10th International Conference on Mass Customization and Personalization – Community of Europe (MCP - CE 2022)
Toward the Sustainable, User-Centric and Smart Industry 5.0
September 21-23, 2022, Novi Sad, Serbia

PREPROCESSING TECHNIQUES TO IMPROVE SATISFIABILITY-BASED PRODUCT CONFIGURATORS

Vinzent Brömauer, Tomáš Balyo, Noemi Christensen, Tobias Ostertag
CAS Software AG, Karlsruhe, Germany

Abstract: *One of the most successful and versatile approaches to designing a product configurator system is to utilize the power of Boolean Satisfiability (SAT) solving. Such a system is called a SAT-based Product Configurator. SAT solving is a well studied and very competitive research field. One of the most important techniques contributing to the success of state-of-the-art SAT solvers is preprocessing. However, to our best knowledge, these techniques are not yet being used in SAT-based product configurators. The goal of this paper is to find out which preprocessing algorithms can be successfully utilized in this specific application of SAT solving. Due to some theoretical properties of product configuration, many of the preprocessing algorithms used in standard SAT solving cannot be applied. We identified four techniques that can be used and evaluated them experimentally within the commercial product configurator Merlin using real industrial configuration benchmarks. We discovered, that the usage of particular preprocessing techniques can significantly reduce the size of the rule set, which leads to the reduction of the configurator's memory usage.*

Key Words: ***Preprocessing, Satisfiability, Product Configuration***

1. INTRODUCTION

Configuration software presents the user with the ability to define a product with features and their possible values. The user then defines constraints around these features, creating a set of rules out of which a finished product can be configured. During the configuration process the user assigns desired values to the properties of a product and the software is responsible for determining whether the given configuration satisfies the rules. The problem of evaluating the given set of rules can be reduced to solving the Boolean satisfiability problem [1]. One technique that has not yet been researched in SAT based product confgurators is preprocessing [2], which aims to simplify the set of rules defined by the user. This work details an effort to apply preprocessing techniques in the commercial SAT based product configurator Merlin by CAS Software. The main goal of this work is to implement a module that seamlessly improves the performance of the SAT solver used in Merlin by preprocessing the ruleset. The growth of customer workspaces also puts bigger strains on hardware. Increased waiting times during configuration may increase usability concerns. By preprocessing we aim to reduce the response time and memory usage of the configurator. In order to achieve this result, we pose the following central question: Which individual and combinations of preprocessing techniques for CNF-based formulas are applicable to an incremental SAT solver result in the greatest speedup of the solving time in a product configurator like Merlin.

2. PRELIMINARIES

In this Section we define the relevant terms and algorithms required in rest of the paper. In particular, we define the basic terminology for Boolean SAT solving and give an overview of relevant preprocessing algorithms.

2.1. Boolean SAT Solving

A Boolean variable is variable with two possible values: True and False. A literal is a Boolean variable (positive literal) or its negation (negative literal). A clause is a disjunction (or) of literals and a term is a conjunction (and) of literals. The size of a clause/term is the number of literals it contains.

A CNF (Conjunctive Normal Form) formula is conjunction of clauses and a DNF (Disjunctive Normal Form) formula is a disjunction of terms.

A truth assignment function assigns a value (True or False) to each Boolean variable in a given formula. An assignment satisfies a CNF (DNF) formula if it satisfies each of its clauses (at least one of its terms). An assignment satisfies a clause if it satisfies at least one of its literals and a term if it satisfies all of its literals. Finally, an assignment satisfies a positive (negative) literal if its corresponding variables has the value true (false) assigned.

If there is truth assignment that satisfies a given a formula we call this formula satisfiable. The problem of Satisfiability (SAT) is to determine whether a given formula is satisfiable and if yes, then finding a satisfying assignment. An algorithm or tool that can solve the SAT problem is called a SAT Solver.

In many applications of SAT a long sequence of very similar formulas is being solved. In such cases it is beneficial to use a so-called incremental SAT solver [3]. An incremental SAT solver provides an API which can

be used to use the SAT solver interactively [4]. This can bring significant performance benefits, on the other hand, it limits the number of usable preprocessing and inprocessing algorithms [5].

