The small model property: How small can it be? ⋆

Amir Pnueli[1] Yoav Rodeh[1] Ofer Strichman[1] Michael Siegel[2]

[1]Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Rehovot, Israel. Email: {amir—yrodeh—ofers}@wisdom.weizmann.ac.il
[2]Infineon Technologies AG, Otto-Hahn-Ring 6 D-81730 Munich, Germany.
Email: michael.siegel@infineon.com

Abstract. Efficient decision procedures for equality logic (quantifier-free predicate calculus + the equality sign) are of major importance when proving logical equivalence between systems. We introduce an efficient decision procedure for the theory of equality based on finite instantiations. The main idea is to analyze the structure of the formula, and compute accordingly a small domain to each variable such that the formula is satisfiable iff it can be satisfied over these domains. We show how the problem of finding these small domains can be reduced to an interesting graph theoretic problem. This method enabled us to verify formulas containing hundreds of integer and floating point variables that could not be efficiently handled with previously known techniques.

Keywords: Finite instantiation, Equality logic, Uninterpreted functions, Compiler verification, Translation validation, Range allocation.

1 Introduction

Proving logical equivalence between systems is of major interest in the formal methods arena. Proving such equivalence between two versions of a hardware design, or proving that an implementation corresponds to some specification or another, more abstract implementation, are typical examples of such comparisons. In some cases there is a need to prove a weaker claim, namely that each computation of the implementation system S_1, corresponds to a computation allowed by the specification system S_2. In this case we say that S_1 *refines* S_2. In our case, we tried to prove such a refinement relation between source and target code serving as the input and output of a compiler, and thus to verify that the compilation process was correct (see [11], [10] and [7] for more details about this project). The advantages of this approach, to which we refer as *translation validation* (TV), over simply verifying the compiler itself, are that in general the TV process is simpler, and it is semantics-dependent rather than product dependent. Hence, as long as the semantics of the input and output languages are unchanged, this approach is insensitive to changes in the compiler.

Equivalence/refinement proofs are roughly divided into two major steps. In the first step, one needs to construct a *verification condition* φ, which is simply a logical formula that is valid if and only if the two compared systems are equivalent. As a second step, the verification condition is validated via a *decision procedure.*

⋆ This research was supported in part by the John von Neuman 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. Parts of this paper appeared before in [PRSS99].

For obvious reasons, it is essential that the decision procedure used in the second step be fully automatic. The most prevalent decision procedures currently in use are based on the representation of formulas by the special data structure of Ordered Binary Decision Diagrams (OBDDs) [3]. OBDDs are directly applicable only to Boolean formulas. Formulas with variables ranging over a small finite domain can be encoded as Boolean formulas and thus enjoy the advantages of OBDDS. In our application we have many variables ranging over very large domains such as the integers or reals, thus special abstraction techniques are called for.

In addition to symbolic representation by OBDDs, there are other methods for proving equivalence efficiently. Theorem-provers such as PVS [12] typically solve this problem by computing the congruence closure over ground terms and uninterpreted functions (see below) [8]. However, this method is not very efficient in the presence of many disjunctions or If-Then-Else constructs, due to its tendency to split on each such branch. OBDDs are more efficient dealing with disjunctions due to their canonical structure. The verification condition associated with the correct compilation problem has a large number of these branching constructs, and therefore we chose an OBDD based decision procedure.

When proving equivalence, or refinement, it is often possible to abstract away all functions, except the equality sign and Boolean operators, by replacing them with *Uninterpreted Functions* (UIFs) (If-Then-Else constructs are left interpreted). The abstracted formula holds less information and therefore can be represented by a significantly smaller OBDD. It was Ackerman [1] who first showed the reduction of such abstracted formulas to function-free formulas of the theory of equality (i.e. Boolean combination of atomic equality formulas), 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 the formula $(z = F(x, y) \wedge u = y) \rightarrow z = F(x, u)$ 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 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 that 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 is conservative: it can not generate false positives, and therefore 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 finite 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 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 with respect to equality, will also falsify the formula. Since there cannot be more than n classes, the $[1..n]$ range is sufficient regardless of the formula's structure. We will later show that a smaller state space of $n!$ can be achieved by first ordering the variables, and then allocating the ranger $1..i$ to the i'th variable. We will also show that analyzing the formula's structure can lead to significantly smaller domains.

