SMT: Equality Logic With Uninterpreted Functions

Koller, Lukas

June 2020

1 Introduction

Consider the problem of verifying the correctness of a program. One approach is, given pre-conditions, to check if the output of the program satisfies certain post-conditions. A large formula is constructed from the program and its pre- and post-conditions. Thus, the problem of verifying the program comes down to checking the satisfiability of the constructed formula.

Satisfiability modulo theories (SMT) solvers are widely used in program verification. As assigning and comparing values is a common task in programs, it is convenient to use equalities in the formula. Hence, a SMT solver has to be able to reason about equality. To reason about equality, any modern-day SMT solver contains a solver for the *theory of equality logic with uninterpreted functions* (EUF) with an implementation of the *congruence closure* algorithm.

This paper gives an introduction to SMT and EUF. The essential congruence closure algorithm is explained and an efficient implementation of the algorithm is given using *Directed-Acylic-Graphs* (DAGs) and the *Union-Find* algorithm.

Ackermann (1954) showed that EUF is decidable. Based on this, Shostak (1978) formulated a decision procedure for formulas in EUF with the congruence closure algorithm. Soon after efficient implementations using DAGs and the Union-Find algorithm followed (Nelson and Oppen, 1980; Downey et al., 1980). Today any modern-day SMT solver contains an efficient implementation of the congruence closure algorithm as part of a solver for EUF (Moura and Bjørner, 2008).

2 Satisfiability Modulo Theories (SMT)

Satisfiability modulo theories (SMT) is the satisfiability problem for formulas with respect to some first-order theory, or combinations of first-order theories. A first-order theory extends Boolean logic with specific predicates, functions, and quantifiers (non-logical symbols), thus SMT is a generalization of the Boolean satisfiability problem (SAT). Compared to SAT, SMT allows a richer and more convenient representation of formulas, as SMT can reason about equality, linear arithmetic, bit-vectors, and other first-order theories.

Example 2.1 (First-Order Theories)**.** Examples for first-order theories are:

- *Theory of Equality Logic* introduces equality (=).

$$\neg(x_1 = x_2) \wedge x_1 = 4$$

- *Theory of Linear Arithmetic* adds arithmetic functions, such as + and −, and arithmetic predicates, like = or <.

$$(3y + 2x - 1 = 0) \wedge (0 < x)$$

- *Theory of Bit Vectors* allows the use of binary numbers and binary operators, such as bit-shift ($\gg$), XOR ($\oplus$), or binary negation ($\sim$).

$$(a \gg 2) = c \wedge c \oplus d$$

3 Theory of Equality Logic With Uninterpreted Functions

The *theory of equality logic with uninterpreted functions* (EUF) extends Boolean logic and adds the equality predicate (=). In EUF the equality is a binary predicate, which evaluates to True or False based on the axioms for an equivalence relation:

$$\begin{aligned}
&\forall x.\ x = x && \text{(Reflexivity)}\\
&\forall x.\ \forall y.\ x = y \implies y = x && \text{(Symmetry)}\\
&\forall x.\ \forall y.\ \forall z.\ x = y \wedge y = z \implies x = z && \text{(Transitivity)}
\end{aligned}$$

Compared to Boolean logic variables in EUF are non-binary and are defined over an infinite domain, such as $\mathbb{N}$ or $\mathbb{R}$. Functions in EUF are *uninterpreted* and only maintain the property of *functional congruence*:

Definition 3.1 (Functional Congruence)
For each $n > 0$ and n-ary function f

$$\forall \bar{x}, \bar{y}.\ \bigwedge_{i=1}^{n} x_i = y_i \implies f(\bar{x}) = f(\bar{y})$$

By using uninterpreted functions the details and characteristics of functions are ignored. This can generalize and simplify theorems and proofs. However, some properties of a function can be lost when replacing the function with an uninterpreted function.