2.2. Preprocessing Algorithms

The general steps a SAT solving based application undertakes to come to a solution are clause building, preprocessing and clause evaluation. These steps are generally not rigid. Clause building and preprocessing might overlap whereas clauses are inserted, they are already modified and simplified. One such technique called forward subsumption is further examined below. The evaluation step itself might consist of some preprocessing techniques. The DPLL Algorithm is based on Pure Literal Elimination and Unit Propagation, which both modify and reduce clause sizes. Clauses themselves are meant to be adapted, changed, removed and added during every part of the solving process. Another common technique is to combine preprocessing steps with the solving process. This is called inprocessing and won't be examined in this work [6].

The idea of preprocessing is that by modifying and simplifying the formula we can aid the solver in finding solutions more quickly. Due to the complexity of the problem, simply shortening the ruleset by removing duplicate or tautological terms may already be beneficial to the result. It is important to understand that depending on the solving algorithm shorter but more complex rulesets are more difficult than longer but simpler ones. As such it is the point of some preprocessing techniques to introduce redundancies into the formula that shorten certain subtasks. Another intricacy is the order of applying certain techniques. Certain techniques may work better if applied after other techniques. Or it may be the case that one technique makes another obsolete. The conclusion is that applying preprocessing correctly is an important part of the implementation. As a whole preprocessing is a vital part of optimized solving.

There are numerous preprocessing algorithms in literature, however, only a few of them are easily applicable in incremental SAT solving. We identified four techniques for this work, which we will define next.

2.2.1 Subsumption

Subsumption represents a basic but powerful optimization technique which allows us to remove entire clauses from a formula. In a CNF formula F, a clause C is subsumed by a clause D if the set of literals in clause C is a (non-strict) superset of the literals in clause D. Subsumed clauses obviously do not contribute to the logical complexity of the formula. We can remove them without changing whether the formula is satisfiable or not.

2.2.2 DNF Subsumption

Subsumption can be also used on the terms of a DNF formula. A term C is subsumed by a term D if the set of literals in C is a (non-strict) superset of the literals in clause D. Like subsumed clauses, subsumed terms also do not contribute to the logical complexity of the formula and cane be removed. Additionally, we can apply subsumption to a pair of DNF formulas connected by conjunction, if the terms of one DNF are a subset of the terms in the other DNF.

2.2.3 Self-Subsuming Resolution

Like subsumption, self-subsuming resolution deals with pairs of clauses and the overlap of their literals. Unlike subsumption however we are not only concerned with removing an entire clause, but also with strengthening it if removal is not possible. Clause strengthening happens when we remove literals from a clause. Suppose we have a formula with clauses (C or l) and (D or -l) which represent two clauses with an added literal. If C subsumes D, we can simply strengthen (D or -l) to D.

We prove that strengthening the clause does not modify the satisfiability of the formula. We again think of the satisfiability of a formula F containing (C or l) and (D or -l). Anything that satisfies clause (C or l) with the exception of l also satisfies D due to their relation under subsumption. If we assert l=false the formula has not yet been satisfied as clause C remains. To satisfy it one of the literals of C has to be satisfied which in turn satisfies D. If on the other hand we assert l=true, we still need to satisfy one literal of D. In both cases the literal -l of the clause (D or -l) is irrelevant to the satisfiability of the formula and therefore can be removed.

2.2.4 Failed Literal Probing

One common technique used in both preprocessing and solving is Unit Propagation. A Unit Clause is a Clause that only contains a single literal. The single literal of a unit clause is called a unit literal. We collect all unit clauses of a given formula and assume that variables are set to satisfy these unit clauses. Following the assertion on the truth value of these variables we are able to simplify the formula as follows. We can remove all the clauses that contain any of the true literals and remove all false literals from each clause that contains any of them. Some clauses get shorter and may even become unit clauses. Therefore, we can repeat the process recursively until no new unit clause emerges. If for some clause all of its literals get removed, then we have determined that the formula is unsatisfiable.

Failed Literal Probing makes heavy use of literal propagation. A literal L is a failed literal with respect to a formula F if unit propagation derives an empty clause on (F and L). The logical consequence of the conflict is that F implies -L and we can therefore add (-L) as a unit clause. Furthermore, we can simplify the formula by removing all the clauses that contain -L and remove L from each clause that contains L.