There were several attempts to cope with this problem before. Here we will mention two of them.

Bryant, German and Velev [4, 6] proposed two distinct approaches to this problem. In the first approach they suggest to examine the structure of the formula and distinguish a class of terms, called 'p-terms' (positive terms), which can be replaced by constants while preserving validity. These terms can only appear in equalities that do not appear under a logical negation or in the input cone of function applications. By replacing these terms with unique constants, the tested state-space is reduced immensely, which makes the decision procedure highly efficient. Other variables are given an increasing range of values, which leads in the worst case (when there are no p-terms) to a state space of $n!$. Practically, however, a significant part of many typical hardware designs can be expressed by p-terms, which implies that most variables are replaced by constants. Since our method reduces the domain of all variables, the two methods are complementary and can be combined: p-terms can be replaced by unique constants, while the domain of the other terms can be calculated by our method.

In the second approach, they encode each equality $x_i = x_j$ with a new Boolean variable e_{ij}, and maintain the transitivity of equality by adding appropriate constraints to the formula. The additional transitivity constraints should prevent an assignment to a cycle of size n s.t. exactly $n-1$ edges are assigned true. For example, given the cycle of size 3 $x_1 = x_2, x_2 = x_3, x_3 = x_1$, the constraints should prevent an assignment such that $e_{12} = e_{23} = \text{TRUE}$ and $e_{13} = \text{FALSE}$. Explicit enumeration of all chord-free cycles in the formula can be exponential, and they therefore suggest a preprocessing stage: they add edges s.t. the resulting graph is chordal (i.e. every cycle of size 4 or more has a chord). Since all chord-free cycles in such a graph are triangles, the number of cycles to be enumerated is bounded by $\binom{n}{3}$. Experiments have showed that this sparse enumeration method is more efficient than the first method. It also outperforms an earlier method suggested in [9], which also encodes equalities with Boolean variables, but preserves transitivity in a different way. Rather than adding constraints, they construct a BDD based on the encoded Boolean formula, and then restrict their search over this BDD to paths that are consistent with transitivity of equality. For example, a path with $e_{i,j} = e_{j,k} = \text{TRUE}$ and $e_{i,k} = \text{FALSE}$ is inconsistent. A comparative study showed that this method can not scale up [6].

There are two reasons why in spite of this success we believe that finite instantiation can perform better in many cases. First, in the worst case, where all variables are compared to each other, Boolean encoding results in a state space of $O(2^{n^2})$. For the same formula, allocating an increasing range of values results in a state space of $n!$, which is exponentially smaller. Second, the Boolean encoding method of [6] increases the formula itself by adding transitivity constraints. This is in contrast to our method that checks the original formula, hence transitivity is preserved implicitly without adding constraints.

The verification conditions we considered 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 on 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 OBDD-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 many useful contexts. 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 extremely 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 while constructing CVT [11, 10], a verification tool that proves refinement between source and target code of a compiler. Although the same problem was mentioned in the past usually in the context of proving equivalence between hardware designs, our running example will follow a typical formula from the former domain.

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

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

where '+' and '*' have been abstracted into F and G, respectively. Clearly, if the abstracted version is valid then so is the original concrete one.

Next, we perform the Ackerman reduction [1], 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:

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

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.

