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

8

Satisfiability Modulo Theories

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 specifies 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:

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

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 well-established interpretation of those symbols.

There exist 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.

In this chapter, we discuss the two most commonly used approaches for implementing SMT solvers, namely *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 based on lazy encoding use specialized theory solvers in combination with SAT solvers to decide the satisfiability of formulas within a given theory.

8.1 Definitions and Notations

A theory in predicate logic consists of two parts:

- A *signature* Σ, which is a set of constants, functions, and predicate symbols 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 common semantic of the operators.

Definition 8.1 (Theory) A theory is as a pair $(\Sigma; \mathcal{A})$, where Σ is a *signature* that defines the set of constant, function, and predicate symbols used in the theory. 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. We assume the equality symbol $=$ to be included in every signature.

A formula φ in SMT regarding a fixed theory $\mathcal{T} = (\Sigma, \mathcal{A})$ may only consist of logical connectives, variables, quantifiers and symbols from Σ.

Linear Integer Arithmetic. A first example for a theory is *linear integer arithmetic*. The constant symbols of this theory consist of the set $\mathbb{Z}$. The function symbols are $+$ and $-$ and predicate symbols for this theory are $=$, $\neq$, $<$. $\leq$, $>$ and $\geq$. The set of axioms $\mathcal{A}$ gives the well-established meaning to these function and predicate symbols.

Therefore, for the theory of linear integer arithmetic $\mathcal{T}_{LIA}$ we have:

- $\Sigma = \mathbb{Z} \cup \{+, -\} \cup \{=, \neq, <, \leq, >, \geq\}$
- $\mathcal{A}$ defines the usual meaning to all symbols:
  - Constant symbols are mapped to the corresponding value in $\mathbb{Z}$.
  - $+$ is interpreted as the function $0 + 0 \to 0, 0 + 1 \to 1, \ldots$. $-$ follows it analogous interpretation.
  - The predicate symbols are interpreted as their respective comparison operator.

An 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, $\mathcal{T}$-Literals and $\mathcal{T}$-Formulas

Definition 8.2 ($\mathcal{T}$-terms) A $\mathcal{T}$-*term* is either a constant or variables $x, y, \ldots$. An application 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, $x - y$.

Definition 8.3 ($\mathcal{T}$-atom) 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: $x + 2 > 0$, $5 \leq 2$, $x - y > 10$.

Definition 8.4 ($\mathcal{T}$-literal) A $\mathcal{T}$-*literal* is a $\mathcal{T}$-atoms or its negation.

Definition 8.5 ($\mathcal{T}$-formula) A $\mathcal{T}$-*formula* is a well-formed predicate logic formula consisting of $\mathcal{T}$-literals and logical connectives, variables and quantifiers.

An example for a $\mathcal{T}$-formula is: $\forall x \, \forall y \; x + y = y + x$

For the remainder of this chapter, we will consider only the *quantifier-free* fragment of a given theory.

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 $\mathcal{T}_{LIA}$:

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

Under the model $\mathcal{M}_0 = \{x \to 5, y \to 1\}$ the formula φ evaluates to *false*. Under the model $\mathcal{M}_1 = \{x \to 0, y \to 1\}$ the formula φ evaluates to *true*.

Definition 8.6 ($\mathcal{T}$-satisfiablility) 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 $\mathcal{T}$, this is $\mathcal{M} \vDash A$ for every $A \in \mathcal{A}$, that satisfies φ.

Definition 8.7 ($\mathcal{T}$-validity) 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 8.8 ($\mathcal{T}$-entailment) A set of formulas $\varphi_1, \ldots \varphi_n$ $\mathcal{T}$-entails a formula ψ, written as $\varphi_1, \ldots \varphi_n \vDash_{\mathcal{T}} \psi$, if every model of $\mathcal{T}$ that satisfies all formulas $\varphi_1, \ldots \varphi_n$ satisfies ψ as well.

Definition 8.9 ($\mathcal{T}$-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 not $\mathcal{T}$-valid.

8.1.1 Theory of Equality and Uninterpreted Functions

We continue with an in depth discussion of the *theory of equality and uninterpreted functions* $\mathcal{T}_{EUF}$ in detail. $\mathcal{T}_{EUF}$ is one of the simplest first-order theories and we will therefore use it to discuss lazy encoding and eager encoding.

Uninterpreted functions serve as a way to describe uninterpreted relations between elements of the theory. They have no explicit property apart from *functional congruence*, meaning that a function always returns the same output for the same input. Uninterpreted functions are often used as an abstraction technique to remove unnecessarily complex or irrelevant details of a system being modeled.

Definition 8.10 (Theory of Equality and Uninterpreted Functions) The theory of equality and uninterpreted functions $\mathcal{T}_{EUF}$ has the signature

$$\Sigma_{EUF} := \{a, b, c, \ldots\} \cup \{f, g, h, \ldots\} \cup \{=, P, Q, R, \ldots\},$$

where

- $a, b, c, \ldots$ are constant symbols,
- $f, g, h, \ldots$ are function symbols, and
- $P, Q, R, \ldots$ are predicate symbols.