Example 3.1 (Uninterpreted Functions & Commutativity)**.** For some fixed x_1,x_2,y_1, and y_2 the following formula is valid as + is commutative.

$$x_1 = y_1 \wedge x_2 = y_2 \implies x_1 + x_2 = y_2 + y_1$$

When generalizing + with an uninterpreted function symbol f, we obtain:

$$x_1 = y_1 \wedge x_2 = y_2 \implies f(x_1, x_2) = f(y_2, y_1)$$

The formula is no longer valid as the uninterpreted function f is not commutative. A constraint can be added to the formula to keep the commutativity.

$$x_1 = y_1 \wedge x_2 = y_2 \implies f(x_1, x_2) = f(y_2, y_1) \vee f(x_1, x_2) = f(y_1, y_2)$$

3.1 Congruence Closure Algorithm

The congruence closure algorithm was originally described by Shostak (1978). With this algorithm, the satisfiability of a conjunction ($\wedge$) of equalities and inequalities with uninterpreted functions can be determined.

Algorithm (Congruence Closure)

Let F be a conjunction of equalities and inequalities with uninterpreted functions.

$$F : (\bigwedge_{i=1}^{m} s_i = t_i) \wedge (\bigwedge_{j=m+1}^{n} s_j \neq t_j)$$

Let S be the set of all equalities and inequalities in F, and let T denote the set of all terms and subterms in F.

A partition of T is constructed as follows:

(1) initially put all terms and subterms in their own congruence class

$$\{\{t\} \mid t \in T\}$$

(2) for all $1 \leq i \leq m$

   a. with $s_i = t_i$ merge the congruence classes of s_i and t_i

   b. propagate the new congruence with symmetry, transitivity, and functional congruence

The constructed partition on T induces a *congruence relation* $\sim$ on T. $\sim$ is a congruence relation, because it satisfies the axioms for an equivalence relation (reflexive, symmetric, and transitive), and also respects functional congruence.

A *congruence closure* is the smallest congruence relation that contains another relation R. $\sim$ is the congruence closure that contains all equalities in F (Shostak, 1978), hence the algorithm is called congruence closure algorithm.

The satisfiability of the formula F can be determined with the following theorem (Shostak, 1978).

Theorem 1
F is satisfiable $\iff \nexists s_i, t_i \in T$ such that $s_i \sim t_i$ and $(s_i \neq t_i) \in S$.

Proof Idea.

$\Longrightarrow$ *(Soundness).* Assume F is satisfiable. There has to exists a model for F. All $s_i, t_i \in T$ with $s_i \sim t_i$ must have the same value in the model, because the model satisfies reflexivity, symmetry, transitivity, and functional congruence. Hence there cannot be an inequality $(s_i \neq t_i) \in S$.

$\Longleftarrow$ *(Completeness).* Assume there are no $s_i, t_i \in T$ such that $s_i \sim t_i$ and $(s_i \neq t_i) \in S$. The goal is to construct a model for F. This is done by constructing a *Herbrand model* for F. A Herbrand model assigns a value to each term in the *term universe* $T_\infty = \bigcup_{i=0}^{\infty} T_i$. T_i is inductively defined as follows:

$$T_0 = T, \text{ and } T_{i+1} = \{f(t_1, \dots, t_r) \mid t_i \in T_i\} \cup T_i$$

where f ranges over all function symbols that appear in F. The term universe contains all possible terms that can be constructed by using all constants and function symbols that appear in F. The height of the term universe is infinite because functions can be arbitrarily nested.

For all terms $t \in T$, the model can directly assign values based on $\sim$. For all remaining terms in the term universe, values are assigned with an inductive construction based on functional congruence.

If for a term $t = f(x_1, \dots, x_r) \in T_{j+1} - T_j$ there exists a functional congruent term in T_j, then the model can assign the value from the functional congruent term to t. Otherwise the model assigns a new value to t. The new value is obtained by evaluating $f(x_1, \dots, x_r)$ with the values of its arguments. For all arguments x_i, values already have been assigned as they are all contained in T_j.

The correctness of this construction is then proven by induction over the height of the term universe (Shostak, 1978).

