Finding All Minimal Unsatisfiable Subsets

Maria Garcia de la Banda
School of Comp. Sci and Soft. Eng
Monash University, 3800
Australia
mbanda@csse.monash.edu.au

Peter J. Stuckey, Jeremy Wazny
Dept of Comp. Sci and Soft. Eng
University of Melbourne, 3010
Australia
{pjs,jeremyrw}@cs.mu.oz.au

ABSTRACT

An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. A number of forms of error diagnosis, including circuit error diagnosis and type error diagnosis, require finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error. In this paper we give algorithms for efficiently determining all minimal unsatisfiable subsets for any kind of constraints. We show how taking into account notions of independence of constraints and using incremental constraint solvers can significantly improve the calculation of these subsets.

Categories and Subject Descriptors

I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms

Algorithms

Keywords

Minimal unsatisfiable subsets, constraint solving

1. INTRODUCTION

A set of constraints is unsatisfiable if it has no solution. An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. A number of forms of error diagnosis, including circuit error diagnosis and type error diagnosis, require finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error.

There is a significant amount of work that deals with minimal unsatisfiable sets, particularly in the areas of explanation (e.g. [9]), intelligent backtracking (e.g. [5, 3, 4]) and nogood creation (e.g. [13]). However, the vast bulk of this work is only interested in finding a single minimal unsatisfiable subset. This is usually achieved by relying on some kind of justification recording, and then post-processing the recorded unsatisfiable set to eliminate unnecessary constraints. In many cases a non-minimal unsatisfiable subset is used.

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

Our motivation for examining the problem of finding all minimal unsatisfiable subsets of a set of constraints arises from type error debugging. In Hindley-Milner type inference and checking, a program is mapped to a system of Herbrand constraints and a type error results when this system is unsatisfiable. An explanation of the type error is given by a minimal unsatisfiable subset of the Herbrand constraints

Example 1. Consider the following fragment of Haskell code

```
f [] y = []
f (x:xs) y = if (x < y) then (f xs y) else xs
g xs y = 'z' > (f xs y)
```

which defines a function `f` that returns a list, and then erroneously compares the result of that function to character `'z'`. The Chameleon type debugging system [14] finds a single minimal unsatisfiable set of constraints that causes the type error and underlines the associated program fragments. If that minimal unsatisfiable set included the constraints posed by the base case in the definition of function `f`, the Chameleon system would show the following

```
f [] y = []
f (x:xs) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

while if the minimal unsatisfiable set included the constraints posed by the recursive case in the definition of function `f`, the Chameleon system would instead show the following

```
f [] y = []
f (x:xs) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

Some explanations may be easier to understand than others if, for example they involve fewer constraints. Hence, deriving all minimal unsatisfiable subsets allows us to choose the "simplest" explanation.

A similar problem arises in circuit error diagnosis. Using the methodology of Reiter [11], all minimal unsatisfiable constraint sets need to be determined in order to locate the minimal sets of components that could be acting abnormally.

Example 2. Consider the full adder circuit shown in Figure 1 where G1 and G3 are AND gates, G2 and G4 are

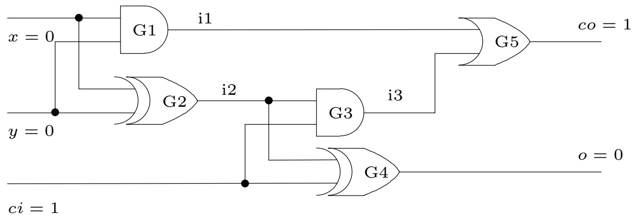

Figure 1: A full adder with observations illustrating erroneous behaviour.

XOR gates, and G5 is an OR gate. The observations of the inputs and outputs show that the circuit is not acting correctly. There are two minimal unsatisfiable subsets involving the constraints of the circuit, one involving the constraints of gates $G2$ and $G4$, and another involving the constraints of gates $G1$, $G2$ and $G3$.

Minimal unsatisfiable sets are used to obtain *diagnoses*, i.e., minimal sets of gates which if working abnormally (with all other gates working normally) would explain the behaviour. Diagnoses are obtained by determining minimal sets of gates that intersect with those appearing in the constraints of every minimal unsatisfiable subset. For the example, the diagnoses for the circuit in Figure 1 are $\{G2\}$, $\{G1, G4\}$ and $\{G3, G4\}$.

Finding all minimal unsatisfiable subsets of a system of constraints is a challenging problem because, if done naively, it involves examining every possible subset. Previous work has concentrated on its use in diagnosis of circuit errors. Hou [8] explored how to avoid visiting all subsets, but gave an erroneous pruning rule that sometimes missed answers. Han and Lee [7] corrected this algorithm, added pruning rules that reduced the number of subsets required to be explored, and also noted how to eliminate satisfiability checks in some cases.

This paper investigates how to improve the calculation of all minimal unsatisfiable subsets. It shows how to

- use preprocessing steps to reduce the size of the set of constraints we need to explore,
- reason about independence and redundancy of constraints to reduce the size and number of sets of constraints we must explore, and
- use incremental satisfaction to more quickly drive the search toward satisfiable subsets, reducing the number of sets of constraints we must explore.

Using all these techniques together allows us to solve much larger problems than previous techniques.

The remainder of the paper proceeds as follows. We first give some background definitions in Section 2. Next, we examine the best previous approach to the problem we are aware of, that of Han and Lee, in Section 3. In Section 4 we show how we can improve this approach using preprocessing, independence and incrementality. In Section 5 we show the results of experiments comparing different optimizations. Finally, in Section 6 we conclude.

2. BACKGROUND

Let us start by introducing the notation which will be used herein. A constraint domain $\mathcal{D}$ defines the set of possible values of variables. A *valuation* θ, written $\{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$, $d_i \in \mathcal{D}, 1 \leq i \leq m$, maps each variable v_i to a value d_i in the domain.

A *constraint* c is a relation on a tuple of variables $vars(c)$. Let $vars(c) = (v_{i_1}, \ldots, v_{i_n})$ then c defines a subset $vals(c)$ of D^n. A valuation $\theta \equiv \{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$ is a *solution* of constraint c if $(d_{i_1}, \ldots, d_{i_n}) \in vals(c)$.

We often treat $vars(c)$ as a set (rather than a tuple) of variables. If C is a set of constraints then $vars(C) = \cup_{c \in C} vars(c)$. A valuation θ is a solution of a set of constraints C if θ is a solution of each constraint $c \in C$.

The *size*, $||c||_c$, of a constraint c is $|vars(c)|$. The size $||C||_c$ of a constraint set C, is $\sum_{c \in C} ||c||_c$. The size of a set of sets $\mathbf{A}$, denoted $||\mathbf{A}||$ is $\sum_{A \in \mathbf{A}} |A|$.

A set of constraints C is *satisfiable* iff there exists a solution θ of C. Otherwise it is *unsatisfiable*. We assume an algorithm $sat(C)$ which returns $true$ if C is satisfiable and $false$ otherwise. A constraint set C is a *minimal unsatisfiable* constraint set if C is unsatisfiable and each $C' \subset C$ is satisfiable.

We will also be interested in *incremental* satisfaction algorithms. Incremental satisfiability checks constraints one at a time. Hence, to answer the question $sat(\{c_1, \ldots, c_n\})$ we compute the answers to questions $sat(\{c_1\})$, $sat(\{c_1, c_2\})$, $\ldots$, $sat(\{c_1, \ldots, c_{n-1}\})$ and finally $sat(\{c_1, \ldots, c_n\})$. We describe incremental satisfiability algorithms as a procedure $isat(c_n, state)$ which takes a new constraint c_n and an internal state representing a set of constraints $\{c_1, \ldots, c_{n-1}\}$, and returns a pair $(result, state')$ where $result = sat(\{c_1, \ldots, c_n\})$ and $state'$ is a new internal state representing constraints $\{c_1, \ldots, c_n\}$.

