A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas

Maher Mneimneh[1], Inês Lynce[2], Zaher Andraus[1],
João Marques-Silva[2], and Karem Sakallah[1]

[1] University of Michigan,
{maherm, zandrawi, karem}@umich.edu
[2] Technical University of Lisbon, Portugal
{ines,jpms}@sat.inesc-id.pt

Abstract. We tackle the problem of finding a smallest-cardinality MUS (SMUS) of a given formula. The SMUS provides a succinct explanation of infeasibility and is valuable for applications that rely on such explanations. We present a branch-and-bound algorithm that utilizes iterative MAXSAT solutions to generate lower and upper bounds on the size of the SMUS, and branch on specific subformulas to find it. We report experimental results on formulas from DIMACS and DaimlerChrysler product configuration suites.

1 Introduction

Explaining the causes of infeasibility of Boolean formulas has practical applications in numerous fields: electronic design, formal verification, and artificial intelligence. In design applications, for example, a large Boolean function is formed such that a feasible design is obtained when the function is satisfiable, and design infeasibility is indicated when the function is unsatisfiable. An example of this is the routing of signal wires in an FPGA. We are usually interested in a "minimal" explanation of infeasibility that excludes irrelevant information. For Boolean formulas in conjunctive normal form (CNF), the notion of minimality is defined as follows. Consider an unsatisfiable CNF formula φ. An unsatisfiable subformula (a US) of φ is a minimal-unsatisfiable subformula (MUS) if it becomes satisfiable whenever any of its clauses is removed. Algorithms for finding MUSes are presented in [1, 2, 7].

Since an unsatisfiable formula might have many MUSes, specific ones might be of greater value based on the application. For example, in verification applications the quality of refinement affects the number of iterations in the abstraction-refinement flow. While a US represents a set of spurious behaviors, an MUS represents a larger set of spurious behaviors. Thus, an MUS in general, and a smallest cardinality MUS (SMUS) in particular, tend to be more effective in reducing the number of refinement steps. Lynce et al. [4] presented an algorithm that computes an SMUS by implicitly searching all USes of a formula.

In this paper, we tackle the problem of finding an SMUS by a branch-and-bound algorithm that utilizes iterative MAXSAT solutions to generate lower and upper bounds on the size of the SMUS, and branch on specific subformulas to find it. The paper is

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 467 – 474, 2005.

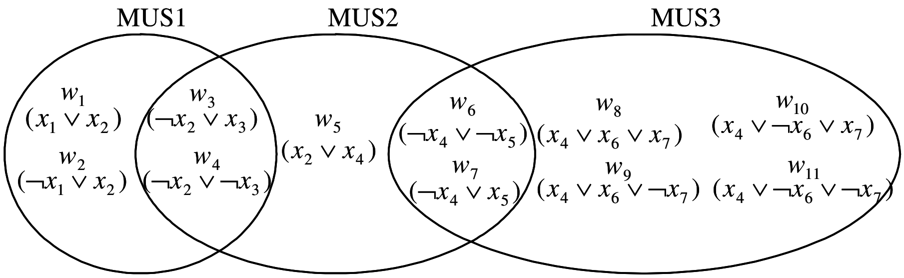

Fig. 1. The formula $\prod$ of (1) and its MUSes

organized as follows. In Section 2, we review basic definitions and notations. In Section 3, we present our algorithm for finding the SMUS. Results on unsatisfiable formulas from DIMACS and DaimlerChrysler Automotive Product Configuration benchmarks are presented in Section 4.

2 Preliminaries

Consider an unsatisfiable formula $\varphi = w_1 w_2 \ldots w_m$. A US α of φ is an MUS if removing every clause results in a formula that is satisfiable. α is an SMUS if it is an MUS and for all other MUSes β of φ, $|\alpha| \le |\beta|$. We denote the set of MUSes of φ by $Muses(\varphi)$. The MAX-SAT problem finds a satisfiable subformula α of φ with the maximum number of clauses; we call $\varphi - \alpha$ a MAX-SAT solution of φ. We solve MAX-SAT by reducing it to an integer optimization problem as follows. We define a set of m new Boolean *clause selector* variables $Y = \{y_1, y_2, \ldots, y_m\}$, and construct a new formula $\phi = (\neg y_1 \vee w_1)(\neg y_2 \vee w_2)\ldots(\neg y_m \vee w_m)$. The MAX-SAT solution is obtained by maximizing the objective $y_1 + y_2 + \ldots + y_m$ subject to the clauses of ϕ. Consider the Boolean formula:

