Deciding Equality Formulas by Small Domains Instantiations *

Amir Pnueli, Yoav Rodeh, Ofer Shtrichman, and Michael Siegel

Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Rehovot, Israel,
email{amir—yrodeh—ofers}@wisdom.weizmann.ac.il

Abstract. We introduce an efficient decision procedure for the theory of equality based on finite instantiations. When using the finite instantiations method, it is a common practice to take a range of $[1..n]$ (where n is the number of input non-Boolean variables) as the range for all non-Boolean variables, resulting in a state-space of n^n. Although various attempts to minimize this range were made, typically they either required various restrictions on the investigated formulas or were not very effective. In many cases, the n^n state-space cannot be handled by BDD-based tools within a reasonable amount of time. In this paper we show that significantly smaller domains can be algorithmically found, by analyzing the structure of the formula. We also show an upper bound for the state-space based on this analysis. This method enabled us to verify formulas containing hundreds of integer and floating point variables.

Keywords: Finite Instantiation, equality logic, uninterpreted functions, compiler verification, translation validation, Range Allocation.

1 Introduction

Automated validation techniques for formulas of the theory of equality become increasingly important as the advantages of abstraction and the use of uninterpreted functions (UIFs) become more evident. UIFs are mainly useful when proving equivalence between two models. Proving design equivalence or comparing a specification to an implementation are two typical examples of such equivalence proofs. In our case, we proved equivalence between source and target code serving as the input and output of a compiler, and thus verified that the compilation process was correct (see [PSS98b], [PSS99] and [Con95] for more details about this project).

When verifying equivalence between two formulas, it is often possible to abstract away all functions, except the equality sign and Boolean operators, by replacing them with UIFs. An abstracted formula holds less information and therefore can be represented by a significantly smaller BDD. It was Ackerman [Ack54] who first showed the reduction of such abstracted formulas to function-free formulas of the theory of equality, while preserving validity. He suggested doing so by replacing each occurrence of a function with a new variable, and adding constraints that preserve their functionality as an antecedent of the formula, rewriting $(z = F(x, y) \wedge u = y) \rightarrow z = F(x, u)$ into $((x = x \wedge y = u) \rightarrow f_1 = f_2) \rightarrow ((z = f_1 \wedge u = y) \rightarrow z = f_2)$.

The abstraction process itself does not preserve validity and may transform a concrete valid formula such as $x + y = y + x$ into the invalid formula $F(x, y) = F(y, x)$ which does not hold for all functions F. However, in many useful contexts, such as the verification of compilers which do not perform extensive arithmetical optimizations, the process of abstraction is often justified. At least we can rely on the fact that the process of abstraction into UIFs never generates false positives, and that if the abstract version is found valid, this is also the case with the concrete formulas it abstracts.

After performing such an abstraction followed by Ackerman's reduction, the resulting formula is an equality formula, and enjoys the small model property (i.e. it is satisfiable iff it is satisfiable over a finite domain). Therefore, the next step is the calculation of a finite domain, such that the formula is valid iff it is valid over all interpretations of this finite domain. The latter can be checked with a finite state decision procedure. A known 'folk theorem' is that it is enough to give each variable the range $[1..n]$ (where n is the number of non-Boolean input variables), resulting in a state-space of n^n. It is not difficult to see that

* This research was supported in part by the Minerva Center for Verification of Reactive Systems, a gift from Intel, a grant from the U.S.-Israel bi-national science foundation, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.

this range is sufficient for preserving the validity or invalidity of the formula. If a formula is not valid, there is at least one assignment that makes the formula false. Any assignment that partitions the variables into the same equivalence classes will also falsify the formula (the absolute values are of no importance). Since there can not be more than n classes, the $[1..n]$ range is sufficient regardless of the formula's structure. In this paper we will show that analyzing the formula's structure can lead to significantly smaller domains. For example, a trivial improvement is to construct a graph whose vertices are the formula's non-Boolean variables, and the edges represent the comparisons between them. Then, instead of giving a range of $[1..n]$ to all variables, give to each variable the range $[1..k]$, where k is the size of the component it belongs to ($k \leq n$). Experiments with 'real-life' problems has shown us that this simple partitioning can be very effective.

Hojati et. al ([HIKB96], [HKGB97]) tried to avoid the $[1..n]$ range by first considering the explicit DNF of the formula. Given the formula in this form, they 'colored' the comparison graph of each clause (a graph based on the disequalities in the formula) and chose the maximal chromatic number (the number of colors needed for coloring the graph) as the range for each variable. As a second step, they tried to approximate the maximum number of disequalities needed to satisfy the formula, in a general formula. Given that number, a uniform range of $[1..k]$ is sufficient, where k is calculated on the basis of this number. It seems that finding a good approximation is very hard. Although several heuristics are suggested, it is unclear how well they work. They also indicated an inherent problem with finding a good BDD ordering in the presence of Ackerman constraints (the *stripping assertions* in the notation of their paper).

Sajid et al [SGZ+98] approached this problem from a different angle. Since non-Boolean variables appear in the formula only when compared to one another, they suggest encoding each such comparison with a new Boolean variable, and ensuring transitivity of equality by restricting the BDD traversing accordingly. Although this traversing procedure is proved by the authors to be worst-case exponential, it proved to be more efficient than finite instantiations with the $[1..n]$ range.

Even with this range, which we show in this paper is not tight, it is not always the case that this kind of encoding results in a smaller state-space (as was mentioned by the authors themselves). Consider, for example, a formula where all variables are compared to each other (graphically, this is a clique of n vertices). In this case, $n \cdot (n-1)/2$ new Boolean variables will be introduced, each represented by a BDD variable. Finite instantiations with a range of $[1..n]$, on the other hand, will require only $n \cdot \log n$ BDD variables.