3. RELATED WORK

In the world of practical SAT solving there exist many implementations to various algorithms. An annual competition called the SAT Competition [7] pits

submitted solvers against challenging and large problems. One of the tracks focusses on incremental solvers. Among the top-ranking solvers is CaDiCal [8]. CaDiCals described goal was to be an easy to understand and modifiable SAT solver that focusses on documentation.

One early and prominent adoption of SAT solving algorithms is MiniSat [9] which was featured in the SAT competition in 2005 [10] which among other things used preprocessing techniques to reduce memory usage [11].

To our best knowledge, there is no previous published work regarding using preprocessing in a SAT based product configurator.

4. IMPLEMENTATION DETAILS

In this Section we will discuss some implementation details regarding the preprocessing algorithms described in Subsection 2.2.

4.1. Subsumption

A trivial implementation of subsumption would compare every clause with every other clause resulting in a quadradic time complexity in the number of clauses. The comparison would determine whether clause A has all literals of clause B. Realistically, a large majority of clauses will only contain two or three literals. Further implementation details split up subsumption in forward and backward subsumption. Forward subsumption checks if a clause C is subsumed by a clause D in a formula whereas backwards subsumption checks if formula contains clauses C that are subsumed by a given clause D. We can apply forward subsumption when adding new clauses such as when generating our clause set. Backward subsumption already assumes a complete formula and simply iterates over every clause. For the purpose of streamlining the application of techniques we chose the latter.

Due to the potentially hundreds of thousands of clauses found in certain use cases of Merlin, a trivial implementation would not suffice. There are several ways to speed up subsumption. One common optimization is to only compare a clause to every clause that also contains its literals. To do that we keep a full occurrence list that maps each literal onto a set of clauses that contain that literal.

For backwards subsumption we now check the size of the occurrence list of each literal in a clause and pick the smallest one. We can do this because a clause that can be subsumed by our previous clause must contain all its literals.

The subsuming check itself can be optimized further as well. Trivially, if clause B is shorter than clause A, then clause A cannot subsume clause B. Additionally, in certain cases we are able to determine that subsumption is not possible by checking against a clause's signature [8]. We calculate the signature by applying a logical or to the hash of each literal. The signature is intended to function as a lossy form of a bitmask. The number of literals found in formulas is generally in the 10.000 to 40.000 range, but it is far less than the largest possible number represented through a 32-bit integer. To still make use of all 32 bits of the signature and improve the possible effect this method can have, we use a custom hash function on the integer representation of the literals. Using the signature we can express our exit condition as follows. We denote the signature of clause A with Sig(A)

- bitwise invert Sig(B) obtaining ~Sig(B)
- compute bitwise AND of Sig(A) and ~Sig(B)
- compare with zero

In other words, the signature of a clause C that is subsumed by another clause D has a 1 bit in at least at every position the signature of clause D has. So if the bitwise AND of the first with the negated second is zero, we know that the second signature fulfills our condition and the clause may potentially be subsumed. We cannot prove whether that is actually the case without doing a full check. But a failure to meet this condition asserts that subsumption is not possible, allowing us to skip the full check.

Algorithm 1. *Self-subsuming resolution*

Algorithm SELF-SUBSUME-(clause C, clause D): returns a literal or true or false. Literals are named $L_1, L_2, ..., L_n$

```
1:  maysubsume := true
2:  literal := 0
3:  for L ∈ C do
4:      if L ∈ D then
5:          continue
6:      else if ¬L ∈ D then
7:          if maysubsume = true then
8:              maysubsume ← false
9:              literal ← L
10:             continue
11:         else
12:             return false
13:         end if
14:     else
15:         return false
16:     end if
17: end for
18: if literal = 0 then
19:     return true
20: else
21:     return literal
22: end if
```

4.2. Self-subsuming Resolution

The check for self-subsumption can easily be integrated into our previous subsumption algorithm but we allow at most one negated literal of C in the potentially subsumed clause D. This complicates the subsumption check substantially.

This integration allows us to do both checks simultaneously while profiting from our previous improvements through the usage of an occurrence list. In the pseudocode detailing this combined approach in Algorithm 1 we can see, that if the initial check for the existence of the literal in the maybe subsumed clause fails we fall back onto a second check. We keep track of both the literal that may be removed due to self-subsumption and also make sure only one such case is allowed. This culminates in a rather complex procedure that nonetheless allows us to apply these two techniques simultaneously.