$$\varphi = (x_1 \vee x_2)(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)(\neg x_2 \vee \neg x_3)(x_2 + x_4)(\neg x_4 \vee \neg x_5)(\neg x_4 \vee x_5)$$
$$(x_4 \vee x_6 \vee x_7)(x_4 \vee x_6 \vee \neg x_7)(x_4 \vee \neg x_6 \vee x_7)(x_4 \vee \neg x_6 \vee \neg x_7) \quad (1)$$

We call w_i $1 \le i \le 11$ the *ith* clause of φ. φ has three MUSes that are illustrated with a Venn diagram in Figure 1. MUS1 is an SMUS. There are several possible MAX-SAT solutions for φ. Two of them are $\{w_3, w_7\}$, and $\{w_1, w_6\}$.

3 Computing a Smallest MUS

A simple approach to compute an SMUS of a formula is to generate all MUSes and then select the smallest MUS. This approach is hindered by the fact that the number of MUSes of a formula can be exponential in the number of its variables. To solve this problem efficiently, we present a branch-and-bound algorithm to compute the SMUS.

3.1 Lower and Upper Bounds

We utilize iterative MAX-SAT solutions to get a lower bound on the size of the SMUS. Let us consider φ in (1) and its extended φ' with selector variables. We have seen that one possible MAX-SAT solution is $\{w_3, w_7\}$. From this, and the properties of MAX-SAT, we can conclude that every MUS of φ contains w_3 or w_7 (or both), and consequently contains at least one clause. To improve this lower bound, we repeat the above process by finding another MAX-SAT solution that contains clauses other than w_3 and w_7. This can be achieved by adding the constraints (y_3) and (y_7) to the MAX-SAT optimization problem to get: Maximize $\sum_{i=1}^{11} y_i$ subject to $(\varphi')(y_3)(y_7)$. $\{w_4, w_6\}$ is a possible solution. Thus, every MUS must contain one of these clauses. In the third iteration, we have the following optimization problem: Maximize $\sum_{i=1}^{11} y_i$ subject to $(\varphi')(y_3)(y_7)(y_4)(y_6)$ which has the solution $\{w_1, w_5, w_8\}$.

After adding the constraints (y_1) , (y_5) and (y_8) , the optimization problem becomes UNSAT because of the MUS $\{w_3, w_4, w_5, w_6, w_7\}$. The aggregated set of clauses from MAX-SAT solutions is $\varphi_{maxsat} = \{w_1, w_3, w_4, w_5, w_6, w_7, w_8\}$. At this point, we can conclude that any MUS contains at least 3 clauses since:

$$\begin{aligned}\forall \alpha \subseteq \varphi \quad \alpha \text{ is MUS} \rightarrow \{w_3, w_4, w_1\} \subseteq \alpha \vee \{w_3, w_4, w_5\} \subseteq \alpha \vee \\ \{w_3, w_4, w_8\} \subseteq \alpha \vee \ldots \vee \{w_7, w_6, w_1\} \subseteq \alpha \vee \ldots \vee \\ \{w_7, w_6, w_8\} \subseteq \alpha\end{aligned} \tag{2}$$

Thus, the number of iterations of MAX-SAT is a lower bound (LB) on the number of clauses of the SMUS of φ .

To obtain an upper bound on the size of the SMUS, we can generate all the MUSes of the subformula φ_{maxsat} . The upper bound on the size of the SMUS is the size of the smallest MUS found in φ_{maxsat} . For our example φ_{maxsat} has a single MUS of size 5, and consequently the upper bound is 5.

