Effective Preprocessing with Hyper-Resolution and Equality Reduction

Fahiem Bacchus[1] and Jonathan Winter[1]

Department of Computer Science, University Of Toronto,*
Toronto, Ontario, Canada
[fbacchus|winter]@cs.toronto.edu

Abstract. HypBinRes, a particular form of hyper-resolution, was first employed in the SAT solver 2CLS+EQ. In 2CLS+EQ, HypBinRes and equality reduction are used at every node of a DPLL search tree, pruning much of the search tree. This allowed 2CLS+EQ to display the best all-around performance in the 2002 SAT solver competition. In particular, it was the only solver to qualify for the second round of the competition in all three benchmark categories. In this paper we investigate the use of HypBinRes and equality reduction in a preprocessor that can be used to simplify a CNF formula prior to SAT solving. We present empirical evidence demonstrating that such a preprocessor is extremely effective on large structured problems, including making some previously unsolvable problems solvable. The preprocessor is also able to solve a number of non-trivial instances by itself. Since the preprocessor does not have to worry about undoing changes on backtrack, nor about keeping track of reasons for intelligent backtracking, we are able to develop a new algorithm for applying HypBinRes that can be orders of magnitude more efficient than the algorithm employed inside of 2CLS+EQ. The net result is a technique that improves our ability to solve hard problems SAT problems.

1 Introduction

In this paper we investigate the use of a particular hyper-resolution rule, HypBinRes, along with equality reduction to preprocess CNF encoded SAT theories. HypBinRes is an inference rule that attempts to discover new binary clauses. These binary clauses are in turn used to detect that a literal is either forced or must be equivalent to other literals. In either case the input formula can then be reduced to one that does not contain that literal.

The HypBinRes rule was developed as part of the SAT solver 2CLS+EQ [1]. This solver was designed to further investigate the use of additional reasoning at every node of a DPLL search tree in order to prune the search tree [2, 3]. In the 2002 SAT competition 2CLS+EQ displayed the best all around performance, being the only solver to qualify for the second round of the competition in all three benchmark categories: industrial, handmade, and random problems. Furthermore, 2CLS+EQ was the top contributor to the SOTA (state of the art) solver. That is, it solved 18 problems that were not solved

* This research was supported by the Canadian Government through their NSERC program.

by any other solver, (second was zchaff which was the sole solver of 15 problems) [4]. This performance demonstrated that the right kind of additional reasoning can be very effective. Furthermore, in [2] empirical evidence was presented demonstrating that it is the specific use of HypBinRes and equality reduction that is key to 2CLS+EQ's performance.

The competition results demonstrated two other things about the use of HypBinRes. First, it can be quite expensive to utilize inside of the DPLL search, often resulting in a significant slow down in the per-node search rate of the solver. On some problems, the pruning produced is so dramatic that there is a significant net improvement in solution times. However, on many problems the overheads are such that state of the art DPLL SAT solvers, like zchaff [5], can solve the problem faster, even though they search many more nodes. Second, there are a number of problems on which HypBinRes and equality reduction is so effective that 2CLS+EQ can solve the problem without doing any search.

These two observations lead us to investigate the use of HypBinRes and equality reduction as a preprocessor for simplifying SAT problems prior to invoking a DPLL search. First, much of the expense in the implementation of HypBinRes comes from the fact that it was being used dynamically inside of a DPPL search. Using HypBinRes dynamically means that it must maintain sufficient information to allow all of the changes it makes to the theory to be undone on backtrack. Furthermore, because 2CLS+EQ utilizes intelligent backtracking, information also has to be maintained so that the reasons for failures can be computed. Since at each node HypBinRes and equality reduction can produce huge changes to the theory, computing and maintaining all of this information becomes quite expensive. All of that extra work can be avoided in a preprocessor. Second, that HypBinRes with equality reduction was actually able to solve some hard problems prior to search, gave us reason to believe that it could usefully simplify other problems even if it was not able to solve them completely.

In this paper we report on the results of our investigation into the use of HypBinRes and equality reduction as a preprocessor. A short summary being that such a preprocessor is often extremely effective in improving net solution times, in contrast with the mixed results about preprocessing reported in [6]. In the sequel we will first describe HypBinRes and equality reduction in more detail. Then we will sketch a new algorithm suitable for implementing it in a preprocessor. Empirical results from an implementation of this algorithm are presented next, followed by some conclusions.

2 HypBinRes+eq

HypBinRes is a rule of inference involving a hyper-resolution step (i.e., a resolution step that involves more than two input clauses). It takes as input a single n-ary clause ($n \geq 2$) $(l_1, l_2, ..., l_n)$ and $n-1$ binary clauses each of the form $(\bar{l}_i, \ell)$ ($i = 1, \ldots, n-1$). It produces as output the new binary clause (ℓ, l_n). For example, using HypBinRes hyper-resolution on the inputs (a, b, c, d), $(h, \bar{a})$, $(h, \bar{c})$, and $(h, \bar{d})$, produces the new binary clause (h, b).

HypBinRes is equivalent to a sequence of ordinary resolution steps (i.e., resolution steps involving only two clauses). However, such a sequence would generate clauses of intermediate length while HypBinRes side-steps this, only generating the final binary

clause. In a SAT solver it is generally counter productive to add all of these intermediate clauses to the theory.[1] However, can be very useful to add the final binary clause.

It should also be noted that if the input n-ary clause is itself binary, HypBinRes reduces to the simple resolution of binary clauses. For example, HypBinRes on the "n-ary" clause (a, b) and the clause $(h, \bar{a})$ yields the new binary clause (h, b).

