Yet Another Decision Procedure for Equality Logic

Orly Meir[1] and Ofer Strichman[2]

[1] Computer science department, Technion , Israel. orlym@cs.technion.ac.il
[2] Information Systems Engineering, Technion, Israel. ofers@ie.technion.ac.il

Abstract. We introduce a new decision procedure for Equality Logic. The procedure improves on Bryant and Velev's SPARSE method [4] from CAV'00, in which each equality predicate is encoded with a Boolean variable, and then a set of transitivity constraints are added to compensate for the loss of transitivity of equality. We suggest the Reduced Transitivity Constraints (RTC) algorithm, that unlike the SPARSE method, considers the *polarity* of each equality predicate, i.e. whether it is an equality or disequality when the given equality formula φ^{E} is in Negation Normal Form (NNF). Given this information, we build the Equality Graph corresponding to φ^{E} with two types of edges, one for each polarity. We then define the notion of *Contradictory Cycles* to be cycles in that graph that the variables corresponding to their edges cannot be simultaneously satisfied due to transitivity of equality. We prove that it is sufficient to add transitivity constraints that only constrain Contradictory Cycles, which results in only a small subset of the constraints added by the SPARSE method. The formulas we generate are smaller and define a larger solution set, hence are expected to be easier to solve, as indeed our experiments show. Our new decision procedure is now implemented in the UCLID verification system.

1 Introduction

Equality Logic with Uninterpreted Functions is a major decidable theory used in verification of infinite-state systems. Well-formed expressions in this logic are Boolean combinations of Equality predicates, where the equalities are defined between *term-variables* (variables with some infinite domain) and Uninterpreted Functions. The Uninterpreted Functions can be reduced to equalities via either Ackermann's [1] or Bryant et al.'s reduction [2] (from now on we will say Bryant's reduction), hence the underling theory that is left to solve is that of Equality Logic.

There are many examples of using Equality Logic and Uninterpreted Functions in the literature. Proving equivalence of circuits after *custom-design* or *retiming* (a process in which the layout of the circuit is changed in order to improve computation speed) is a prominent example [3, 6]. *Translation Validation* [15], a process in which the input and output of a compiler are proven to be semantically equivalent is another example of using this logic. Almost all theorem

provers that we are aware of support this logic, either explicitly or as part of their support of more expressive logics.

Related work The importance of this logic led to several suggestions for decision procedures in the last few years [17, 9, 13, 2, 4, 16], almost all of which are surveyed in detail in the full version of this article [11]. Due to space limitations here we will only mention the most relevant prior work by Bryant and Velev [4], called the SPARSE method. In the SPARSE method, each equality predicate is replaced with a new Boolean variable, which results in a purely propositional formula that we denote by $\mathcal{B}$ ($\mathcal{B}$ for $\mathcal{B}$oolean). Transitivity constraints over these Boolean variables are then conjoined with $\mathcal{B}$, to recover the transitivity of equality that is lost in the Boolean encoding. So, for example, given the equality formula: $v_1 = v_2 \wedge v_2 = v_3 \wedge \neg(v_1 = v_3)$ the SPARSE method reduces it to the Boolean formula $\mathcal{B} = e_{1,2} \wedge e_{2,3} \wedge \neg e_{1,3}$ and conjoins $\mathcal{B}$ with the transitivity constraints $e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}, e_{1,2} \wedge e_{1,3} \rightarrow e_{2,3}$ and $e_{1,3} \wedge e_{2,3} \rightarrow e_{1,2}$. The conjoined formula is satisfiable if and only if the original formula is satisfiable.

In order to decide which constraints are needed, following the SPARSE method one needs to build a graph in which each equality predicate is an edge and each variable is a vertex. With a simple analysis of this graph the necessary constraints are derived. This is where our method is different from the SPARSE method: unlike the graph considered by the SPARSE method, the graph we build has two kinds of edges: one for equalities and one for disequalities, assuming the Equality formula is given to us in Negation Normal Form (NNF) . Given this extra information, about the *polarity* of each equality predicate, we are able to find a small subset of the constraints that are generated by the SPARSE method, that are still sufficient to preserve correctness. This results in a much simpler formula that is easier for SAT to solve, at least in theory.

We base our procedure on a theorem that we state and prove in Section 4. The theorem refers to what we call *Simple Contradictory Cycles*, which are simple cycles that have exactly one disequality edge. In such cycles, the theorem claims, we need to prevent an assignment that assigns FALSE to the disequality edge and TRUE to the rest. And, most importantly, these are the *only kind* of constraints necessary. The proof of this theorem relies on a certain property of NNF formulas called *monotonicity with respect to satisfiability* that we present in Section 3. In Section 5 we show an algorithm that computes in polynomial time a set of constraints that satisfy the requirements of our theorem. In Section 6 we present experimental results. Our new procedure is now embedded in the UCLID [5] verification tool and is hence available for usage. In Section 7 we conclude the paper and present directions for future research.