3.2 Branch-and-Bound

Given $Muses(\varphi_{maxsat})$ and the initial LB and UB, if LB is equal to UB, then an MUS whose size is UB is an SMUS for φ . If this is not the case, we search the remaining MUSes of φ (the ones not in φ_{maxsat}) for an MUS (if any) whose size is smaller than UB. We achieve this by recursively branching on specific subformulas of φ , and bounding the search using LB and UB. The subformulas we branch on are $\varphi - \delta_i$ where $\delta_i \in \Delta$ and Δ is the set of all MAX-SAT solutions of φ_{maxsat} . Each of the subformulas in this recursion returns its SMUS if it is smaller than the one currently found in φ (and consequently φ 's UB is updated). Otherwise, an empty set is returned. For the running example, all MAX-SAT solutions of φ_{maxsat} are: $\{w_3\}$, $\{w_4\}$, $\{w_5\}$, $\{w_6\}$ and $\{w_7\}$. Thus, five recursive calls are made on the subformulas $\varphi - \{w_3\}$, $\varphi - \{w_4\}$, $\varphi - \{w_5\}$, $\varphi - \{w_6\}$, and $\varphi - \{w_7\}$.

To understand why this approach efficiently searches the space of $Muses(\varphi) - Muses(\varphi_{maxsat})$, we have to address the following questions. Why will branching on all the specified subformulas "implicitly" search every MUS in $Muses(\varphi) - Muses(\varphi_{maxsat})$? How does a child subformula use its parent's lower bound and MAX-SAT solution to compute its own lower bound? Finally, why is this an efficient solution? The first question addresses completeness, and its answer is omitted due to space limitations. We address the last two questions in what follows.

We use parent lower bound (pLB) and parent upper bound (pUB) to designate the upper and lower bounds of the parent formula, and current upper bound (cUB) and current lower bound (cLB) to designate the upper and lower bounds of a subformula $\varphi - \delta_i$. To understand how to compute $\varphi - \delta_i$'s cLB, let us consider the subformula $\varphi - \{w_3\}$. It is easy to verify that $Muses(\varphi - \{w_3\})$ includes all MUSes of φ except the ones that contain w_3. Since $Muses(\varphi - \{w_3\}) \subseteq Muses(\varphi)$, then MUSes of $\varphi - \{w_3\}$ satisfy (2), and pLB (that of φ) holds for $\varphi - \{w_3\}$. In fact, since all the MUSes of $\varphi - \{w_3\}$ do not contain w_3, they have to satisfy a stricter version of (2):

Algorithm 1 FindSMUS

```
FindSMUS(φ)
    smus = FindSMUSRec(φ, ϕ , 0, φ·size());
    if(smus == ϕ) print "NO MUS. Formula is Satisfiable";
    else print smus;
FindSMUSRec(set φ, set pMaxSat, int pLB, int pUB)
    if(IsSat(φ)) return ϕ ;
    (comp,numIter,cMaxSat)=IterateMaxSat(φ,pMaxSat,pUB-pLB);
    if(!comp) return ϕ;
    cLB = pLB + numIter;
      φ_maxsat = cMaxSat + pMaxSat    ;
    (muses, allMaxSats) = FindAllMuses(φ_maxsat);
    cMUS = Smallest(muses);
    cUB = cMUS.size();
    smallestTillNow = (cUB < pUB)? cUB: pUB;
    set smallestMUS = (cUB < pUB)? cMUS: ϕ;
    if(smallestTillNow<=cLB+1) return smallestMUS;
    foreach (ms in allMaxSats)
        φ_new = φ − ms ;
        maxsat_rec = φ_maxsat − ms ;
        recMUS=FindSMUSRec(φ_new, maxsat_rec, cLB, smallestTillNow);
        if(recMUS!= ϕ)
            smallestTillNow = recMus.size();
            smallestMUS = recMUS;
            if(smallestTillNow == cLB + 1) return smallestMUS;
    return smallestMUS;
```

Fig. 2. The algorithm for finding an SMUS

$$\forall \alpha \subseteq \varphi - \{w_3\} \quad \alpha \text{ is MUS} \rightarrow \{w_7, w_4, w_1\} \subseteq \alpha \vee \{w_7, w_4, w_5\} \subseteq \alpha \vee \ldots \vee \{w_6, w_7, w_8\} \subseteq \alpha \tag{3}$$