HypBinRes could also be used to generate unit clauses, if we allow it to consider one more binary clause. For example, (a, b, c, d), $(h, \bar{a})$, $(h, \bar{b})$, $(h, \bar{c})$, and $(h, \bar{d})$, when hyper resolved together produces the unit clause (h). Equivalently, one can do as we do in our implementation. We can apply HypBinRes as specified above and then a separate single step of ordinary resolution of binary clauses. In our example, the HypBinRes step uses only the first 3 binary clauses would produce (h, d), then an ordinary resolution step with clause $(h, \bar{d})$ produces (h).

Once binary clauses are available equality reduction can performed. If the theory F contains $(\bar{a}, b)$ as well as $(a, \bar{b})$ (i.e,, $a \Rightarrow b$ as well as $b \Rightarrow a$), then we can generate a new formula EqReduce(F) by equality reduction. Equality reduction involves (a) replacing all instances of b in F by a, (b) removing all clauses which now contain both a and $\bar{a}$, (c) removing all duplicate instances of a (or $\bar{a}$) from all clauses. This process might generate new binary clauses.

For example, EqReduce$(\{(a, \bar{b}), (\bar{a}, b), (a, \bar{b}, c), (b, \bar{d}), (a, b, d)\}) = \{(a, \bar{d}), (a, d)\}$. Clearly EqReduce$(F)$ will have a satisfying truth assignment if and only if F does. Furthermore, any truth assignment for EqReduce(F) can be extended to one for F by assigning b the same value as a.

Finally, we can apply the standard reduction of unit clauses. If we have a unit clause (ℓ) in the theory, we can remove all clauses containing ℓ, and then remove $\bar{\ell}$ from all remaining clauses. We use UR(ℓ) to denote such an application of this inference rule. The iterative application of UR until no more unit clauses remain is commonly known as unit propagation UP.

We can apply unit reduction (UR), HypBinRes, and equality reduction to a CNF theory until no more new inferences can be made with these rules. We call the resultant theory the *HypBinRes+eq-closure*. A theory in which these three inference rules can infer nothing new is called HypBinRes+eq-closed. Interestingly, a Church-Rosser result holds for this collection of inference rules.

Theorem 1. *The HypBinRes+eq-closure of a CNF theory $\mathcal{F}$ is unique up to renaming. That is, the order in which the inference rules are applied is irrelevant, as long as we continue until we cannot apply them anymore.*

Proof. We show that these inference rules satisfy the Church-Rosser property. Namely, if a CNF theory T can be reduced to T_1 or T_2 by zero or more applications of the above three inference rules, then there exists another expression that both T_1 and T_2 can be reduced to (up to renaming[2]). From this we immediately obtain the theorem.

First, we show that for any sequence of two rule applications r_1 and r_2, there is some other sequence of rule applications $f(r_1, r_2)$ such that $r_2[r_1[T]]$ is equivalent to $f(r_1, r_2)[r_2[T]]$ up to renaming. This is shown by exhaustive case analysis. Each case

[1] These clauses are not like conflict clauses. Adding conflict clauses does appear to be useful.

[2] Renaming might be necessary because the EqReduce rules might be applied in different ways.

is easy, but there are many of cases. Hence, we only give a couple of examples. Say that r_1 is $\{(a, x), (a, y), (\bar{x}, \bar{y}, z)\} \vdash (a, z)$, and r_2 is $\{(a, z), (c, z), (\bar{c}, \bar{a}, \bar{s})\} \vdash (s, z)$. r_2 depends of r_1, and thus might not be applicable to T. In this case we view $r_2[T]$ as being a null operation, i.e., $r_2[T] = T$. Hence, $r_2[r_1[T]]$ is equivalent to $r_2[r_1[r_2[T]]]$. Another case is with the same r_1 but with r_2 being UR(x). Now $r_2[r_1[T]]]$ is equivalent to $r'_1[r_2[T]]$ where r'_1 is $\{(a, y), (\bar{y}, z)\} \vdash (a, z)$. From this result it follows that for any theory $r[\pi[T]]$ where π is a sequence of rule applications, there exists an alternate sequence π' such that $r[\pi[T]]$ is equivalent to $\pi'[r[T]]$: we simply push r in one step at a time.

Second, we observe that for any two theories T_1 and T_2 equivalent up to renaming, and any inference rule r_1, Church-Rosser holds for $r_1[T_1]$ and T_2. We simply rename the literals in r_1 according to the renaming function between T_1 and T_2 and apply the renamed r_1 to T_2: $r'_1[T_2]$. The result is clearly equivalent to $r_1[T_1]$. That is, the empty sequence and r'_1 transform $r_1[T_1]$ and T_2 to equivalent theories.

Finally, we consider two theories, T_1 and T_2 for which Church-Rosser holds. We show that for any rule application r Church-Rosser still holds for $r[T_1]$ and T_2. Church-Rosser means that there exists two sequences π_1 and π_2 such that $\pi_1[T_1]$ is equivalent to $\pi_2[T_2]$. Hence, $r[\pi_1[T_1]]$ is equivalent to $r'[\pi_2[T_2]]$ where r' is r appropriately renamed. By our first result, there are sequences π'_1 and π'_2 such that $\pi'_1[r[T_1]]$ is equivalent to $r[\pi_1[T_1]]$, and $\pi'_2[r'[T_2]]$ is equivalent to $r'[\pi_2[T_2]]$. Thus π_1 applied to $r[T_1]$ and $\pi'_2; r'$ applied to T_2 make these two theories equivalent. With the previous base case and this inductive step we have shown that Church-Rosser holds.