However, when we look at our previous optimizations for subsumption, multiple issues arise. First, the occurrence list now is not accurate anymore when we allow one literal to be negated. To fix this issue we can additionally keep track of all clauses in the occurrence lists of its negated literals. This diminishes the speedup gained through this technique but is necessary for the purpose of self-subsuming resolution.

A similar issue arises with our previously described method of calculating a signature for the purpose of exiting out of a subsumption check early. Here again we may choose to additionally hash the negated literal, though that also reduces the effectiveness of that method. Once these problems have been dealt with however, more issues arise. In the following we assume that a subsumption check actually occurs and is not skipped due to our previous optimizations.

One thing to consider is the order in which we compare clauses. A clause that was strengthened may subsume more clauses than in its unstrengthened state. During backward subsumption we made use of its transitive property to allow us to disregard the order of subsumption. Strengthening clauses however breaks that assumption.

Due to our optimization around occurrence lists the first subsumption check would have been skipped either way but we still need to disregard the previous result. This is a bigger problem that complicates our approach to this technique but there are two solutions. First, we can adapt backwards subsumption to instead work based on a queue. One after the other we insert clauses and perform a full self-subsuming resolution check against other clauses. Whenever a clause gets changed we append it to the queue. When the queue is empty we simply add the next not yet examined clause. This allows us to keep our backwards approach but may potentially introduce duplicate checks of the same clause. In our previous example we can perform the subsumption check on clause two with clause three (which fails), then strengthen due to self-subsuming resolution and lastly repeat the subsumption of the latter two clauses which now succeeds.

Second, we may instead adapt forward subsumption. We may add clauses one by one with the smallest clause first. Then using a One-Watched-Literal approach we store the literal with the least occurrences. When adding a new clause, we simply traverse all One-Watched-Literal lists and check for self-subsuming resolution. If the clause is subsumed, we do not add it at all. For the purpose of reusing our existing approach we integrated self-subsuming resolution into backward subsumption.

5. EXPERIMENTAL EVALUATION

During the development and evaluation of the work presented here multiple testing techniques were used. On a larger scale we used customer workspaces that were given to the development team of Merlin for the purpose of testing our efforts on real world use cases.

Several factors went into the specifics under which each statistic was created. For pure tests of effectiveness such as the number of clauses removed by subsumption (see Figure 1), the actual runtime of the optimization was not necessarily important. If the technique did not remove anything at all, no implementation quirk could change that. After examining backward subsumption we did not examine forward subsumption as the end result should be the same. After concluding that the technique seemed useful enough, as with subsumption, more effort was put into measuring speed. Here however another issue arises due to the nature of the programming

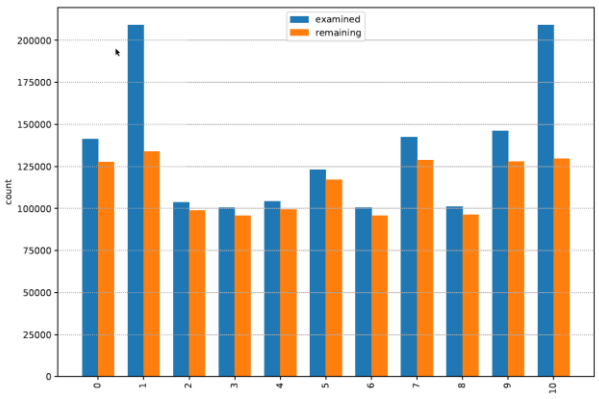

Fig. 1. *Effectiveness of subsumption on products from the largest available customer workspace*

language that Merlin was developed in. Java bytecode as generated by the javac compiler runs on a platform--independent virtual machine. To combat effects such as virtual machine warmup and other potential minimal lags due to first time initialization, all performance measuring tests were run three times back to back [12]. The statistics of the first two runs were discarded and only the third taken into consideration. Performance comparison tests were done back to back (i.e. test run with subsumption and without such as in Figure 1) in that same manner.

