Disk Based Software Verification via Bounded Model Checking

Fernando Brizzolari, Igor Melatti and Enrico Tronci
Dep. of Computer Science
Univ. of Rome “La Sapienza”, 00198 Rome, Italy
{brizzolari,melatti,tronci}@di.uniroma1.it

Giuseppe Della Penna
Dep. of Computer Science
Univ. of L’Aquila, 67010 Coppito, Italy
dellapenna@di.univaq.it

Abstract

One of the most successful approach to automatic software verification is SAT based Bounded Model Checking (BMC). One of the main factors limiting the size of programs that can be automatically verified via BMC is the huge number of clauses that the backend SAT solver has to process. In fact, because of this, the SAT solver may easily run out of RAM.

We present two disk based algorithms that can considerably decrease the number of clauses that a BMC backend SAT solver has to process in RAM. Our experimental results show that using our disk based algorithms we can automatically verify programs that are out of reach for RAM based BMC.

1 Introduction

1.1 Motivations

Model checking [25] technology is enabling automatic verification of larger and larger programs. For example see [40] for Java programs verification via explicit model checking (e.g. [20]) and [12] for C programs verification via SAT based Bounded Model Checking (BMC, e.g. see [7, 6]).

The main obstruction to automatic Software Verification via Model Checking is the huge number of reachable states that even a moderate size program may have (state explosion) (e.g. see [2]). Indeed, state explosion may force us to give up a verification task because of lack of RAM.

Many approaches have been studied to counteract state explosion in software verification. For example, see [14, 38, 27] for approaches exploiting parallelism or randomization in an explicit model checking framework.

As for BMC based software verification, it has seen tremendous advances thanks to the use of state-of-the-art SAT solvers (e.g. see SATO [36, 42], zChaff [41, 29], MiniSat [16, 28]) as BMC backends. In fact, using BMC based model checkers it is now possible to automatically verify nontrivial C programs (e.g. see CBMC [12, 9]).

Moreover, also (discrete time) linear hybrid systems (e.g. see [3, 19]) defining specifications for embedded software can be effectively verified using SAT based BMC (e.g. see [24]).

Of course BMC has its own limitations too. In fact, the size of the Conjunctive Normal Form (CNF) generated by the BMC frontend can get huge as the system to be verified or the verification horizon grow. This, in turn, fills in the SAT solver RAM thus limiting the size of the systems that can be verified as well as the verification horizon.

CNF preprocessing [15] reduces the size of a CNF by applying suitable transformations to CNF clauses. The resulting CNF is smaller and often can be handled by a SAT solver whereas the original CNF could not.

To the best of our knowledge, the state-of-the-art CNF preprocessor is SatELite [15, 35]. SatELite performs all of its computations in RAM and reduces the CNF size by exploiting subsumption, self-subsuming resolution, and variable elimination by substitution. Preprocessing CNFs with SatELite allows handling of verification problems out of reach for state-of-the-art SAT solvers alone, like MiniSat.

Needless to say, SatELite does not solve all of our problems. In fact, SatELite performs all of its computations in RAM and therefore may itself run out of memory. This has motivated our search for disk based CNF preprocessing algorithms.

Our approach rests on the following observations. First, even relatively small unsatisfiable CNFs may be hard to solve for a SAT solver. Second, quite often large satisfiable CNFs are not hard if the SAT solver had enough RAM. Thus a CNF preprocessing that can reduce the size of a large satisfiable CNF can enlarge the class of problems tractable with BMC. This is exactly what our preprocessing algorithms aim to do: reduce the size of large (satisfiable) CNFs in order to make them tractable for a SAT solver. Of course our algorithms will work on any CNF, however we expect them to be effective mainly on satisfiable CNFs.

1.2 Our Contribution

We present two disk based CNF preprocessing algorithms. To the best of our knowledge there are no disk based CNF preprocessing algorithms previously published. Our algorithms are mainly useful on large CNFs that cannot be handled by SatELite because of lack of RAM.
1.2.1 DSATshrink

The first algorithm, *Disk SAT Shrink* (DSATshrink), takes as input a CNF $F$ and returns a smaller CNF $G$ which is satisfiable if $F$ is satisfiable. DSATshrink (Sects. 3, 4, 5 and 6) applies to $F$ the following transformations: *Boolean Constraint Propagation* (BCP), *Conse of Influence* (COI) and *Thinning* (i.e. elimination of equality constraints). The RAM versions of BCP, COI and Thinning are well known and widely used algorithms. For example, all (DPLL based) SAT solvers implement a RAM based BCP (e.g. as in SATO, zChaff, MiniSat) and many BMC frontends (e.g. VIS [39], NuSMV [31]) implement RAM based Thinning and COI (but not BCP). DSATshrink usefulness stems from the fact that it uses *disk based algorithms* to implement BCP, COI and Thinning as well. This allows DSATshrink to handle CNFs that cannot be handled by SatELite.