We will use two different example constraint domains. The first, $\mathcal{H}$, deals with Herbrand equations, that is, equations over uninterpreted function symbols, such as the constraint arising in Hindley-Milner typing. The complexity of $sat(C)$ for this class of constraints is $O(n)$ where n is the number of symbols in the constraint C [10]. The amortized incremental complexity of $isat(c, state)$ is $O(a^{-1}(n))$ where a^{-1} is the inverse Ackerman function, thus effectively constant.

The second constraint domain, $\mathcal{B}$, deals with Boolean constraints, that is terms made up using the Boolean operators 1 (true), 0 (false), $\neg$ negation, $\wedge$ (conjunction) $\vee$ disjunction, $\oplus$ (exclusive or), $\Rightarrow$ (implication), and $\Leftrightarrow$ (bi-implication). These are the constraints arising in circuit diagnosis. The best known complexity for sat for $\mathcal{B}$ is exponential, it is of course the classic NP-complete problem SAT. The complexity of $isat$ is also exponential. Approaches to incremental satisfiability of Boolean constraints which are efficient in practice, are possible using Reduced Ordered Binary Decision Diagrams (ROBDDs) [2].

3. BASIC ALGORITHM

The problem we address is the following: given a set of constraints S, find all minimal unsatisfiable subsets of S. The idea is, of course, to minimize the amount of work required (by reducing the number of sets, their size, etc) without missing any minimal unsatisfiable subset.

The problem is inherently difficult, even for very simple

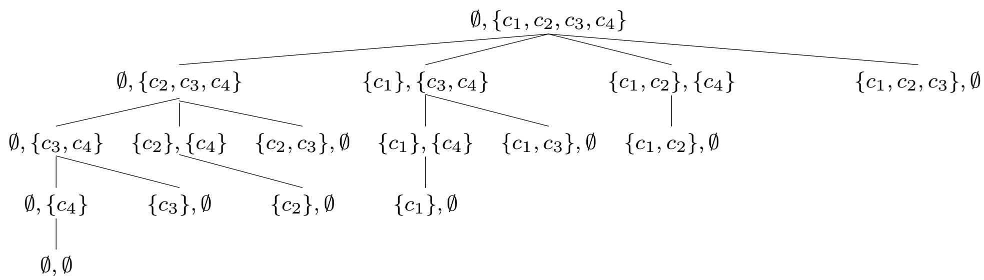

Figure 2: Visiting all subsets of $\{c_1, c_2, c_3, c_4\}$

sat satisfiability checks. This is because it has (at least) exponential complexity due to the exponential number of minimal unsatisfiable subsets.

Example 3. Consider the set of Herbrand equations

$$\{x_0 = a, x_n = b\} \cup \bigcup_{0 \leq i < n} \{x_i = u_i, u_i = x_{i+1}, x_i = v_i, v_i = x_{i+1}\}$$

Then the minimal unsatisfiable subsets are exactly those of the form

$$\{x_0 = a, x_n = b\} \cup \bigcup_{0 \leq i < n} \{x_i = y_i, y_i = x_{i+1} \mid y_i = u_i \text{ or } y_i = v_i\}$$

of which there are clearly an exponential number.

In [7] Han and Lee define a method for deriving all minimal unsatisfiable sets of a given set S based on an ordered traversal of the subsets of S. The method relies on two components. The first is the *CS-tree*, a tree in which nodes represent subsets of S and no two nodes represent the same subset. Thus, the CS-tree defines an order among the subsets which guarantees that the same subset will never be tested twice.

To achieve this, each set S labeling a node is broken into two parts: D, the elements which must appear in all subsequent subsets (and will therefore appear in all its descendants and right-sibling nodes), and P the elements that do not need to appear in all subsequent subsets (and will therefore be used to obtain children nodes with different labels). A node is labeled by a pair of sets D, P which represents the subset $D \cup P$. This node has $|P|$ children, obtained by eliminating one by one the constraints in P and adding them to the D part of the rightmost sibling, i.e., the first child node will be labeled by $D \cup (P - \{c_1\})$, the second by $(D \cup \{c_1\}) \cup P - \{c_1, c_2\}$, the third by $(D \cup \{c_1, c_2\}) \cup P - \{c_1, c_2, c_3\}$, etc.

Example 4. Figure 2 shows the CS-tree for the set $S = \{c_1, c_2, c_3, c_4\}$ which has root node $\emptyset, \{c_1, c_2, c_3, c_4\}$.

It is easy to see that given a CS-tree labeled by $\emptyset, S$, the tree contains all proper subsets of S without duplicates.

The following pseudo-code defines function $\mathsf{all\_subsets}(D, P, \emptyset)$ which visits all nodes of a CS-tree with root D, P, and returns the set of all sets of the form $D \cup X$ where $X \subseteq P$ in $\mathbf{A}$. The CS-tree is traversed in a depth-first manner with each iteration of the while loop visiting a child of the root and each inner call to `all_subsets` visiting the descendants of that child.

$\mathsf{all\_subsets}(D,\ P,\ \mathbf{A})$
  $\mathbf{A} := \mathbf{A} \cup \{D \cup P\}$
  while $\exists c \in P$
    $P := P - \{c\}$
    $\mathbf{A} := \mathsf{all\_subsets}(D,\ P,\ \mathbf{A})$
    $D := D \cup \{c\}$
  endwhile
  return A

The second component of Han and Lee's method is an algorithm that uses the CS-tree to detect all minimal unsatisfiable subsets. This is done by traversing the tree depth first testing at each node the satisfiability of its label $D \cup P$. If it is satisfiable, there is no need to visit its children (they will also be satisfiable). Otherwise, it continues visiting each child collecting minimal unsatisfiable subsets. If after visiting all children no minimal unsatisfiable subset A is found such that $A \subset D$, then D is a minimal unsatisfiable set.

The following pseudo-code defines `min_unsat1` as a simple modification of `all_subsets` which returns in **A** only those nodes in the CS-tree which are labeled by a minimal unsatisfiable subset.

$\mathsf{min\_unsat1}(D,\ P,\ \mathbf{A})$
  if $(sat(D \cup P))$ **return A**
  while $\exists c \in P$
    $P := P - \{c\}$
    $\mathbf{A} := \mathsf{min\_unsat1}(D,\ P,\ \mathbf{A})$
    $D := D \cup \{c\}$
  endwhile
  if $(\neg \exists A \in \mathbf{A}$ such that $A \subset D)$ $\mathbf{A} := \mathbf{A} \cup \{D\}$
  return A

The above algorithm avoids visiting some nodes in the CS-tree: children of consistent nodes are not visited. Han and Lee provided an extra pruning rule which potentially decreases the number of nodes visited. The rule avoids visiting children and right-siblings of a node labeled D, P if D is a superset of a minimal unsatisfiable subset already found (i.e., of an A already in $\mathbf{A}$). This is because D cannot then be a minimal unsatisfiable set, and neither can the D' part of the label of any of its children and right-siblings, since $D \subseteq D'$. Finally, the algorithm can be improved by avoiding the satisfiability test if $D \cup P$ is a superset of some $A \in \mathbf{A}$, in which case it is guaranteed to be unsatisfiable.

The following pseudo-code defines `min_unsat2` which incorporates these two modifications.

$\mathsf{min\_unsat2}(D,\ P,\ \mathbf{A})$
  if $(\neg \exists A \in \mathbf{A}$ such that $A \subset D \cup P)$
    if $(sat(D \cup P))$ **return A**

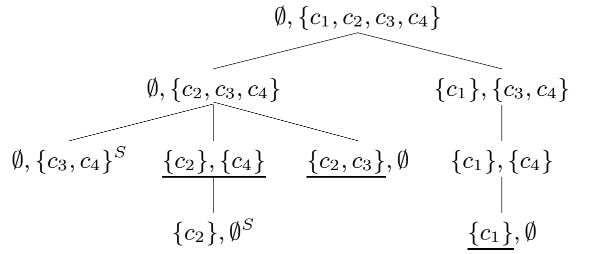

Figure 3: Finding all minimal unsatisfiable subsets of $\{c_1, c_2, c_3, c_4\}$

while $\exists c \in P$
    $P := P - \{c\}$
    $\mathbf{A}$:= min_unsat2(D, P, $\mathbf{A}$)
    $D := D \cup \{c\}$
    if ($\exists A \in \mathbf{A}$ such that $A \subset D$)
        return $\mathbf{A}$
endwhile
return $\mathbf{A} \cup \{D\}$

Note that the test (if $\exists A \in \mathbf{A}$ such that $A \subset D$) needs to be performed inside the while loop so that, if it succeeds, we can avoid generating the right-siblings. Also, the test is performed once D has been set to $D \cup \{c\}$, thus corresponding to the D of the immediate right-sibling.