Example 3.2 (Congruence Closure Algorithm: unsatisfiable formula)**.**

$$f(a, b) = a \land f(f(a, b), b) \neq a$$

- initial partition:

$$\{\{a\},\ \{b\},\ \{f(a, b)\},\ \{f(f(a, b), b)\}\}$$

- impose $f(a, b) = a$:

$$\{\{a,\ f(a, b)\},\ \{b\},\ \{f(f(a, b), b)\}\}$$

- $a \sim f(a,b)$, with functional congruence $f(a,b) \sim f(f(a,b),b)$:

$$\{\{a,\ f(a,b),\ f(f(a,b),b)\},\ \{b\}\}$$

The partition yields $f(f(a,b),b) \sim a$, but the formula contains the inequality $f(f(a,b),b) \neq a$. With Theorem 1 the formula is unsatisfiable.

Example 3.3 (Congruence Closure Algorithm: satisfiable formula)**.**

$$a = b \wedge b = c \wedge g(f(a),b) = g(f(c),a) \wedge f(a) \neq b$$

- initial partition:

$$\{\{a\},\ \{b\},\ \{c\},\ \{f(a)\},\ \{f(c)\},\ \{g(f(a),b)\},\ \{g(f(c),a)\}\}$$

- impose $a = b$:

$$\{\{a,\ b\},\ \{c\},\ \{f(a)\},\ \{f(c)\},\ \{g(f(a),b)\},\ \{g(f(c),a)\}\}$$

- impose $b = c$:

$$\{\{a,\ b,\ c\},\ \{f(a)\},\ \{f(c)\},\ \{g(f(a),b)\},\ \{g(f(c),a)\}\}$$

- $a \sim c$, with functional congruence $f(a) \sim (c)$:

$$\{\{a,\ b,\ c\},\ \{f(a),\ f(c)\},\ \{g(f(a),b)\},\ \{g(f(c),a)\}\}$$

- $f(a) \sim f(c)$ and $b \sim a$, with functional congruence $g(f(a),b) \sim g(f(c),a)$:

$$\{\{a,\ b,\ c\},\ \{f(a),\ f(c)\},\ \{g(f(a),b),\ g(f(c),a)\}\}$$

There are no inequalities that contradict $\sim$. From Theorem 1 follows that the formula is satisfiable.

Satisfiability of Arbitrary EUF-Formulas

The congruence closure algorithm can only decide the satisfiability for conjunctions of equalities and inequalities. The satisfiability for an arbitrary formula in EUF is determined by determining the unsatisfiability of the *disjunctive normal form* (DNF) of the negated formula. A DNF is unsatisfiable if all disjuncts are unsatisfiable. As each disjunct in a DNF is a conjunction, the unsatisfiability of each disjunct is determined with the congruence closure algorithm.

The disadvantage of this approach is that the DNF of a formula can be exponentially larger than the original formula. This results in the decision procedure for an arbitrary formula in EUF having exponential worst-case runtime. That is expected because the decision problem for arbitrary formulas in EUF is NP-complete (Shostak, 1978).

3.2 Efficient Implementation of the Congruence Closure Algorithm With Union-Find

One efficient implementation of the congruence closure algorithm uses the *Union-Find* algorithm and represents a formula as a *Directed-Acyclic-Graph* (DAG). The implementation given here is based on an implementation given by Nelson and Oppen (1980).

3.2.1 Union-Find Algorithm

Union-Find is an algorithm to efficiently maintain a partition on a set of elements. Every partition induces an equivalence relation with equivalence classes. In Union-Find the membership of an element to an equivalence class is only maintained by a reference to a single representative element of the equivalence class. This allows Union-Find to have a near-constant (inverse Ackermann's function) worst-case time complexity (Tarjan, 1975).

Figure 1 shows a graphical representation of a partition (result from example 3.2) in Union-Find.

Two operations are defined to manipulate the partition:

- FIND(v): returns the representative element of the equivalence class, which contains the element v.
- UNION(u,v): combines the two equivalence classes that contain the element u and the element v to one single equivalence class.