Now we can conclude that since no rules can be applied in the HypBinRes+eq-closure it must be the case that we that any two sequences of rules reaching closure must yield the same theory (to renaming). Church-Rosser holds between the two theories obtained by these two sequences. Hence, if these theories were different, there would be a non-empty sequence of rules applicable to at least one of them (to move them both to an equivalent theory). That is, the theories could not both be HypBinRes+eq-closed.

The practical significance of Theorem 1 is that we are free to apply these inference rules in any order; we are guaranteed to reach the same final result. We now turn our attention to an algorithm for computing the HypBinRes+eq-closure. Our new algorithm does not implement HypBinRes directly, rather it exploits the close relation between HypBinRes and unit propagation.

2.1 UP and HypBinRes+eq

Unit propagation is the iterative procedure of applying all unit reduction rules until no more unit clauses remain. Unit propagation can also be done on a trial basis. That is we can choose a literal to set to be true and then perform unit propagation. We call this *unit propagating a literal*, and denote it UP(a), where a is the literal that has been initially set to true. When UP(a) causes another literal ℓ to become true we use the notation UP$(a) \vdash \ell$. If UP$(a) \vdash \ell$ as well as UP$(a) \vdash \bar{\ell}$, we have detected that a is a *failed literal*, and it must be the case that the original theory $\mathcal{F} \vdash \bar{a}$. We can then reduce $\mathcal{F}$ by performing UP$(\bar{a})$.

In the sequel we will generally suppress mention of the underlying CNF theory, $\mathcal{F}$, upon which the various the reasoning processes are being run.

Theorem 2. *UP is more powerful than a single HypBinRes resolution step, but not as powerful as a sequence of HypBinRes resolution steps. More precisely:*

1. *If (a, b) can be produced by a single HypBinRes step, then either UP$(\bar{a}) \vdash b$ or UP$(\bar{b}) \vdash a$.*
2. *There are theories from which a binary clauses (a, b) can be produced from a sequence of HypBinRes steps, but neither UP$(\bar{a}) \vdash b$, nor UP$(\bar{b}) \vdash a$.*
3. *In a theory with no unit clauses (we can remove all units by an initial unit propagation phase), if UP$(\bar{a}) \vdash b$ then there is a sequence of HypBinRes steps that produce (a, b).*

Proof. (1) Any HypBinRes step is of the form $\{(l_1, l_2, ..., l_n), (\ell, \bar{l}_1), \ldots, (\ell, \bar{l}_{n-1})\}$, $\vdash (\ell, l_n)$, and UP$(\bar{\ell}) \vdash l_n$. Note that it need not be the case that UP$(\bar{l}_n) \vdash \ell$.

(2) An example is $\{(a, x), (a, y), (\bar{x}, \bar{y}, c), (\bar{c}, h), (\bar{c}, i), (\bar{i}, \bar{h}, q), (b, m), (b, o), (\bar{m}, \bar{o}, \bar{q})\}$. The binary clauses (a, c), $(\bar{c}, q)$, and $(\bar{q}, b)$ can be produced by 3 HypBinRes steps, after which two more resolution steps produce (a, b). However, UP$(\bar{a}) \vdash \{x, y, c, h, i, q\}$, but not b, and UP$(\bar{b}) \vdash \{m, o, \bar{q}\}$ but not a.

(3) We prove this result by induction. First we define an ordering on the literals entailed by UP$(\bar{a})$. Stage 0 of UP$(\bar{a})$ involves reducing the theory by removing a from all clauses, and removing all clauses containing $\bar{a}$. All literals appearing in unit clauses of the reduced theory are said to be entailed at step one. At stage 1 the one step literals are used to further reduce the clauses of the theory, and all literals appearing in new unit clauses are said to be entailed at step two. In general, the literals entailed at step i are those appearing in unit clauses of the reduced theory produced at stage $i - 1$. We prove the theorem by induction on the stage at which b is produced.

If b is a step 1 literal then the clause (a, b) must have appeared in the initial theory: i.e., a zero length sequence of HypBinRes steps suffices.

Say b is entailed at step i, and that it was entailed by the clause $(l_1, \ldots, l_k, b)$ becoming unit. Hence, the negation of each of the l_i was entailed at earlier steps, and by induction for each there is a sequence of HypBinRes steps producing the binary clause $(a, \bar{l}_j)$ for each $j \in \{1, \ldots, k\}$. Hence, one more step of HypBinRes suffices to produce (a, b).

2.2 Achieving HypBinRes+eq-closure with UP

Achieving HypBinRes+eq-closure involves repeatedly applying HypBinRes, UR, and equality reduction until nothing new can be inferred. Theorem 2 shows that we can achieve HypBinRes-closure by repeatedly applying UP on the literals of the theory.

More precisely, we first reduce the theory by unit propagating all unit clauses it might contain. Then for each remaining literal ℓ we can perform UP(ℓ), adding to the theory a new binary clause $(\bar{\ell}, a)$ for every literal a such that UP$(\ell) \vdash a$. By (1) above, one pass over all of the literals ensures that we find all binary clauses that can be inferred by one HypBinRes step. Adding the entailed binary clauses then ensures that the second pass can find all binary clauses inferable by two HypBinRes steps. By (2) we must add the entailed binary clauses found in the first pass, else UP would not be powerful enough. Adding these clauses makes all of the inputs to the second HypBinRes step

available in the theory, and by (1) allows UP to capture the second HypBinRes step. These passes are continued until we find no new binary clauses; clearly at this stage we have achieved HypBinRes-closure: there is no instance of the HypBinRes rule that can be applied.