Example 5. Consider the system of constraints $c_1, \ldots, c_4$ where the minimal unsatisfiable subsets are $\{c_1\}$, $\{c_2, c_3\}$ and $\{c_2, c_4\}$. The nodes visited by min_unsat2($\emptyset, \{c_1, \ldots, c_4\}, \emptyset$) are shown in Figure 3. Satisfiable nodes are marked S, while minimal unsatisfiable sets are underlined.

Nodes ($\{c_1, c_3\}, \emptyset$), ($\{c_1, c_2\}, \{c_4\}$), ($\{c_1, c_2\}, \emptyset$), and ($\{c_1, c_2, c_3\}, \emptyset$) which appear in Figure 2, are no longer visited because their D components are a superset of the already (by depth-first search) discovered minimal unsatisfiable set $\{c_1\}$.

4. IMPROVING THE ALGORITHM

The algorithm min_unsat2 is essentially[1] that defined by Han and Lee [7]. In the following sections we discuss improvements to this basic algorithm that can be achieved by taking into account the characteristics of the problem being solved, the connections among constraints, and the particular characteristics of the *sat* solver.

4.1 Preprocessing

In the context of type errors, it is often the case that some constraints occur in all minimal unsatisfiable subsets. This results from the fact that there may be in essence only one error, but because type information is heavily redundant there are multiple minimal unsatisfiable subsets all of which share a number of crucial constraints associated to that error.

We can straightforwardly detect constraints occurring in all minimal unsatisfiable subsets by using the following obvious result.

Lemma 1. *If a set of constraints C is unsatisfiable and there exists a constraint $c \in C$ such that $C - \{c\}$ is satisfiable, then c belongs to every minimal unsatisfiable subset of C.*

[1]Their presentation is different, but the underlying algorithms are identical.

Proof. Let us assume there is a minimal unsatisfiable subset M of C such that $c \notin M$. But then $M \subseteq C - \{c\}$ which is satisfiable. Contradiction. □

The following code uses the above lemma to define min_unsat3 which moves constraints in all unsatisfiable subsets into the first argument D, since we do not need to consider sets which do not include them.

min_unsat3($D, P, \mathbf{A}$)
    D' := in_all_unsat($D \cup P, P$)
    return Min($D \cup D'$,$P - D'$,$\mathbf{A}$)

in_all_unsat(C,P)
    $D := \emptyset$
    foreach $c \in P$
        if $sat(C - \{c\})$ $D := D \cup \{c\}$
    endfor
    return D

Here Min is a place holder for any of min_unsat* versions defined in this paper. To use min_unsat3 as a preprocessing step for Han and Lee's algorithm we simply replace Min by min_unsat2. Note that in_all_unsat only tests the constraints in P since those in D are already known to be in subsets that will be examined.

The cost of in_all_unsat(C, P) is $|P|$ calls to *sat*. If there are $m = |D'|$ constraints in every minimal unsatisfiable constraint, the subsequent call to Min($D \cup D'$,$P - D'$,$\emptyset$) may examine up to m less nodes for each non leaf call in the original call Min(D,P,$\emptyset$). Therefore, if calls to *sat* are cheap and there is a significant probability of finding at least one constraint common to all unsatisfiable subsets, the cost is almost certainly worthwhile.

Although we could use this optimization at any time during the search, it is most useful if performed once, at the beginning of the search. It is unlikely to discover new information later in the search, and the cost then becomes significant.

Example 6. Consider finding all minimal unsatisfiable subsets of $\{c_1, \ldots, c_5\}$ where the answers are $\{c_1, c_5\}$, $\{c_2, c_5\}$, $\{c_3, c_5\}$, $\{c_4, c_5\}$. Then

$$\text{min\_unsat2}(\emptyset, \{c_1, c_2, c_3, c_4, c_5\}, \emptyset)$$

examines 31 subsets, requiring 23 calls to *sat*, while

$$\text{min\_unsat2}(\{c_5\}, \{c_1, c_2, c_3, c_4\}, \emptyset)$$

examines 11 subsets, requiring 9 calls to *sat* to which we have to add another 5 calls from in_all_unsat.

4.2 Independence

For the following two improvements we use the notion of independence of constraints: constraints C_1 and C_2 are independent iff the solutions of $C_1 \cup C_2$ can be obtained by simply pairwise combining the solutions of C_1 and C_2 separately. In other words, if they do not influence each others solutions.

A simple (and incomplete) test for constraint independence relies on examining the constraint graph. Given a set of constraints C, the *constraint graph*, $g(C)$ is a bi-partite graph with a *variable node* (labeled v) for each $v \in vars(C)$, a *constraint node* (labeled c) for each $c \in C$, and an edge (v, c) for each $v \in vars(c)$.

LEMMA 2. *If C is a minimal unsatisfiable constraint set, then $g(C)$ is connected, i.e., the set of constraint nodes cannot be partitioned into two sets such that there is no path connecting nodes in the two sets.*

PROOF. Let us reason by contradiction, and thus assume that we can partition the constraint nodes into two non-empty disjoint subsets C_1 and C_2. Let $V_1 = vars(C_1)$ and $V_2 = vars(C_2)$. Then $V_1 \cap V_2 = \emptyset$ otherwise C_1 and C_2 would be connected. Since we have also assumed that C is a minimal unsatisfiable set, C_1 (C_2) must be satisfiable. Thus, there exist θ_1 (θ_2) solutions of C_1 (C_2) on the variables V_1 (V_2). But then $\theta = \theta_1 \cup \theta_2$ must be a proper valuation (since the variables are disjoint) and thus a solution of C. Contradiction. □

We can use this result to improve the min_unsat2 algorithm by first partitioning $C = D \cup P$ into connected subsets $X_1, \cdots X_n$ and then eliminating any partition which does not include D, since we are only interested in exploring supersets of D. The code is as follows:

```
min_unsat4(D, P, A)
    let X be the connected subsets of D ∪ P
    foreach X ∈ X
        if X ⊇ D
            P := X − D
            A := Min(D, P, A)
    endfor
    return A
```

where Min can again be any of the min_unsat* versions developed herein. Note that the only time when there can be more than one partition that passes the test $X \supseteq D$ is when $D = \emptyset$. Also note that partitioning the constraint graph is $O(||C||_c)$. Checking whether $X \supseteq D$ for all X is $O(|D|)$ since we need only traverse each constraint in D checking it belongs to the same partition. In the worst case min_unsat4 calls *sat* on each partition of $D \cup P$ which is certainly cheaper than a single call to $sat(D \cup P)$.

Note, however, that in the worst case min_unsat4 may indirectly cause the visitation of the node $(D, \emptyset)$ multiple times.

Example 7. Returning to the constraints of Example 5, suppose the variables of each constraint are $vars(c_1) = \{v_1, v_2\}$, $vars(c_2) = \{v_2, v_3, v_4\}$, $vars(c_3) = \{v_3, v_4, v_5\}$, and $vars(c_4) = \{v_4, v_5\}$. The constraint graph for this problem is shown in Figure 4.