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

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

Example. By using 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 defined uninterpreted functions.

8.2 Eager Encoding

The *eager encoding approach* for solving SMT formulas involves *translating the original formula to an equisatisfiable propositional formula in*, potentially multiple, *preprocessing steps.* The translations are done as a preprocessing step to encode enough relevant consequences of the axioms $\mathcal{A}$ of $\mathcal{T}$ into the propositional 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, where the translations are of course dependent on $\mathcal{T}$. After encoding the relevant consequences of $\mathcal{A}$ into the propositional formula, the formula can be given to any off-the-shelf SAT solver.

In order to perform eager encoding of a formula φ, any algorithm follows the following general steps:

(I) Replace any unique $\mathcal{T}$-atom in the original formula φ with a fresh propositional variable to get a propositional formula $\hat{\varphi}$.

(II) Generate a propositional formula φ_{cons} that constrains the values of the introduced propositional variables to preserve the information of the theory.

(III) Invoke a SAT solver on the propositional formula $\varphi_{prop} \coloneqq \hat{\varphi} \wedge \varphi_{cons}$ that corresponds to an equisatisfiable propositional formula to φ.

Note that steps (I) and (II) might be split into multiple steps to encode the consequences of $\mathcal{A}$.

In the following, we will discuss the exact steps needed to translate a formula within $\mathcal{T}_{EUF}$ to an equisatisfiable propositional formula.

Elimination of Function Applications - via Ackermann Algorithm

The first step for transforming a formula φ_{EUF} in $\mathcal{T}_{EUF}$ to a propositional 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, i.e. the congruence property.

The Ackermann algorithm performs the following steps:

- Generate the formula $\hat{\varphi}_{EUF}$ by replacing every function application and predicate application in φ_{EUF} with a fresh variables.
- Generate 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 1

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.
Replacing the function instances with fresh variables yields:

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

Next, 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 2

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.

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

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

Example 3

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.

$$\begin{aligned}
\varphi_{FC} \equiv & (x = y) \rightarrow (f_x = f_y) \wedge \\
& (x = g_y) \rightarrow (f_x = f_{g_y}) \wedge \\
& (y = g_y) \rightarrow (f_y = f_{g_y}) \wedge \\
& (x = y) \rightarrow (g_x = g_y) \\
\hat{\varphi}_{EUF} \equiv & (f_x = f_{g_y} \wedge f_y \neq y) \wedge \\
& (f_x = f_y \wedge f_y = y) \wedge \\
& (f_y = f_x \wedge y \neq f_{g_y}) \wedge \\
& (f_y = g_x \wedge f_y = y) \\
\varphi_E \equiv & \varphi_{FC} \wedge \hat{\varphi}_{EUF}
\end{aligned}$$

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} encodes transitivity constraints.

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 8.11 (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 4

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.

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

Example 5

Perform graph-based reduction to translate a formula in $\mathcal{T}_E$ 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.

$$\begin{aligned}\hat{\varphi}_E \equiv & (e_{z=f_{xz}} \leftrightarrow e_{f_{xy}=x}) \wedge (\neg e_{y=x} \vee e_{f_{yz}=f_{xy}} \vee e_{x=z}) \rightarrow e_{z=f_{xz}} \\ \varphi_{TC} \equiv & \; true \\ \varphi_{prop} \equiv & \varphi_{TC} \wedge \hat{\varphi}_E\end{aligned}$$

8.3 Lazy Encoding

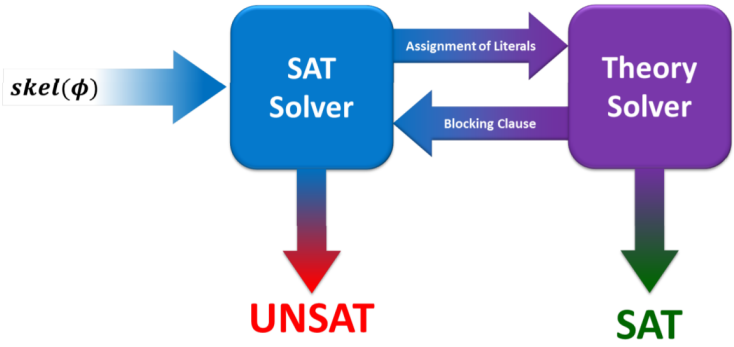
The *lazy encoding aproach* is based on the interaction between a SAT solver and a so-called *theory solver*. A theory solver is able to decide the satisfiability of the *conjunctive fragment of a theory*. In contrast to eager encoding,the lazy approach starts with no constraints at all, and adds constraints when required during the computation.

The principle of lazy encoding is shown in Figure 8.1. To decide if a $\mathcal{T}$-formula φ is $\mathcal{T}$-satisfiable, the propositional skeleton skel(φ) is given to a SAT solver.

Definition 8.12 (Propositional Skeleton) The *propositional skeleton* skel(φ) of a formula φ is obtained by replacing each occurance of a $\mathcal{T}$-literal with a propositional variable.