In a more recent work, Bryant, German and Velev [BGV99] restricted the logic to formulas that contain positive equalities only, i.e. the outcome of any equality test between terms can only be part of a monotonically positive Boolean formula. This restriction disallows the use of the outcome of equalities in control decisions. Given this restricted logic, they were able to substitute UIFs with unique constants that serve as 'witnesses' in case the formula is false. This replacement naturally reduced the state-space immensely, and made the decision procedure highly efficient. Although they chose the same case study examined by [SGZ+98], the results are not given in a way that they can be compared.

The verification conditions we considered were not restricted to positive equality. They had the form of an implication between two conjuncts ($\bigwedge_{i=1}^{n} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$), typically with several thousand clauses in each side (the left and right hand side corresponded to the transition relations of the target and source code respectively), and more than a thousand variables. The abstraction process added several hundred more variables, and thousands of constraints. Several hundreds of these variables were integer and floating-point, while the rest were Boolean. Although we decomposed the formula, we still had many verification conditions with 150 integer variables or more. Since the size of the domain is crucial to the time required to complete the proof with a BDD-based tool, the n^n state-space (where $n > 150$ in our case) was naturally far too large to handle.

In the next section, we present a precise definition of the problem we consider: deciding validity (satisfiability) of equality formulas, and explain how it naturally arises in the context of translation validation. In Section 3 we outline our general solution strategy, which is a computation of a small set of domains (ranges) R such that the formula is satisfiable iff it is satisfiable over R, followed by a test for R-satisfiability performed by a standard BDD package. The remaining question is how to find such a set of small domains. To answer this question, we show how it can be reduced to a graph-theoretic problem. The rest of the paper focuses on algorithms, which, in most cases, produce tractably small domains. In Section 4, we describe the basic algorithm. The soundness proof of the algorithm is given in Section 5. In Section 6, we present several improvements to the basic algorithm, and analyze their effect on the upper bound of the resulting

state-space. We describe experimental results from an industrial case study in Section 7, and conclude in Section 8 by considering possible directions for future research.

2 The Problem: Deciding Equality Formulas

Our interest in the problem of deciding equality formulas arose within the context of the *Code Validation Tool* (CVT) that we developed as part of the European project *Sacres*. The focus in this project is on developing a methodology and a set of tools for the correct construction of safety critical systems.

CVT is intended to ensure the correctness of the code generator incorporated in the *Sacres* tools suite which automatically translates high level specifications into running code in C and Ada (see [PSS98b], [PSS99], [Con95]). Rather than formally verifying the code generator program, CVT verifies the correctness of every individual run of the generator, comparing the source with the produced target language program and checking their compatibility. This approach of *translation validation* seems to be promising in many other contexts, where verification of the operation of a translator or a compiler is called for.

We will illustrate this approach by a representative example. Assume that a source program contained the statement $z := (x_1 + y_1) \cdot (x_2 + y_2)$ which the translator we wish to verify compiled into the following sequence of three assignments:

$u_1 := x_1 + y_1;\ u_2 := x_2 + y_2;\ z := u_1 \cdot u_2,$

introducing the two auxiliary variables u_1 and u_2.

For this translation, CVT first constructs the verification condition

$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 \cdot u_2 \quad \rightarrow \quad z = (x_1 + y_1) \cdot (x_2 + y_2),$

whose validity we wish to check.

The second step performed by CVT in handling such a formula is to abstract the concrete functions appearing in the formula, such as addition and multiplication, by abstract (uninterpreted) function symbols. The abstracted version of the above implication is:

$u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2) \quad \rightarrow \quad z = G(F(x_1, y_1), F(x_2, y_2))$

Clearly, if the abstracted version is valid then so is the original concrete one.

Next, we perform the Ackerman reduction [Ack54], replacing each functional term by a fresh variable but adding, for each pair of terms with the same function symbol, an extra antecedent which guarantees the functionality of these terms. Namely, that if the two arguments of the original terms were equal, then the terms should be equal. It is not difficult to see that this transformation preserves validity.

Applying the Ackerman reduction to the abstracted formula, we obtain the following equality formula:

$$\varphi: \quad \left(\begin{array}{c} (x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \rightarrow g_1 = g_2) \wedge \\ u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \end{array} \right) \quad \rightarrow \quad z = g_2 \tag{1}$$

Note the extra antecedent ensuring the functionality of F by identifying the conditions under which f_1 should equal f_2 and the similar requirement for G.

This shows how equality formulas such as φ of Equation (1) arise in the process of translation validation.

Equality Formulas. Even though the variables appearing in an equality formula such as φ are assumed to be completely uninterpreted, it is not difficult to see that a formula such as φ is generally valid (satisfiable) iff it is valid (respectively, satisfiable) when the variables appearing in the formula range over the integers. This leads to the following definition of the syntax of equality formulas that the method presented in this paper can handle.

Let $x_1, x_2, \ldots$ be a set of *integer variables*, and $b_1, b_2, \ldots$ be a set of *Boolean variables*. We define the set of *terms* $\mathcal{T}$ by

$\mathcal{T} \quad ::= \quad$ *integer constant* $\mid \ x_i \ \mid$ **if** Φ **then** $\mathcal{T}_1$ **else** $\mathcal{T}_2$

The set of equality formulas Φ is defined by

$\Phi \quad ::= \quad b_j \ \mid \ \neg\Phi \ \mid \ \Phi_1 \vee \Phi_2 \ \mid \ \mathcal{T}_1 = \mathcal{T}_2 \ \mid$ **if** Φ_0 **then** Φ_1 **else** Φ_2

Additional Boolean operators such as $\wedge$, $\rightarrow$, $\leftrightarrow$, can be defined in terms of $\neg, \vee$.

For simplicity, we will not consider in this paper the cases of integer constants and Boolean variables. The full version of the algorithms which deals with the general case is presented in [PRSS98].