Execution for min_unsat4$(\emptyset, \{c_1, c_2, c_3, c_4\}, \emptyset)$ (where Min = min_unsat2 in min_unsat4 and the recursive call in min_unsat2 is replaced by min_unsat4) proceeds exactly as those for

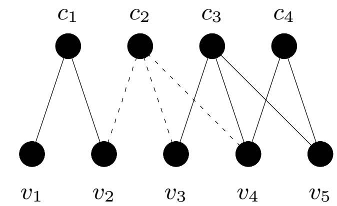

Figure 4: Constraint graph for the constraints $\{c_1, c_2, c_3, c_4\}$ with edges for c_2 shown dotted.

min_unsat2$(\emptyset, \{c_1, c_2, c_3, c_4\}, \emptyset)$ until we reach the call to min_unsat4$(\{c_1\}, \{c_3, c_4\}, \mathbf{A})$. At this point the constraint set $\{c_1, c_3, c_4\}$ becomes disconnected and can be partitioned into two different sets, $\{c_1\}$ and $\{c_3, c_4\}$. Since only the first partition contains D, we do not address the second partition and the call min_unsat2$(\{c_1\}, \emptyset, \mathbf{A})$ immediately adds $\{c_1\}$ to $\mathbf{A}$.

In total, the original min_unsat2 examines 9 subsets and makes 9 calls to *sat* while min_unsat4 examines only 8 subsets and makes 7 calls to *sat*.

A more complex (and more complete) test for constraint independence disregards connections with variable nodes whose associated variables have been set to a fixed value. Intuitively, these connections are "dead" and thus cannot be the cause for dependency. Formally, a set of constraints C fixes a variable $v \in vars(C)$ if there exists a value $d \in \mathcal{D}$ such that for every solution θ of C: $\theta(v) = d$.

LEMMA 3. *Let C be a minimal unsatisfiable constraint set with subset $D \subseteq C$ which fixes variables W. Then, the constraints $C - D$ are connected in $g(C)$ even after the variable nodes associated to variables W are removed from the graph.*

PROOF. Let us first assume D is unsatisfiable. Then, the only minimal unsatisfiable constraint set $C \supseteq D$ must be $C = D$. Clearly the result holds since $C - D$ is empty.

Let us now assume D is satisfiable. Suppose to the contrary that we can partition C into non-empty disjoint subsets C_1 and C_2 which are not connected in $g(C)$ with variables W removed. Now, $D \cup C_1$ and $D \cup C_2$ must be satisfiable and hence have solutions θ_1 and θ_2, respectively. The only variables that C_1 and C_2 can share are W, since otherwise they would be connected. But every solution of D gives these variables the same value. Hence, $\theta_1(v) = \theta_2(v), v \in W$. Thus $\theta = \theta_1 \cup \theta_2$ is a correct valuation and a solution of C. Contradiction. □

We can use the above result to increase the partitioning performed by min_unsat4 by eliminating the connections in the graph of $D \cup P$ from variables fixed by D. This is managed by the code below.

```
min_unsat5(D, P, A)
    let X be the connected subsets of D ∪ P
            with variables fixed by D removed
    foreach X ∈ X
        if X ∪ D is connected
            P := X − D
            A := Min(D, P, A)
    endfor
    return A
```

Note that we now need to check that the partition X when unioned with D is connected (as opposed to checking that $X \supseteq D$), since D might not be a subset of X due to the removal of fixed variables. Also note that, as in min_unsat4, we may visit the node $(D, \emptyset)$ multiple times.

In order to use this optimization effectively we need to efficiently determine variables that are fixed by a constraint D. One can obtain a good approximation of the fixed variables for many constraint domains by representing the groundness dependencies of each individual constraint by Horn clauses, and then using a linear time Horn clause solver to compute fixed variables.

Example 8. Consider the Herbrand equations $x = list(u), y = pair(x, w), u = int$, then the corresponding Horn groundness dependencies are $x \rightarrow u, u \rightarrow x, y \rightarrow x, y \rightarrow w, x \wedge w \rightarrow y, u$ which result in u and x being found to be ground.

Improvements in circuit error diagnosis that make use of the constraint graph have been investigated before in [6]. Here, the authors cluster constraints in the initial constraint graph until it becomes a tree, and then use this for diagnosis. This is clearly distinct from the methods we use which simply rely on connectedness of the constraint graph.

4.3 Always satisfiable

Another optimization consists in detecting constraints that can never take part in a minimal unsatisfiable set since their addition to any satisfiable subset will always give a satisfiable set. These "always satisfiable" constraints can be eliminated.

LEMMA 4. *Let $S_1 \cup S_2$ be a set of constraints such that for all $U \subseteq S_2$ we have that $sat(U)$ implies $sat(U \cup S_1)$. Then all minimal unsatisfiable subsets of $S_1 \cup S_2$ are subsets of S_2.*

PROOF. For a minimal unsatisfiable subset M we have that $\neg sat(M)$, and for each $M' \subset M$, $sat(M')$. Suppose $U = M - S_1 \neq M$, then $sat(U)$ but then $sat(U \cup S_1)$ and $U \cup S_1 \supseteq M$ so contradiction. □

This concept is in some sense related to independence since such constraints will also be independent of the minimal unsatisfiable sets. Unfortunately, most incomplete independence tests (like the two used in the previous section) are not powerful enough to detect this kind of independence.

A common approach to detect these kind of constraints is to use an analysis based on "degrees of freedom". This kind of analysis locates constraints which contain at least one variable which is completely "free", i.e., it can take any value in the domain.

Example 9. The Boolean constraints C_{add} modeling the adder circuit shown in Figure 1 are

$$i1 \Leftrightarrow x \wedge y, i2 \Leftrightarrow x \oplus y, i3 \Leftrightarrow i2 \wedge ci, o \Leftrightarrow ci \oplus i2, co \Leftrightarrow i1 \vee i3$$

The Boolean constraints modeling the observed inputs and outputs are

$$\neg x, \neg y, ci, \neg o, co.$$

Let C be defined as $C_{add} \cup \{\neg x, \neg y, ci, \neg o\}$, i.e., all constraints except for the last observation co. Then, variable co occurs only once in C. Hence, the constraint $co \Leftrightarrow i1 \vee i3$ can always be satisfied by choosing the appropriate value for co. Furthermore, once this constraint is removed the constraint $i3 \Leftrightarrow i2 \wedge ci$ can always be satisfied because $i3$ can take any value. Therefore, we can show that $S_1 = \{i3 \Leftrightarrow i2 \wedge ci, co \Leftrightarrow i1 \vee i3\}$ and $S_2 = C - S_1$ satisfy the conditions of Lemma 4. Note that $co \Leftrightarrow i1 \vee i3$ is indeed independent of the rest of constraints but none of the two independence tests used in the previous section would be capable of detecting this.

We can use the above result in the algorithm by searching each set $D \cup P$ looking for constraints which are always satisfiable. If they appear in D, we can finish since D cannot then be a subset of a minimal unsatisfiable subset. Otherwise, we can remove them from P.

min_unsat6(D, P, $\mathbf{A}$)
  let $Z \subseteq D \cup P$ be the subset such that
    $\forall U' \subseteq D \cup P \;\; sat(U') \Rightarrow sat(U' \cup Z)$
  if $(Z \cap D \neq \emptyset)$ **return** $\mathbf{A}$
  $P := P - Z$
  return Min(D, P, $\mathbf{A}$)

The implementation of the above optimization uses a simple degrees of freedom analysis to detect always satisfiable constraints: constraints of the form $v = t$ or $v \Leftrightarrow t$ are always satisfiable if variable v appears nowhere else. Such constraints are removed and the check repeated until no constraints of this form appear. By keeping a list of the occurrences of each variable on the left hand side of an equation, and a count of other occurrences, our (incomplete) analysis can determine the always satisfiable constraints (of this form) in linear time.

4.4 Entailment

Any constraint $c \in P$ which is entailed (and thus redundant) with respect to D can be removed from P, since it cannot be part of a minimal unsatisfiable subset.

LEMMA 5. *Let $U \subseteq P$ be a set of constraints such that $D \Rightarrow U$, then no minimal unsatisfiable subset of $D \cup P$ which is a superset of D intersects with U.*

PROOF. Suppose to the contrary that minimal unsatisfiable subset $M \subseteq D \cup P$ such that $M \cap U \neq \emptyset$. Now $sat(M - U)$ since it is a strict subset of a minimal unsatisfiable subset M, but then $M - U \supseteq D$, and hence $M - U \Rightarrow D \Rightarrow U$, and thus $sat((M - U) \cup U)$ or equivalently $sat(M)$. Contradiction. □

Checking for entailment can be as expensive as the satisfiability checks it may avoid. However, some simple (and incomplete) kinds of entailment checking are straightforward. For example, by determining which variables are fixed by D and their values, we can cheaply check for entailment of constraints by simply evaluating the constraint with those values.