We can continue the iterations of MAX-SAT on the formula $\varphi - \{w_3\}$ starting with the set $\varphi_{maxsat} - \{w_3\}$. In other words, the initial optimization problem for

$\varphi_{maxsat} - \{w_3\}$ is: Maximize $y_1 + y_2 + y_4 \ldots + y_{11}$ subject to $(\phi - \{(\neg y_3 \vee \neg x_2 \vee \neg x_3)\})(y_7)(y_4)(y_6)(y_1)(y_5)(y_8)$. A possible solution for this problem is $\{w_9\}$. By combining this with (3), we know that the current lower bound for $\varphi - \{w_3\}$ is four. The next solutions are $\{w_{10}\}$ and $\{w_{11}\}$ (at this point, the optimization problem is unsatisfiable). Following the above reasoning, we have cLB = pLB + numIter where numIter is the number of MAX-SAT iterations in the subformula. Thus, cLB for $\varphi - \{w_3\}$ is 6, and consequently, the smallest MUS in $\varphi - \{w_3\}$ has at least 6 clauses. In fact, $\varphi - \{w_3\}$ has a single MUS of size 6. As a result, $\varphi - \{w_3\}$ does not contain an MUS smaller than the best we have till now (5 clauses). The above conclusion can be reached with fewer computations by noting that numIter must be at most pUB - pLB. If the optimization problem remains satisfiable after pUB-pLB iterations, the current subformula does not contain an MUS smaller than pUB and the search is bound.

Let us consider the next recursive call on the subformula $\varphi - \{w_6\}$ and the MAX-SAT solution $\varphi_{maxsat} - \{w_6\}$. We know that pLB and pUB are 3 and 5 respectively. The optimization problem is: Maximize $y_1 + \ldots + y_5 + y_7 + \ldots + y_{11}$ subject to $(\phi - \{(\neg y_6 \vee \neg x_4 \vee \neg x_5)\})(y_3)(y_7)(y_4)(y_1)(y_5)(y_8)$. $\{w_2\}$ is a solution. At this point, the optimization problem is unsatisfiable; consequently cLB = 4. Next, we generate all MUSes of the current MAX-SAT solution: $\{w_1, w_3, w_4, w_5, w_7, w_2\}$. We get the MUS: $\{w_1, w_2, w_3, w_4\}$. Since the size of this MUS is equal to cLB, we have found the smallest MUS in $\varphi - \{w_6\}$. Since this MUS is smaller than cUB for φ, we update cUB to reflect the smallest MUS we have up to this point. The recursive calls on the remaining subformula can proceed with pLB = 3 and pUB = 4. No smaller MUS is found in these formulas. Thus the smallest MUS for φ is $\{w_1, w_2, w_3, w_4\}$.

An additional optimization can be applied to enhance the above algorithm. Consider φ again. From (2), we know that each MUS contains at least 3 clauses. If there is an MUS that contains exactly three clauses then it must be in φ_{maxsat}. Since we did not find an MUS of size 3 in φ_{maxsat} then we know that the SMUS of φ has size at least cLB + 1. Using this observation, and after returning from the branch of the subformula $\varphi - \{w_6\}$, and updating cUB of φ to 4, we conclude that we have found the SMUS since cUB = cLB + 1.

The pseudo code for algorithm that follows from the above description is illustrated in Figure 2. FindMusRec() is the recursive procedure for finding the SMUS. It takes as arguments the formula φ, the parent's MAX-SAT clauses pMaxSat, the parent lower bound pLB, and the parent upper bound pUB. If φ is satisfiable, it contains no MUS and the empty set is returned. Otherwise, the procedure calls IterateMaxSat() using the arguments φ, pMaxSat, and pUB-pLB. IterateMaxSat() returns three values. The Boolean variable comp is set to 0 if the optimization problem remains satisfiable after running pUB-pLB iterations, and is set to 1 otherwise. If comp is set to 1, numIter is the number of iterations, and is cMaxSat is the set current MAX-SAT clauses. If comp is 0 then the formula does not contain an MUS smaller than cUB and the empty set is returned. Otherwise, cLB is set to pLB + numIter, and all the MUSes and MAX-SAT solutions of φ_{maxsat} are computed. If the smallest of these MUSes is equal to cLB or cLB

+ 1, it is returned as the sMUS for φ . If this is not the case, we branch on all MAX-SAT solutions of φ_{maxsat} in a depth-first manner. After each branch terminates, we update the smallest MUS of φ and designate it as an SMUS if its size is equal to cLB + 1. After all recursive calls end, the SMUS is returned.