If the SAT solver returns *unsatisfiable* when given skel(φ), we can deduce that the propositional structure of φ is unsatisfiable, and therefore φ cannot be $\mathcal{T}$-satisfiable. If, however, the SAT solver returns *satisfiable*, we obtain a satisfying assignment for the truth values of the $\mathcal{T}$-literals in φ. This assignment is a formula in the conjunctive fragment of $\mathcal{T}$, which we pass to a dedicated theory solver. If the theory solver returns *satisfiable*, we have found an assignment of truth values to the theory literals that is *consistent* with the axioms $\mathcal{A}$ in $\mathcal{T}$. If, however, the theory solver returns *unsatisfiable* the current assignment is not consistent with $\mathcal{T}$ and we have to consider a different assignment. 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 potential satisfying assignment obtained from the SAT solver is different from the current one. If the SAT solver cannot produce a satisfying assignment, we deduce that φ is unsatisfiable.

Figure 8.1: 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.



8.3.1 Congruence Closure - A Theory Solver for $\mathcal{T}_{EUF}$

Theory Solvers. Theory solvers are dedicated for a specific theory, or a fragment of a theory. The common practice is to write theory solvers just for deciding *conjunctions of* $\mathcal{T}$*-literals.*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.

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.

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 inequalities 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 6

Use the congruence closure algorithm to check whether the following formula is satisfiable.

$$\begin{aligned}\varphi :=& f(a) = e \wedge f(c) \neq f(e) \wedge a = f(b) \wedge \\ & f(b) \neq c \wedge b \neq a \wedge f(a) = d \wedge \\ & d \neq f(c) \wedge b = d \wedge a \neq e \wedge c = d\end{aligned}$$

Solution.

$$\begin{aligned}&\{\underline{f(a)}, e\}, \{a, f(b)\}, \{\underline{f(a)}, d\}, \{b, d\}, \{c, d\}, \{f(c)\}, \{f(e)\} \\ &\{f(a), e, \underline{d}\}, \{a, f(b)\}, \{b, \underline{d}\}, \{c, \underline{d}\}, \{f(c)\}, \{f(e)\} \\ &\{f(a), \underline{e}, d, b, \underline{c}\}, \{a, f(b)\}, \{f(c)\}, \{f(e)\} \\ &\{f(a), e, d, \underline{b}, \underline{c}\}, \{a, f(b)\}, \{f(c), f(e)\} \\ &\{f(a), e, d, b, c\}, \{a, f(b), f(c), f(e)\}\end{aligned}$$

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 7

Use the congruence closure algorithm to check whether the following formula is satisfiable.

$$\begin{aligned}\varphi :=& f(b) = a \wedge c \neq d \wedge f(e) = b \wedge \\ & d \neq f(b) \wedge f(a) = f(e) \wedge b \neq f(b) \wedge \\ & a \neq e \wedge f(a) = e \wedge a = c \wedge \\ & f(b) \neq e \wedge d = f(c)\end{aligned}$$

Solution.

$$\begin{aligned}&\{f(b), a\}, \{f(e), b\}, \{\underline{f(a)}, f(e)\}, \{\underline{f(a)}, e\}, \{a, c\}, \{d, f(c)\} \\ &\{f(b), a\}, \{\underline{f(e)}, b\}, \{f(a), \underline{f(e)}, e\}, \{a, c\}, \{d, f(c)\} \\ &\{f(b), \underline{a}\}, \{f(a), f(e), e, b\}, \{\underline{a}, c\}, \{d, f(c)\} \\ &\{f(b), a, c\}, \{f(a), f(e), \underline{e}, \underline{b}\}, \{d, f(c)\} \\ &\{f(b), a, c, f(a), f(e), e, b\}, \{d, f(c)\}\end{aligned}$$

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 8

Use the congruence closure algorithm to check whether the following formula is satisfiable:

$$\begin{aligned}\varphi :=& x = y \wedge v = w \wedge z = f(w) \wedge z \neq x \wedge w \neq f(y) \wedge \\ & f(x) = w \wedge f(z) = f(x) \wedge f(z) = f(v)\end{aligned}$$

Solution.

$$\begin{aligned}&\{x, y\}, \{v, \underline{w}\}, \{z, f(w)\}, \{f(x), \underline{w}\}, \{f(z), f(x)\}, \{f(z), f(v)\} \\ &\{x, y\}, \{v, w, \underline{f(x)}\}, \{z, f(w)\}, \{f(z), \underline{f(x)}\}, \{f(z), f(v)\} \\ &\{x, y\}, \{v, w, f(x), \underline{f(z)}\}, \{z, f(w)\}, \{\underline{f(z)}, f(v)\} \\ &\{x, y\}, \{v, w, f(x), f(z), f(v)\}, \{z, f(w)\}\end{aligned}$$

We have to check the following two inequalities:

- $z \neq x$✓
- $w \neq f(y)$✓

Since both are not violated we can conclude that φ is SAT.

Chapter 7 is 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.

List of Definitions