Example 10. Consider the constraints $D = \{\neg x, \neg co, i1 \Leftrightarrow x \wedge y\}$ and $P = \{i2 \Leftrightarrow x \oplus y, i3 \Leftrightarrow i2 \wedge ci, o \Leftrightarrow ci \oplus i2, co \Leftrightarrow i1 \vee i3, \neg y, ci, \neg o\}$. Then $D \Rightarrow co \Leftrightarrow i1 \vee i3$ since in all solutions of D we have $\{i1 \mapsto 0, co \mapsto 0\}$, which ensures the constraint is satisfied.

Hence we can remove the constraint $co \Leftrightarrow i1 \vee i3$ from P since it will never be part of a minimal unsatisfiable set.

Entailment checking for Herbrand constraints is also cheap, by determining if the most general unifier of D makes the terms in an equation in P identical.

4.5 Cheap Solvers

The original algorithm of Han and Lee [7] was developed for diagnosis of circuits where the satisfiability test *sat* is very expensive. But other solvers, such as the linear unification solver for Herbrand constraints, are cheap, which chagnes some considerations.

First, checking that $D \cup P$ is a superset of a previous answer in order to avoid the call to *sat*, may not be beneficial since the *sat* check may be cheaper than detecting whether it is a superset or not. Similarly, we can replace the test which checks that D is not a superset of a previous answer by $sat(D)$. Finally, if D is unsatisfiable then we do not need to check any other subsets of $D \cup P$, since only D can be a minimal unsatisfiable subset.

This leads to the code below which replaces min_unsat2. Note that due to the changes to the loop break (when D becomes unsatisfiable then we add D to the set $\mathbf{A}$ of answers, if it is not subsumed and return) execution will never fall out of the while loop.

```
min_unsat7(D, P, A)
    if (sat(D ∪ P)) return A
    while ∃c ∈ P
        P := P − {c}
        A := min_unsat7(D, P, A)
        D := D ∪ {c}
        if (¬sat(D))
            if (¬∃A ∈ A such that A ⊂ D)) A := A ∪ {D}
            return A
    endwhile
    %% return A
```

Example 11. The extra *sat* check can discover answers earlier. Imagine we have the set of constraint $\{c_1, \ldots, c_4\}$ with minimal unsatisfiable sets $\{c_1, c_2\}$ and $\{c_3, c_4\}$, and we traverse the sets as in Figure 2. Then, the nodes traversed by min_unsat2 are shown in Figure 5 while min_unsat7 will discover the unsatisfiable set $\{c_1, c_2\}$ at the root node, just before calling min_unsat7($\{c_1, c_2\}, \{c_4\}, \mathbf{A}$). Thus, it will not explore this subtree.

4.6 Incremental Solvers

As Han and Lee pointed out, the order in which subsets of each node are investigated can have significant impact on the total search space.

Example 12. Returning to the constraints of Example 5, if we build the tree by reversing the ordering of the constraints, we obtain a tree of calls as shown in Figure 6.

Nodes marked U are determined as unsatisfiable without calling *sat*, since they are supersets of already determined minimal unsatisfiable sets.

For this order there are 14 nodes and 11 satisfiability checks compared to 9 nodes and 9 satisfiability checks for the order shown in Figure 3.

As the size of constraint sets increases, the differences get bigger. For example, for 5 constraint $\{c_1, \ldots, c_5\}$ with minimal unsatisfiable subsets $\{c_1\}$, $\{c_2, c_3\}$, $\{c_2, c_4\}$ and $\{c_2, c_5\}$ the first order requires 14 nodes and calls to *sat* while the reverse order leads to 28 nodes and 17 calls to *sat*.

It is then important to determine orders which are likely to reduce the search. Constraint solvers which are *incremental* can help us in this task. This is not only because we can reuse some of the work performed by the incremental *isat* thus reducing its cost, but also because we can gain important information from the constraints by adding them one by one.

The idea is to start with a state representing D, and add constraints from P one by one, until we discover unsatisfiability. The last added constraint c must be part of a minimal unsatisfiable set (by Lemma 1), and it should therefore be considered next in the order.

To take maximum advantage of the incremental solver, we will pass three extra arguments to our function: $T \supseteq D$ representing the subset of $D \cup P$ currently proved to be satisfiable by the incremental solver, $state_T$ the state representing the constraints in T, and $state_D$ the state representing the constraints in D. Intuitively, $state_T$ is obtained from $state_D$ by adding some constraints from P while looking for the next constraint to be chosen.

```
min_unsat8(D, state_D, P, T, state_T, A)
    result := true
    while (result && ∃c ∈ P − T)
        T := T ∪ {c}
        last_T := state_T
        (result, state_T) := isat(c, state_T)
    endwhile
    if (result) return A
    A := min_unsat8(D, state_D, P − {c}, T − {c}, last_T, A)
    D := D ∪ {c}
    (result, state_D) := isat(c, state_D)
    if (¬result)
        if (¬∃A ∈ A such that A ⊂ D))
            A := A ∪ {D}
        return A
    A := min_unsat8(D, state_D, P − {c}, D, state_D, A)
    return A
```

Example 13. Consider the constraints of Example 5 and the execution of (a) min_unsat8($\emptyset, true, \{c_1, c_2, c_3, c_4\}, \emptyset, true, \mathbf{A}$) with the reverse (bad) ordering as the underlying one.

Initially, $D = \emptyset$ and $P = P_0 = \{c_1, c_2, c_3, c_4\}$. We add the constraints one by one in reverse order discovering unsatisfiability when $c = c_2$, leaving $T = \{c_2, c_3, c_4\}$. This causes a call to (b) min_unsat8($\emptyset, \_, \{c_1, c_3, c_4\}, \{c_3, c_4\}, \_, \mathbf{A}$) (where _ represents some state information). This call adds c_1 to T which causes unsatisfiability, thus finishing the while loop and calling (c) min_unsat8($\emptyset, \_, \{c_3, c_4\}, \{c_3, c_4\}, \_, \mathbf{A}$) which immediately returns since $P - T$ is empty.

Returning to call (b) we add $\{c_1\}$ to D detecting unsatisfiability, so $\{c_1\}$ is added to $\mathbf{A}$ and we return execution to call (a). We add c_2 to D which remains satisfiable and call (d) min_unsat8($\{c_2\}, \_, \{c_1, c_3, c_4\}, \{c_2\}, \_, \mathbf{A}$). This call adds c_4 which immediately causes failure finishing the while loop and invoking (e) min_unsat8($\{c_2\}, \_, \{c_1, c_3\}, \{c_2\}, \_, \mathbf{A}$). This call adds c_3 which again causes failure finishing the loop and invoking (f) min_unsat8($\{c_2\}, \_, \{c_1\}, \{c_2\}, \_, \mathbf{A}$). This call adds c_1 which yet again causes failure and invokes (g) min_unsat8($\{c_2\}, \_, \emptyset, \{c_2\}, \_, \mathbf{A}$) which immediately succeeds. Returning to invocation (f) we add c_1 to c_2 and discover an unsatisfiable subset which is not added to $\mathbf{A}$ since it is a superset of an existing answer. Returning to invocation (e) we add c_3 to c_2, which is also unsatisfiable and the new answer is added. Returning to invocation (d) we add c_4 to c_2, which again is unsatisfiable and the new answer is added.