2 Reducing Equality Logic to Propositional Logic

We consider the problem of deciding whether an Equality Logic formula φ^{E} is satisfiable. The following framework is used by both [4] and the current work to reduce this decision problem to the problem of deciding a propositional formula:

1. Let E denote the set of equality predicates appearing in φ^{E}. Derive a Boolean formula $\mathcal{B}$ by replacing each equality predicate $(v_i = v_j) \in E$ with a new Boolean variable $e_{i,j}$. Encode disequality predicates with negations, e.g., encode $i \neq j$ with $\neg e_{i,j}$.
2. Recover the lost transitivity of equality by conjoining $\mathcal{B}$ with explicit *transitivity constraints* jointly denoted by $\mathcal{T}$ ($\mathcal{T}$ for $\mathcal{T}$ransitivity). $\mathcal{T}$ is a formula over $\mathcal{B}$'s variables and, possibly, auxiliary variables.

The Boolean formula $\mathcal{B} \wedge \mathcal{T}$ should be satisfiable if and only if φ^{E} is satisfiable. Further, we should be able to construct a satisfying assignment to φ^{E} from an assignment to the $e_{i,j}$ variables. A straightforward method to build $\mathcal{T}$ in a way that will satisfy these requirements is to add a constraint for every cyclic comparison between variables, which disallow TRUE assignment to exactly $k-1$ predicates in a k-long simple cycle.

In [4] three different methods to build $\mathcal{T}$ are suggested, all of which are better than this straightforward approach, and are described in some detail also in [11]. We need to define *Non-Polar Equality Graph* in order to explain the SPARSE method, which is both theoretically and empirically the best of the three:

Definition 1 (Non-Polar Equality Graph). *Given an Equality Logic formula φ^E, the* Non-Polar Equality Graph *corresponding to φ^E is an undirected graph (V, E) where each node $v \in V$ corresponds to a variable in φ^E, and each edge $e \in E$ corresponds to an equality or disequality predicate in φ^E.*

The graph is called *non-polar* to distinguish it from the graph that we will use later, in which there is a distinction between edges that represent equalities and those that represent disequalities. We will simply say Equality Graph from now on in both cases, where the meaning is clear from the context.

The SPARSE method is based on a theorem, proven in [4], stating that it is sufficient to add transitivity constraints only to *chord-free* cycles (a chord is an edge between two non-adjacent nodes). A *chordal* graph, also known as *triangulated graph*, is a graph in which every cycle of size four or more has a chord. In such a graph only triangles are chord-free cycles. Every graph can be made chordal by adding auxiliary edges in linear time. The SPARSE method begins by making the graph chordal, while referring to each added edge as a new auxiliary $e_{i,j}$ variable. It then adds three transitivity constraints for each triangle. We will denote the transitivity constraints generated by the SPARSE method with $\mathcal{T}^S$.

Example 1. Figure 1 presents an Equality Graph before and after making it chordal. The added edge $e_{0,6}$ corresponds to a new auxiliary variable $e_{0,6}$ that appears in $\mathcal{T}^S$ but not in $\mathcal{B}$. After making the graph chordal, it contains 4 triangles and hence there are 12 constraints in $\mathcal{T}^S$. For example, for the triangle (v_1, v_2, v_3) the constraints are: $e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$, $e_{1,3} \wedge e_{2,3} \rightarrow e_{1,2}$ and $e_{1,2} \wedge e_{1,3} \rightarrow e_{2,3}$.

□

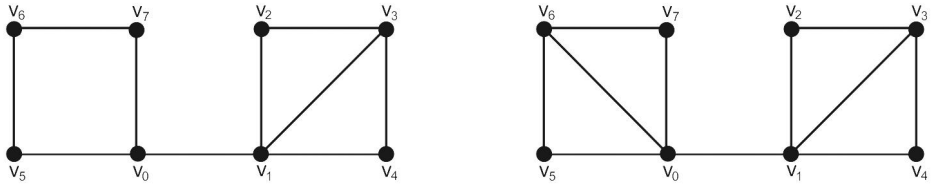


Fig. 1. A non-chordal Equality Graph (left) and its chordal version.

We will show an algorithm for constructing a Boolean formula $\mathcal{T}^R$ (the superscript R is for Reduced) which is, similarly to $\mathcal{T}^S$, a conjunction of transitivity constraints, but contains only a subset of the constraints in $\mathcal{T}^S$. $\mathcal{T}^R$ is not logically equivalent to $\mathcal{T}^S$; it has a larger solution set. Yet it maintains the property that $\mathcal{B} \wedge \mathcal{T}^R$ is satisfiable if and only if φ^{E} is satisfiable, as we will later prove. This means that $\mathcal{T}^R$ not only has a subset of the constraints of $\mathcal{T}^S$, but it also defines a less constrained search space (has more solutions than $\mathcal{T}^S$). Together these two properties are likely to make the SAT instance easier to solve. Since the complexity of both our algorithm and the SPARSE method are similar, we can claim dominance over the SPARSE method, although practically, due to the unpredictability of SAT, such claims are never 100% true.