Another important aspect of this type of data analysis is the consideration whether the data gathered was trustable in the first place. The testing conditions were accounted for, the code itself was validated with handwritten regression tests and multiple runs sought to eliminate any coincidental slowdowns through vm warmup. But is the result plausible at all? Corresponding analysis was largely done through finding more metrics to keep track of, such as clause size and literal count. We can for example see that the tests on the large customer workspace we use for performance testing builds products that consist of a large majority of smaller clauses. We previously pointed out that subsumption works best when the subsuming clause has few literals. Therefore, a clause size distribution approaching an exponential decay could plausibly remove 35% of clauses. Even though some products consist of 200.000 clauses, they do not contain more literals than formulas half their size. In theory one should be able to observe a stark contrast between the percentage of clauses removed from the former. And looking at the data presented in Figure 1 we can see exactly that effect. Likewise, the other tests on a much smaller customer workspace revealed that even though these product formulas are quite small in comparison, they do in fact have many literals. Where earlier we had 40.000 literals for 200.000 clauses, here we have 11.000 literals for 16.000 clauses. In this case we observed a drastically lower percentage of clauses subsumed.

We can immediately see that on the second and last product that both contain over 200.000 clauses. Subsumption was able to remove 35\% and 38\% of clauses respectively, shrinking the formulas to around 130.000. The application of the technique to this product took around 5 seconds. Further, there seems to be a

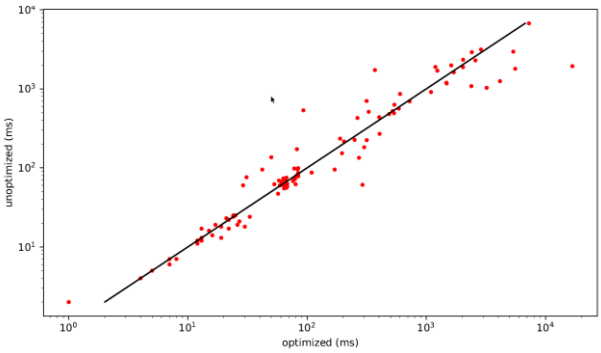

Fig. 2. *Scatterplot of optimized and unoptimized runtime tests on the largest customer workspace available*

correlation between the size of the formula and both the runtime and number of clauses that subsumption is able to remove. This is plausible considering the quadratic nature of looking at pairs in a set.

Next, we will more closely examine the runtime of the solver. We first examine what impact the techniques have on the formula and follow with a benchmark of a set of tests. To measure a possible impact on the runtime we perform the previous tests again and closely monitor the solver. Again, we use the largest workspace available and perform the subsumption technique. Figure 2 depicts a scatterplot. Each data point gathered from the tests (of which there are 109) is marked as a red dot. The x-coordinate of each dot consists of the runtime of the SAT solver with subsumption and the y-coordinate the runtime without any preprocessing. The black diagonal line represents a reference where x = y. If the red dot is above the line the subsumed runtime of that test took longer and vice versa. Additionally, we make use of a logarithmic scale to magnify outliers and get a better distribution across the graphic.

We can observe a clustering of results at the diagonal indicating that no significant speedup or slowdown occurred. As the runtime of the tests get higher though we see more fluctuation culminating in a drastic slowdown towards the higher end of runtime. Looking closer at the numbers, the total runtime of all tests with optimization is 77.599 seconds and 53.234 seconds without. We take the sum of the 95% quantile of both data sets to get an idea of the general speedup without outliers. Using the subsumption optimization, the 109 tests have a total runtime, filtered to the 95% of lower values, of 35.304 seconds. Without any optimization, the 95% quantile of the runtimes of the tests summed up is 32.887 seconds. This data suggests that we do in fact have a small speedup over the course of all tests, though we did not consider the time taken for preprocessing. Even though subsumption was able to remove a large number of clauses, the actual speedup seems to be minimal though existing.

To further examine the impact that subsumption has on the properties of the solver we can plot the statistics taken from both runs and compare them. Figure 3 shows that as a whole all four stats generally went up. The solver had to consistently restart more, even breaking a third restart. Interrupts, solving time and branches also generally went up although there are isolated cases where it slightly decreased.