Equality reduction and unit propagation can now added to compute the HypBinRes+eq-closure. One obvious way to see this is to consider the iterative process were we wait until HypBinRes closure is achieved, then perform all equality reduction and unit propagations, then iterate these steps again until we find nothing new. By Theorem 1 this particular sequence of operations will compute the HypBinRes+eq-closure. In practice, however, the flexibility ensured by Theorem 1 is very important for efficiency. For example, it is always a good idea to perform UR immediately whenever we find a unit clause.

Part (3) of Theorem 2 tells us that we do not achieve anything greater than HypBinRes-closure using multiple applications of UP: UP cannot infer anything more than HypBinRes. Hence the process just described computes precisely the HypBinRes+eq-closure.

2.3 A Real Algorithm

The process described in the previous section would make a hopelessly inefficient algorithm. However, we can develop an efficient algorithm by using UP in a more refined manner that tries to avoid consuming too much space and wasted work. The basic idea is that we can often tell when UP(ℓ) for some literal ℓ will not yield anything new. A good example of this is when UP$(a) \vdash \ell$ and UP(a) yields nothing new—UP(ℓ) cannot either. Space is also an issue with the previous process. If we add a binary clause $(\bar{\ell}, a)$ for every a such that UP$(\ell) \vdash a$, we could end up storing the transitive closure of the binary subtheory, which can be quadratic in the number of literals. This would make it impossible to deal with the large CNF theories that are now commonplace.

Our algorithm utilizes the implicit implication graph represented by a set of binary clauses [7]: the nodes are all of the literals in the theory and each binary clause (a, b) represents the two edges $\bar{a} \Rightarrow b$ and $\bar{b} \Rightarrow a$. In the following discussion we will interchangeably refer to a set of binary clauses as an implication graph and vice versa. Our implementation actually works with sets of binary clauses, performing operations on the implication graph (like traversing it) by corresponding operations on the binary clauses.

First we remove all unit clauses from the input CNF by doing an initial unit propagation. Then all of the input binary clauses are collected, and used to represent an implication graph. The aim of the algorithm is to generate an augmented implication graph (new set of binary clauses) that satisfies the following property: if (a, b) is present in the HypBinRes+eq-closure, then in the implication graph b is reachable from $\bar{a}$ and a is reachable from $\bar{b}$. In particular, the clause (a, b) need not be in the final set of binary clauses, but it must be derivable by resolution steps involving only the computed set of binary clauses. Thus we avoid materializing the transitive closure of the implication graph.[3]

[3] The original 2CLS+EQ algorithm did explicitly represent the transitive closure of the implication graph as do the two other preprocessors that reason with the binary clauses, 2SIMPLIFY

Table 1 Graph Search Algorithm for computing HypBinRes-closure

```
Visit(ℓ)
1.  if ℓ is MARKED return
2.  CurrentImplicants := {}
3.  foreach l s.t. (l̄, l) is in the implication graph
4.    if l ∈ CurrentImplicants
5.       delete (l̄, l) from the implication graph.
6.    else
7.       Visit(l)
8.       CurrentImplicants ∪= DescendantsOf(l)
9.  UPImplicants := {l s.t. UP(ℓ) ⊢ l}
10. NewImplicants := UPImplicants - CurrentImplicants
11. foreach l ∈ NewImplicants
12.   if l ∈ CurrentImplicants
13.      continue
14.   else
15.      add (l̄, l) to implication graph.
16.      Visit(l)
17.      CurrentImplicants ∪= DescendantsOf(l)
18. MARK ℓ
19. return
```

The basic algorithm is presented Table 2.3. It is based on a depth first post-order traversal of the implication graph $\mathcal{G}$. The traversal is started at the set of literals (nodes) that have no parents in $\mathcal{G}$, i.e., they do not appear in any binary clauses, only their negations do. When the search completes its visit of a literal that literal is marked. The mark indicates that unit propagating that literal will not be able to discover anything new (at least for now).

First the algorithm visits all current children of the literal ℓ, recursively achieving a marked status for each child. As the children are visited the set of literals currently reachable from ℓ are accumulated (line 8). It can be that a new edge is added to the graph when recursively solving a child, and that new edge might reach some other child l of ℓ. This means that there is now another path to l from ℓ rather than the direct edge $(\bar{\ell}, l)$. To keep the graph small, the direct edge can be deleted (line 5). After all of the current children are marked, ℓ is unit propagated and all of the entailed literals accumulated (line 9). Note that ℓ entails all of its children *simultaneously*, so its set of unit implicants will in general be larger than the union of its children's unit implicants.

Any unit implicant not in the set of currently reachable literals then needs to be added to the graph (in order to converge on the HypBinRes-closure). This is done at lines 11–17. Each of these new children can then be visited to ensure that they are properly marked. Again, the algorithm tries to minimize the number of new children added, by skipping those that are already reachable from some previously processed child (line 13).

Note that the algorithm makes no attempt to minimize the size of the implication graph (although this is an option if space is very tight). For example, it does not go

[8] and 2CL_SIMP [3]. All of these algorithms have difficulty dealing with larger formulas. The preprocessor we report on here follows more the "lean" approach suggested in [9, 10].

back to check whether previous children might be reachable from children processed later—this would be too expensive.