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

$$\mathcal{T} \quad ::= \quad \textit{integer constant} \;\mid\; x_i \;\mid\; \textbf{if } \Phi \textbf{ then } \mathcal{T}_1 \textbf{ else } \mathcal{T}_2$$

The set of equality formulas Φ is defined by

$$\Phi \quad ::= \quad b_j \;\mid\; \neg\Phi \;\mid\; \Phi_1 \vee \Phi_2 \;\mid\; \mathcal{T}_1 = \mathcal{T}_2 \;\mid\; \textbf{if } \Phi_0 \textbf{ then } \Phi_1 \textbf{ else } \Phi_2$$

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

3 The Solution: Instantiations over Small Domains

Our solution strategy for checking whether a given equality formula φ is satisfiable can be summarized as follows:

1. Determine, in polynomial time, a *range allocation* $R : Vars(\varphi) \mapsto 2^{\mathbb{N}}$, by mapping each integer variable $x_i \in Vars(\varphi)$ into a small finite set of integers $R(x_i)$, such that φ is satisfiable (valid) iff it is satisfiable (respectively, valid) over some R-interpretation (i.e. an interpretation in which each variable x_i is assigned an integer from $R(x_i)$).
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 $Vars(\varphi) = \{x_1, \ldots, x_n\}$, then the complexity of R is $|R| = |R(x_1)| \times |R(x_2)| \times \cdots \times |R(x_n)|$. 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 $R^* : x_1, \ldots, x_n \mapsto \{0\}$.

Thus, the answer to the question posed in our title is: 1. 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.[1]

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 is given in a positive form (sometimes called Negation Normal 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*. This means that if S_1 and S_2 are two consistent subsets of atomic formulas of φ (where φ is given in positive form), such that $S_1 \subseteq S_2$, and $\hat{S}_i = \bigwedge_{\psi \in S_i} \psi$, then $\hat{S}_1 \models \varphi$ implies $\hat{S}_2 \models \varphi$. Any equality formula can be brought into a 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 $At(\varphi)$ 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 $x_1{=}x_2 \wedge x_1{\neq}x_2$.

The following lemma constitutes the monotonicity of adequacy with respect to sets of atomic formulas.

Lemma 1. *Let* $\Phi(A)$ *be the set of formulas* φ *such that* $At(\varphi) = A$. *If* R *is adequate for* $\Phi(A)$ *and* $B \subseteq A$, *then* R *is adequate for* $\Phi(B)$.

Proof. Let $\Delta = A - B$ and let ψ be an an arbitrary formula such that $\psi \in \Phi(B)$. Construct $\psi' = \psi \wedge \bigwedge_{\delta \in \Delta}(\delta \vee \neg\delta)$. Clearly ψ is satisfiable iff ψ' is satisfiable. Since $\psi' \in \Phi(A)$ and R is adequate for $\Phi(A)$, then ψ' is satisfiable iff it is R-satisfiable. By construction, if ψ' is R-satisfiable, then ψ is also R-satisfiable. So our chain is: ψ is satisfiable $\leftrightarrow$ ψ' is satisfiable $\leftrightarrow$ ψ' is R-satisfiable $\rightarrow$ ψ is R -satisfiable. Thus, since ψ is an arbitrary formula belonging to $\Phi(B)$, R is adequate for $\Phi(B)$. □

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 one of the following two patterns:

[1] It can easily be shown that if in addition there are k constants in Φ_n, and c_{max} is the largest constant, then the range allocation $R : x_i \mapsto [c_1..c_k, c_{max}+1..c_{max}+1+i]$ is adequate and results in a complexity of $(k+n)!/k!$.

1. A chain of the form $x_1 = x_2, x_2 = x_3, \ldots, x_{r-1} = x_r$ together with the formula $x_1 \neq x_r$.
2. A chain of the form $x_1 = x_2, x_2 = x_3, \ldots, x_{r-1} = x_r$ where x_1 and x_r represent different constants.

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 $R: x_1, x_2, x_3 \mapsto \{0\}$ is satisfactory for the atomic formula set $\{x_1 = x_2, x_2 = x_3\}$, while the allocation $R: x_1 \mapsto \{1\}, x_2 \mapsto \{2\}, x_3 \mapsto \{3\}$ 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 $R: x_1 \mapsto \{1\}, x_2 \mapsto \{1, 2\}$.