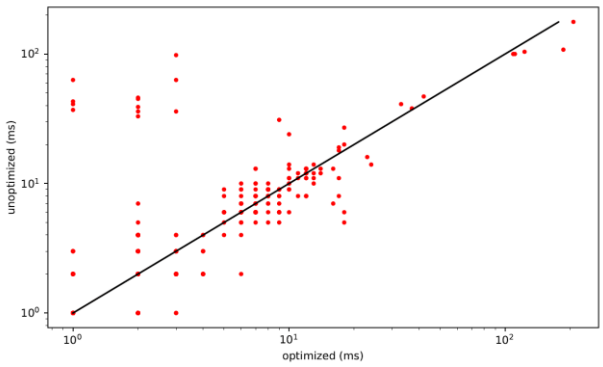

Fig. 4. *Scatterplot of optimized and unoptimized runtime tests on a smaller customer workspace.*

For further data we applied the same methodology to a smaller customer workspace (see Figure 4). There were multiple noteworthy observations made. First of we have a large volatility in data. While the average seems to match with the neutral diagonal, due to the very short runtime of the tests we observe a far heavier spread. This can be explained in part due to the small runtime of these tests no matter the optimization. As we can see on the x and y axis, the tests regularly finish within a few milliseconds. Therefore, it is possible that the data contains a large amount of coincidental slowdown on both tests that obfuscate our measurements.

The sum of the solving time of all tests without subsumption is 3.296 seconds and 2.812 seconds with subsumption. The same 95\% quantile of the runtime reveals that with optimization we have a total runtime of 1.497 seconds whereas without optimization we get 1.846 seconds. As opposed to the large workspace here we see several tests that feature substantial speedups using our optimization. While the runtime of these tests is still very small and quite scattered, it does seem like there is a potential for speedup. Though there is also a range of tests that are slower with optimization.

There are several possible explanations for these results. First, subsumed clauses are able to be deleted because they do not contribute to the complexity of the formula, but provide redundancies that may allow a solver to find contradictions sooner, therefore offsetting the additional clauses that need to be evaluated. Further, as we will discuss later, a small number of DNF clauses may contribute to a large part of the complexity of a formula than a large number of Or clauses. We assume that the main problem of this technique is that it is not able to reduce the number of literals found in the clauses. Despite reducing the clause size of two products in the large workspace from 200.000 to 130.000, the number of literals stays constant at 40.000. However, because Merlin contains an incremental solver, removing literals in of itself is not allowed. We can at best propagate them and add them as unit clauses.

For now, though we suppose that subsumption is a marginally effective preprocessing mechanism for our use case under the condition that the number of clauses is above a certain threshold and the ratio between the number of clauses and the number of literals is anywhere above 2:1. Additionally, further implementation improvements such as literal sorting may be necessary to further speed up of the subsumption check.

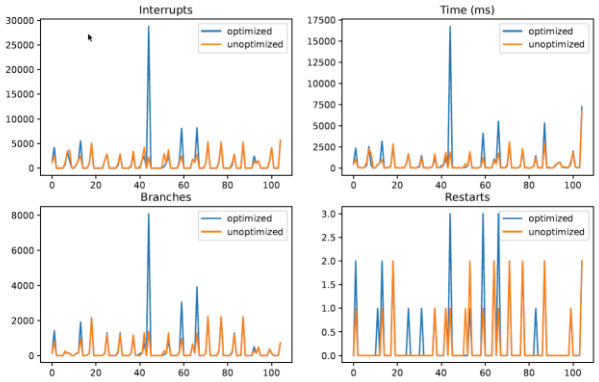

Fig. 3. *Solver statistics comparing subsuming (blue) and default (orange) benchmarks*

5.1. Self-Subsuming Resolution

In our experiments, the technique was not able to strengthen any clause and the number of subsumed clauses is equal to number of simply subsumed clauses. We may additionally examine the number of clauses we are able to strengthen in another workspace but the results seem to be the same. Applying the technique to the smaller workspace used in other benchmarks gave the same results which we do not elaborate on again.

We may consider the plausibility of this technique. In order for it to work at all we need two clauses that are nearly identical but one specific literal has to be negated. Even across products with 200.000 clauses this did not seem to occur even once. While we would be able to create a situation in which we may strengthen a clause, this in of itself seems to not occur coincidentally in the specific use cases that customers of Merlin work with.

5.2. Failed Literal Probing

We probed a small customer workspace for failed literals. We observed that although some products contain over 11.000 literals, none of them have been determined to be failed. We repeated the check on the biggest customer workspace we have available, however we get no positive results. The quite strict conditions of failed literals seem to not be fulfilled, even if the product consists of over 200.000 clauses.