3 Basic Definitions

We will assume that our Equality formula φ^{E} is given in Negation Normal Form (NNF), which means that negations are only applied to atoms, or equality predicates in our case. Every formula can be transformed to this form in linear time in the size of the formula. Given an NNF formula, we denote by $E_=$ the set of (unnegated) equality predicates, and by $E_{\neq}$ the set of disequalities (negated) equality predicates. Our decision procedure, as the SPARSE method, relies on graph-theoretic concepts. We will also use Equality Graphs, but redefine them so they refer to polarity information. Specifically, each of the sets $E_=$, $E_{\neq}$ corresponds in this graph to a different set of edges. We overload these notations so they refer both to the set of predicates and to the edges that represent them in the Equality Graph.

Definition 2 (Equality Graph). *Given an Equality Logic formula φ^E, the* Equality Graph *corresponding to φ^E, denoted by $G^E(\varphi^E)$, is an undirected graph $(V, E_=, E_{\neq})$ where each node $v \in V$ corresponds to a variable in φ^E, and each edge in $E_=$ and $E_{\neq}$ corresponds to an equality or disequality from the respective equality predicates sets $E_=$ and $E_{\neq}$. By convention $E_=$ edges are dashed and $E_{\neq}$ edges are solid.*

As before, every edge in the Equality Graph corresponds to a variable $e_{i,j} \in \mathcal{B}$. It follows that when we refer to an assignment of an edge, we actually refer to an assignment to its corresponding variable. Also, we will simply write G^{E} to denote an Equality Graph if we do not refer to a specific formula.

Example 2. In Figure 2 we show an Equality Graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ corresponding to the non-polar version shown in Figure 1, assuming some Equality Formula φ^{E} for which $E_{=} : \{(v_5 = v_6), (v_6 = v_7), (v_7 = v_0), (v_1 = v_2), (v_2 = v_3), (v_3 = v_4)\}$ and $E_{\neq} : \{(v_0 \neq v_1), (v_5 \neq v_1), (v_5 \neq v_4), (v_6 \neq v_4)\}$ □

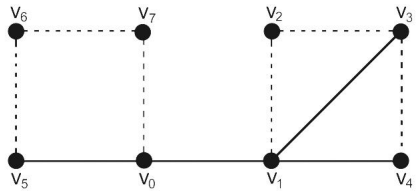


Fig. 2. The Equality Graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ corresponding to the non-polar version of the same graph shown in Figure 1.

We now define two types of paths in Equality Graphs.

Definition 3 (Equality Path). *An* Equality Path *in an Equality Graph G^E is a path made of $E_{=}$ (dashed) edges. We denote by $x =^* y$ the fact that x has an Equality Path to y in G^E, where $x, y \in V$.*

Definition 4 (Disequality Path). *A* Disequality Path *in an Equality Graph G^E is a path made of $E_{=}$ (dashed) edges and a single $E_{\neq}$ (solid) edge. We denote by $x \neq^* y$ the fact that x has a Disequality Path to y in G^E, where $x, y \in V$.*

Similarly, we will use a *Simple Equality Path* and a *Simple Disequality Path* when the path is required to be loop-free. In Figure 2 it holds, for example, that $v_0 =^* v_6$ due to the simple path v_0, v_7, v_6; $v_0 \neq^* v_6$ due to the simple path v_0, v_5, v_6; and $v_7 \neq^* v_6$ due to the simple path v_7, v_0, v_5, v_6.

Intuitively, Equality Path between two variables implies that it might be required to assign both variables an equal value in order to satisfy the formula. A Disequality Path between two variables implies the opposite: it might be required to assign different values to these variables in order to satisfy the formula. For this reason the case in which both $x =^* y$ and $x \neq^* y$ hold in $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$, requires special attention. We say that the graph, in this case, contains a *Contradictory Cycle*.

Definition 5 (Contradictory Cycle). *A Contradictory Cycle in an Equality Graph is a cycle with exactly one disequality (solid) edge.*

Several characteristics of Contradictory Cycles are: 1) For every pair of nodes x, y in a Contradictory Cycle, it holds that $x =^* y$ and $x \neq^* y$. 2) For every

Contradictory Cycle C, either C is *simple* or a subset of its edges forms a Simple Contradictory Cycle. We will therefore refer only to simple Contradictory Cycles from now on. 3) It is impossible to satisfy simultaneously all the predicates that correspond to edges of a Contradictory Cycle. Further, this is the only type of subgraph with this property.

The reason that we need polarity information is that it allows us to use the following property of NNF formulas.