Dealing with equivalence reduction and unit propagation of forced literals is conceptually straightforward. Equivalent literals can be detected with a depth-first search using Tarjan's strongly connected components (SCC) algorithm [11] (or more modern improvements). And forced literals (detected during the unit propagations of the algorithm) can be unit propagated. The complexity lies with finding ways to perform these operations incrementally and in more efficient orders. For example, we want to interrupt the graph search to unit propagate forced literals as soon as they are detected.

To make equivalent literal detection incremental we do an initial SCC and equality reduction prior to invoking **Visit**. Subsequently, we restrict the SCC computation so that it is only invoked when a new binary clause has been added. Furthermore we only search in the neighborhood of the new edges for newly created components (any new SCC must include one of the new edges). In this way we avoid examining the entire graph for every small change.

For forced literals, whenever a failed literal is detected, we interrupt the graph search and immediately unit propagate the negation of the failed literal. This marks some of the nodes in the graph as true or false, and can also generate new binary clauses that are immediately added to the graph. The graph search is then continued. To deal with the changes in the graph we make it backtrack immediately from any true or false node, otherwise it continues as before. Hence, it will only traverse the new edges in that part of the graph it has not yet searched. We utilize the literal marks to ensure that it eventually goes back to consider the new edges in the part it has already searched

All of the inference rules utilize the literal marks to inform the graph search of any incremental work it still needs to do. The mark represents the condition that unit propagating the literal will not yield any new edges in the graph. So whenever one of the rules is activated, it unmarks those literals that might now generate something new under unit propagation. There are only two cases:

(1) A new binary clause (a, b) is added to the theory. This happens when an n-ary clause is reduced to binary. The new clause represents the new edges $\bar{a} \Rightarrow b$ and $\bar{b} \Rightarrow a$. This means that $\bar{a}$ and $\bar{b}$ along with anything upstream of them could now potentially generate new unit implicants. Hence, all these literals are unmarked so that the graph search can reconsider them.

(2) An n-ary clause $(l_1, \ldots, l_{k+1})$ is reduced in size to the clause $(l_1, \ldots, l_k)$. This could happen from a unit propagation forcing $\bar{l}_{k+1}$ or from $\bar{l}_{k+1}$ becoming equivalent to one of the other l_i. For any literal ℓ such that UP$(\ell) \vdash \bar{l}_i$ for any $i \in \{1, \ldots, k\}$ it could that UP(ℓ) now makes this clause unit. For example, it could be that UP$(\ell) \vdash \{\bar{l}_1, \ldots, \bar{l}_{k-1}\}$. Previously, this would have only reduced the clause to binary, but now that the clause has been reduced in size we get that UP$(\ell) \vdash l_k$. Hence, we must unmark ℓ. In general, we unmark all literals upstream of any of the $\bar{l}_i$. Note that (2) is simply a generalization of (1).

Table 2 The Hypre Preprocessing Algorithm

Hypre

```
1.  Unit Propagate all unit clauses.
2.  Find all SCC and perform all EqReduce steps
3.  UNMARK all nodes in implication graph
4.  while there is an UNMARKED node
5.   foreach ℓ s.t. ℓ is marked, and has no parents
6.      Visit(ℓ)
7.      Perform Incremental SCC
9.      UNMARK all nodes according to the two cases above.
10. end.
```

Visit(ℓ)

```
1.    if ℓ is MARKED return
2.    CurrentImplicants := {}
3.    foreach l s.t. (¬ℓ, l) is in the implication graph
4.      if l ∈ CurrentImplicants
5.         delete (¬ℓ, l) from the implication graph.
6.      else
7.         Visit(l)
7.5        if ℓ is MARKED return
8.         CurrentImplicants ∪= DescendantsOf(l)
9.    UPImplicants := {l s.t. UP(ℓ) ⊢ l}
9.1   if contradiction detected
9.2      UP(¬ℓ)
9.3      MARK all literals whose truth value is set
9.4      UNMARK all nodes according to the two cases above.
10.   NewImplicants := UPImplicants - CurrentImplicants
11.   foreach l ∈ NewImplicants
12.     if l ∈ CurrentImplicants
13.        continue
14.     else
14.5       Note that these new edges do not cause unmarking
15.        add (¬ℓ, l) to implication graph.
15.5       Note that these visits cannot detect a contradiction
15.6       since line 9 didn't.
16.        Visit(l)
17.        CurrentImplicants ∪= DescendantsOf(l)
18.   MARK ℓ
19.   return
```

With this unmarking process, we run the graph search until there is no unmarked literal (or a contradition is detected, or all clauses become satisfied). Unit propagations of forced literals are done immediately, and strongly connected component detection and equivalent literal reduction performed after the graph search is completed. Both the unit propagations as well as the equality reductions might remove node marks, in which case we may have to perform another iteration. Once no more changes can be made we have achieved HypBinRes+eq-closure.

The final algorithm is presented in Table 2.3. The changes to **Visit** are indicated by fractional line numbers.

3 Empirical Results

Perhaps the most dramatic demonstration of the power of HypBinRes+eq-closure comes from the two problems c6288-s, and c6288 from João Marques-Silva's MITERS test

Table 3 Performance on two "hard" MITERS problems.