3 The Solution: Instantiations over Small Domains

As previously explained, the problem solved in this paper is to check whether a given equality formula φ is satisfiable. Our solution strategy can be summarized as follows:

1. Determine, in polynomial time, a *range allocation* $R : \mathit{Vars}(\varphi) \mapsto 2^{\mathbb{N}}$, by mapping each integer variable $x_i \in \varphi$ into a small finite set of integers, such that φ is satisfiable (valid) iff it is satisfiable (respectively, valid) over some R-interpretation.
2. Encode each variable x_i as an enumerated type over its finite domain $R(x_i)$, and use a standard BDD package to construct a BDD B_φ. Formula φ is satisfiable iff B_φ is not identical to 0.

We define the complexity of a range allocation R to be the size of the state-space spanned by R, that is, if $\mathit{Vars}(\varphi) = \{x_1, \ldots, x_n\}$, then $|R| = |R(x_1)| \times |R(x_2)| \times \cdots \times |R(x_n)|$.

Obviously, the success of our method depends on our ability to find range allocations with small complexity.

3.1 Some Simple Bounds

In theory, there always exists a *singleton* range allocation R^*, satisfying the above requirements, such that R^* allocates each variable a domain consisting of a single natural, i.e., $|R^*| = 1$. This is supported by the following trivial argument. If φ is satisfiable, then there exists an assignment $(x_1, \ldots, x_n) = (z_1, \ldots, z_n)$ satisfying φ. It is sufficient to take $R^* : x_1 \mapsto \{z_1\}, \ldots x_n \mapsto \{z_n\}$ as the singleton allocation. If φ is unsatisfiable, it is sufficient to take $R^* : x_1, \ldots, x_n \mapsto \{0\}$.

However, finding the singleton allocation R^* amounts to a head-on attack on the primary NP-complete problem. Instead, we generalize the problem and attempt to find a small range allocation which is adequate for a *set* of formulas Φ which are "structurally similar" to the formula φ, and includes φ itself.

Consequently, we say that the range allocation R is *adequate* for the formula set Φ if, for every equality formula in the set $\varphi \in \Phi$, φ is satisfiable iff φ is satisfiable over R.

First, let us consider Φ_n, the set of all equality formulas with at most n variables.

Claim 1 (Folk theorem) *The uniform range allocation $R : \{x_1, \ldots, x_n\} \mapsto [1..n]$ with complexity n^n is adequate for Φ_n.*

We can do better if we do not insist on a *uniform* range allocation which allocates the same domain to all variables. Thus the range allocation $R : x_i \mapsto [1..i]$ is also adequate for Φ_n and has the better complexity of $n!$. In fact, we conjecture that $n!$ is also a lower bound on the size of range allocations adequate for Φ_n.

The formula set Φ_n utilizes only a simple structural characteristic common to all of its members, namely, the number of variables. As a result Φ_n groups together many formulas of radically different nature. No wonder that the best size of adequate range allocations for the whole set is so high.

By paying attention to additional structural similarities of formulas, we were able to form smaller sets of formulas and managed to obtain much smaller adequate range allocations, which we proceed to describe in the rest of this paper.

3.2 An Approach Based on the Set of Atomic Formulas

We assume that φ has no constants or Boolean variables, and is given in a positive form, i.e. negations are only allowed within atomic formulas of the form $x_i \neq x_j$. An important property of formulas in positive form is that they are monotonically satisfied, i.e. if S_1 and S_2 are two subsets of atomic formulas of φ (where φ is given in positive form), and $S_1 \subseteq S_2$, then $S_1 \models \varphi$ implies $S_2 \models \varphi$. Any equality formula can be brought into such positive form, by expressing all Boolean operations such as $\rightarrow$, $\leftrightarrow$ and the *if-then-else* construct in terms of the basic Boolean operations $\neg$, $\vee$, and $\wedge$, and pushing all negations inside.

Let $\mathit{At}(\varphi)$ be the set of all atomic formulas of the form $x_i = x_j$ or $x_i \neq x_j$ appearing in φ, and let $\Phi(\varphi)$ be the family of all equality formulas which have the same set of atomic formulas as φ. Obviously $\varphi \in \Phi(\varphi)$. Note that the family defined by the atomic formula set $\{x_1 = x_2, x_1 \neq x_2\}$ includes both the satisfiable formula $x_1{=}x_2 \vee x_1{\neq}x_2$ and the unsatisfiable formula $x_1{=}x_2 \wedge x_1{\neq}x_2$.

For a set of atomic formulas A, we say that the subset $B = \{\psi_1, \ldots, \psi_k\} \subseteq A$ is *consistent* if the conjunction $\psi_1 \wedge \cdots \wedge \psi_k$ is satisfiable. Note that a set B is consistent iff it does not contain a chain of the form $x_1 = x_2$, $x_2 = x_3$, $\ldots$, $x_{r-1} = x_r$ together with the formula $x_1 \neq x_r$.

Given a set of atomic formulas A, a range allocation R is defined to be *satisfactory* for A if every consistent subset $B \subseteq A$ is R-satisfiable.

For example, the range allocation $R\colon x_1, x_2, x_3 \mapsto \{0\}$ is satisfactory for the atomic formula set $\{x_1 = x_2,\ x_2 = x_3\}$, while the allocation $R\colon x_1 \mapsto \{1\},\ x_2 \mapsto \{2\},\ x_3 \mapsto \{3\}$ is satisfactory for the formula set $\{x_1 \neq x_2,\ x_2 \neq x_3\}$. On the other hand, no singleton allocation is satisfactory for the set $\{x_1 = x_2,\ x_1 \neq x_2\}$. A minimal satisfactory allocation for this set can be given by $R\colon x_1 \mapsto \{1\},\ x_2 \mapsto \{1, 2\}$.

Claim 2 *The range allocation R is satisfactory for the atomic formula set A iff R is adequate for $\Phi(A)$ the set of formulas φ such that $At(\varphi) = A$.*

Thus, we concentrate our efforts on finding a small range allocation which is satisfactory for $A = At(\varphi)$ for a given equality formula φ. In view of the claim, we will continue to use the terms satisfactory and adequate synonymously.