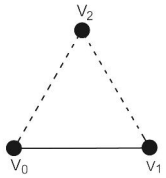
Theorem 1 (Monotonicity of NNF). *Let ϕ be an NNF formula and α be an assignment such that $\alpha \models \phi$. Let the* positive set *S of α be the positive literals in ϕ assigned* TRUE *and the negative literals in ϕ assigned* FALSE*. Every assignment α' with a positive set S' such that $S \subseteq S'$ satisfies ϕ as well.*

The same theorem was used, for example, in [14]. As an aside, when this theorem is applied to CNF formulas, which are a special case of NNF, it is exactly the same as the *pure literal rule.*

4 Main Theorem

The key idea that is formulated by Theorem 2 below and later exploited by our algorithm can first be demonstrated by a simple example.

Example 3. For the Equality Graph below (left), the SPARSE method generates $\mathcal{T}^S$ with three transitivity constrains (recall that it generates three constraints for each triangle in the graph, regardless of the edges' polarity). We claim, however, that the single transitivity constraint $\mathcal{T}^R = (e_{0,2} \wedge e_{1,2} \rightarrow e_{0,1})$ is sufficient.



	α^R	α^S
$e_{0,1}$	TRUE	TRUE
$e_{1,2}$	TRUE	TRUE
$e_{0,2}$	FALSE	TRUE

To justify this claim, it is sufficient to show that for every assignment α^R that satisfies $\mathcal{B} \wedge \mathcal{T}^R$, there exists an assignment α^S that satisfies $\mathcal{B} \wedge \mathcal{T}^S$. Since this, in turn, implies that φ^{E} is satisfiable as well, we get that φ^{E} is satisfiable if and only if $\mathcal{B} \wedge \mathcal{T}^R$ is satisfiable. Note that the 'only if' direction is implied by the fact that we use a subset of the constraints defined by $\mathcal{T}^S$.

We are able to construct such an assignment α^S because of the monotonicity of NNF (recall that the polarity of the edges in the Equality Graph are according to their polarity in the NNF representation of φ^{E}). There are only two satisfying assignments to $\mathcal{T}^R$ that do not satisfy $\mathcal{T}^S$. One of these assignments is shown in the α^R column in the table to the right of the drawing. The second column shows a corresponding assignment α^S, which clearly satisfies $\mathcal{T}^S$. But we still need to prove that every formula $\mathcal{B}$ that corresponds to the above graph, is still

satisfied by α^S if it was satisfied by α^R. For example, for $\mathcal{B} = (\neg e_{0,1} \vee e_{1,2} \vee e_{0,2})$, both $\alpha^R \models \mathcal{B} \wedge \mathcal{T}^R$ and $\alpha^S \models \mathcal{B} \wedge \mathcal{T}^S$ hold. Intuitively, this is guaranteed to be true because α^S is derived from α^R by flipping an assignment of a positive (un-negated) predicate ($e_{0,2}$) from FALSE to TRUE. We can equivalently flip an assignment to a negated predicate ($e_{0,1}$ in this case) from TRUE to FALSE.

A formalization of this argument requires a reference to the monotonicity of NNF (Theorem 1): Let S and S' denote the positive sets of α^R and α^S respectively. Then in this case $S = \{e_{1,2}\}$ and $S' = \{e_{1,2}, e_{0,2}\}$. Thus $S \subset S'$ and hence, according to Theorem 1, $\alpha^R \models \mathcal{B} \rightarrow \alpha^S \models \mathcal{B}$. □

We need several definitions in order to generalize this example into a theorem.

Definition 6 (A constrained Contradictory Cycle). *Let $C = (e_s, e_1, \ldots, e_n)$ be a Contradictory Cycle where e_s is the solid edge. Let ψ be a formula over the Boolean variables in $\mathcal{B}$ that encodes the edges of C. C is said to be* constrained *in ψ if the assignment $(e_s, e_1, \ldots, e_n) \leftarrow (F, T, \ldots, T)$ contradicts ψ.*

Recall that we denote by $\mathcal{T}^S$ the formula that imposes transitivity constraints in the SPARSE method, as defined in [4] and described in Section 2. Further, recall that the SPARSE method works with chordal graphs, and therefore all constraints are over triangles. Our method also makes the graph chordal, and the constraints that we generate are also over triangles, although we will not use this fact in Theorem 2, in order to make it more general.

Definition 7 (A Reduced Transitivity Constraints function $\mathcal{T}^R$). *A Reduced Transitivity Constraints (RTC) function $\mathcal{T}^R$ is a conjunction of transitivity constraints that maintains these two requirements:*

R1 For every assignment α^S, $\alpha^S \models \mathcal{T}^S \rightarrow \alpha^S \models \mathcal{T}^R$ (the solution set of $\mathcal{T}^R$ includes all the solutions to $\mathcal{T}^S$).
R2 $\mathcal{T}^R$ constrains all the simple Contradictory Cycles in the Equality Graph G^E.