1.2.2 DSATsplit

The second algorithm, *Disk SAT Split* (DSATsplit), is a disk based implementation of the first iterations of the DPLL algorithm [13] used by most SAT solvers (e.g. as in SATO, zChaff, MiniSat). DSATsplit (Sect. 7) splits the input CNF $F$ into two subproblems as follows. First, DSATsplit selects a literal $l$ (using a strategy similar to VSIDS [29]) and then uses a disk based BCP to compute the CNF $F_l$ ($F_{\neg l}$), obtained by setting literal $l$ to true (false). If the size of $F_l$ can be handled by the SAT solver (MiniSat in our case) then DSATsplit calls the SAT solver to solve $F_l$. Analogously for $F_{\neg l}$. If $F_l$ and $F_{\neg l}$ are too big to be handled by the SAT solver, the above splitting process is repeated with another literal (as in DPLL) until a manageable CNF is obtained or we run out of time. Of course as soon as we find a satisfiable CNF during the splitting process we can stop the whole procedure and return a solution. Shortly, DSATsplit is an effective *disk based* implementation of a DPLL-like wrapping to MiniSat.

1.3 Related Works

Not surprisingly, CNF preprocessing has been extensively studied in an effort of finding effective tradeoffs between amount of reduction achieved and preprocessing time.

The works in [4, 8, 23, 30, 37] propose RAM-based techniques focusing on deriving units, implications and equivalent literals. Most of these techniques are embedded in MiniSat or in other SAT solvers outperformed by MiniSat [34]. Since we will be using MiniSat as our SAT solver, we will be using (obliviously) the above techniques when passing the preprocessed CNF to MiniSat.

The works in [22, 21, 17] propose RAM-based simplifications of digital circuits. Such simplifications are performed before the generation of the CNF, whereas our focus here is on CNF preprocessing.

Techniques involving CNF splitting have been proposed for parallel (e.g. see [18, 43] and Grid-based (e.g. see [11, 10]) SAT solvers. Note however that in such cases splitting aims at partitioning the original CNF among computational nodes in order to minimize communication between them. On the other hand, our focus here is on reducing the CNF size.

1.4 Experimental Results

SatELite is a state-of-the-art CNF preprocessor and MiniSat is a state-of-the-art SAT solver. Thus, to understand *if* and *when* it is advantageous to use our disk based algorithms, we can compare them with SatELite using MiniSat as a SAT solver to handle the processed CNFs for all preprocessing algorithms (i.e. SatELite, DSATshrink, DSATsplit). This is done in Sect. 8 with CNFs generated from CBMC and VIS. Our findings can be summarized as follows.

As for DSATshrink, our experimental results show that DSATshrink can make tractable by MiniSat CNFs that cannot even be handled by SatELite. For example, using DSATshrink and MiniSat with 1GB of RAM we can solve a SAT problem with 9 million variables and 33 million clauses. This problem cannot be solved using SatELite and MiniSat.

As for DSATsplit, our experimental results show that DSATsplit can solve problems that are out of reach for SatELite as well as for DSATshrink. For example, using DSATsplit and MiniSat with 1GB of RAM we can solve a SAT problem with 8.4 million variables and 29 million clauses. This problem cannot be solved by MiniSat neither with a SatELite preprocessing nor with a DSATshrink preprocessing.

Computation times, as to be expected, are our bottleneck here: when enough RAM is available SatELite+MiniSat (i.e. SatELite followed by MiniSat) is faster than DSATshrink+MiniSat and than DSATsplit+MiniSat. Moreover, DSATshrink+MiniSat is typically faster than DSATsplit+MiniSat when the CNF produced by DSATshrink can be handled by MiniSat. For large problems both SatELite and MiniSat run out of memory while DSATshrink+MiniSat or DSATsplit+MiniSat can handle such large problems within about 20 hours of computation on our PC.

2 Background

We denote with $\mathbb{B}$ the set of boolean values, that is, $\mathbb{B} = \{0, 1\}$. As usual, 0 stands for *false* and 1 for *true*. A literal is a boolean variable or the logical negation of a boolean variable. A clause $C$ is a disjunction ($\vee$) of literals. A unit clause is a clause with just one literal. A CNF is a conjunction ($\wedge$) of clauses. As usual, we also regard a CNF (clause) as a set of clauses (literals). We denote with $|F|$ the number of clauses in CNF $F$, and with $|C|$ the number of literals in clause $C$. Finally, given a literal $l$, we denote with $F_l$ the CNF obtained by assigning $1$ to $l$. Moreover, a literal $l$ is said to be *pure* iff $\neg\exists C \in F$ s.t. $\neg l \in C$ (i.e., $l$ always appear in positive or in negated form).