Table 1. *Table displaying the number of DNF Formulas contained in a test, the number of combined DNF Terms and the number of DNF Terms after subsumption was applied*

DNFClauses	DNFTerms	DNFTerms after
2131	23064	20572
2205	45281	23056
1288	9900	8349
850	4299	3312
1293	9917	8366
1348	10018	8467
850	4299	3312
2120	22492	20276
1011	4781	3794
2775	71582	25843
2703	58110	24214

It seems much more likely that a failed literal is created as a result of clause strengthening itself. And while failed literal elimination in theory does just that, if nothing is found, there is no propagation and thus no clause strengthening. As such, failed literal probing on its own does not seem like a promising preprocessing mechanism in the context of Merlin. We can clearly see that failed literal probing is unable to modify the formula. Therefore, we conclude that failed literal probing is not useful for our use case.

5.3. DNF Subsumption

We now discuss the effectiveness of subsumption on DNF formulas. We first apply subsumption of DNF Terms to our smaller workspace again to check for the number of terms we are able to remove. Subsumption was not able to remove any term. The same result can be seen in any of the other test products, though that is easily explainable by the miniscule number of terms. Previous benchmarks showed us that subsumption on less than 16.000 clauses/terms only had a minor effect. And here we have the added problem that these terms are split up into eight individual groups.

When applying the technique to our large workspace however we see a massive reduction in the number of terms. Table 1 displays the number of clauses, terms and terms after applying subsumption. We can for example see that the second to last product contained over 70.000 terms of which more than 45.000 (64% of terms) were subsumed. For comparison, from a relative perspective this is more than the subsumption applied to Or Clauses was able to remove.

We also run the benchmark and measure the solving time required for each test in a large workspace and apply the subsumption technique to DNF Clauses. In Figure 5 we can see a clear slowdown of the solving speed. While faster tests retained their runtime, the slower ones became even slower. We can sum up the total runtime and see that without optimization we get a runtime of 53.234 seconds and with subsumption we get 121.075 seconds. As the small workspace did not find any subsumed terms a runtime benchmark was unnecessary.

5.4. Discussion of the Results

We sought to find techniques for optimizing formulas in incremental SAT. In our research we examined subsumption, self-subsuming resolution and failed literal probing and applied these to our use case. There were other techniques that were considered but ultimately disregarded. Pure literal elimination is a technique commonly used in regular SAT and is not directly applicable to incremental SAT. Further, blocked clause elimination was considered for a closer examination as there is existing research on its application in incremental SAT. However due to the requirements of inprocessing which we do not apply in this work, the technique was disregarded as well. We implemented the subsumption technique both for CNF and DNF type formulas. In both cases major reductions in formula size can be observed depending on the ratio of clauses to the

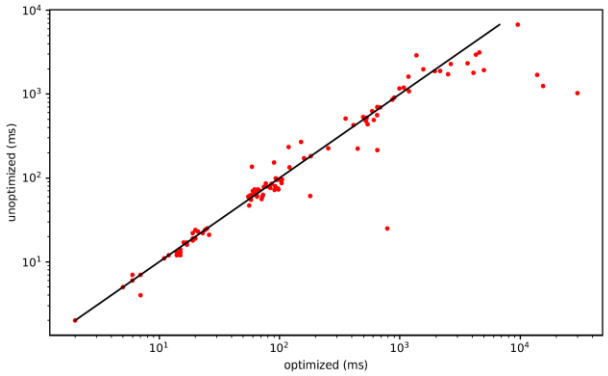

Fig. 5. *Scatterplot of optimized and unoptimized runtime tests on a small customer workspace using subsumption of DNF Clauses*

number of unique literals and the total size of the formula. Further measuring the runtime of the solving step, we did not observe a general pattern of speedup or slowdown though both occurred. In a workspace with products that generally have 200.000 clauses and 40.000 literals we observed a trend towards slowing down in cases where the runtime of the test was already large. However, in a smaller workspace we instead occasionally observed a speedup. The biggest discovery was the amount of terms that could be removed from DNF Clauses through the subsumption technique as it broke 65% in one case. Our benchmarks of self-subsuming resolution and failed literal elimination showed that neither are able to modify the formula at all.