R1 implies that $\mathcal{T}^R$ is less constrained than $\mathcal{T}^S$. Consider, for example, a chordal Equality graph in which all edges are solid (disequalities): in such a graph there are no Contradictory Cycles and hence no constraints are required. In this case $\mathcal{T}^R =$ TRUE, while $\mathcal{T}^S$ includes three transitivity constraints for each triangle.

Theorem 2 (Main). *An Equality formula φ^E is satisfiable if and only if $\mathcal{B} \wedge \mathcal{T}^R$ is satisfiable.*

Due to R1, the proof of the 'only if' direction ($\Rightarrow$) is trivial. To prove the other direction we show in [11] an algorithm for reconstructing an assignment α^S that satisfies $\mathcal{T}^S$ from a given assignment α^R that only satisfies $\mathcal{T}^R$.

5 The Reduced Transitivity Constraints Algorithm

We now introduce an algorithm that generates a formula $\mathcal{T}^R$, which satisfies the two requirements R1 and R2 that were introduced in the previous section.

The RTC algorithm processes *Biconnected Components* (BCC) [7] in the given Equality Graph.

Definition 8 (Maximal Biconnected Component). *A* Biconnected Component *of an undirected graph is a maximal set of edges such that any two edges in the set lie on a common simple cycle.*

We can focus on BCCs because we only need to constrain cycles, and in particular Contradictory Cycles. Each BCC that we consider contains a solid edge e_s and all the Contradictory Cycles that it is part of. In line 5 of RTC we make the BCC chordal. Since making the graph chordal involves adding edges, prior to this step, in line 4, we add solid edges from G^{E} that can serve as chords. After the graph is chordal we call GENERATE-CONSTRAINTS, which generates and adds to some local cache all the necessary constraints for constraining all the Contradictory Cycles in this BCC with respect to e_s. When GENERATE-CONSTRAINTS returns, all the constraints that are in the local cache are added to some global cache. The conjunction of the constraints in the global cache is what RTC returns as $\mathcal{T}^R$.

RTC (Equality Graph $G^{\mathrm{E}}(V, E_{=}, E_{\neq})$)

1: global-cache $= \emptyset$
2: **for all** $e_s \in E_{\neq}$ **do**
3: Find $B(e_s)$ = maximal BCC in G^{E} made of e_s and $E_{=}$ edges;
4: Add to $B(e_s)$ all edges from $E_{\neq}$ that connect vertices in $B(e_s)$;
5: Make the graph $B(e_s)$ chordal; ▷ (The chords can be either solid or dashed)
6: GENERATE-CONSTRAINTS ($B(e_s)$, e_s);
7: global-cache = global-cache ∪ local-cache;
8: $\mathcal{T}^R$ = conjunction of all constraints in the global cache;
9: return $\mathcal{T}^R$;

GENERATE-CONSTRAINTS (Equality Graph $G^{\mathrm{E}}(V, E_{=}, E_{\neq})$, edge $e \in G^{\mathrm{E}}$)

1: **for all** triangles $(e_1, e_2, e) \in G^{\mathrm{E}}$ such that

- $e_1 \wedge e_2 \rightarrow e$ is not in the local cache
- $source(e) \neq e_1 \wedge source(e) \neq e_2$

do
2: $source(e_1) = source(e_2) = e$;
3: Add $e_1 \wedge e_2 \rightarrow e$ to the local cache;
4: GENERATE-CONSTRAINTS (G^{E}, e_1); ▷ expand e_1
5: GENERATE-CONSTRAINTS (G^{E}, e_2); ▷ expand e_2

GENERATE-CONSTRAINTS iterates over all triangles that include the solid edge $e_s \in E_{\neq}$ with which it is called first. It then attempts to implicitly *expand* each such triangle to larger cycles that include e_s. This expansion is done in the recursive calls of GENERATE-CONSTRAINTS. Given the edge e, which is part of a

cycle, it tries to make the cycle larger by replacing e with two edges that 'lean' on this edge, i.e. two edges e_1, e_2 that together with e form a triangle. This is why we refer to this operation as expansion. There has to be an indication in which 'direction' we can expand the cycle, because otherwise when considering e.g. e_1, we would replace it with e and e_2 and enter an infinite loop. For this reason we maintain the *source* of each edge. The *source* of an edge is the edge that it replaces. In the example above when replacing e with e_1, e_2, $source(e_1) = source(e_2) = e$. So in the next recursive call, where e_1 is the considered edge, due to the second condition in line 1 we *do not* expand it through the triangle (e, e_1, e_2).

Each time we replace the given edge e by two other edges e_1, e_2, we also add a transitivity constraint $e_1 \wedge e_2 \rightarrow e$ to the local cache. Informally, one may see this constraint as enforcing the transitivity of the expanded cycle, by using the transitivity enforcement of the smaller cycle. In other words, this constraint guarantees that if the expanded cycle violates transitivity, then so does the smaller one. Repeating this argument all the way down to triangles, gives us an inductive proof that transitivity is enforced for all cycles. A formal proof of correctness of RTC appears in [11].