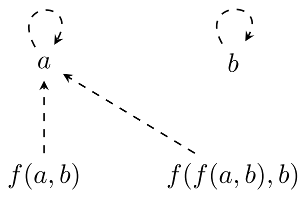


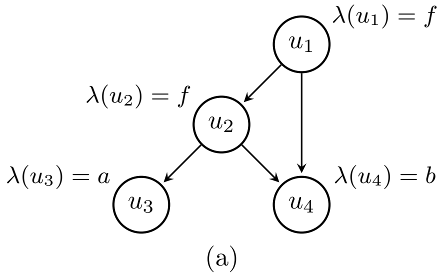
Figure 1: The graphical representation of the partition $\{\{a,\ f(a,b),\ f(f(a,b),b)\},\ \{b\}\}$ in Union-Find is shown here. A dashed edge represents the reference to the representative element of an equivalence class.

3.2.2 Directed-Acyclic-Graph (DAG)

A *Directed-Acyclic-Graph* (DAG) is a directed graph with no directed cycle. A DAG $G = (V, E)$ consists of a set of nodes V, and a set of edges E. Every node $v \in V$ has a label $\lambda(v)$. The number of outgoing edges from v is denoted by $\delta(v)$. The i-th successor of v is denoted by $v[i] = u \in V$, for $1 \leq i \leq \delta(v)$.

Every term and subterm in a formula F is represented by a node in the DAG. The label for each node corresponds to the root constant or function symbol of the associated term or subterm. An edge is inserted from every function symbol to all its arguments.

The construction of a DAG for a formula only considers the relationship between function symbols and arguments. Equalities and inequalities in the formula are not considered for the construction of the DAG. Figure 2 shows two DAGs that each represent a formula.



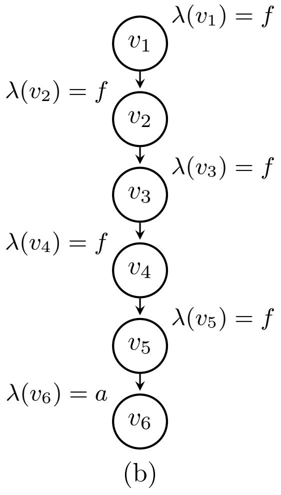


Figure 2: (a) DAG representing the formula $f(a,b) = a \wedge f(f(a,b),b) \neq a$, where e.g. u_2 represents the subtrem $f(a,b)$. (b) DAG representing the formula $f^5(a) = a \wedge f^3(a) = a \wedge f(a) \neq a$.

3.2.3 Implementation

The Union-Find algorithm is sufficient to maintain and propagate an equivalence relation, but the partition constructed by the congruence closure algorithm induces a congruence relation. The induced congruence relation also respects functional congruence. To also maintain and propagate functional congruence, the Union operation is extended by Merge.

An auxiliary function Congruent is defined to determine if two nodes represent congruent subterms.

```
function CONGRUENT($u$,$v$)
    if $\lambda(u) \neq \lambda(v) \vee \delta(u) \neq \delta(v)$ then return FALSE
    for $1 \leq i \leq \delta(u)$ do
        if FIND($u[i]$) $\neq$ FIND($v[i]$) then return FALSE
    return TRUE
```

```
function MERGE($u$,$v$)
    if FIND($u$) $\neq$ FIND($v$) then
        let $P_u$ and $P_v$ be sets of all congruent predecessors for $u$ and $v$
        UNION($u, v$)
        for $(x, y) \in P_u \times P_v$ do
            if FIND($x$) $\neq$ FIND($y$) $\wedge$ CONGRUENT($x, y$) then
                MERGE($x, y$)
```

MERGE uses the UNION operation and additionally propagates functional congruence. Representing a formula as a DAG allows an efficient propagation of functional congruence in MERGE. Functional congruence is propagated from function arguments to function symbols. As function arguments are successors of function symbols in the DAG, functional congruence is propagated to congruent predecessors. Every congruence class has an associated set that contains all nodes that are predecessors to any node in the congruence class. Let P_u denote this set for the congruence class that contains the node u. With every UNION operation the corresponding predecessor sets are merged.