3 CNF Preprocessing on Disk

In this Section we give an overview of our DSATshrink algorithm. DSATshrink takes as input a CNF $F$ and returns a CNF $F'$ s.t. $F'$ is satisfiable iff $F'$ is satisfiable.
4 Thinning

The set of clauses \(\{(-l_1 \lor l_2), (l_1 \lor -l_2)\}\) is equivalent to the equality constraint \(l_1 = l_2\). Analogously, \(\{(-l_1 \lor -l_2), (l_1 \lor l_2)\}\) is equivalent to the inequality constraint \(l_1 \neq l_2\). Thus, it is possible to simplify a CNF \(F\) containing one of the above sets of clauses by replacing \(l_2\) with \(l_1\) (respectively \(-l_1\)) and deleting the two clauses. This is done by the Thinning algorithm, implemented in function DskThinning of Fig. 1. Note that in this way we reduce the number of clauses and variables in the given CNF. Function DskThinning works as follows.

First, in function DskFindEqIneqConstr in Fig. 1 we read the input CNF \(F\) and look for consecutive clauses defining an equality (inequality) constraint between, say, literals \(l_1, l_2\). Each time such clauses are found, DskFindEqIneqConstr updates array toBeSub by setting toBeSub[\(l_2\)] = \(l_1\) (toBeSub[\(l_1\)] = \(-l_1\)). Upon termination DskFindEqIneqConstr returns in array toBeSub the substitution to be carried out on \(F\). That is we have toBeSub[\(l_2\)] = \(l_1\) iff \(l_2\) may be replaced by \(l_1\) in \(F\). Note that we only look for consecutive clauses defining equality. Since bounded model checkers typically generate this kind of clauses consecutively DskFindEqIneqConstr usually detects most of the equality or inequality constraints contained in \(F\).

Equalities may chain. That is we may have toBeSub[\(l_3\)] = \(l_2\) and toBeSub[\(l_2\)] = \(l_1\). In such a case we should replace \(l_3\) with \(l_1\) rather than with \(l_2\). That is we should compute the transitive closure of the equalities in toBeSub. Moreover, to avoid loading the SAT solver with unused variable indexes, we should rename variables in order to avoid gaps of unused indexes. All these operations are carried out by function TransClosureSubs in Fig. 1 which updates toBeSub accordingly. Note that TransClosureSubs works on array toBeSub stored in RAM so it is quite fast. Moreover TransClosureSubs takes space \(O(V)\), where \(V\) is number of variables in the CNF. This fits in RAM without any problem. For example, even with a naive implementation of toBeSub as an array of (4 bytes) \(int\), with \(10^8\) variables in the CNF we would have need 4 \(\times 10^8\) bytes of RAM.

Function DskApplyChanges in Fig. 1 reads the clauses in \(F\), applies to each of them the substitutions in toBeSub and appends the clauses to a new temporary file \(G\). Finally, DskApplyChanges removes \(F\), sets \(\hat{F}\) file pointer to \(G\) file pointer and returns to NSubs the number of substitutions performed.

The above sequence of operations (DskFindEqIneqConstr, TransClosureSubs, DskApplyChanges) may generate new equality or inequality constraints. For this reason the above operations are in the body of a do-while loop which terminates when a fixpoint has been reached, that is when no more substitutions are possible (NSubs = 0).

The thinning process may create expanded unit clauses of the form \(\{(l_1 \lor l_2), (l_1 \lor -l_2)\}\) which indeed represent the unit clause \((l_1)\). Function DskFindMaskedUnit in Fig. 1 detects and simplifies all such expanded unit clauses. This entails a last scan to the disk file containing \(F\), and the creation of a new CNF file which is returned as the output of DskThinning.

5 COI

The COI [5] reduction algorithm has been originally designed for digital hardware verification. COI preprocessing removes from a digital circuit gates that do not contribute (directly or indirectly) to the circuit signals (variables) occurring in the property to be verified.

To use COI preprocessing in a CNF context we need to overlay a logic gate structure on the given CNF. Of course this, in general, is neither possible nor computationally feasible. However for CNF generated from BMC problems this is usually possible and computationally feasible. For example this is the case for the CNF generated by VIS and CBMC.