Claim 2 *The range allocation R is satisfactory for the atomic formula set A iff R is adequate for $\Phi(A)$.*

Proof. ($\Rightarrow$) Falsely assume that there exists $\varphi \in \Phi(A)$ such that φ is satisfiable but not R-satisfiable. Let B be the subset of atomic formulas in φ that are evaluated to TRUE in a satisfying assignment of φ. Clearly B is consistent and $B \subseteq A$. Therefore since R is satisfactory for A, then B is R-satisfiable. But if B is R-satisfiable then φ is also R-satisfiable, a contradiction.

($\Leftarrow$) Falsely assume that R is adequate for $\Phi(A)$ but there exists a consistent subset B, $B \subseteq A$, such that B is not R-satisfiable. According to Lemma 1, since R is adequate for $\Phi(A)$ then it is also adequate for $\Phi(B)$. Let $\varphi = \bigwedge_{\psi \in B} \psi$. Since B is consistent then φ is satisfiable. From the fact that $\varphi \in \Phi(B)$ and that R is adequate for $\Phi(B)$, we conclude that φ is also R-satisfiable, which, by the construction of φ, contradicts our assumption that B is not R-satisfiable. $\square$

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). Other 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:

$$\neg\varphi: \quad \begin{pmatrix} (x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \end{pmatrix} \wedge \ z \neq g_2,$$

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 $A_=$ and $A_{\neq}$ 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 G represents a variable. Vertices representing mixed variables are called *mixed vertices*. We will use dashed edges to represent $G_=$-edges and solid ones to represent $G_{\neq}$-edges.

An inconsistent subset $B \subseteq A$ will either contain a dashed path between two constants or 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 graph G corresponding to the formula $\neg\varphi$. Note that there are three contradictory cycles in this graph: $(g_2 - g_1 - z), (u_1 - f_1)$ and $(u_2 - f_2)$.

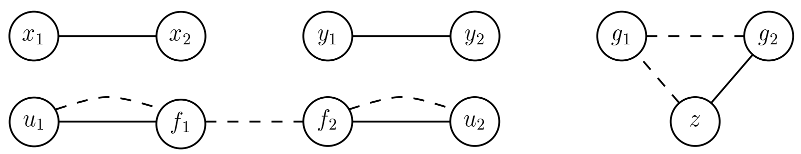

Fig. 1. The Graph $G : G_{\neq} \cup G_=$ representing $\neg\varphi$

Let us now give a self-contained definition of the range-allocation graph-theoretic problem. In the next section we will suggest a heuristic for solving it.

The ***Range-Allocation*** **problem** We are given a graph $G(V, E, D)$ with two types of edges, E and D, where the vertices represent either variables or constants. If $(x_i, x_j) \in E$, we say that there is an equality constraint between x_i and x_j. Similarly, if $(x_i, x_j) \in D$, we say that there is a disequality constraint between these variables. We mark E edges with dashed lines, and D edges with solid lines.

We say that a subset of edges $B \in E \cup D$ is *consistent* if it does not contain a cycle of size $k > 1$, consisting of one solid edge $(x_i, x_j) \in D$ and $k-1$ edges from E, and it does not contain a dashed path between two constants. We denote the vertices connected by the edges in B by $\overline{B}$.

Given G, we wish to associate for each vertex x_i a set of integers $R(x_i)$, such that for all consistent subsets $B \in E \cup D$, there is a possible assignment of values from $R(x_i)$ to x_i for all $x_i \in \overline{B}$, that preserves all the constraints imposed by the edges in B.

The problem is to find in polynomial time the sets $R(x_i)$ such that their state-space $|R|$ is minimal ($|R| = \prod_{i=1..|V|} |R(x_i)|$). □

Note that if all subsets B are consistent, it is immediate to find an allocation R s.t. $|R| = 1$: first, allocate a unique value to each connected component in $G_=$, and assign this value to all the variables in this component. Then, assign a unique value to each node in $G_{\neq} - G_=$.

4 The Basic Range Allocation Algorithm

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

I. Eliminating constants and pre-processing

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

A. For each constant-vertex c_i in G do:
   1. Assign $R(c_i) := c_i$ [2].
   2. Assign $R(x_j) := R(x_j) \cup \{c_i\}$ for each vertex x_j, s.t. there is a $G_=$-path from c_i to x_j not through any other constant-vertex.
   3. Remove c_i from the graph.