Problem	HyPre	Berkmin	2SIMPLIFY +Berkmin	2CL_SIMP +Berkmin	Zchaff	2SIMPLIFY +Zchaff
c6288-s	1.05	> 604,800	> 30,000	> 604,800	> 604,800	> 604,800.0
c6288	1.05	> 604,800	> 30,000	> 604,800	> 604,800	> 604,800.0

suite. Both of these problems are detected to be UNSAT by the preprocessor. Table 3 shows the time required by the preprocessor and by the SAT solvers BerkMin 5.61[4], and zchaff. All times reported are in CPU seconds on a 3GB, 2.4 GHx Pentium-IV. These two solvers were used in our tests are they are probably the most powerful current SAT solvers.

It can be seen that although the preprocessor solves both problems is about a second, the unsimplified problem is unsolvable by either of these state of the art SAT solvers. We ran each problems for a week before aborting the run. We also tried two other preprocessors (discussed in more detail in Section 4) 2SIMPLIFY [8] and 2CL_SIMP [3]. Both of these preprocessors do some form of binary clause reasoning, but as can be seen from the table, neither are effective on this problem.

Similar results are achieved on the BMC2 test suite from the 2002 competition, Table 4. Again all of these problems were solved by the preprocessor (all are SAT). These problems were also solved by the original 2CLS+EQ solver without search. However, as noted above the implementation of HypBinRes+eq-closure in 2CLS+EQ is much less efficient. This is verified by the results of the table.

Table 4 Performance on the BMC2 suite

Problem	HyPre	2CLS+EQ	Berkmin	Zchaff	Problem	HyPre	2CLS+EQ	Berkmin	Zchaff
BMC2-b1	0.01	0.03	0.00	0.02	BMC2-b4	2.33	7.44	10.56	36.81
BMC2-b2	0.05	0.16	0.05	0.13	BMC2-b5	28.31	321.60	520.92	3492.23
BMC2-b3	0.31	0.84	0.62	1.89	BMC2-b6	214	11,193	3,426	>20,000

Table 5 shows some additional results summed over families of problems. Most of these families came from the 2002 SAT competition. The families 03_rule and 06_rule come from the IBM Formal Verification Benchmark suite (we did not run Zchaff on these families). The number in brackets after the family name is the number of problems in the family. The time given is the total time to "solve" all problems in the family in CPU seconds. In these experiments a single problem time-out of 20,000 CPU seconds was imposed, and 20,000 was added to the total time to "solve" the family for each failed problem. The number in brackets after the total time is the number of problems the solver failed to solve. For the preprocessor "failure to solve" means that the problem was not resolved at preprocessing time. The first set of times is the time required by the

[4] The most recent public release.

Table 5 Performance on various families of problems. Time in CPU seconds. Bracketed numbers indicate number of failures for that family. Numbers in **bold** face indicate family/solver combinations where preprocessing reduced the net solution times (i.e. preprocessing plus solving), or allowed more problems to be solved.

Family (#probs)	HyPre	Berkmin-Orig	Berkmin-Pre	Zchaff-Orig	Zchaff-Pre
BMC (76)	18,819 (53)	41,751 (1)	29,254 (1)	48,225 (1)	30,009 (1)
BMCTA (2)	1,210 (2)	7,835 (0)	**6,015** (0)	34,741 (1)	**24,977** (1)
Cache (5)	23,620 (5)	61,771 (2)	42,327 (2)	30,725 (1)	43,932 (2)
w10 (4)	1,377 (4)	516 (0)	65 (0)	1,603 (0)	615 (0)
Checker (4)	8 (4)	4,092 (0)	**3,191** (0)	1,763 (0)	2,318 (0)
Comb (3)	352 (3)	22,017 (1)	**21,196** (1)	60,000 (3)	**36,845** (1)
IBM-Easy (2)	1,454 (2)	1,092 (0)	61 (0)	678 (0)	2 (0)
IBM-Med (2)	47,880 (2)	5,075 (0)	6 (0)	13,342 (0)	6 (0)
IBM-Hard (3)	1,767 (3)	38,997 (1)	**3,560** (0)	47,581 (2)	**28,920** (0)
Lisa (29)	16 (29)	107,379 (2)	135,612 (3)	185,091 (7)	**111,482** (2)
f2clk (3)	674 (3)	22,663 (1)	**19,451** (0)	40,662 (2)	**28,687** (1)
fifo (4)	875 (3)	56,014 (2)	**1,015** (0)	29,040 (1)	**8,929** (0)
ip (4)	735 (4)	1,466 (0)	**145** (0)	30,667 (1)	**1,687** (0)
w08 (3)	8,672 (2)	8,542 (0)	151 (0)	27,670 (1)	**309** (0)
rule_03 (20)	4,409 (14)	134,258 (6)	**10,501** (0)		
rule_06 (20	15,509 (5)	113,360 (4)	**5,587.62** (0)		

preprocessor, then for each of the two SAT solvers we give the time require to solve the original problems, then the time require to solve the preprocessed problems (i.e., those that had not be resolved by the preprocessor).

The data shows that HypBinRes+eq-closure almost always makes the problems easier to solve. Only for Cache with Zchaff, and Lisa with Berkmin does the time to process the family increase. The preprocessor also allows Berkmin to solve one more problem from the IBM-Hard family, one more from the f2clk, two more from the fifo family, 6 more from the 03_rule family, and 4 more from the 06_rule family. It improves Zchaff even more, allowing Zchaff to solve 2 more from Comb, 2 more from IBM-Hard, five more from Lisa, one more from f2clk, fifo, ip, and w08. Preprocessing on rare occasion makes a problem harder as in the case for one problem in the Lisa family for Berkmin, and one problem in the Cache family for Zchaff. Interestingly, for the Lisa family and Berkmin, the preprocessing allowed Berkmin to solve one problem it could not solve before, and stopped it from solving one that it could solve before.