Example 4. Figure 3 (left) shows the result of the iterative application of line 3 in RTC for each solid edge in the graph shown in Figure 2. By definition, after this step each BCC contains exactly one solid edge. Figure 3 (right) demonstrates the application of lines 4 and 5 in RTC: in line 4 we add $e_{1,3}$, and in line 5 we add $e_{0,6}$, the only additional chords necessary in order to make all BCCs chordal. The progress of GENERATE-CONSTRAINTS for this example is shown in Table 1.

Iteration	Component	edge to expand	source of edge	Triangle	added constraint
1	a	$e_{0,5}$	-	$(e_{0,5}, e_{5,6}, e_{0,6})$	$e_{0,6} \wedge e_{5,6} \rightarrow e_{0,5}$
2	a	$e_{0,6}$	$e_{0,5}$	$(e_{0,6}, e_{6,7}, e_{0,7})$	$e_{6,7} \wedge e_{0,7} \rightarrow e_{0,6}$
3	b	$e_{1,4}$	-	$(e_{1,4}, e_{3,4}, e_{1,3})$	$e_{1,3} \wedge e_{3,4} \rightarrow e_{1,4}$
4	b	$e_{1,3}$	$e_{1,4}$	$(e_{1,3}, e_{2,3}, e_{1,2})$	$e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$
5	c	$e_{1,3}$	-	$(e_{1,3}, e_{2,3}, e_{1,2})$	$e_{1,2} \wedge e_{2,3} \rightarrow e_{1,3}$

Table 1. The progress of GENERATE-CONSTRAINTS when given the graph of Figure 3 (not including steps where the function returns because the triangle contains the source of the expanded edge). In line 5 the constraint is already in the local cache, and hence not added again.

□

5.1 Complexity of RTC and improvements

Lines 3-5 in RTC can all be done in time linear in the size of the graph (including the process of finding BCCs [7]). The number of iterations of the main loop in RTC

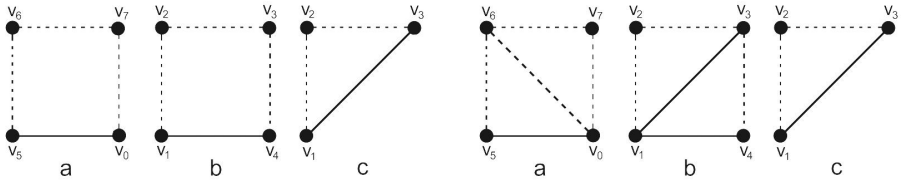


Fig. 3. The BCCs found in line 3 (left) and after lines 4 and 5 in RTC (right).

is bounded by the number of solid edges in the graph. GENERATE-CONSTRAINTS, in each iteration of its main loop, either adds a new constraint or moves to the next iteration without further recursive calls. Since the number of possible constraints is bounded by three times the number of triangles in the graph, the number of recursive calls in GENERATE-CONSTRAINTS is bounded accordingly.

Improvements: To reduce complexity, we only use a global cache, which reduces the number of added constraints and the overall complexity, since we never generate the same constraint twice and stop the recursion calls earlier if we encounter a constraint that was generated in a previous BCC. The correctness proof for this improvement is rather complicated and appears in the full version of this paper [11].

We are also currently examining an algorithm that is more strict than RTC in adding constraints: RTC constrains *all* contradictory cycles, not only the simple ones, which we know is sufficient according to Theorem 2. This algorithm checks whether the cycle that is currently expanded is simple or not. This leads to certain complications that require to continue exploring the graph even when encountering a constraint that is already in the cache. This, in turn, can lead to a worst-case exponential time algorithm, that indeed removes many redundant constraints but is rarely better than RTC according to our experiments, when considering the total solving time. Whether there exists an equivalent algorithm that works in polynomial time is an open question.

6 Experimental Results

UCLID **benchmarks.** Our decision procedure is now integrated in the UCLID [5] verification system. UCLID is a tool for analyzing the correctness of models of hardware and software systems. It can be used to model and verify infinite-state systems with variables of integer, Boolean, function, and array types. The applications of UCLID explored to date include microprocessor design verification, analyzing software for security exploits, verification of a compiler through Translation Validation and verifying distributed algorithms.

UCLID reports to RTC the edges of the Equality Graph corresponding to the verified formula including their polarity, and RTC returns a list of transitivity

constraints. The Boolean encoding (the generation of $\mathcal{B}$), the elimination of Uninterpreted Functions, various simplifications and the application of the Positive Equality algorithm [2], are all applied by UCLID as before. The comparison to the SPARSE method of [4], which is also implemented in this tool and fed exactly the same formula, is therefore fair.