B. Remove all $G_{\neq}$ edges which do not lie on a contradictory cycle.

C. For every singleton vertex (a vertex comprising a connected component by itself) x_i, add to $R(x_i)$ a fresh value u_i. 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 $x_k \in C_=$.

[2] Obviously there is no need to allocate a range for constants. This step is presented for the sake of symmetry with step II.A.1.

We refer to the values that were added in steps I.A.1, I.C, II.A.1 and 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 that are assigned their characteristic values in steps I.A.1, I.C 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. Consequently, we require that all fresh values are larger than the largest constant C_{max}. This assumption is necessary only for simplifying the proof in later sections.

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. Since the values given in this step are distributed on the $G_=$ graph in step II.A.2, we would like to keep this set as small as possible. Furthermore, we would like to partition the graph fast, in order to limit this distribution. A rather simple, yet effective heuristic for this purpose is to choose vertices according to a "greedy" criterion, where mixed vertices are chosen in the order of their (descending) degree in $G_{\neq}$. Among vertices with equal degrees in $G_{\neq}$, we choose the one with the highest degree on $G_=$. We expect that further research will reveal other, more efficient ordering methods.

Note that, excluding some special cases, 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. Consequently, minimizing this set corresponds to the well known Minimal Vertex Cover problem, and we therefore choose to denote this set from now on by *mvc*.

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

For demonstration purposes, consider φ where g_1 is replaced by the constant '3'. In this case the component $z - g_1 - g_2$ will be handled as follows: in step I.A '3' will be added to $R(g_2)$ and $R(z)$. The edge $z - g_2$ will then be removed in step I.B and a distinct fresh value will be added to each of these variables in step I.C.

□

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 $a_B(x_i) \in R(x_i)$. We then continue by proving that this assignment satisfies B.

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 $a_B(x_i) \in R(x_i)$, according to the following rules:

R1 If x_i is connected by a (possibly empty) $G_{=}(B)$-path to an individually allocated vertex x_j, assign to x_i the minimal value of $char(x_j)$ among such x_j's.
R2 Otherwise, assign to x_i its communally assigned value $char(x_i)$.

To show that all vertices are assigned a value by this procedure, we observe that every vertex is allocated a characteristic value before it is removed. It can be an individual characteristic value in steps I.A.1, I.C and II.A.1, or a communal value allocated in step II.B. Every vertex x_i that has an individual characteristic value can be assigned a value $a_B(x_i)$ by **R1**, because it has at least the empty $G_{=}$-path leading to an individually allocated vertex, namely itself. All other vertices are allocated a communal value that makes them eligible to a value assignment by **R2**.

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 and 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 **R1**, 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 **R1**, 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.C.
- By **R1**, g_2 is assigned the value $char(g_2) =$'6' which it received in step II.A.
- By **R2**, z and g_1 are assigned the value '9' which they received in step II.B. □

Claim 3 *The assignment procedure is feasible (i.e. the value assigned to a node by the procedure belongs to its R-set).*

Proof. Consider first the two classes of vertices that are assigned a value by **R1**. 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 by **R2**. Every vertex that is removed in step I.C or II.A is clearly assigned a value by **R1**. 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 4 *If B is a consistent set then the assignment a_B satisfies B.*

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

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 $a_B(x_i) = a_B(x_j)$. 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 **R1**, 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 **R2**, 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. To show that $a_B(x_i) \neq a_B(x_j)$, we distinguish between three cases:

- If both x_i and x_j were assigned values by **R1**, 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 by **R1** while the other acquired its value by **R2**, then since any communal value is distinct from any individually allocated value, $a_B(x_i)$ must differ from $a_B(x_j)$.
- The remaining case is when both x_i and x_j were assigned values by **R2**. The fact that they were not assigned values in **R1** implies that their characteristic values are not individually but communally allocated. Falsely assume that $a_B(x_i) = a_B(x_j)$. This 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 (moreover, this path was still part of the graph in the beginning of step II.B). 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. According to the loop condition of this step, in the end of this step there are no mixed vertices left, which rules out the possibility that (x_i, x_j) was still part of the graph at this stage. Thus, at least one of these vertices was individually allocated in step II.A.1, and consequently, the component it belongs to is assigned a value by **R1**, in contrast to our assumption. □

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

In this section we present several improvements to the basic algorithm, which can significantly decrease the size of the resulting state-space.