Frequently, especially for Berkmin, once we add the time to perform the preprocessing the gains in total solution time are minimal, or even negative. The net gains for Zchaff are better. Nevertheless, the preprocessor almost always makes the problem easier, so only in the case of IBM-Med does it cause a serious slow down in the total solution time. We feel that this is acceptable given that it also allows some problems to be solved that previously could not be solved.

Table 6 provides some data about typical reductions in size in the CNF formulas produced by the preprocessor (these are of course formulas that are unresolved by pre-

Table 6 Size reduction on various problems instances. #Vars is the number of variables, #N-ary is the number of non binary clauses, and #Bin is the number of binary clauses

Problem	Original			After Processing		
	#Vars	#N-ary	#Bin	#Vars	#N-ary	#Bin
BMC-b10	42,405	42,327	98,804	10,229	14,978	35,263
BMC-b74	209,211	208,117	501,283	181,057	182,006	472,656
BMCTA-b1	64,909	87,909	108,066	40,705	73,866	158,966
BMCTA-b2	87,029	118,197	144,802	55,871	100,978	215,793
Cache-b1-s1-0	113,080	96,120	326,995	63,872	70,160	191,079
Cache-b2-s2-0	227,210	195,260	666,835	117,991	131,310	377,647
w10-b3	32,745	21,123	82,290	12,059	13,995	46,922
w10-b4	36,291	23,499	91,621	13,657	15,856	53,158
Checker-b3	1,155	33,740	5,528	1,029	30,158	4,903
Checker-b4	1,188	34,732	5,688	1,062	31,150	5,069
Comb-b2	31,933	21,364	91,097	15,707	17,517	84,952
Comb-b3	4,774	3,342	12,988	2,405	2,806	9,552
IBM-Easy-b2	29,605	35,046	115,506	15,276	29,297	59,389
IBM-Easy-b3	48,109	58,023	156,958	17,967	34,014	94,779
IBM-Med-b1	212,091	207,243	2,313,020	64,779	133,805	1,204,190
IBM-Med-b2	125,646	122,454	1,378,757	29,165	69,545	339,252
IBM-Hard-b2	22,984	27,508	90,298	11,788	21,803	49,480
IBM-Hard-b3	33,385	41,040	121,986	14,757	26,583	72,340
Lisa-b28	1,453	6,954	968	1,347	6,494	1,439
Lisa-b21	2,125	10,464	1,576	1,447	6,724	1,260
f2clk-b2	27,568	20,352	58,761	10,395	11,478	56,342
f2clk	34,678	25,662	74,001	14,895	16,388	84,901
fifo-b3	194,762	118,508	407,666	29,724	52,026	87,117
fifo-b4	259,762	158,108	543,766	41,574	72,826	122,374
ip-b3	49,967	36,407	124,732	14,042	15,592	48,462
ip-b4	66,131	48,275	165,208	19,226	21,352	65,791
w08-b2	120,367	81,080	344,201	34,861	40,515	211,796
w08-b3	132,555	89,489	379,993	40,856	47,091	243,426

processing). The data shows that for the most part the preprocessor is able to achieve a significant reduction in the number of variables in the theory. Furthermore, it is clear that for problems like IBM-Med-b1, which contains more than 2 million binary clauses, it would be impossible to maintain the transitive closure of the binary clauses. (In fact, 2CLS+EQ, whose algorithm for achieving HypBinRes+eq-closure involves realizing the transitive closure, is unable to complete the preprocessing of the larger problems.) The implicit representation of these clauses in an implication graph used in our implementation avoids this blowup. In fact, we see that generally the final theory has a reduced number of binary clauses, even though the outputed set of clauses contains within its transitive closure all implications of HypBinRes+eq.

On a number of families, e.g., Bart, Homer, GridMN, Matrix, Polynomial, sha, and fpga-routing-2 the preprocessor had no effect. In these problems there are no useful binary clauses in the original theory.

It can be noted however, that at least for the Bart family using HypBinRes+eq dynamically during the search had dramatic benefits even though it was useless as a preprocessor. In particular, from data gathered during the SAT-2002 competition 2CLS+EQ was able to solve all 21 instances of the Bart family in 1.1 seconds with an average of only 25 nodes expanded during the search. Berkmin on the other hand took 77.6 seconds to solve the family and zchaff required 39,708 seconds and still was only to solve 5 of the 21 problems.

4 Previous Work

Some of the processing done in our preprocessor is similar to techniques used in previous preprocessors, e.g., COMPACT [12], 2SIMPLIFY [8], 2CL_SIMP [3], and COMPRESSLITE[14].

COMPACT, and COMPRESSLITE both try to detect failed literals, with COMPRESSLITE doing a more elaborate test including finding literals forced by both ℓ and $\bar{\ell}$. Both of these preprocessors use unit propagation to do their work. However, the inferences performed are not as powerful. In particular, COMPACT employs only the failed literal rule: if UP$(a) \vdash$ FALSE then force $\bar{a}$. By Theorem 2, HypBinRes will also detect all failed literals. COMPRESSLITE on the other hand employs two inference rules

1. If UP$(\ell) \vdash a$ and UP$(\bar{\ell}) \vdash a$, then force a.
2. If UP$(\ell) \vdash a$ and UP$(\bar{\ell}) \vdash \bar{a}$ then perform EqReduce replacing a by ℓ.