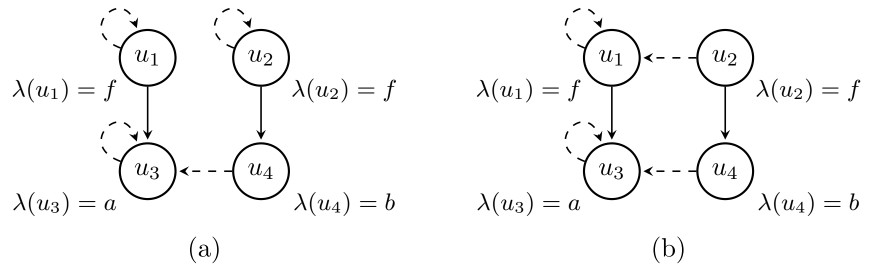


Figure 3: A simple example to demonstrate the propagation of functional congruence. A dashed edge again represents the reference to the representative element of an equivalence class. In (a) a DAG representing the formula $a = b \wedge f(a) = f(b)$ is shown. The equation $a = b$ was just imposed. (b) shows the same DAG, but the new congruence was propagated via functional congruence, resulting in the partition $\{\{a, b\}, \{f(a), f(b)\}\}$.

In Figure 3 the propagation of a new congruence via functional congruence is shown in a simple example. The equality $a = b$ is imposed in Figure 3 (a), by merging the congruence classes of a and b. This is done by changing the reference of u_3 from itself to u_4, hence u_3 and u_4 are now in the same congruence class. The predecessor set for u_3 is $P_{u_3} = \{u_1\}$, and for u_4 it is $P_{u_4} = \{u_2\}$. In this case the only pair of predecessors from both sets is (u_2, u_1). Because u_2 and u_1 are functionally congruent, the congruence is propagated to u_2 and u_1 by merging their congruence classes in Figure 3 (b).

With CONGRUENT and MERGE the decision procedure based on the congruence closure algorithm is implemented as follows:

Let F be a conjunction of equalities and inequalities with uninterpreted functions.

$$F : (\bigwedge_{i=1}^{m} s_i = t_i) \wedge (\bigwedge_{j=m+1}^{n} s_j \neq t_j)$$

Let $\tau(t)$ be the node in the DAG for F, that corresponds to the term or substerm t in F.

```
1: for $1 \leq i \leq m$ do
2:     MERGE($\tau(s_i), \tau(t_i)$)
3: for $m + 1 \leq j \leq n$ do
4:     if FIND($\tau(s_j)$) = FIND($\tau(t_j)$) then return UNSATISFIABLE
5: return SATISFIABLE
```

Nelson and Oppen (1980) have shown that the congruence relation constructed in this implementation is the congruence closure that contains all equalities in F. From this follows that for any subterms s and t in F

$$\textsc{Find}(\tau(s)) = \textsc{Find}(\tau(t)) \iff s \sim t$$

holds. Where $\sim$ is the congruence closure, that contains all equalities in F. With this, the correctness of the implementation can be verified.

If the procedure returns UNSATISFIABLE then for some $m + 1 \leq j \leq n$ there exists s_j and t_j, with $\textsc{Find}(\tau(s_j)) = \textsc{Find}(\tau(t_j))$. s_j and t_j are in the same congruence class. As F contains the inequality $s_j \neq t_j$, from Theorem 1 follows that F is unsatisfiable.

When the procedure returns SATISFIABLE then there is no $m+1 \leq j \leq n$ such that s_j and t_j are in the same congruence class. From Theorem 1 follows that F is satisfiable.

3.2.4 Complexity

Let $G = (V, E)$ be the DAG, which represents the formula F. Let $n = |V|$ be the number of nodes in G, and let $m = |E|$ be the number of edges in G.

With n nodes in G there can be at most $n - 1$ Merge operations until all nodes are in the same congruence class. Hence there can be at most be $n - 1 = \mathcal{O}(n)$ recursive Merge operations in total. From this it can be proven that for any sequence of Merge operations the number of Congruent calls is bounded by $\mathcal{O}(mn)$, and the number of Find calls from Congruent (line 4) is bounded by $\mathcal{O}(m^2)$ (Nelson and Oppen, 1980).