We partition the set A into the two sets $A = A_= \cup A_\neq$, where $A_=$ contains all the equality formulas in A, while $A_\neq$ contains the disequalities. Variable x_i is called a *mixed variable* iff $(x_i, x_j) \in A_=$ and $(x_i, x_k) \in A_\neq$ for some $x_j, x_k \in Vars(\varphi)$.

Note that the sets $A_=(\varphi)$ and $A_\neq(\varphi)$ for a given formula φ can be computed without actually carrying out the transformation to positive form. All that is required is to check whether a given atomic formula has a positive or negative *polarity* within φ. Where the polarity of a sub-formula p is determined according to whether the number of negations enclosing p is even (positive polarity) or odd (negative polarity). Additional considerations apply to sub-formulas involving the *if-then-else* construct.

Example 1. Let us illustrate these concepts on the formula φ of Equation (1), whose validity we wished to check.

Since our main algorithm checks for satisfiability, we proceed to form the positive form of $\neg\varphi$, which is given by:

$$\neg\varphi\colon \quad \left(\begin{array}{l} (x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ \quad u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \end{array}\right) \ \wedge \ z \neq g_2,$$

and therefore

$$A_= : \{(f_1 = f_2),\ (g_1 = g_2),\ (u_1 = f_1),\ (u_2 = f_2),\ (z = g_1)\}$$
$$A_\neq : \{(x_1 \neq x_2),\ (y_1 \neq y_2),\ (u_1 \neq f_1),\ (u_2 \neq f_2),\ (z \neq g_2)\}$$

Note that u_1, u_2, f_1, f_2, g_2 and z in this example are mixed variables.

As explained above, the sets $A_=$ and $A_\neq$ can be computed directly by counting the number of negations enclosing the atomic formulas in φ without transforming to positive form or even explicitly negating φ. For example, the comparison $x_1 = x_2$ in φ is contained within two negations implied by appearing on the left-hand side of two (nested) implications. Since we are considering $\neg\varphi$, this amounts to 3 negations. Since 3 is odd, we add $x_1 \neq x_2$ to $A_\neq$. In a similar way, the comparison $f_1 = f_2$, being under 2 negations, is added to $A_=$. □

This example would require a state-space of 11! if we used the upper bound without further calculations (and a state-space of 11^{11} if we used the $[1..n]$ range). We will later show that this example possesses an adequate range allocation of size 16.

3.3 A Graph-Theoretic Representation of the Sets $A_=$, $A_\neq$

The sets $A_\neq$ and $A_=$ can be represented by two graphs, $G_=$ and $G_\neq$ defined as follows:

(x_i, x_j) is an edge on $G_=$, the *equalities graph*, iff $(x_i = x_j) \in A_=$.

(x_i, x_j) is an edge on $G_\neq$, the *disequalities graph*, iff $(x_i \neq x_j) \in A_\neq$.

We refer to the joint graph as G. Each vertex in these graphs represents a variable, and therefore some of them represent mixed variables. We refer to these vertices as *mixed vertices*.

An inconsistent subset $B \subseteq A$ will appear as a *contradictory cycle* i.e. a cycle consisting of a single $G_{\neq}$ edge and any positive number of $G_{=}$ edges.

In Fig. 1, we present the graphs corresponding to the formula $\neg\varphi$, where $G_{=}$ edges are represented by dashed lines and $G_{\neq}$ edges are represented by solid lines. Note that there are three contradictory cycles in this graph: $(g_2 - g_1 - z), (u_1 - f_1)$ and $(u_2 - f_2)$.

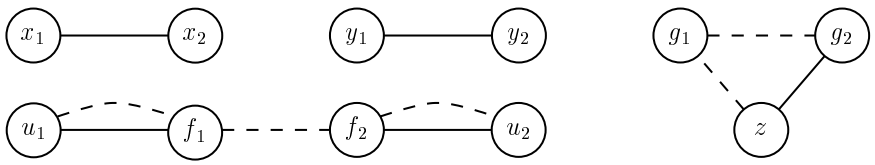

Fig. 1. The Graph $G : G_{\neq} \cup G_{=}$ representing $\neg\varphi$

4 The Basic Range Allocation Algorithm

Following is a two-step algorithm for computing an economic range allocation R for the variables in a given formula φ.

I. Pre-processing

Initially, $R(x_i) = \emptyset$, for all vertices $x_i \in G$.

A. Remove all $G_{\neq}$ edges which do not lie on a contradictory cycle.
B. For every singleton vertex (a vertex comprising a connected component by itself) x_i, add to $R(x_i)$ a fresh value u_i, and remove x_i from the graph.

II. Value Allocation

A. While there are mixed vertices in G do:
   1. Choose a mixed vertex x_i. Add u_i, a fresh value, to $R(x_i)$.
   2. Assign $R(x_j) := R(x_j) \cup \{u_i\}$ for each vertex x_j, s.t. there is a $G_{=}$-path from x_i to x_j.
   3. Remove x_i from the graph.
B. For each (remaining) connected $G_{=}$ component $C_{=}$, add a common fresh value $u_{C_{=}}$ to $R(x_k)$, for every $x_k \in C_{=}$.

We refer to the fresh values u_i added to vertex x_i in steps I.B and II.A.1, and $u_{C_{=}}$ added to $x_k \in C_{=}$ in step II.B, as the *characteristic* values of these vertices. We write $char(x_i) = u_i$ and $char(x_k) = u_{C_{=}}$. Note that every vertex is assigned a single characteristic value. Vertices which are assigned their characteristic values in steps I.B and II.A.1 are called *individually assigned vertices*, while the vertices assigned characteristic values in step II.B are called *communally assigned vertices*. We assume that fresh values are assigned in ascending order, so that $char(x_i) < char(x_j)$ implies that x_i was assigned its characteristic value before x_j.