4 Experimental Results

To experimentally evaluate the effectiveness of our algorithm, we implemented it in C++ and used Satzoo [6] to solve MAX-SAT problems. For generating all MUSes we use the algorithms in [3]. All experiments were conducted on a 2 GHz Pentium 4 machine having 1 GB of RAM and running the Linux operating system.

Table 1 lists the results for representative aim benchmarks from the DIMACS set. The number of clauses of the SMUS range between 10% and 40% of the total number of clauses. The short run time is due to the fact that the total number of MUSes in these formulas is very small.

Table 2 presents the results for representative unsatisfiable formulas from the DaimlerChrysler Automotive Product Configuration Benchmarks [5]. To our knowledge, there exists no previous work that shows the sizes of the SMUSes for these benchmarks. The last column reports the number of MUSes obtained by running the algorithm in [3]. In some cases, the algorithm does not terminate and consequently either no

Table 1. Results on Representative Aim Benchmarks

Benchmark	Variables	Clauses	SMUS Size	Time (sec)
aim-50-1_6-no-2	50	80	32	0.08
aim-50-2_0-no-1	50	100	22	0.01
aim-100-1_6-no-1	100	160	47	0.14
aim-100-2_0-no-2	100	200	39	0.1
aim-200-1_6-no-1	200	320	55	0.36
aim-200-2_0-no-1	200	400	53	0.3

information or a lower bound on the number of MUSes is provided. The total number of MUSes for these formulas ranges from 1 to more than a million. The size of the SMUS ranges from 0.1% to 5.5% (the average size is 1%) of the size of its formula. This shows that the SMUSes for these formulas are very small. Even in the cases where the number of MUSes is extremely large, our algorithm was able to efficiently find the SMUS. This shows the effectiveness of the implicit search utilized by the branch-and-bound process. For C202_FS_SZ_84, C202_FW_SZ_103, C210_FW_SZ_90, and C210_FW_SZ_91 the number of MUSes in φ_{maxat} was very large. To limit the run time, a cut-off of 500 seconds was used when generating all MUSes. The size column for these benchmarks has the format n1:n2 where n1 is the LB and n2 is the smallest MUS found before time-out. The difference between the best MUS found in the time limit and the lower bound for these formulas is 6, 8, 6, and 14 respectively. Thus, even