In Sections 5.1, 5.2 we present our algorithm to reconstruct the logic gate structure of a given CNF. Our algorithm is inspired to that in [33], however, unlike the one in [33] our algorithm is disk based. This allows us to handle

```plaintext
FN. DskThinning(CNF F) {
    do { toBeSub = DskFindEqIneqConstr(F);
           toBeSub = TransClosureSubs(toBeSub);
         NSubs = DskApplyChanges(F, toBeSub);
     } while (NSubs > 0);
    return DskFindMaskedUnit(F);
}

Figure 1. Function DskThinning

More in detail, DSATshrink consists of three disk based algorithms: Thinning, COI and BCP. Thinning removes from the given CNF clauses defining equality or inequality constraints between pair of variables by choosing for each such pairs only one witness variable to appear in the CNF. COI removes from the given CNF clauses all clauses not containing relevant variables w.r.t. the specification constraints. Finally, BCP propagates unit clauses and pure literals forced assignments.

Algorithms to implement the above algorithms are well known. For example BCP is used in many SAT solver (e.g. see SATO, zChaff, MiniSat) and COI algorithms are used in many model checkers (e.g. VIS, NuSMV). Finally, a transformation close to Thinning is described in SatELite.

However, the algorithms presented in the previous literature store all clauses in RAM. Thus, if the CNF is too large they run out of memory. This is unfortunate since we have noticed that quite often preprocessing with even just one of DSATshrink algorithms can drastically reduce the size of the problem at hand thus making it manageable for a SAT solver like MiniSat or zChaff.

The above observation led us to design disk based algorithms for Thinning, COI and BCP with the goal of getting a CNF suitable for, say, MiniSat. More specifically, we look for algorithms that never store all CNF clauses in RAM and only access (input as well as temporary) disk files in a sequential in order to keep computation times reasonable. Sects. 4, 5 and 6 describes our algorithms meeting the above requirements.

4 Thinning

The set of clauses \(\{(-l_1 \lor l_2), (l_1 \lor -l_2)\}\) is equivalent to the equality constraint \(l_1 = l_2\). Analogously, \(\{(-l_1 \lor -l_2), (l_1 \lor l_2)\}\) is equivalent to the inequality constraint \(l_1 \neq l_2\). Thus, it is possible to simplify a CNF \(F\) containing one of the above sets of clauses by replacing \(l_2\) with \(l_1\) (respectively \(-l_1\)) and deleting the two clauses. This is done by the Thinning algorithm, implemented in function DskThinning of Fig. 1. Note that in this way we reduce the number of clauses and variables in the given CNF. Function DskThinning works as follows.

First, in function DskFindEqIneqConstr in Fig. 1 we read the input CNF \(F\) and look for consecutive clauses defining an equality (inequality) constraint between, say, literals \(l_1, l_2\). Each time such clauses are found, DskFindEqIneqConstr updates array toBeSub by setting toBeSub[\(l_2\)] = \(l_1\) (toBeSub[\(l_1\)] = \(-l_1\)). Upon termination DskFindEqIneqConstr returns in array toBeSub the substitution to be carried out on \(F\). That is we have toBeSub[\(l_2\)] = \(l_1\) iff \(l_2\) may be replaced by \(l_1\) in \(F\). Note that we only look for consecutive clauses defining equality. Since bounded model checkers typically generate this kind of clauses consecutively DskFindEqIneqConstr usually detects most of the equality or inequality constraints contained in \(F\).

Equalities may chain. That is we may have toBeSub[\(l_3\)] = \(l_2\) and toBeSub[\(l_2\)] = \(l_1\). In such a case we should replace \(l_3\) with \(l_1\) rather than with \(l_2\). That is we should compute the transitive closure of the equalities in toBeSub. Moreover, to avoid loading the SAT solver with unused variable indexes, we should rename variables in order to avoid gaps of unused indexes. All these operations are carried out by function TransClosureSubs in Fig. 1 which updates toBeSub accordingly. Note that TransClosureSubs works on array toBeSub stored in RAM so it is quite fast. Moreover TransClosureSubs takes space \(O(V)\), where \(V\) is number of variables in the CNF. This fits in RAM without any problem. For example, even with a naive implementation of toBeSub as an array of (4 bytes) \(int\), with \(10^8\) variables in the CNF we would have need 4 \(\times 10^8\) bytes of RAM.

Function DskApplyChanges in Fig. 1 reads the clauses in \(F\), applies to each of them the substitutions in toBeSub and appends the clauses to a new temporary file \(G\). Finally, DskApplyChanges removes \(F\), sets \(\hat{F}\) file pointer to \(G\) file pointer and returns to NSubs the number of substitutions performed.

The above sequence of operations (DskFindEqIneqConstr, TransClosureSubs, DskApplyChanges) may generate new equality or inequality constraints. For this reason the above operations are in the body of a do-while loop which terminates when a fixpoint has been reached, that is when no more substitutions are possible (NSubs = 0).