The tree of calls is shown in Figure 7. Invocations dealing

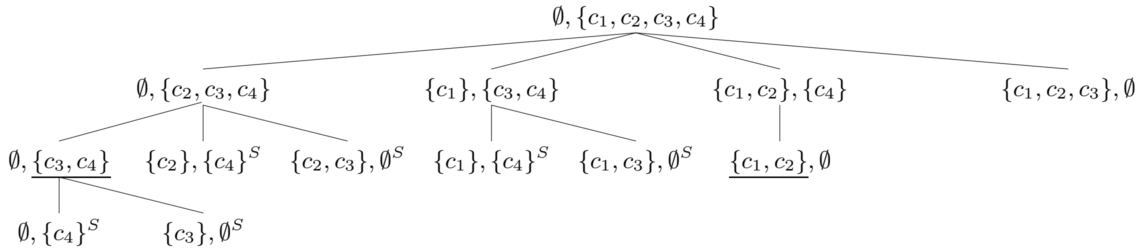

Figure 5: Finding all minimal unsatisfiable subsets of $\{c_1, c_2, c_3, c_4\}$

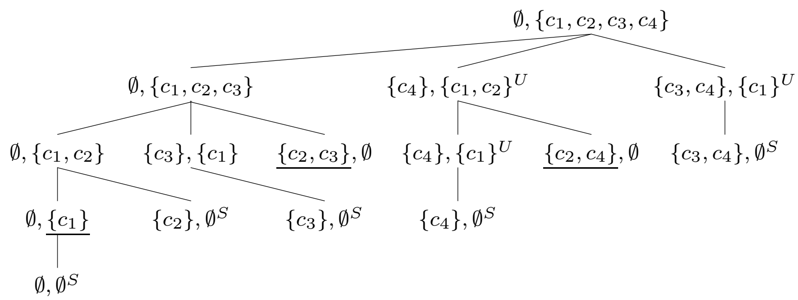

Figure 6: Finding all minimal unsatisfiable subsets of $\{c_4, c_3, c_2, c_1\}$

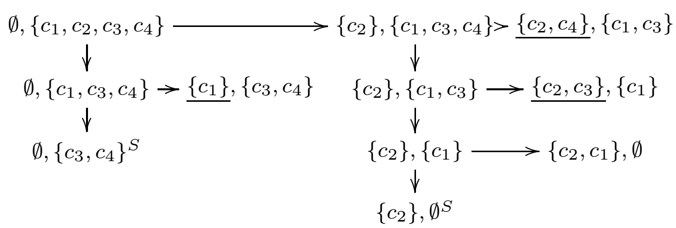

Figure 7: Finding all minimal unsatisfiable subsets of $\{c_4, c_3, c_2, c_1\}$ using an incremental solver

with the same set $D \cup P$ are shown horizontally aligned. Overall there are 12 calls to *isat* and 6 subsets (nodes of the CS-tree) are explored.

If we count the size of constraints sets passed to *sat* in Example 5 there are effectively 18 calls to *isat* and 9 subsets explored. In the bad order (illustrated by Example 12) there are effectively 21 calls to *isat* and 15 subsets explored.

Note that the incremental version is far less sensitive to the underlying constraint order since it dynamically determines its own.

Example 14. Consider the constraints from Example 5 and the call $\mathsf{min\_unsat8}(\emptyset, \mathit{true}, \{c_1, c_2, c_3, c_4\}, \emptyset, \mathit{true}, \mathbf{A})$ using the natural (good) order. This leads to the tree of calls where there are 11 calls to *isat* and 6 subsets explored.

Theorem 1. *The call $\mathsf{min\_unsat8}(\emptyset, \mathrm{true}, C, \emptyset, \mathrm{true}, \emptyset)$ returns all minimal unsatisfiable subsets of C.*

Proof. The invariant maintained by the algorithm is that for any call $\mathsf{min\_unsat8}(D_0, \_, P_0, T_0, \_, \mathbf{A})$ D_0 is satisfiable, T_0 is satisfiable, $D_0 \subseteq T_0 \subseteq D_0 \cup P_0$, and $\mathbf{A}$ contains any minimal unsatisfiable subsets of $D_0 \cup P_0$ which are not supersets of D_0. The call returns all minimal unsatisfiable subsets M of $D_0 \cup P_0$.

Clearly the invariant holds for the initial call.

At the end of the while loop. In the first case $T = D_0 \cup P_0$ is satisfiable in which case there are no unsatisfiable subsets of $D_0 \cup P_0$, and the call returns correctly. In the second case $T - \{c\}$ is satisfiable. The recursive call $\mathsf{min\_unsat8}(D_0, \_, P_0 - \{c\}, T - \{c\}, \_, \mathbf{A})$ satisfies the invariants since D_0 is satisfiable, $T - \{c\}$ is satisfiable, and $\mathbf{A}$ contains the minimal unsatisfiable subsets of $D_0 \cup P_0 \supseteq D_0 \cup P_0 - \{c\}$ that are not supersets of D_0.

After the return from the recursive call we have by the invariant that $\mathbf{A}$ holds all minimal unsatisfiable subsets of $D_0 \cup P_0 - \{c\}$.

We add c to D_0 and check satisfiability. If the resulting set is unsatisfiable we add it to $\mathbf{A}$ if it there is no smaller subset already in $\mathbf{A}$. This is correct since any minimal unsatisfiable subset M of $D_0 \cup \{c\}$ must be in $\mathbf{A}$ already by the calling invariant since it will not be a superset of D_0. If none exists then $D_0 \cup \{c\}$ is a minimal unsatisfiable subset. Now $\mathbf{A}$ already contains all minimal unsatisfiable subsets of $D_0 \cup P_0$ which (a) are not supersets of D_0 or (b) which are subsets of $D_0 \cup P_0 - \{c\}$. Hence, with the possible addition $D_0 \cup \{c\}$, we have determined all minimal unsatisfiable subsets of $D_0 \cup P_0$, and, upon return, correctness is maintained.

Otherwise, $D_0 \cup \{c\}$ is satisfiable. For the second recursive call $\mathsf{min\_unsat8}(D_0 \cup \{c\}, \_, P_0 - \{c\}, D_0 \cup \{c\}, \_, \mathbf{A})$ the satisfiability conditions clearly hold. $\mathbf{A}$ already contained all the minimal unsatisfiable subsets of $D_0 \cup P_0 = (D_0 \cup \{c\}) \cup (P_0 - \{c\})$ not supersets of D_0. We need to ensure it now holds all the subsets of $D_0 \cup P_0$ not supersets of $(D_0 \cup \{c\})$. These are minimal unsatisfiable subsets of $D_0 \cup P_0 - \{c\}$ which are already in $\mathbf{A}$.

On returning from the recursive call we have that $\mathbf{A}$ contains all minimal unsatisfiable subsets of $(D_0 \cup \{c\}) \cup (P_0 - \{c\}) = D_0 \cup P_0$, and hence the return is correct. $\square$

Note that although it may appear that we need to keep two (or even three) states of solvers for this algorithm, in fact we can recover $state_D$ (and $last_T$) from $state_T$ by backtracking, a facility available with all the Herbrand solvers used and, hence, we only need one solver state.

We can use the incremental version together with independence, always satisfiable and entailment improvements by extending the calls to $\mathsf{Min}(D, P, \mathbf{A})$ occurring in those versions to $\mathsf{min\_unsat8}(D, state_D, P, D, state_D, \mathbf{A})$, since none of these change the set D.

4.7 Don't care constraints

Many applications of minimal unsatisfiable subset detection contain constraints which can be disregarded because they are not of interest for the application. For example, circuit diagnosis applications such as those used by Han and Lee, are only interested in constraints corresponding to gates. Constraints corresponding to inputs, outputs and connections are not interesting to the diagnosis. Similarly, when debugging type errors, constraints corresponding to function application $(e_1\ e_2)$ are not very useful for providing explanations, since there is no part of the text to highlight.

Hence, we can split the initial set of constraints C, into the constraints we are interested in C_c and those we do not care about C_d. We can then simply assume that the constraints in C_d are in all minimal unsatisfiable subsets, i.e. add C_d to the initial D set. Indeed, such an assumption is implicit in earlier discussions on circuit diagnosis, since the observations are always considered to be present. This preprocessing can significantly reduce the complexity of the problem and even change the number of answers, since we are now comparing unsatisfiable sets only on the constraints C_c.