We used all the relevant UCLID benchmarks that we are aware of (all of which happen to be unsatisfiable). We compared RTC and the SPARSE method using the two different reduction methods of Uninterpreted Functions: Ackermann's reduction [1] and Bryant's reduction [2]. This might cause a bias in our results not in our favor: the reduction of Uninterpreted Functions to Equality Logic results in Equality Graphs with specific characteristics. In [11], we explain the difference between the two reductions and why this influences our results. Here we will only say that when Bryant's reduction is used, all edges corresponding to comparisons between arguments of functions are 'double', meaning that they are both solid and dashed. In such a case RTC has no advantage at all, since every cycle is a contradictory cycle. This does not mean that when using this reduction method RTC is useless: recall that we claim for theoretical dominance over the SPARSE method. It only means that the advantage of RTC is going to be visible if there is a large enough portion of the Equality Graph that is not related to the reduction of Uninterpreted Functions, rather to the formula itself.

The SAT-solver we used for both RTC and the SPARSE method was zCHAFF (2004 edition) [12]. For each benchmark we show the number of generated transitivity constraints, the time it took zCHAFF to solve the SAT formula, the run time of UCLID, which includes RTC but not zCHAFF time and the total run time. Table 2 (top) compares the two algorithms, when UCLID uses Bryant's reduction with Positive Equality. Indeed, as expected, in this setting the advantage of RTC is hardly visible: the number of constraints is a little smaller comparing to what is generated by the SPARSE method (while the time that takes RTC and the SPARSE method to generate the transitivity constraints is almost identical, with a small advantage to the SPARSE method), and correspondingly the runtime of zCHAFF is smaller, although not significantly. We once again emphasize that we consider this as an artifact of the specific benchmarks we found; almost all equalities in them are associated with the reduction of the Uninterpreted Functions. As future research we plan to integrate in our implementation the method of Rodeh et al. [16] which, while using Bryant's reduction, not only produces drastically smaller Equality Graphs, but also does not necessarily require a double edge for each comparison of function instances. This is expected to mostly neutralize this side effect of Bryant's reduction. Table 2 (bottom) compares the two algorithms when Ackermann's reduction is used. Here the advantage of RTC is seen clearly, both in the number of constraints and the overall solving times. In particular, note the reduction from a total of 222,807 constraints to 67,769 constraints.

Random formulas. In another set of experiments we generated hundreds of random formulas and respective Equality Graphs, while keeping the ratio of vertices to edges similar to what we found in the real benchmarks (about 1 vertex to

Benchmark set	# files	SPARSE method				RTC			
		Constraints	uclid	zChaff	total	Constraints	uclid	zChaff	total
TV	9	16719	148.48	1.08	149.56	16083	151.1	0.96	152.0
Cache.inv	4	3669	47.28	40.78	88.06	3667	54.26	38.62	92.8
Dlx1c	3	7143	18.34	2.9	21.24	7143	20.04	2.73	22.7
Elf	3	4074	27.18	2.08	29.26	4074	28.81	1.83	30.6
OOO	6	7059	26.85	46.42	73.27	7059	29.78	45.08	74.8
Pipeline	1	6	0.06	37.29	37.35	6	0.08	36.91	36.99
Total	26	38670	268.19	130.55	398.7	38032	284.07	126.13	410.2
TV	9	103158	1467.76	5.43	1473.2	9946	1385.61	0.69	1386.3
Cache.inv	4	5970	48.06	42.39	90.45	5398	54.65	44.14	98.7
Dlx1c	3	46473	368.12	11.45	379.57	11445	350.48	8.88	359.36
Elf	5	43374	473.32	28.99	502.31	24033	467.95	28.18	496.1
OOO	6	20205	78.27	29.08	107.35	16068	79.5	24.35	103.8
Pipeline	1	96	0.17	46.57	46.74	24	0.18	46.64	46.8
q2	1	3531	30.32	46.33	76.65	855	32.19	35.57	67.7
Total	29	222807	2466.02	210.24	2676.2	67769	2370.56	188.45	2559.0

Table 2. RTC vs. the SPARSE method using Bryant's reduction with positive equalities (top) and Ackermann's reduction (bottom). Each benchmark set corresponds to a number of benchmark files in the same family. The column 'uclid' refers to the total running time of the decision procedure without the SAT solving time.

4 edges). Each benchmark set was built as follows. Given n vertices, we randomly generated 16 different graphs with $4n$ random edges, and the polarity of each edge was chosen randomly according to a predefined ratio p. We then generated a random CNF formula $\mathcal{B}$ with $16n$ clauses (each clause with up to 4 literals) in which each literal corresponds to one of the edges. Finally, we generated two formulas, $\mathcal{T}^S$ and $\mathcal{T}^R$ corresponding to the transitivity constraints generated by the SPARSE and RTC methods respectively, and sent the concatenation of $\mathcal{B}$ with each of these formulas to three different SAT solvers, HaifaSat [8], Siege_v4 [10] and zChaff 2004.