The thinning process may create expanded unit clauses of the form \(\{(l_1 \lor l_2), (l_1 \lor -l_2)\}\) which indeed represent the unit clause \((l_1)\). Function DskFindMaskedUnit in Fig. 1 detects and simplifies all such expanded unit clauses. This entails a last scan to the disk file containing \(F\), and the creation of a new CNF file which is returned as the output of DskThinning.

5 COI

The COI [5] reduction algorithm has been originally designed for digital hardware verification. COI preprocessing removes from a digital circuit gates that do not contribute (directly or indirectly) to the circuit signals (variables) occurring in the property to be verified.

To use COI preprocessing in a CNF context we need to overlay a logic gate structure on the given CNF. Of course this, in general, is neither possible nor computationally feasible. However for CNF generated from BMC problems this is usually possible and computationally feasible. For example this is the case for the CNF generated by VIS and CBMC.

In Sections 5.1, 5.2 we present our algorithm to reconstruct the logic gate structure of a given CNF. Our algorithm is inspired to that in [33], however, unlike the one in [33] our algorithm is disk based. This allows us to handle
larger CNF than [33] can. On the other hand our gate recon-
struction algorithm may fail to recognize some of the gate
structures recognized by [33].

In Sect. 5.3 we present our disk based COI algorithm for
CNF preprocessing.

5.1 Logic Gates Identification

Table 1 gives the set of clauses used to represent gates

\[
\begin{align*}
&\text{OR} (A \lor B), \quad \text{AND} (A \land B), \quad \text{XOR} (A \oplus B) \quad \text{and} \quad \text{XNOR} (A \equiv B). \\
&\text{Note that in Table 1 } v_x \text{ represents the variable}
\end{align*}
\]

associated with the boolean expression \( x \). We say that a set
of clauses is a logic gate (or just a gate) if it has one of the
forms in Table 1.

We say that a set \( F_{LG} = \{F_1, \ldots, F_k\} \) is a logic gate
structure for the CNF \( F \) iff, for all \( 1 \leq i \leq k \), \( F_i \subseteq F \),
\( F_i \) is a logic gate and for all \( j \neq i \), \( F_i \cap F_j = \emptyset \). We also
define the constraint set of \( F \) as \( F_{CM} = F \setminus \{v_1, F_i\} \). All
clauses in \( F_{CM} \) are referred to as constraints.

<table>
<thead>
<tr>
<th>Form</th>
<th>Clauses</th>
<th>Form</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \lor B )</td>
<td>( \neg v_B \lor \neg v_A \lor v_B \lor v_A \lor \neg v_{(A \lor B)} )</td>
<td>( A \land B )</td>
<td>( v_B \lor v_A \lor \neg v_{(A \land B)} )</td>
</tr>
<tr>
<td>( A \land B )</td>
<td>( \neg v_B \lor v_A \lor \neg v_{(A \land B)} )</td>
<td>( A \lor B )</td>
<td>( v_B \lor v_A \lor \neg v_{(A \land B)} )</td>
</tr>
</tbody>
</table>

Table 1. CNF translation of some logic gates

The COI reduction algorithm cannot directly operate on
a CNF \( F \), but it needs to know the partition of \( F \) in gates and
constraints, i.e. it needs \( F_{LG} \) and \( F_{CM} \).

We devised a disk based algorithm, DskBuildGates, to compute \( F_{LG} \) and \( F_{CM} \) from a CNF \( F \). Our algorithm
is inspired by the technique described in [33], but unlike [33] we never store in RAM the full CNF. Shortly,
DskBuildGates is a predictive parser [1] which sequentially
reads the disk file containing \( F \). As soon as a set of
clauses \( G \) defining a gate is recognized, \( G \) is appended to
the file defining \( F_{LG} \). All clauses that are not recognized as
logic gates are considered constraints and thus appended to
the file defining \( F_{CM} \).

Note that DskBuildGates only recognizes gates whose clauses appear consecutively in the disk file containing \( F \). This is the typical situations for CNF generated from a BMC problem.

5.2 Logic Gate Input-Output Variables

Let \( g \in F_{LG} \) be a logic gate. We denote with \( \text{output\_var}(g) \) the output variable of \( g \), namely \( v_{\text{op}} \)
in Table 1, where \( \text{op} = \lor, \land, \oplus, \equiv \).

Of course in an actual CNF file variables are not labeled
with formulas. Thus finding the output variable of a (set of clauses defining a) gate may not be obvious. For the case of the \( \land \) and \( \lor \) operators form Table 1 we immediately see that the output variable is the only one that

\[
\begin{align*}
&\text{Figure 2. Function DskCOI}
\end{align*}
\]

appears in all clauses. Thus for such gates output variables are easily defined.

For the case of the XOR (\( \oplus \)) and XNOR (\( \equiv \)) gates instead all variables appear in the same way. For such gates we simply assume that the output variable is the one with the largest index. Of course this hypothesis is false in general, however it does hold for many BMC frontends (e.g. VIS, CBMC). Note that indeed this hypothesis is also the main limitation of our algorithm with respect to the one in [33]. This approach, however, avoids us storing CNF clauses in RAM, our goal here.