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. This reduces the state space of the entire graph from 48 to 16.

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 $x_1, \ldots, x_m$. Also 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 constraints imposed by the subset B, only if there exists a path of the form:

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

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

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.D 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 the constants and the variables $x_1, \ldots, x_m$. Add an edge between x_i and x_r iff x_i and x_r are incompatible.
3. Find a minimal coloring for G_{inc} (refer to the constants as vertices with pre-determined color), 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.D.3 as the characteristic value of x_i.

Like the case of minimal vertex covering, step I.D.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 polynomial approximation 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 $R_3 : u_1 \mapsto \{0, \underline{1}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{\underline{0}\}, u_2 \mapsto \{0, \underline{1}\}$ □

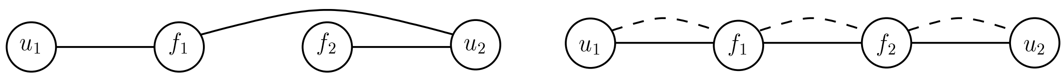

Fig. 2. (a) The Graph G_{inc} contains an edge between every pair of incompatible vertices (b) Illustrating selective assignments

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). $\square$

Assume that $C_=$ is a remaining connected $G_=$ component with no mixed vertices. It is not hard to see that the range allocated for all variables in $C_=$ is the same before the application of step II.B. We denote this common range by $R(C_=)$. Let $y_1 \dots y_k \notin C_=$ be all the vertices that 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*: $k < |R(C_=)|$, or $R(C_=) - \bigcup_{i=1}^{k} R(y_i) \neq \emptyset$.

When condition *Con* holds, we can always assign in **R2** a common value to the vertices in $C_=$ that is different than the values assigned to $y_1, \dots, y_k$.

Example 7. Consider the component $\{u_2\}$ in the graph of Fig. 2(b). $R(u_2) = \{0, 1\}$ with $|R(u_2)| = 2$, while $\{u_2\}$ has only one $G_{\neq}$-neighbor: f_2. Consequently, we can skip the assignment of the fresh value '2' to u_2. $\square$

Let us now generalize this analysis. For this purpose we define the *restricted set-cover* problem:

> Given a multiset of sets $\mathcal{S}$ and a set $\mathcal{U}$, does there exist a cover that assigns to each $u \in \mathcal{U}$ a unique $S_u \in \mathcal{S}$, such that $u \in S_u$ and for every $v \neq u \in \mathcal{U}$, $S_u \neq S_v$?

Note that S_u and S_v may be identical sets, but it is required that they are distinct members of the multiset $\mathcal{S}$. Clearly if $|\mathcal{U}| > |\mathcal{S}|$ or if $\exists v \in \mathcal{U}$ s.t. $\forall i.\ v \notin S_i$, then the answer is negative. These are two easy preliminary tests for this problem. If we associate $\mathcal{S}$ with the multiset $\{R(y_1), \dots, R(y_k)\}$, and $\mathcal{U}$ with the set $R(C_=)$, then condition *Con* reflects exactly this preliminary test. However, if this test fails, a more exhaustive search is called for. Since the number of solid neighbors of each dashed component and their associated R-sets is relatively small, this does not present a serious computational problem.

To incorporate selective assignments into the algorithm, we modify step II.B of the basic algorithm to read as follows:

B. For each (remaining) connected $G_=$ component $C_=$, if there is a restricted set cover of $R(C_=)$ by $\{R(y_1), \dots, R(y_k)\}$, where $y_1, \dots, y_k$ are the $G_{\neq}$ neighbors of $C_=$, add a common fresh value $u_{C_=}$ to $R(x_k)$, for every $x_k \in C_=$.