The presented description of the algorithm leaves open the order in which vertices are chosen in step II.A, which has a strong impact on the size of the resulting state-space. The set of vertices that are removed in this step can be seen as a *vertex cover* of the $G_{\neq}$ edges, i.e., a set of vertices V such that every $G_{\neq}$ edge has at least one of its ends in V. To keep this set as small as possible, we apply the known "greedy" heuristic for the Minimal Vertex Cover problem, and accordingly we denote this set by *mvc*. We choose mixed vertices following a descending degree on $G_{\neq}$. Among vertices with equal degrees on $G_{\neq}$, we choose the one with the highest degree on $G_{=}$. This heuristic seems not only to find a small vertex cover, it also partitions the graph rather rapidly, which is important for minimizing the size of the R-sets. However, we expect that further research will reveal other, more efficient ordering methods.

Example 2. The following trace represents the application of the Basic Range Allocation algorithm to the formula $\neg\varphi$, where each step is represented by a single line:

Step/ var	x_1	x_2	y_1	y_2	u_1	f_1	f_2	u_2	g_2	z	g_1	Remarks
Step I.A												Removed edges $(x_1 - x_2), (y_1 - y_2)$
Step I.B	0	1	2	3								Removed x_1, x_2, y_1, y_2
Step II.A (f_1)					4	4	4	4				Removed f_1
Step II.A (f_2)							4,5	4,5				Removed f_2
Step II.A (g_2)									6	6	6	Removed g_2
Step II.B					4,7							
Step II.B								4,5,8				
Step II.B										6,9	6,9	
Final R-sets	0	1	2	3	4,7	4	4,5	4,5,8	6	6,9	6,9	State-space = 48

□

5 The Algorithm is Sound

In this section we argue for the soundness of the basic algorithm. We begin by describing a procedure which, given the allocation R produced by the basic algorithm and a consistent subset B, assigns to each variable $x_i \in G$ an integer value $a(x_i) \in R(x_i)$. We then continue by proving that this assignment guarantees that every consistent subset is satisfied, and that it is always feasible.

An Assignment Procedure

Given a consistent subset B and its representative graph $G(B)$, assign to each vertex $x_i \in G(B)$ a value $a(x_i) \in R(x_i)$, according to the following rules:

1. If x_i is connected by a (possibly empty) $G_=(B)$-path to an individually assigned vertex x_j, assign to x_i the minimal value of $char(x_j)$ among such x_j's.
2. Otherwise, assign to x_i its communally assigned value $char(x_i)$.

Example 3. Consider the R-sets that were computed in example 2. Let us apply the assignment procedure to a subset B that contains all edges excluding both edges between u_1 to f_1, the dashed edge between g_1 and g_2, and the solid edge between f_2 and u_2. The assignment will be as follows:

- By step 1, f_1, f_2 and u_2 are assigned the value $char(f_1)$ ='4', because f_1 was the first mixed vertex in the sub-graph $\{f_1, f_2, u_2\}$ that was removed in step II.A, and consequently it has the minimal characteristic value.
- By step 1, x_1, x_2, y_1 and y_2 are assigned the characteristic values '0', '1', '2', '3' respectively, which they received in step I.B.
- By step 1, g_2 is assigned the value $char(g_2)$ ='6' which it received in step II.A.
- By step 2, z and g_1 are assigned the value '9' which they received in step II.B. □

Claim 3 *The assignment procedure satisfies every consistent subset B.*

Proof. We have to show that all constraints implied by the set B are satisfied by the assignment.

Consider first the case of two variables x_i and x_j which are connected by a $G_=(B)$ edge. We have to show that $a(x_i) = a(x_j)$. Since x_i and x_j are $G_=(B)$-connected, they belong to the same $G_=(B)$-connected component. If they were both assigned a value in step 1, then they were assigned the minimal value of an individually assigned vertex to which they are both $G_=(B)$-connected. If, on the other hand, they were both assigned a value in step 2, then they were assigned the communal value assigned to the $G_=$ component to which they both belong. Thus, in both cases they are assigned the same value.

Next, consider the case of two variables x_i and x_j which are connected by a $G_{\neq}(B)$ edge. We have to show that $a(x_i) \neq a(x_j)$. We distinguish between three cases:

- If both x_i and x_j were assigned values in step 1, they must have inherited their values from two distinct individually allocated vertices. Because, otherwise, they are both connected by a $G_{=}(B)$ path to a common vertex, which together with the (x_i, x_j) $G_{\neq}(B)$-edge closes a contradictory cycle, excluded by the assumption that B is consistent.
- If one of x_i, x_j was assigned a value in step 1 while the other acquired its value in step 2, then since any communal value is distinct from any individually allocated value, $a(x_i)$ must differ from $a(x_j)$.
- The remaining case is when both x_i and x_j were assigned values in step 2. The fact that they were not assigned values in step 1 implies that their characteristic values are not individually but communally allocated. If $a(x_i) = a(x_j)$ it means that x_i and x_j were allocated their communal values in the same step II.B of the allocation algorithm, which implies that they had a $G_{=}$-path between them. Hence, x_i and x_j belong to a contradictory cycle, and the solid edge (x_i, x_j) was therefore still part of G in the beginning of step II.A. By definition of *mvc*, at least one of them was individually assigned in step II.A.1, and consequently, according to the assignment procedure, the component it belongs to is assigned a value in step 1, in contrast to our assumption. We can therefore conclude that our assumption that $a(x_i) = a(x_j)$ was false. □

Claim 4 *The assignment procedure is feasible (i.e. the R-sets include the values required by the assignment procedure).*

Proof. Consider first the two classes of vertices that are assigned a value in step 1. The first class includes vertices that are removed in step I.B. These vertices have only one (empty) $G_{=}(B)$ path to themselves, and are therefore assigned the characteristic value they received in this step. The second class includes vertices that have a (possibly empty) $G_{=}(B)$ path to a vertex from *mvc*. Let x_i denote such a vertex, and let x_j be the vertex with the minimal characteristic value that x_i can reach on $G_{=}(B)$. Since x_i and all the vertices on this path were still part of the graph when x_j was removed in step II.A, then according to step II.A.2, $char(x_j)$ was added to $R(x_i)$. Thus, the assignment of $char(x_j)$ to x_i is feasible.