The set \( \text{output\_vars}(g) \) of a gate \( g \in F_{LG} \) is the set of variables of \( g \) that are not output variables.

5.3 COI Reduction Algorithm

Once \( F_{LG} \) and \( F_{CM} \) have been computed, the COI re-
duction may take place.

Indeed, without constraints, any acyclic set of logic gates yields a satisfiable CNF, since any combinational circuit
will provide output values given input values. Therefore, the SAT solver actually needs to check only the satisfiability of the set of clauses that represent the constraints together with the logic gates whose output is (directly or indirectly) involved in such constraints. Such set of clauses is computed by function DskCOI of Fig. 2, to which, unless otherwise stated, the following discussion refers to.

First, DskCOI computes \( F_{LG} \) and \( F_{CM} \) from the input
CNF \( F \), by calling DskBuildGates described in Sect. 5.1.

Function Dsk2Gates1Output checks if in \( F_{LG} \) there are two gates with the same output. In such a case COI preprocessing cannot be done and DskCOI just returns \( F \). Of course for well formed BMC problem this situation should never arise. However our input is a CNF. Thus to avoid giving possibly wrong answers we should make sure that our input CNF satisfies our working hypotheses.

Function DskLoadGraphQueue sequentially reads files \( F_{LG} \) and \( F_{CM} \) and builds in RAM the following structures: an oriented graph \( G \) representing the logic gate structure and a queue \( Q \) representing constraints. Graph \( G \) vertices are the variables of \( F \). There is an edge \( (v_1, v_2) \) in \( G \) iff \( v_i \) (\( i = 1, 2 \)) is the output of a gate \( g_i \) and the output of \( g_2 \) is the input of \( g_2 \). Queue \( Q \) consists of all variables in \( F_{CM} \).

Function ContainsCycles checks if the graph \( G \) is acyclic. In such a case COI preprocessing cannot be done and DskCOI just returns \( F \). Since \( G \) is in RAM we can use well known techniques for cycle detection in ContainsCycles, for example see [32,
Before doing any further work DskBCP checks if $A$ contains contradicting assignments. In such a case, (a CNF representing) UNSAT is returned.

Function DskApplyAssign sequentially read clauses from $F$ and for each clause $C$ in $F$ does the following. First, a new empty file $G$ is created to hold the clauses produced by DskApplyAssign. If there exists a literal $l \in C$ s.t. $A[l] = 1$ then $C'$ is appended to $G$, where $C' = C \setminus \{l \in C \mid A[\neg l] = 1\}$. If $C'$ is empty (that is $\forall l \in C \ A[\neg l] = 1$) then the problem is UNSAT and DskApplyAssign returns $-1$. Finally, DskApplyAssign removes $F$, sets $F$ file pointer to $G$ file pointer and returns to NSubs the number of 1s in $A$, that is the number of units clauses and pure literals in $F$.

After DskApplyAssign, function DskBCP checks if NSubs is $-1$. In such a case (a CNF representing) UNSAT is returned.

The do-while loop is repeated until NSubs becomes 0, that is no units clauses or pure literals are present in $F$. When this happens the fixpoint has been reached.

Finally, function DskRescaleVars, by reusing indexes of unused variables in $F$, renames the variables in $F$ in order to remove index gaps (thus saving on SAT solver RAM). Bitvector $P$ is used here, since it presents the set of literals occurring in $F$. This step entails a last (sequential) scan of the disk file.

### 7 Splitting CNFs

In this Section we present our disk-based splitting algorithm DSAtsplit. DSAtsplit takes as input a CNF $F$ and returns SAT if $F$ is satisfiable, UNSAT otherwise.

Essentially DSAtsplit uses a DPLL [13] schema to split the given CNF $F$ into smaller and smaller CNFs until we obtain CNFs that are small enough to be handled by a RAM based SAT solver. At that point a SAT solver (MiniSat in our case) is called on the small enough CNFs.

As it is usually done for DPLL, we present DSAtsplit here in a recursive form although, for efficiency reasons, our implementation is indeed iterative.

DSAtsplit is implemented by function DskSplit of Fig. 4.

Function DskSplit behaves as follows. If $F$ does not have too many clauses (namely, $|F| \leq m$) we try to solve it using our backend SAT solver. If the SAT solver does not run out of memory (res $\not= \text{OutOfMem}$), DskSplit returns the SAT solver result to the callee. Note that if the answer is SAT, the result is propagated to the other recursive calls, thus SAT will be the final response of the algorithm. A global variable $m$ is used as an estimation of the number of clauses the backend SAT solver can handle. If the SAT solver runs out of memory (res $= \text{OutOfMem}$), then we decrease $m$ by a factor of $\gamma$ ($\gamma = 0.05$ in our experiments) and a splitting phase takes place.

