different from the current, $\mathcal{T}$-inconsistent assignment. This loop is repeated until we encounter one of the following two terminal cases: (1) the SAT solver suggests an assignment that the theory solver finds to be consistent with $\mathcal{T}$, in which case φ is $\mathcal{T}$-satisfiable. (2) we have added so many blocking clauses that the SAT solver cannot find any more assignments, in which case φ is not $\mathcal{T}$-satisfiable. As every blocking clause excludes (at least) one of only finitely many assignments, the loop is guaranteed to terminate.

9.3.1 Theory Solvers and Congruence Closure

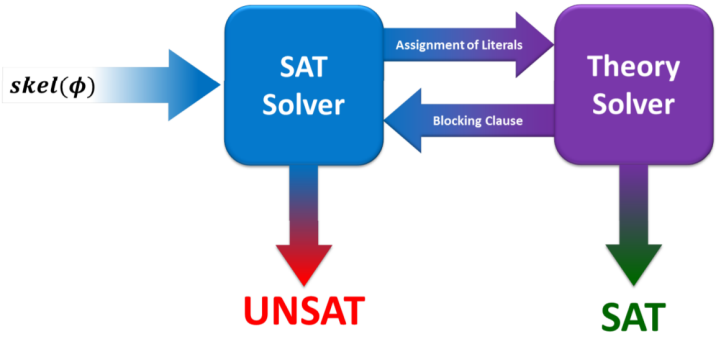
Theory Solvers. Theory solvers are specialized on deciding a specific background theory, or a fragment of a background theory $\mathcal{T}_{EUF}$. The common practice is to write theory solvers just for deciding *conjunctions of literals*; i.e., atomic formulas and their negations. The main advantage of theory-specific solvers is that one can use whatever specialized algorithms and data structures are best for the theory in question, which typically leads to better performance.

The role of the theory solver is to accept a set of literals and report whether the set is $\mathcal{T}$-satisfiable or not. The congruence closure algorithm is the most common theory solver for $\mathcal{T}_{EUF}$.

Congruence Closure

Given a conjunction of $\mathcal{T}_{EUF}$-literals, the congruence closure algorithm computes a set of *congruence classes*, such that all terms in the same congruence class are equal. Congruence classes are computed in the following way.

Figure 9.1: The propositional skeleton of φ is given to a SAT solver. If a satisfying assignment is found, it is checked by a theory solver. If the assignment is consistent with the theory, φ is $\mathcal{T}$-satisfiable. Otherwise, a blocking clause is generated and the SAT solver searches for a new assignment. This is repeated until either a $\mathcal{T}$-consistent assignment is found, or the SAT solver cannot find any more assignments.



1. All terms for which there is a (positive) equality in the conjunction of literals are put into the same congruence class. All remaining terms are put in singleton classes.
2. Any two classes that contain common terms are merged. This accounts for the transitivity of the equality predicate.
3. Classes are merged based on function congruence. That is, if two classes both contain an instance of the same uninterpreted function, and corresponding parameters are already in the same congruence class (which means that they are equal), the classes of the function instances are merged.
4. Repeat step 2 and 3 until no more merging can be done.
5. In the last step, all the disequalities from the set of input literals are checked against the merged congruence classes. If there is a disequality that contradicts the congruence classes (both its terms are in the same congruence class), the conjunction of literals is unsatisfiable. If no such disequality exists, the conjunction of literals is satisfiable.

Example. Use the congruence closure algorithm to check whether the following formula is satisfiable. $f(a) = e \wedge f(c) \neq f(e) \wedge a = f(b) \wedge f(b) \neq c \wedge b \neq a \wedge f(a) = d \wedge d \neq f(c) \wedge b = d \wedge a \neq e \wedge c = d$

Solution.

$$\{\underline{f(a)}, e\}, \{a, f(b)\}, \{\underline{f(a)}, d\}, \{b, d\}, \{c, d\}, \{f(c)\}, \{f(e)\}$$
$$\{f(a), e, \underline{d}\}, \{a, f(b)\}, \{b, \underline{d}\}, \{c, \underline{d}\}, \{f(c)\}, \{f(e)\}$$
$$\{f(a), \underline{e}, d, b, \underline{c}\}, \{a, f(b)\}, \{f(c)\}, \{f(e)\}$$
$$\{f(a), e, d, \underline{b}, \underline{c}\}, \{a, f(b)\}, \{f(c), f(e)\}$$
$$\{f(a), e, d, b, c\}, \{a, f(b), f(c), f(e)\}$$

Checking the disequality $f(c) \neq f(e)$ leads to the result that the assignment is UNSAT, since $f(c)$ and $f(e)$ are in the same congruence class.

Example. Use the congruence closure algorithm to check whether the following formula is satisfiable. $\varphi \coloneqq f(b) = a \wedge c \neq d \wedge f(e) = b \wedge d \neq f(b) \wedge f(a) = f(e) \wedge b \neq f(b) \wedge a \neq e \wedge f(a) = e \wedge a = c \wedge f(b) \neq e \wedge d = f(c)$

Solution.

$$\{f(b), a\}, \{f(e), b\}, \{\underline{f(a)}, f(e)\}, \{\underline{f(a)}, e\}, \{a, c\}, \{d, f(c)\}$$
$$\{f(b), a\}, \{\underline{f(e)}, b\}, \{f(a), \underline{f(e)}, e\}, \{a, c\}, \{d, f(c)\}$$
$$\{f(b), \underline{a}\}, \{f(a), f(e), e, b\}, \{\underline{a}, c\}, \{d, f(c)\}$$
$$\{f(b), a, c\}, \{f(a), f(e), \underline{e}, \underline{b}\}, \{d, f(c)\}$$
$$\{f(b), a, c, f(a), f(e), e, b\}, \{d, f(c)\}$$

Checking the disequality $f(b) \neq e$ leads to the result that the assignment is UNSAT, since $f(b)$ and e are in the same congruence class.

Example. Use the congruence closure algorithm to check whether the following formula is satisfiable: $\varphi \coloneqq x = y \wedge v = w \wedge z = f(w) \wedge z \neq x \wedge w \neq f(y) \wedge f(x) = w \wedge f(z) = f(x) \wedge f(z) = f(v)$

Solution.

$\{x, y\}, \{v, \underline{w}\}, \{z, f(w)\}, \{f(x), \underline{w}\}, \{f(z), f(x)\}, \{f(z), f(v)\}$
$\{x, y\}, \{v, w, \underline{f(x)}\}, \{z, f(w)\}, \{f(z), \underline{f(x)}\}, \{f(z), f(v)\}$
$\{x, y\}, \{v, w, \underline{f(x)}, \underline{f(z)}\}, \{z, f(w)\}, \{\underline{f(z)}, f(v)\}$
$\{x, y\}, \{v, w, f(x), \underline{f(z)}, f(v)\}, \{z, f(w)\}$

$z \neq x$✓
$w \neq f(y)$✓
φ is SAT

Chapter 9 was based on the following books.

- A. Biere, M. Heule, H. van Maaren, and T. Walsh: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, (2009)
- Georg Hofferek: Controller Synthesis with Uninterpreted Functions. PhD Thesis. 2014. Graz University of Technology.