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.

Finally, as a result of this change, we also modify the assignment procedure as follows: Given a consistent subset B and its representative graph $G(B)$, assign values to the vertices in $G(B)$ according to the following two (ordered) rules:

R1' For all $x_i \in G(B)$ s.t. x_i is connected by a (possibly empty) $G_=(B)$-path to an individually allocated vertex x_j, assign to x_i the minimal value of $char(x_j)$ among such x_j's.

R2' For each $G_=(B)$ component $C_=$ whose vertices have not been assigned a value, choose a value z, such that $z \neq a_B(y_i)$ for all vertices $y_i \neq C_=$ that are $G_{\neq}$ neighbors of $C_=$, and assign z to all $x_i \in C_=$.

We now make the following claim:

Claim 6 *The revised procedure is sound.*

Proof. Assume that $C_=$ is a remaining connected $G_=$ component with no mixed vertices. As stated before, the range allocated for all variables in $C_=$ is the same before the application of step II.B. Let x be a vertex in $C_=$ and let $y_1 \dots y_k \notin C_=$ be all the vertices that are $G_{\neq}$-neighbors of vertices in $C_=$. By definition, if there is no restricted set cover of $R(x)$ by $\{R(y_1), \dots, R(y_k)\}$, then the following holds: under every assignment α to $y_1 \dots y_k$, there exists a value in $R(x)$ that is different than $\alpha(y_1) \dots \alpha(y_k)$. More formally:

$$\forall \alpha(y_1), \dots, \alpha(y_k).\ \exists v \in R(x).\ v \neq \alpha(y_1) \wedge \dots \wedge v \neq \alpha(y_k)$$

Since all vertices in $C_=$ have the value v in their range, then any consistent component B that contains vertices from $C_=$ and a subset of $y_1 \dots y_k$ can be satisfied. Hence, the algorithm remains sound. $\square$

6.3 An Upper Bound

We present an upper bound for the size of the state-space computed by our algorithm, which is better, in most cases, than the naive bounds presented in Section 3.1. 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, $y_k \leq m_k$), and let c_k denote the number of constants in $G^k_=$.

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

1. Vertices $x_1, \dots, x_{y_k}$, such that x_i is the first vertex to be individually assigned color i. For these vertices, $|R(x_i)| \leq c_k + i$ and together they contribute $\frac{(c_k+y_k)!}{c_k!}$ or less to the state space.
2. The rest of the individually assigned vertices $x_{y_k+1}, \dots, x_{m_k}$. For these vertices, $|R(x_i)| \leq c_k+y_k$, and together they contribute $(c_k + y_k)^{(m_k-y_k)}$ or less to the state space.
3. The remaining vertices $x_{m_k+1}, \dots, x_{n_k}$ that, in the worst case, will be assigned an additional common value. For these vertices, $|R(x_i)| \leq c_k + y_k + 1$ and together they contribute $(c_k + y_k + 1)^{n_k-m_k}$ or less to the state-space.

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

$$|R| \ \leq \ \prod_k \frac{(c_k + y_k)!}{c_k!} \cdot (c_k + y_k)^{m_k-y_k} \cdot (c_k + y_k + 1)^{n_k-m_k} \tag{2}$$

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 $G_= \equiv G_{\neq}$) that results in a state space of $\frac{(c+n)!}{c!}$. If $c = 0$, the state space is $n!$, which is 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 an OBDD-based tool in order to validate formulas containing several hundred integer and floating-point variables.

The Range Allocation algorithm proved to be very effective in our case. One of the reasons for this has to do with a module called *auto-decomposition* (described in [10]) that 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 type of graph results 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 with 150 integer variables or more, 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 consist of many disconnected $G_=$ components with a very small number of $G_{\neq}$ edges.