As in DPLL, a literal $l$ is chosen (function DskPickALiteral), and two new CNFs, $F_l$ and $F_{\neg l}$ are generated. Differently from RAM based DPLL, however, $F_l$ and $F_{\neg l}$ are generated as disk files, so that they can be passed to the recursive calls of DskSplit.
Function DskAssignAndBCP takes a literal l and a CNF F, computes \( F_1 \) and passes it to the disk BCP procedure of Sect. 6. The resulting CNF is put back into \( F \) by using a temporary file. As a matter of fact the two steps above are done in one single pass on the \( F \) file by preloading into the BCP assignment bivector (A in Sect. 6) the literal \( l \).

Function DskPickALiteral chooses a literal \( l \) for splitting. To this end, DskPickALiteral makes a trade-off between the VSIDS heuristic (i.e., selecting the literal which occurs most in \( F \)) and a heuristic that tries to maximize the number of clauses that become unit clauses. Function DskPickALiteral accomplishes this by computing two arrays, \( P \) for VSIDS and \( N \) for the other heuristic. Namely, for all literals \( l \), \( P[|l|] = |\{C \in F \text{ s.t. } l \in C\}| \) and \( N[|l|] = |\{C \in F \text{ s.t. } |C| = 2 \wedge \notin C\}| \). Note that only a sequential read of the CNF file is needed for computing these two arrays. A trade-off between the two heuristics is then used to choose the literal. Namely, we choose the literal \( l \) s.t. \( \alpha P[|l|] + \beta N[|l|] \) is maximum, for suitable parameters \( \alpha \) and \( \beta \). The parameter \( \alpha \) is fixed. We found experimentally that 1.0 is a reasonable value. The parameter \( \beta \) is instead computed as \( \frac{|F'| - |F|}{|F'|} \), where: \( F' \) is the parent CNF (i.e., the one who resolved in \( F \) after splitting) and \( |F'| \) is the number of unit clauses in \( F' \). This allows us to take into account the effectiveness of our heuristics for generating unit clauses. The idea is that, if too few unit clauses have been generated, then \( \beta \) has to be increased.

8 Experimental Results

We implemented algorithms DSATshrink (Sections 3, 4, 5, 6) and DSATsplit (Sect. 7) in a tool named DiskSAT. In this Section, we report the experimental results obtained using DiskSAT. Our results show that using our approach we can complete BMC of systems out of reach for state-of-the-art tools.

In order to have a meaningful benchmark for our experiments, we consider two categories of models.

In the first category, we consider software verification problems, namely CNFs generated by CBMC. To this end, we collected from the web a number of C programs implementing standard computer science algorithms, added assert’s as required from CBMC, defined for each BMC problem the number of unwindings and, finally, generated CNFs using CBMC. We expect DSATshrink to be the most effective DiskSAT preprocessing on this kind of experiments.

In the second category, we consider hardware verification problems, namely CNFs generated by VIS. To this end, for a given BMC verification horizon, we consider some of the examples in the VIS 2.1 standard distribution. Note that this category will be useful only to experiment with DSATsplit, since the CNFs output by VIS are already pre-processed (in RAM) for thinning clauses and COI reduction.

In Tab. 2 we show the benchmarks we use in our experiments. Column ID denotes the identifier of the corresponding model, and will be used in the following Tables reporting the experimental results. The number included in the ID shows either the number of unwindings for CBMC or the verification horizon for VIS. Column Name gives a short description of the model itself. Column From shows if the corresponding CNF is generated by CBMC (version 1.7) or VIS (version 2.1). Finally, columns \#Vars and \#Cls give, respectively, the number of variables and clauses of the corresponding CNF.

Note that, as denoted by the suffix of the experiments ID, most of the CNFs we show are SAT, our target here. For the sake of completeness, we also have an UNSAT CNF, bsort21unsat, which is denoted with an asterisk.

Our experiments are organized as follows. To have a uniform comparison for all verification experiments we set a memory limit of 1GB of RAM (no limit is instead set for disk). For the splitting technique we also set a time limit (our real bottleneck here) of 20 hours. Given this, we run experiments both with state-of-the-art techniques and with our two approaches (DSATshrink and DSATsplit). We finally collect our results and compare them.

As state-of-the-art tools, we use: the MiniSatSAT solver and the preprocessor SatELite, which is known to improve MiniSat performances.

In all tables the results on CBMC-generated CNFs are obtained with a Dual-Core 32-bits 3GHz Pentium 4 with 1 GB of RAM whereas the results on VIS-generated CNFs are obtained with a 2 Quad-Core Xeon 3GHz Pentium 4 with 8GB of RAM. Moreover, all time results are in seconds, while memory occupation results are in MBs.