Next, consider the vertices that are assigned a value in step 2. Every vertex that is removed in step I.B or II.A is clearly assigned a value in step 1. All the other vertices are communally assigned a value in step II.b. In particular, the vertices that do not have a path to an individually assigned vertex are assigned such a value. Thus, the two steps of the assignment procedure are feasible. □

Claim 5 *φ is satisfiable iff φ is satisfiable over R.*

Proof. By claims 3 and 4, R is satisfactory for $A_{=} \cup A_{\neq}$. Consequently, by claim 2 R is adequate for $\Phi(At(\varphi))$, and in particular R is adequate for $\Phi(\varphi)$. Thus, by the definition of adequacy, φ is satisfiable iff φ is satisfiable over R. □

6 Improvements of the Basic Algorithm

There are several improvements to the basic algorithm, which can significantly decrease the size of the resulting state-space. Here, we present some of them.

6.1 Coloring

Step II.A.1 of the basic algorithm calls for allocation of *distinct* characteristic values to the mixed vertices. This is not always necessary, as we demonstrate in the following small example.

Example 4. Consider the subgraph $\{u_1, f_1, f_2, u_2\}$ from the graph of Fig. 1. Application of the basic algorithm to this subgraph may yield the following allocation, where the assigned characteristic values are underlined: $R_1 : u_1 \mapsto \{0, \underline{2}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{0, \underline{1}\}, u_2 \mapsto \{0, 1, \underline{3}\}$. This allocation leads to a state-space complexity of 12.

By relaxing the requirement that all individually assigned characteristic values should be distinct, we can obtain the allocation $R_2 : u_1 \mapsto \{0, \underline{2}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{\underline{0}\}, u_2 \mapsto \{0, \underline{1}\}$ with a state-space complexity of 4.

It is not difficult to see that R_2 is adequate for the considered subgraph. □

We will now explore some conditions under which the requirement of distinct individually assigned values can be relaxed while maintaining adequacy of the allocation.

Assume that the mixed vertices are assigned their individual characteristic values in the order $x_1, \ldots, x_m$. Assume that we have already assigned individual *char* values to $x_1, \ldots, x_{r-1}$ and are about to assign a *char* value to x_r. What may be the reasons for not assigning to x_r the value of $char(x_i)$ for some $i < r$? Examining our assignment procedure, such an assignment may lead to violation of the B-constraints only if there exists a path of the form:

$$x_i \text{ - - - } \cdots \text{ - - - } x_j \text{ ——— } x_k \text{ - - - } \cdots \text{ - - - } x_r$$

where for every individually assigned vertex x_p on the $G_=$-path from x_i to x_j (including x_j), $i \leq p$, and equivalently for every vertex x_q on the $G_=$-path from x_r to x_k (including x_k), $r \leq q$.

This observation is based on the way the assignment procedure works: it assigns to all vertices in a connected $G_=(B)$ component the characteristic value of the mixed vertex with the lowest index. Thus, if there exists a vertex x_p on the path from x_i to x_j s.t. $p < i$, then x_j will not be assigned the value $char(x_i)$. Consequently, there is no risk that the assignment procedure will assign x_j and x_k the same value, even if the characteristic values of x_i and x_r are equal.

We refer to vertices that have such a path between them as being *incompatible* and assign them different characteristic values.

Assigning Values to Mixed Vertices with Possible Duplication.

To allow duplicate characteristic values, we add the following as step I.C of the algorithm.

1. Predetermine the order $x_1, \ldots, x_m$, by which individually assigned variables will be allocated their characteristic values.
2. Construct an *incompatibility graph* G_{inc} whose vertices are $x_1, \ldots, x_m$ and there is an edge connecting x_i to x_r iff x_i and x_r are incompatible.
3. Find a minimal coloring for G_{inc}, i.e. assign values ('colors') to the vertices of G_{inc} s.t. no two neighboring vertices receive the same value. Due to the preprocessing step, we require that each connected component is colored with a unique 'pallet' of colors.

Step II.A.1 should be changed as follows:

1. Choose a mixed vertex x_i. Add to $R(x_i)$ the color c_i that was determined in step I.C.3 as the characteristic value of x_i.

Like the case of minimal vertex covering, step 3 calls for the solution of the NP-hard problem of minimal coloring. In a similar way, we resolve this difficulty by applying one of the approximation algorithms (e.g. one of the "greedy" algorithms) for solving this problem.

Example 5. Once more, let us consider the subgraph $\{u_1, f_1, f_2, u_2\}$ of Fig. 1. The modified version of the algorithm identifies the order of choosing the mixed vertices as f_1, f_2. The incompatibility graph G_{inc} for this ordering simply consists of the two vertices f_1 and f_2 with no edges. This means that we can color them by the same color, leading to the allocation $R_2 : u_1 \mapsto \{0, \underline{2}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{\underline{0}\}, u_2 \mapsto \{0, \underline{1}\}$, presented in Example 4.

For demonstration purposes, assume that all four vertices in this component were connected by additional edges to other vertices, and that the removal order of step II.A was determined to be : f_1, f_2, u_2, u_1. The resulting G_{inc} is depicted in Fig. 2(a). By the definition of G_{inc}, every two vertices connected on this graph must have different characteristic values. For example f_1 and u_2 cannot have the same characteristic value because $G(B)$ can consist of both the solid edge (f_2, u_2) and the dashed edge (f_1, f_2) (in the original graph). Since according to the assignment procedure the value we assign to f_1 and f_2 is determined by $char(f_1)$, it must be different than $char(u_2)$.