For $\mathcal{O}(n)$ Merge operations there are $\mathcal{O}(n)$ Find calls from line 2, plus $\mathcal{O}(mn)$ Find calls from line 6, and $\mathcal{O}(m^2)$ Find calls from Congruent. In total there are $\mathcal{O}(m^2)$ Find calls, which take $\mathcal{O}(m^2)$ time. Each Union operation takes constant time (Tarjan, 1975), resulting in $\mathcal{O}(n)$ time for $\mathcal{O}(n)$ Merge operations. The time cost for maintaining the predecessor sets is $\mathcal{O}(n^2)$ (Nelson and Oppen, 1980).

All in all, this results in the worst-case time complexity of $\mathcal{O}(m^2)$ for this implementation.

4 Conclusion

This paper gave an introduction to SMT and EUF. The presented implementation of the congruence closure algorithm has a worst-case time complexity of $\mathcal{O}(m^2)$. Downey et al. (1980) have implemented a faster version ($\mathcal{O}(m \log^2 m)$) of the congruence closure algorithm, by using hash tables instead of DAGs.

An efficient solver for EUF with an implementation of the congruence closure algorithm is part of any modern SMT solver, such as Z3 (Moura and Bjørner, 2008) or CVC4 (Barrett et al., 2011).

With SMT solvers, tools for program verification, such as OpenJML (Cok, 2011), have been created. OpenJML can verify a program by simply annotating code with pre- and post-conditions.

Besides tools for program verification, SMT solvers have numerous other applications, such as testcase generation, static analysis, and hardware verification. SMT solvers have also been integrated into interactive theorem provers for higher-order logic, such as Isabelle/HOL (Nipkow et al., 2002).

In almost all of the applications, SMT solver must be able to efficiently reason about equality.

References

Ackermann, W. (1954). *Solvable cases of the decision problem.* Studies in logic and the foundations of mathematics. North-Holland Pub. Co.

Barrett, Clark, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli (July 2011). "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11).* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Snowbird, Utah. Springer, pp. 171–177. URL: `http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf`.

Cok, David R. (2011). "OpenJML: JML for Java 7 by Extending OpenJDK". In: *Proceedings of the Third International Conference on NASA Formal Methods.* NFM'11. Pasadena, CA: Springer-Verlag, pp. 472–479. ISBN: 9783642203978.

Downey, Peter J., Ravi Sethi, and Robert Endre Tarjan (Oct. 1980). "Variations on the Common Subexpression Problem". In: *J. ACM* 27.4, pp. 758–771. ISSN: 0004-5411. DOI: `10.1145/322217.322228`. URL: `https://doi.org/10.1145/322217.322228`.

Moura, Leonardo De and Nikolaj Bjørner (2008). “Z3: an efficient SMT solver”. In: *TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 337–340.

Nelson, Greg and Derek C. Oppen (Apr. 1980). “Fast Decision Procedures Based on Congruence Closure”. In: *J. ACM* 27.2, pp. 356–364. ISSN: 0004-5411. DOI: `10.1145/322186.322198`. URL: `https://doi.org/10.1145/322186.322198`.

Nipkow, Tobias, Markus Wenzel, and Lawrence C. Paulson (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag. ISBN: 3540433767.

Shostak, Robert E. (July 1978). “An Algorithm for Reasoning about Equality”. In: *Commun. ACM* 21.7, pp. 583–585. ISSN: 0001-0782. DOI: `10.1145/359545.359570`. URL: `https://doi.org/10.1145/359545.359570`.

Tarjan, Robert Endre (Apr. 1975). “Efficiency of a Good But Not Linear Set Union Algorithm”. In: *J. ACM* 22.2, pp. 215–225. ISSN: 0004-5411. DOI: `10.1145/321879.321884`. URL: `https://doi.org/10.1145/321879.321884`.