In the results depicted in Table 3 we chose $n = 200$ (in the UCLID benchmarks n was typically a little lower than that). Each set of experiments (corresponding to one cell in the table) corresponds to the average results over the 16 graphs, and a different ratio p, starting from 1 solid to 10 dashed, and ending with 10 solids to 1 dashed. We set the timeout to 600 seconds and added this number in case the solver timed-out. We occasionally let SIEGE run without a time limit (with both RTC and SPARSE), just in order to get some information about instances that none of solvers could solve in the given limit. All instances were satisfiable (in the low ratio of solid to dashed, namely 1:2 and 1:5 we could not solve any of the instances with any of the solvers even after several hours). The conclusions from the table are that (1) in all tested ratios RTC generates less constraints than SPARSE. As expected, this is more apparent when the ratio is further than

1:1; there are very few contradictory cycles in this kind of graphs. (2) with all three SAT solvers it took longer to solve $\mathcal{B} \wedge \mathcal{T}^S$ than to solve $\mathcal{B} \wedge \mathcal{T}^R$.

ratio	constraints		zChaff		HaifaSat		siege_v4	
solid:dashed	Sparse	RTC	Sparse	RTC	Sparse	RTC	Sparse	RTC
1:10	373068.8	181707.8	581.1	285.6	549.2	257.4	1321.6	506.4
1:5	373068.8	255366.6	600.0	600.0	600.0	600.0	600.0	600.0
1:2	373068.8	308346.5	600.0	600.0	600.0	600.0	600.0	600.0
1:1	373068.8	257852.6	5.2	0.4	5.9	3.0	1.2	0.1
2:1	373068.8	123623.4	0.1	0.01	0.6	0.22	0.01	0.01
5:1	373068.8	493.9	0.1	0.01	0.6	0.01	0.01	0.01
10:1	373068.8	10.3	0.1	0.01	0.6	0.01	0.01	0.01
average	373068.8	161057.3	255.2	212.3	251.0	208.7	360.4	243.8

Table 3. RTC vs. the SPARSE method in random satisfiable formulas listed by the ratio of solid to dashed edges.

While it is quite intuitive why the instances should be easier to solve when the formula is satisfiable — the solutions space RTC defines is much larger, it is less clear when the formula is unsatisfiable. In fact, SAT solvers are frequently faster when the input formula contains extra information that further prunes the search space. Nevertheless, the experiments above on UCLID benchmarks (which, recall, are all unsatisfiable) and additional results on random formulas (see [11]) show that RTC is still better in unsatisfiable instances. We speculate that the reason for this is the following. Let T represent all transitivity constraints that are in $\mathcal{T}^S$ but not in $\mathcal{T}^R$. Assuming $\mathcal{B}$ is satisfiable, it can be proven that $\mathcal{B} \wedge T$ is satisfiable as well [11]. This means that any proof of unsatisfiability must rely on clauses from $\mathcal{T}^R$. Apparently in practice it is rare that the SAT solver finds shortcuts through the T clauses.

7 Conclusions and Directions for Future Research

We presented a new decision procedure for Equality Logic, which builds upon and improves previous work by Bryant and Velev in [4, 3]. The new procedure generates a set of transitivity constraints that is, at least in theory, easier to solve. The experiments we conducted show that in most cases it is better in practice as well, and in any case does not make it worse, at least not in more than a few seconds. RTC does not make full use of Theorem 2, as it constrains all Contradictory Cycles rather than only the simple ones. We have another version of the algorithm, not presented in the article due to lack of space, that handles this problem, but with an exponential price. As stated before, the question whether there exists a polynomial algorithm that does the same or it is inherently a hard problem, is left open.

Acknowledgement We are exceptionally grateful to Sanjit Seshia for the many hours he invested in hooking our procedure to UCLID, and for numerous insightful conversations we had on this and related topics.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. R. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc.* CAV'99, 1999.
3. R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, 2001.
4. R. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. In *Proc. CAV'00*, volume 1855 of *Lect. Notes in Comp. Sci.*, 2000.
5. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.
6. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. CAV'94*, volume 818 of LNCS, pages 68–80. 1994.
7. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 26, page 563. MIT press, 2000.
8. R. Gershman and O. Strichman. Cost-effective hyper-resolution for preprocessing cnf formulas. In T. Walsh and F. Bacchus, editors, SAT'05, 2005.
9. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A. Hu and M. Vardi, editors, *CAV98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
10. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
11. O. Meir and O. Strichman. Yet another decision procedure for equality logic (full version), 2005. ie.technion.ac.il/∼ofers/cav05_full.ps.
12. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
13. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. *Proc.* CAV'99, LNCS, 1999.
14. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and computation*, 178(1):279–293, Oct. 2002.
15. A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. Proc. ICALP'98, volume 1443 of LNCS, pages 235–246. Springer-Verlag, 1998.
16. Y. Rodeh and O. Shtrichman. Finite instantiations in equivalence logic with uninterpreted functions. Proc. CAV'01.
17. R. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583 – 585, July 1978.