Columns in Tabs. 3 and 4 show the results with MiniSat
Table 3. Results for MiniSat

<table>
<thead>
<tr>
<th>ID</th>
<th>Time</th>
<th>Mem</th>
<th>(MS(SE))</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsort80sat</td>
<td>2.4e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>bsort81sat</td>
<td>&gt; 1.5e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>bsort92sat</td>
<td>&gt; 1.5e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>bsort21unsat</td>
<td>1.9e+03</td>
<td>1.9e+03</td>
<td>5.1e+03</td>
</tr>
<tr>
<td>merge31sat</td>
<td>&gt; 1.2e+02</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>merge32sat</td>
<td>&gt; 1.2e+02</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>heap10sat</td>
<td>&gt; 2.4e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>heap15sat</td>
<td>&gt; 2.4e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>heap25sat</td>
<td>&gt; 2.4e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>heap30sat</td>
<td>&gt; 2.4e+01</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>insum50sat</td>
<td>1.6e+04</td>
<td>1.6e+03</td>
<td>4.7e+02</td>
</tr>
<tr>
<td>elev50sat</td>
<td>&gt; 1.3e+03</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
<tr>
<td>palau90sat</td>
<td>&gt; 2.9e+03</td>
<td>2.9e+03</td>
<td>5.3e+02</td>
</tr>
<tr>
<td>maimu70sat</td>
<td>&gt; 1.8e+03</td>
<td>out-mem</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4. Results for SatELite and MiniSat after SatELite

and SatELite. Their columns have the following meaning. **ID** denotes the identifier of a model as in Tab. 2. **MS** shows the results obtained with MiniSat. **SE** shows the results obtained with SatELite. **MS(SE)** shows the results obtained with MiniSat operating on a CNF preprocessed by SatELite. Note that all the results are given in terms of both time and memory. “Out-mem” indicates that the corresponding experiment could not be completed within 1GB of RAM.

Finally, we run DiskSAT on the same CNFs used for MiniSat and SatELite. We recall that CNFs generated by VIS have been already preprocessed for thinning and unit clauses, thus we do not perform disk preprocessing on them.

The corresponding results are in Tabs. 5 and 6. Column **ID** is as in Tab. 2. Column **DP** shows the results obtained with DSATshrink. Column **MS(DP)** shows the results obtained with MiniSat operating on a CNF preprocessed with DSATshrink. Finally, column **Split** shows the results obtained with DSATsplit. Note that, as in Tabs. 3 and 4, all the results are given in terms of both time and memory.

As for the splitting algorithm, we also provide the memory limit (column **MemLim**) used to decide whether to solve or split a given CNF. In order to stress our splitting algorithm, **MemLim** is chosen in the following way: for models which can be solved by preprocessing, we set Mem- and for large CNFs SatELite (and MiniSat) run out of memory.

Finally, in Tab. 7, we report the total time needed by each experiment to complete. Boldface systems are those for which our approaches are the only capable of completing the verification.

As to be expected on unsatisfiable CNFs our algorithms are not that effective. In fact, for the same program (bubble sort, bsort in Tab. 7) we can handle quite large satisfiable verification instances (e.g. bsort80sat, bsort92unsat in Tab. 7) but perform poorly on a not-so-large unsatisfiable instance, bsort21unsat in Tab. 7.

Our results may be summarized as follows. If SatELite terminates it is faster than any of our algorithms. However, for large CNFs SatELite (and MiniSat) run out of memory. In such cases only our algorithms, DSATshrink or DSATsplit, can complete the verification task. Moreover, DSATshrink appears to be more useful for CNFs generated by software verification problems (CBMC), while DSATsplit appears to be more useful for CNFs generated by hardware verification problems (VIS).
9 Conclusions

We have presented two disk based CNF preprocessing algorithms: DSATshrink (Sections 3, 4, 5, 6) and DSATsplit (Sect. 7). Both of our algorithms aim at reducing the size of the given CNF formula thus making it manageable for a SAT solver. Our preprocessing algorithms target large CNF formulas generated from BMC problems. Our experimental results (Sect. 8) show indeed that DSATshrink is typically effective on large CNFs generated from software verification problems (CBMC) whereas DSATshrink is typically effective on large CNFs generated from hardware verification problems (VIS).

As a future work, we think that DSATsplit could be effectively implemented in a distributed way on a network of workstations. We feel that such an issue deserves further investigation.

References

[34] SatElite, 2007.
[36] G. Stålmarck a system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula (1989) swedish patent n. 467 067, 1989.