HypBinRes+eq captures both rules. For (1) HypBinRes will conclude $(\bar{\ell}, a)$ and (ℓ, a) from which a can be inferred. For (2) HypBinRes will conclude $(\bar{\ell}, a)$ and $(\ell, \bar{a})$, from which EqReduce can make the equality reduction. HypBinRes+eq-closure is in fact more powerful that these two rules. For example, COMPRESSLITE is only able to remove 3.3% of the literals from the two Miters problems c6288-s and c6288, whereas our HypBinRes+eq preprocessor can completely solve these problems. Judging from the times quoted in [14] COMPRESSLITE is also currently much less efficient than our implementation.

2SIMPLIFY and 2CL_SIMP are much closer to our work in that both try to do extensive reasoning with binary clauses. 2CL_SIMP does not employ a HypBinRes rule, but it does more extensive subsumption reasoning. 2SIMPLIFY on the other hand employs the implication graph and also implements a rule that on closer inspection turns out to be equivalent to HypBinRes. This rule is called the derive shared implications [8]. The rule is as follows: from a n-ary clause $(l_1, \ldots, l_n)$ the set of literals reachable from l_i (in the implication graph) is computed for all $i \in \{1, \ldots, n\}$. All literals in the intersection of these sets is detected to be forced. For a literal a to be in each of these sets, it means that $l_i \Rightarrow a$ for all i. That is, we have the n binary clauses $(\bar{l}_1, a)$, $\ldots$, $(\bar{l}_n, a)$, and the n-ary clause $(l_1, \ldots, l_n)$. This is clearly a restricted case of HypBinRes. In fact, 2SIMPLIFY searches in a very limited way for literals in all but one of these sets, and

from them learns new binary clauses (this version is precisely HypBinRes). 2SIMPLIFY also performs equality reduction.

However, 2SIMPLIFY does not compute the closure (in fact it does not even detect all single applications of HypBinRes). Nor was any theoretical analysis of the HypBinRes rule provided. Furthermore, like 2CL_SIMP it computes the transitive closure of the implication graph (binary subtheory). Thus on many examples these two programs fail to run. For example, on the BMC family of 76 problems, 2SIMPLIFY aborts on 68 of the problems, and 2CL_SIMP on 45 problems. Even when these programs run they seem to be ineffective on industrial benchmarks, or at least much less effective than the HypBinRes+eq preprocessor. Table 3 provided some evidence of this. Table 7 provides some more.

Table 7 2SIMPLIFY and 2CL_SIMP Performance on the BMC2 suite. Total solution time (preprocessor + solver) shown.

Problem	2CL_SIMP + Berkmin	2SIMPLIFY + Berkmin	Berkmin	Problem	2CL_SIMP + Berkmin	2SIMPLIFY + Berkmin	Berkmin
BMC2-b1	0.01	0.01	0.00	BMC2-b4	16.58	0.47	10.56
BMC2-b2	0.08	0.05	0.05	BMC2-b5	226.88	251.55	520.92
BMC2-b3	0.83	0.40	0.62	BMC2-b6	3,751.03	3,849.22	3,426.58

5 Conclusion

We have presented a new preprocessor for CNF encoded SAT problems. The empirical evidence demonstrates that although it can sometimes take a lot of time to do its work, it generally produces a significant simplification in the theory. This simplification is sometimes the difference between being able to solve the problem and not being to solve the problem. Hence, it is fair to say that it improves our ability to solve hard SAT problems.

6 Acknowledgements

We thank the referees for very useful comments and for pointing out that the binary clause reasoning employed in [3] had also been implemented as a preprocessor.

References

1. Bacchus, F.: Enhancing davis putnam with extended binary clause reasoning. In: Proceedings of the AAAI National Conference. (2002) 613–619
2. Bacchus, F.: Exploring the computational tradeoff of more reasoning and less searching. In: Fifth International Symposium on Theory and Applications of Satisfiability Testing (SAT-2002). (2002) 7–16 Available from www.cs.toronto.edu/~fbacchus/2clseq.html.

3. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. In Johnson, D., Trick, M., eds.: Cliques, Coloring and Satisfiability. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996) 559–586
4. Simon, L., Berre, D.L., Hirsch, E.A.: The sat2002 competition. Technical report, www.satlive.org (2002) available on line at www.satlive.org/SATCompetition/.
5. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proc. of the Design Automation Conference (DAC). (2001)
6. Lynce, I., Marques-Silva, J.P.: The puzzling role of simplification in propositional satisfiability. In: EPIA'01 Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving (EPIA-CSOR). (2001) available on line at sat.inesc.pt/˜jpms/research/publications.html.
7. Aspvall, B., Plass, M., Tarjan, R.: A linear-time algorithms for testing the truth of certain quantified boolean formulas. Information Processing Letters **8** (1979) 121–123
8. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. In: Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI). (2001) 515–522
9. Morrisette, T.: Incremental reasoning in less time and space. submitted manuscript (2002) available from the author e-mail threesat2000@yahoo.com.
10. Van Gelder, A.: Toward leaner binary-clause reasoning in a satisfiability solver. In: Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002). (2002) on line pre-prints available at gauss.ececs.uc.edu/Conferences/SAT2002/sat2002list.html.
11. Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing **1** (1972) 146–160
12. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in random 3-sat. Artificial Intelligence **81** (1996) 31–57
13. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI). (1997) 366–371
14. Berre, D.L.: Exploiting the real power of unit propagation lookahead. In: LICS Workshop on Theory and Applications of Satisfiablity Testing. (2001)