One aspect of the subsumption technique that has not been detailed is the potential for its reduction in memory usage. Among the issues faced with running Merlin is the runtime overhead. We have seen that the number of clauses may exceed 200.000. To have as many clauses present in memory sets certain restrictions on the hardware that may exceed what is available. This underlying problem is a major concern for Merlin at this time. The computer that was used to implement preprocessing techniques itself had to have its RAM increased past 8GB to 32GB to meet the requirements for the server infrastructure. If the subsumption technique is applied, the potential for reducing this barrier of entry may be substantial. To potentially achieve the desired result, we propose the use of a forward subsumption technique as common implementations of backward subsumption may leave clauses in memory. This may be used for both DNF Clauses as well as Or Clauses.

6. CONCLUSION

In this paper we discussed various approaches to preprocess SAT formulas in a product configurator. The conclusion drawn from the results is that while the subsumption technique on CNF and DNF formulas did in fact show great effect in reducing the formula size, this partially resulted in a slowdown. In other cases, such as when applied to smaller CNF formula, subsumption showed occasional speedup. Lastly, we presume the potential for reducing the memory usage through the application of these techniques.

In the work we primarily looked at the end result of having a formula that has been preprocessed. Runtime of preprocessing itself has not been a major factor in our benchmarks and was never directly addressed. While optimizations for individual techniques were examined more closely, there is still major speedup potential. The idea of using a module which requires conversion of rules to and from also does not correspond to the nature of finding optimal and fast solutions. Instead, we focused our efforts on testing whether these techniques were useful at all.

One thing that could have been done better is a broader examination of how other SAT solvers tackle the issue of preprocessing in incremental SAT. For the most part only theoretical approaches were researched. It was also shown that the lack of inprocessing did not allow for certain techniques to be applied. There are more ways to implement classical SAT preprocessing techniques by reintroducing previously removed literals should they be used in a newly introduced clause. Any technique that makes use of this idea was not further examined. Further, an analysis regarding the complexity of a formula in context of Merlin's SAT solver has not been conducted. Techniques that potentially increase the clause size but reduce complexity were not examined as complexity was not a quantified metric in our research.

This work represents a first step to implementing preprocessing in a SAT based product configurator Merlin. Several techniques were examined but there were also restrictions set in place. There is potential for optimizing the memory usage of Merlin by removing clauses with forward subsumption before they get added. There may also be value in researching more techniques used in other incremental SAT solvers including techniques that make use of inprocessing.

REFERENCES

[1] Janota, Mikoláš. SAT solving in interactive configuration. Diss. University College Dublin, 2010.

[2] Biere, Armin, Marijn Heule, and Hans van Maaren, eds. Handbook of satisfiability. Vol. 185. IOS press, 2009.

[3] Hooker, John N. "Solving the incremental satisfiability problem." The Journal of Logic Programming 15.1-2, 1993.

[4] Balyo, Tomáš, et al. "SAT race 2015." Artificial Intelligence 241, 2016.

[5] Nadel, Alexander, Vadim Ryvchin, and Ofer Strichman. "Preprocessing in incremental SAT." International Conference on Theory and Applications of Satisfiability Testing. Springer, Berlin, Heidelberg, 2012.

[6] Manthey, Norbert, Tobias Philipp, and Christoph Wernhard. "Soundness of inprocessing in clause sharing SAT solvers." International Conference on Theory and Applications of Satisfiability Testing. Springer, Berlin, Heidelberg, 2013.

[7] Froleyks, Nils, et al. "Sat competition 2020." Artificial Intelligence 301, 2021.

[8] Fleury, Armin Biere Katalin Fazekas Mathias, and Maximilian Heisinger. "CaDiCaL, kissat, paracooba, plingeling and treengeling entering the SAT

competition 2020." SAT COMPETITION 2020, 2020.
[9] Sorensson, Niklas, and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization." SAT 2005.53, 2005.
[10] Järvisalo, Matti, et al. "The international SAT solver competitions." Ai Magazine 33.1, 2012
[11] Eén, Niklas, and Armin Biere. "Effective preprocessing in SAT through variable and clause elimination." International conference on theory and applications of satisfiability testing. Springer, Berlin, Heidelberg, 2005.
[12] Lion, David, et al. "Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems." 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016.

CORRESPONDENCE

Vinzent Brömauer, brvi1016@h-ka.de
Dr. Tomáš Balyo, tomas.balyo@cas.de
Noemi Christensen, noemi.christensen@cas.de
Tobias Ostertag, tobias.ostertag@cas.de