We also solved these instances with the methods described in [4, 6], which are implemented in the tool Uclid [2]. Fig. 3 summarizes our experiments. Uclid lets the user combine these methods in different ways. As default, it combines the positive equality reduction of [4] and the Boolean encoding of [6]. More specifically, it replaces all p-terms (see section 1) with unique constants, and encode all other equalities, namely the g-terms, with Boolean variables. This combination, empirically, is faster on average than each of these methods by itself. The column titled UCLID#2 describes the results achieved by this combination with our benchmark files. The column UCLID#1 describes results achieved by using the Boolean encoding method of [6] alone, without the positive equality reduction.

As can be seen in the table, range allocation is generally more efficient than both Uclid#1 and Uclid#2. We should mention, however, that the above methods cannot handle the integer constants that are present in our formulas. We defined constants in Uclid by declaring a variable $Zero$ and then declaring each constant c as c applications of the successor function over $Zero$. Uclid has special simplification rules for terms with successor expressions, that their description is beyond the scope of this article. In any case we believe that this could not skew the results significantly because there is a relatively small number of such terms in our formulas.

It is always hard to determine the superiority of one method over another when both have the same theoretical complexity. We mentioned in the introduction some of the reasons that encouraged us to try this approach (it does not impose additional constraints on the formula, and it has a smaller state-space in the worst case). Although the experimental results support this view, we do not claim that it proves that range allocation is better in general. We developed and optimized range allocation for solving this type of formulas, so it is perhaps not surprising that this method was more efficient than others in solving them. Only a small number of terms in these formulas are p-terms, and hence Uclid could not benefit much from its main source of efficiency. A separate series of experiments[5] indicated that formulas derived from symbolically simulating hardware circuits tend to have a much larger portion of p-terms, which makes them much easier to solve with the

Instance	Range Allocation	Uclid#1	Uclid#2
22	0.16	*	*
25	0.2	*	*
27	1.7	170	20.4
32	0.1	1074	1020
37	0.15	2.5	1
38	0.18	0.5	0.5
43	*	*	*
44	0.1	*	*
46	0.13	*	*
49	*	*	*

Fig. 3. Results in seconds, comparing range allocation to two alternative decision procedures. Asterisks (*) represent run times exceeding two hours.

method of [4]. It seems that the best strategy is to combine these methods, i.e. to use positive equality reduction on p-terms and range allocation on the others. This is a typical conclusion when estimating decision procedures for formal verification.

Some of the problems we were working on are now available on a web site [13] for benchmarking purposes. 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 for validating formulas with a large number of integer and floating point variables. The method is relatively simple and easy to implement and apply. There is no need to rewrite the verified formula, and any satisfiability checker can be used as a decision procedure. The experiments show that it is more effective than competing methods, at least for the type of formulas that we experimented with.

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, two known results in graph theory can be applied in the case of *planar* graphs: every planar graph can be colored with 4 colors, and planar graphs are closed under contraction. Without proving it, we claim that these two results imply that the range [1..4] is adequate for any planar graph, regardless of the types of edges. It should be rather interesting to investigate whether 'real-life' formulas have any special structure that 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 order constraints, such as $x_i > x_j$. The $>$ relation can be represented as directed edges on the graph, while the weak ordering $\geq$ can be represented by a combination of a $>$ edge together with a $G_=$ edge. 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

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. 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 E. Brinksma and K.G. Larsen, editors, *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 78–91, Copenhagen, Denmark, July 2002. Springer-Verlag.
3. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
4. R.E. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc.* 11^{th} *Intl. Conference on Computer Aided Verification (CAV'99)*, 1999.
5. R.E. 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.
6. R.E. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. In E.A. Emerson and A.P. Sistla, editors, *Proc.* 12^{th} *Intl. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*. Springer-Verlag, 2000.
7. 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.
8. D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.
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.J. Hu and M.Y. Vardi, editors, *CAV98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
10. A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Int. J. of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1998.
11. A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc.* 25^{th} *Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *LNCS*, pages 235–246. Springer-Verlag, 1998.
12. N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.
13. O. Strichman. Benchmarks for satisfiability checking in the theory of equality with uninterpreted functions:. www.cs.cmu.edu/~ofers/sat/bench.htm.