The only complexity arising from this treatment of don't care constraints is that

- when a minimal unsatisfiable subset M is discovered we instead store $M \cap C_c$ (which means the subset checks succeed more often); and
- optimizations based on independence (min_unsat4 and min_unsat5) need to be slightly modified since only $D \cap C_c$ needs to be connected

5. EVALUATION

In order to investigate the benefits of the optimizations discussed in previous sections, we have built a prototype system in SICStus Prolog which implements each of the optimizations and their possible combinations. In this prototype the algorithms for checking the connectedness of constraints graphs, and for detecting always unsatisfiable and entailment (by fixed variables) constraints are implemented naively. Similarly, the full incremental algorithm is mimicked rather than implemented in full, since the internal state of the constraint solver is not available to the programmer. Hence, we will not compare the results on execution times but, rather, compare the number of different subsets $D \cup P$ explored by the various versions, and the number of calls to *isat* (where we implement *sat* in terms of *isat*). This allows us to better compare incremental and non-incremental versions.

Benchmark	$\lvert C\rvert$	$\lvert C_c\rvert$	$\lvert\mathbf{A}\rvert$
`sum1`	20	18	2
`sum2`	22	13	2
`foldl1`	6	6	1
`foldl2`	8	5	2
`palin-l`	10	5	2
`palin-g`	260	128	5
`list`	12	12	9
`pop2`	24	10	1
`const`	2891	2228	6
`tup`	36	27	1
`ite`	18	11	1
`rotate`	1711	178	8

Table 2: Type error benchmark statistics: number of constraints, number of constraints excluding don't care constraints, and number of minimal unsatisfiable subsets.

The evaluation uses two classes of benchmarks. The first class are Herbrand equation ($\mathcal{H}$) problems arising from type error debugging. We take the sets of constraints generated by the Chameleon [14] system for debugging Haskell programs and use the efficient satisfiability procedure for solving Herbrand equations provided by SICStus Prolog. The only "don't care" constraints in these examples are those constraints arising from function application expressions. The second class of benchmarks are small examples of circuit diagnosis from the REVISE system [12] based on ISCAS85 benchmarks [1] as well as the `add` example of Figure 1. We use the builtin ROBDD based Boolean solver of SICStus Prolog.

Many different algorithms can be obtained by chaining together the various improvements suggested herein. In order to identify the different improvements in the algorithm of interest we will use the notation $x_1 \cdots x_n.x_{n+1} \cdots x_m$ to represent each algorithm, where each x_i represents the digit n of a `min_unsat`n version in which the call to Min (or the recursive calls in min_unsat2, min_unsat7 and min_unsat8) is replaced by a call to the version indicated by x_{i+1}. The exception appears when $i = m$ in which case the the call to Min is replaced by a call to the version appearing after the dot (x_{n+1}). For example, the algorithm described by 3.658 starts with the preprocessing step defined by version min_unsat3, where its call to Min is replaced by min_unsat6, that of min_unsat6 is replaced by min_unsat5, that of min_unsat5 is replaced by min_unsat8, and that of min_unsat8 is replaced by min_unsat6. Also, whenever the don't care optimization is used, we prepend a d to the algorithm description.

Let us start by discussing the results of our evaluation for type error debugging benchmarks. Table 2 shows the sizes of these benchmarks in terms of total number of constraints, number of "interesting" constraints, and number of minimal unsatisfiable subsets. Table 1 shows the number of subsets $D \cup P$ examined by a number of selected algorithms (ignoring any subsets checked by the preprocessing steps). This gives a reasonable comparison of the size of the search space for each algorithm. We limited the number of subsets to be searched to 10000 for each algorithm. The symbol — indicates entries in which the search required more subsets to be examined. In order to select the algorithms that would

Alg	sum1	sum2	foldl1	foldl2	palin-l	palin-g	list	pop2	const	tup	ite	rotate	TOTAL
.1	4614	6020	9	41	256	—	1949	—	—	5808	—	—	18697
.2	4614	5509	9	41	256	—	1292	6240	—	5808	—	—	23769
3.2	1793	2562	0	10	256	3703	1292	0	—	0	0	—	9616
.42	1885	2280	10	33	11	—	606	4385	—	430	4663	—	14303
.52	1882	2281	9	32	11	—	484	4859	—	308	5012	—	14878
.62	182	204	9	22	8	3089	190	704	—	38	149	—	4595
.7	4614	5508	9	41	255	—	1283	6234	—	5808	—	—	23752
.8	22	24	6	10	15	278	123	24	2884	31	13	1713	5143
3.8	5	7	0	5	15	93	123	0	972	0	0	127	1347
.68	16	17	6	10	8	124	31	24	455	31	13	293	1028
3.648	4	5	0	5	8	34	31	0	102	0	0	29	218
d.2	870	581	9	14	32	—	1292	1411	—	2536	—	—	6745
d.8	18	13	6	9	9	139	123	10	2228	27	11	178	2771
d3.648	4	4	0	4	5	23	31	0	216	0	0	22	309

Table 1: Number of subsets examined for type benchmarks.

appear in the table, we first evaluated all possible combinations and then summarized the results by eliminating those whose data did not add much to the discussion. As a result, the table is divided into three parts. The first compares the results for each individual optimization (preprocessing, independence, always satisfiable, etc). The second part selects the best individual optimization (incrementality) and compares the results of adding others to it. The last part evaluates the effect of the don't care optimization on the basic algorithm and on the best algorithm found in each of the two previous parts. Note that the entailment optimization never occurs for the type error problems, and is only rarely applicable for the circuit error diagnosis problems. Thus, these results were also eliminated from the table. Table 3 shows the number of calls to *isat* examined by each selected algorithm (including preprocessing calls). Since the cost of satisfiability is almost constant, this gives a very accurate estimate of the work performed by the solver.

Regarding the comparison between `min_unsat1` (algorithm .1) and `min_unsat2` (.2), it seems that for Herbrand equations the pruning improvements are only moderate. They make `pop2` solvable and occasionally reduce the number of subsets examined by around 1/3. They also reduce the number of calls to *isat* more frequently (once even by 2/3) although usually by less than 1/3.

The preprocessing step `min_insat3` (3.2), which removes constraints in all unsatisfiable subsets, is more beneficial. For the benchmarks with only a single unsatisfiable subset, `foldl1`, `pop2`, `tup` and `ite`, it solves the entire problem. For other benchmarks, it substantially reduces the number of subsets visited, when there are any constraints found via the method, and makes `palin-g` solvable. The extra *isat* checks required for this optimization are almost always repaid, except for the case of `palin-l` and `list`, where the optimization finds no constraints in all minimal unsatisfiable subsets, and the number of *isat* checks is slightly increased. However, considering the low cost of the Herbrand *isat* checks, the increase seems reasonable.

The independence based optimization `min_unsat4` (.42) is clearly useful. In all but one example it reduces the number of subsets considered, often by around 1/2 and sometimes by more than an order of magnitude. The more sophisticated call-graph approach `min_unsat5` (.52), which takes into account fixed variables, only occasionally improves on the simple call graph approach (`list` and `tup`), and often worsens the situation. The cause is that whenever one of the optimizations applies, then we may visit the same set twice. Since the more sophisticated version finds more places to apply the optimization, this happens more often.

The always satisfiable constraints optimization `min_unsat6` (.62) are also highly effective, often reducing the number of sets examined by an order or magnitude or more. Applying this optimization makes two more examples (`palin-g` and `ite`) solvable where before (using .2) they were not.

Replacing subset checks by calls to *isat* in `min_unsat7` (.7) reduces the number of subsets examined very slightly and only in a few cases. Furthermore, it increases the number of *isat* checks quite dramatically. Surprisingly, the comparison of the execution times (not shown) of .2 versus .7 indicates that the benefit in terms of reduced subset checks is not repaid. For these two simple algorithms the prototype times should be reasonably accurate.

The greatest benefit arises from the use of an incremental solver `min_unsat8` (.8). The number of subsets explored is dramatically reduced, since the incremental approach quickly finds large satisfiable subsets. Improvements of two orders of magnitude are common and all the benchmarks are now require examining less than 3000 different subsets.