Since this graph can be colored by two colors, say, f_1 and f_2 colored by 0, while u_1 and u_2 colored by 1, we obtain the allocation $R_3 : u_1 \mapsto \{0, \underline{1}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{\underline{0}\}, u_2 \mapsto \{0, \underline{1}\}$ □

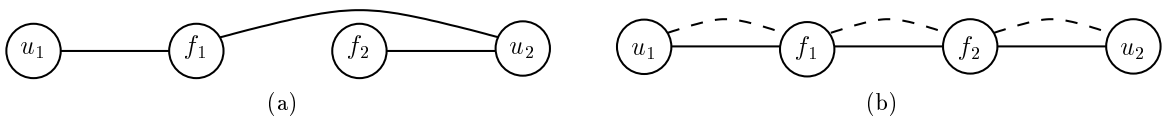

Fig. 2. (a) The Graph G_{inc} (b) Illustrating selective allocation

6.2 Selective Assignments of Characteristic Values in step II.B

Step II.B of the basic algorithm requires an unconditional assignment of a fresh characteristic value to each remaining connected $G_=$ component. This is not always necessary, as shown by the following example.

Example 6. Consider the graph G presented in Fig. 2(b). Applying the Range Allocation algorithm to this graph can yield the ordering f_1, f_2 and consequently the allocation $R_4 : u_1 \mapsto \{0, \underline{3}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{0, \underline{1}\}, u_2 \mapsto \{0, 1, \underline{2}\}$ with complexity 12 (although by the coloring procedure suggested in the previous sub-section u_1 and f_2 can have the same characteristic value, it will not reduce the state-space in this case). Our suggestion for improvement will identify that, while it is necessary to add the characteristic value '3' to $R(u_1)$, the addition of '2' to $R(u_2)$ is unnecessary, and the allocation $R_5 : u_1 \mapsto \{0, \underline{3}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{0, \underline{1}\}, u_2 \mapsto \{0, 1\}$ with complexity 8 is adequate for the graph of Fig. 2(b). □

Assume that $C_=$ is a remaining connected $G_=$ component with no mixed vertices, and let $K = \bigcap_{x \in C_=} R(x)$ be the set of values that are common to the allocations of all vertices in $C_=$ (in fact, it can be proven that for all $x \in C_=$, $R(x)$ is equal). Let $y_1, \ldots, y_k \notin C_=$ be all the vertices which are $G_{\neq}$-neighbors of vertices in $C_=$. The following condition is sufficient for *not* assigning the vertices of $C_=$ a fresh characteristic value:

Condition *Con*: $k < |K|$, or $K - \bigcup_{i=1}^{k} R(y_i) \neq \emptyset$.

Note that when condition *Con* holds, there is always a value in K which is different from the values $y_1, \ldots, y_k$.

For example, when we consider the component $\{u_2\}$ in the graph of Fig. 2(b), we have that $K = \{0, 1\}$ with $|K| = 2$, while this component has only one $G_{\neq}$-neighbor: f_2. Consequently, we can skip the assignment of the fresh value '2' to u_2.

Therefore, we modify step II.B of the basic algorithm to read as follows:

B. For each (remaining) connected $G_=$ component $C_=$, if condition *Con* does not hold, add a common fresh value $u_{C_=}$ to $R(x_k)$, for every $x_k \in C_=$.

A more general analysis of these situations is based on solving a *set-covering* problem (or approximation thereof) for each invocation of step II.B (more details are provided in [PRSS98]). Experimental results have shown that due to this analysis, in most cases step II.B is not activated. Furthermore, condition *Con* alone identifies almost all of these cases without further analysis.

6.3 A new Upper Bound

Let us now revise our calculation of the upper bound of the state-space, taking into considerations the improvements to the basic algorithm. For a dashed connected component $G^k_=$, let $n_k = |G^k_=|$ and let $m_k = |mvc_k|$ (the number of individually assigned vertices in $G^k_=$). Also, let y_k denote the number of colors needed for coloring these m_k vertices (obviously, $y_k \leq m_k$).

When calculating the maximum state-space for the component $G^k_=$, there are three groups of vertices to consider:

1. For every vertex x_i s.t. $i \leq y_k, |R(x_i)| \leq i$. Altogether they contribute $y_k!$ or less to the state-space.
2. For every vertex x_i s.t. $y_k < i \leq m_k, |R(x_i)| \leq y_k$. Altogether they contribute $y_k^{m_k - y_k}$ or less to the state space.
3. For every vertex x_i s.t. $m_k < i \leq n_k, |R(x_i)| \leq y_k + 1$. Each of these vertices can not have more than y_k values when the m_k-th vertex is removed. Then, only one more value can be added to their R-set in step II.B (in fact, this additional element is rarely added, as was explained in the previous sub-section). Altogether these vertices contribute $(y_k + 1)^{n_k - m_k}$ or less to the state-space.

Combining these three groups, the new upper bound for the state-space is:

$$StateSpace \leq \prod_k (y_k!) \cdot y_k^{m_k - y_k} \cdot (y_k + 1)^{n_k - m_k} \quad (2)$$

The worst case, according to formula (2), is when all vertices are mixed ($G_= \equiv G_\neq$), there is one connected component ($n_k = n$), the minimal vertex cover is $m_k = m = n - 1$ and the chromatic number y_k is equal to m_k. Graphically, this is a 'double clique' (a clique where $G_= \equiv G_\neq$) which brings us back to $n!$, the upper bound that was previously derived in Section 3.

7 Experimental Results

As was mentioned in the introduction, we started investigating this problem due to our inability to use a BDD-based tool in order to validate formulas containing several hundred integer and floating-point variables.