Table 2. Results on DaimlerChrysler Benchmarks

Benchmark	Variables	Clauses	SMUS Size	Time (sec)	# MUSes
C168_FW_SZ_107	1583	5939	47	546.54	NA
C168_FW_SZ_41	1583	4727	26	257.39	NA
C168_FW_SZ_66	1583	4751	16	18.84	NA
C168_FW_UT_2463	1804	6756	35	350.41	NA
C168_FW_UT_2469	1804	6767	32	831.46	NA
C168_FW_UT_714	1804	6754	9	14.49	NA
C168_FW_UT_851	1804	6758	8	59.91	102
C170_FR_RZ_32	1528	4067	227	121.33	32768
C170_FR_SZ_58	1528	4083	46	15.329	>16140
C170_FR_SZ_92	1528	4195	131	15.12	1
C170_FR_SZ_96	1528	4068	53	322.76	>172032
C202_FS_RZ_44	1556	5399	18	131.04	>79336
C202_FS_SZ_104	1556	5405	24	4.99	>1109330
C202_FS_SZ_121	1556	5387	22	2.5	4
C202_FS_SZ_122	1556	5385	33	3.84	1
C202_FS_SZ_74	1556	5561	150	36.43	NA
C202_FS_SZ_84	1556	5479	213:219	3878.3	NA
C202_FS_SZ_97	1556	5452	28	62.13	>63936
C202_FW_RZ_57	1561	7434	213	58.34	1
C202_FW_SZ_100	1561	7484	23	173.97	NA
C202_FW_SZ_103	1561	9024	147:155	8606.1	NA
C202_FW_SZ_123	1561	7437	36	14.74	4
C202_FW_SZ_61	1561	7490	18	163.84	NA
C202_FW_SZ_77	1561	7611	156	37.16	NA
C202_FW_SZ_98	1561	7438	7	58.16	NA
C208_FA_RZ_43	1516	4254	8	76.88	>9542
C208_FA_SZ_120	1516	4247	34	3.8	2
C208_FA_SZ_87	1516	4255	18	15.10	12884
C208_FA_UT_3254	1805	6153	40	95.27	17408
C208_FA_UT_3255	1805	6156	40	94.59	52736
C210_FS_RZ_23	1608	4911	31	266.30	NA
C210_FS_RZ_38	1607	4900	25	261.92	>188688
C210_FS_RZ_40	1607	4891	140	36.24	15
C210_FS_SZ_103	1607	4915	45	386.38	NA
C210_FS_SZ_107	1607	4902	15	25.29	NA
C210_FS_SZ_123	1607	5062	176	1401.97	>972463
C210_FS_SZ_78	1607	5071	170	56.72	NA
C210_FW_RZ_57	1628	6390	25	355.00	>129272
C210_FW_RZ_59	1628	6381	140	56.69	15
C210_FW_SZ_106	1628	6405	49	789.58	NA
C210_FW_SZ_111	1628	6393	15	35.17	NA
C210_FW_SZ_128	1628	6401	22	151.33	NA
C210_FW_SZ_90	1628	6977	271:277	6404.18	NA
C210_FW_SZ_91	1628	6709	267:281	6329.91	NA
C220_FV_RZ_12	1530	4017	11	50.58	>56872
C220_FV_RZ_13	1530	4014	10	33.35	6772
C220_FV_RZ_14	1530	4013	11	33.89	80
C220_FV_SZ_46	1530	4014	17	78.44	>5160
C220_FV_SZ_65	1530	4014	23	18.85	>84943

when the number of MUSes is very large, our algorithm provides useful information by generating an MUS whose size is close to the lower bound. We can see a large run time for these formulas. Most of this run time was spent computing φ_{maxsat}.

5 Conclusions

Understanding the causes of infeasibility of Boolean formulas is of interest in various theoretical and practical areas of computer science. Minimal unsatisfiable subformulas provide useful explanations of infeasibility. We have presented an algorithm to find an SMUS of a Boolean formula: an MUS with the least number of clauses. The algorithm utilizes the relation between MAX-SAT and MUSes to construct lower and upper bounds on the size of the SMUS. These bounds are the basis for a branch-and-bound procedure that finds the SMUS by recursively branching on specific subformulas. We have presented novel experimental results on two benchmark suites.

Acknowledgment

This work was funded in part by the National Science Foundation under ITR grant number No. 0205288.

References

[1] R. Bruni and A. Sassano, "Restoring Satisfiability or Maintaining Unsatisfiability by finding *small* Unsatisfiable Subformulae," in *Electronic Notes in Discrete Mathematics*, vol. 9, 2001.

[2] J. Huang, "MUP: A Minimal Unsatisfiability Prover," in Proceedings of Asia South Pacific Design Automation Conference, 2005.

[3] M. Liffiton, Z. Andraus, and K. Sakallah, "From MAX-SAT to Min-UNSAT: Insights and Applications," Technical Report CSE-TR-506-05, University of Michigan, 2005.

[4] I. Lynce and J. Marques-Silva, "On Computing Minimum Unsatisfiable Cores", in *Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT), 2004.

[5] SAT benchmarks from Automotive Product Configuration, http://www-sr.informatik.unituebingen.de/~sinz/DC/

[6] Satzoo, http://www.cs.chalmers.se/~een/Satzoo/

[7] L. Zhang and S. Malik, "Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula," in Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), S. Margherita Ligure - Portofino, Italy, 2003.