When the optimizations are combined the benefits are not always cumulative. For example, comparing 3.8 to .8 in terms of calls to *isat* shows that using the incremental solver the benefit of preprocessing step is usually not paid off, although for the hardest benchmarks `const` and `rotate` it is beneficial. On the other hand, comparing the results of .68, .8 and .62, the benefits of the combination .68 do seem cumulative. In any case, the combination 3.648 visits the least number of subsets in all cases, and overall is the most efficient in terms of the use of the solver. Hence, there is definite synergy between the optimizations.

The treatment of don't care constraint effectively changes the problem, making the constraints we need to deal with considerably smaller without significantly complicating any other optimization. Hence, they should clearly always be applied. Interestingly, the slightly different search for d3.648 explores twice as many subsets as 3.648 for `const`, but overall their is a noticeable saving in calls to *isat*.

Benchmark	$\|C\|$	$\|C_c\|$	$\|\mathbf{A}\|$	$\|\mathbf{A}_c\|$
add	10	5	2	2
c17	13	6	2	2
goel	51	44	8	1
voter	75	59	3	1
plc	28	10	10	1
plc1	28	10	2	2
plc2	28	10	1	1
gte_flat	24	19	4	1

Table 4: Circuit benchmark statistics: number of constraints, number of constraints excluding don't care constraints, and number of minimal unsatisfiable subsets, number of minimal unsatisfiable subsets excluding don't care constraints.

The statistics for the circuit benchmarks are given in Table 4. Note that they are considerably smaller than the type benchmarks. Also note that, for these problems, when we treat some constraints as don't care we actually change the number of minimal unsatisfiable subsets, since many of the answers to the original problem only differ in the don't care constraints. That is why the table shows both sets of numbers.

Table 5 and Table 6 show the same information as the corresponding tables for type benchmarks. Since the Boolean solver is not cheap, we omit the results for min_unsat7. The results of the optimizations are somewhat different on the circuit diagnosis benchmarks. Let us start by comparing the results of each optimization when performed alone. Firstly, min_unsat2 improves upon min_unsat1 really only on the number of calls to *isat*, which is decreased around 33%. The preprocessing step min_unsat3 is again worthwhile on its own, but it almost never pays off when combined with the incremental approach min_unsat8 (algorithm 3.8). The constraint graph approaches min_unsat4 and min_unsat5 are not nearly as effective as before, but still reduce the burden on the solver. The always satisfiable optimization min_unsat6 again provides some benefit, but nowhere near as good as in the type example. Once more the incremental approach min_unsat8 gives the most benefit, and makes all the benchmarks solvable visiting at most 100 subsets. In summary, only algorithms 3.2 and .8 seem to make a significant impact on the results.

Regarding the combinations, the improvements over the straight incremental approach are limited, with algorithm .68 being the winner but only slightly better than algorithm .8. Finally, the treatment of don't care constraints makes a significant difference to min_unsat2, making the plc benchmarks solvable. Interestingly, after disregarding the don't care constraints, most of the benchmarks only have a single unsatisfiable subset, and yet min_unsat3, which solves the problem entirely is still more costly that the incremental approach.

6. CONCLUSION AND RELATED WORK

Finding all minimal unsatisfiable subsets is a challenging problem because it implicitly involves examining each possible subset of a given set of constraints. In this paper we investigated how to reduce as much as possible the number of constraints sets that need to be examined. A number of methods based on independence of constraints can significantly improve the algorithm. The greatest improvement arises from the use of incremental solving algorithms which reduce the order dependence of the algorithm and quickly lead to the discovery of large satisfiable subsets.

7. REFERENCES

[1] F. Brglez, P. Pownall, and R. Hum. Accelerated ATPG and fault grading via testability analysis. In *IEEE Int. Symposium on Circuits and Systems*, pages 695–698, 1985.

[2] R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[3] C. Codognet, P. Codognet, and G. File. Yet another intelligent backtracking method. In *Logic Programming: Proceedings on the Joint International Conference and Symposium*, pages 447–465, 1988.

[4] B. Davey, N. Boland, and P.J. Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS Journal of Computing*, 14(4):373–386, 2002.

[5] B. De Backer and H. Beringer. Intelligent backtracking for CLP languages: an application to CLP(R). In *Logic Programming: Proceedings of the 1991 International Symposium*, pages 405–419, 1991.

[6] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *IJCAI*, pages 1742–1749, 1995.

[7] B. Han and S-J. Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2):281–286, 1999.

[8] A. Hou. A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, 65:281–328, 1994.

[9] Ulrich Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001.

[10] M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[11] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[12] M Schroeder. Revise. http://www.soi.city.ac.uk/~msch/revise/revise.html.

[13] J. Silva and K Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceeding of ICCAD'96*, pages 220–228, 1996.

[14] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/~sulzmann/chameleon.

Alg	sum1	sum2	foldl1	foldl2	palin-l	palin-g	list	pop2	const	tup	ite	rotate	TOTAL
.1	56803	79110	44	239	1023	—	12844	—	—	202128	—	—	352191
.2	40315	55210	44	213	483	—	3998	143889	—	166673	—	—	410825
3.2	9417	19952	35	113	539	138400	4110	1487	—	1590	948	—	176591
.42	1695	2692	38	96	38	—	367	32128	—	6160	20652	—	63866
.52	1738	2715	39	97	38	—	284	44129	—	4905	24747	—	78692
.62	478	704	15	61	15	33120	111	3300	—	454	178	—	38436
.7	108396	135242	74	429	1591	—	16184	362714	—	373128	—	—	997758
.8	428	515	37	72	91	8421	887	1058	147945	1406	502	81440	242802
3.8	371	440	35	76	147	4281	999	1487	47721	1590	948	8546	66641
.68	195	246	24	55	43	2627	152	622	12028	671	168	11156	27987
3.648	341	377	35	74	99	3031	264	1487	7039	1590	948	6934	22219
d.2	7292	5753	44	82	95	—	3998	31851	—	71671	—	—	120786
d.8	262	204	37	35	34	3600	887	246	98635	1059	365	4109	109473
*d*3.648	273	280	35	52	57	2221	264	811	6168	1396	851	2938	15346

Table 3: Number of calls to incremental solver *isat* for type benchmarks

Alg	add	c17	goel	voter	plc	plc1	plc2	gte_flat	TOTAL
.1	68	784	2686	—	—	—	—	450	3988
.2	61	784	2686	—	—	—	—	450	3981
3.2	41	297	297	—	—	—	0	55	690
.42	63	400	2686	—	—	—	—	442	3591
.52	78	400	2686	—	—	—	—	442	3606
.62	42	404	2686	—	—	—	—	378	3510
.8	26	13	92	40	24	12	7	30	244
3.8	16	7	51	36	22	7	0	15	154
.68	21	13	92	40	24	12	7	30	239
3.648	12	7	51	41	23	7	0	15	156
d.2	12	36	109	—	68	34	36	331	626
d.8	7	5	41	4	2	6	4	13	82
*d*3.648	9	5	0	0	0	6	0	0	20

Table 5: Number of subsets examined for circuit benchmarks.

Alg	add	c17	goel	voter	plc	plc1	plc2	gte_flat	TOTAL
.1	511	6856	127168	—	—	—	—	9237	143772
.2	397	4866	122622	—	—	—	—	6668	134553
3.2	334	1393	13886	—	—	—	678	1589	17880
.42	305	578	115108	—	—	—	—	3366	119357
.52	293	578	115108	—	—	—	—	3366	119345
.62	183	578	95370	—	—	—	—	2949	99080
.8	231	180	2750	3612	447	392	257	543	8412
3.8	184	224	2935	8643	1008	823	678	679	15174
.68	170	168	2731	3521	350	272	202	521	7935
3.648	143	218	2935	8616	916	752	678	673	14931
d.2	87	304	4810	—	1019	721	808	4245	11994
d.8	31	31	1828	339	16	46	42	266	2599
*d*3.648	67	98	2247	4385	257	284	267	454	8059

Table 6: Number of calls to *isat* for circuit benchmarks.