The Range Allocation algorithm turned out to be very effective in our case. One of the reasons for this has to do with a module called *auto-decomposition* (described in [PSS99]) which our tool invokes before the range allocation is performed. If the right-hand side of the implication we try to prove is a conjunction of m clauses, then this module decomposes the implication up to m separate formulas. Each of these formulas consists of one clause in the right-hand side, and the *cone of influence* on the left (this is the portion of the formula in the left-hand side that is needed for proving the chosen clause on the right. In the formulas we considered it typically consisted of several hundred clauses). This kind of formulas, when described in terms of the Range Allocation algorithm, represent very unbalanced graphs: $G_=$ is relatively large (all the comparisons on the left-hand side with positive polarity belong to this graph) and $G_\neq$ is very small, resulting in a relatively small number of mixed vertices. This kind of graphs result in very small ranges, and many times a large number of variables receive a single value in their range and thus become constants. We have many examples of formulas containing 150 integer variables or more (which, using the $[1..n]$ range, results in a state-space of 150^{150}), which after performing the Range Allocation algorithm, can be proved in less than a second with a state-space of less than 100. In most cases, these graphs are made of many unconnected $G_=$ components with a very small number of $G_\neq$ edges. Altogether, the Range Allocation algorithm solved more than 99% of the formulas that we originally could not solve in reasonable time. The remaining percent includes large number of Boolean variables, which their effect on the state-space is not handled by our method, and highly condensed graphs (usually with large cliques). We are currently investigating techniques for tackling specific patterns in the graph in order to solve these last hard graphs.

We also tried to conduct a comparative study with [SGZ+98]. Although we had the same input files (the comparison between pipelined and non-pipelined microprocessors, as originally suggested by Burch and Dill [BD94]) as they did, it was nearly impossible to compare the results on this specific example, because of several reasons. First, all the examples examined in their work were easy to solve in fragments of a second by both methods. Thus, the small differences in performance are meaningless as they can be influenced by various other factors. Second, comparing the maximum BDDs required in both methods was also not reliable, since the differences were small and inconsistent. Given the fact that different BDD packages were used (VIS vs. TLV), we could not conclude anything from this factor either. The third reason is that the special BDD-traversing algorithm that is used in the Boolean-encoding method is worst case exponential. Therefore, it is reasonable to predict that on larger examples, this could become the bottle-neck in the verification process.

We predict that a comparison on harder problems will reveal that the two methods are complementary. While the Boolean encoding method is efficient when there is a small number of comparisons between input variables, the Range Allocation algorithm is more efficient when there is a small number of mixed vertices.

We noticed the lack of a good benchmark set for the kind of formulas we investigated. We therefore constructed a web site [BEN] with the examples we worked on, in addition to the case study of Burch and Dill [BD94], followed by the results we achieved. We hope that future research in this field will relate to this set, and additional hard problems will be added.

8 Conclusions and Directions for Future Research

We presented the Range Allocation method, which can be used as the first stage of a decision procedure based on finite instantiations, when validating formulas of the theory of equality. This method proved to be highly effective when we validated formulas with a large number of integer and float variables.

Although the described method does not guarantee optimal results, in almost all cases it gives significantly better results than simply taking the $n!$ upper bound. It is relatively simple and cheap in terms of complexity, easy to implement and enjoys the advantages of finite-instantiations that were mentioned in the introduction. There is no need to rewrite the verified formula, and any satisfiability checker can be used as a decision procedure.

The algorithm described in this paper is a simplified version of the full Range Allocation algorithm. The full algorithm includes several issues that were not discussed here mainly due to lack of space. These issues include several improvements to the algorithm itself, implementation issues and a revised algorithm for handling formulas that include constants. A more comprehensive description of the algorithm can be found in [PRSS98].

The Range Allocation algorithm can be improved in various ways, either based on the $G_=$ and $G_{\neq}$ graphs or not. For example, the *mvc* set is not unique, and the problem of choosing among *mvc* sets that have an equal size is still an open question. Furthermore, given an *mvc* set, the ordering in which the vertices in this set are removed in stage II/a should also be further investigated. It is easy to find examples where the heuristic suggested for this purpose (sorting the vertices by descending degree on $G_{\neq}$) is not optimal. It should also be noted that although the *mvc* set is helpful in restricting the upper-bound of the state-space, it cannot guarantee an optimal ordering. Another possible improvement is the identification of special kind of graphs. For example, the range [1..4] is enough for any *planar* graph (where $G_= \equiv G_{\neq}$). It should be rather interesting to investigate whether 'real-life' formulas have any special structure which can then be solved by utilizing various results from graph theory.

Another possibility for future research is to extend the algorithm to formulas with less abstraction, and more specifically to formulas including the GT, GE, LE and LT signs. The GT and LT signs can be represented as directed edges on the graph, and the GE and LE signs as two edges: a directed edge and a $G_=$ edge (according to their polarity). The definition of consistent subsets should be changed accordingly. For example, directed cycles are also inconsistent subsets (e.g. $\{x > y, y > z, z > x\}$). It seems that adjusting the other parts of the heuristic should not be difficult as well.

References

[Ack54] W. Ackerman. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. CAV'94*, lncs 818, pp 68–80.

[BEN] Benchmarks for satisfiability checking in the theory of equality with uninterpreted functions: www.summer.weizmann.ac.il/∼ofers/sat/bench.htm.

[BGV99] R. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *this volume*, 1999.

[Con95] The Sacres Consortium. Safety critical embedded systems: from requirements to system architecture, 1995. Esprit Project Description EP 20.897, URL http://www.tni.fr/sacres.

[HIKB96] R. Hojati, A. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. In *FMCAD'96*, pp 218 – 232.

[HKGB97] R. Hojati, A. Kuehlmann, S. German, and R. Brayton. Validity checking in the theory of equality using finite instantiations. In *Proc. Intl. Workshop on Logic Synthesis*, 1997.

[PRSS98] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. An efficient algorithm for the range minimization problem. *Technical report*, Weizmann Institute, Dec. 1998.

[PSS98b] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP98* lncs 1443, pages 235–246

[PSS99] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Intl. journal on Software Tools for Technology Transfer (STTT)*, vol 2, 1999.

[SGZ+98] K. Sajid, A. Goel, H. Zhou, A. Aziz, S. Barber, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A.J. Hu and M.Y. Vardi, editors, *CAV'98*, lncs 1427, pp 244–255.