# Incremental Preprocessing Methods for use in BMC [*]

S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker

Albert-Ludwigs-Universität, Freiburg, Germany
{skupfers,lewis,schubert,becker}@informatik.uni-freiburg.de

**Abstract.** Traditional incremental SAT solvers have achieved great success in the domain of Bounded Model Checking (BMC). However, modern solvers depend on advanced preprocessing procedures to obtain high levels of performance. Unfortunately, many preprocessing techniques such as a variable and (blocked) clause elimination cannot be directly used in an incremental manner. This work focuses on extending these techniques and Craig interpolation so that they can be used effectively together in incremental SAT solving (in the context of BMC). The techniques introduced here doubled the performance of our BMC solver on both SAT and UNSAT problems. For UNSAT problems, preprocessing had the added advantage that Craig interpolation was able to find the fixed point sooner, reducing the number of incremental SAT iterations. Furthermore, our ideas seem to perform better as the benchmarks become larger, and/or deeper, which is exactly when they are needed. Lastly, our methods can be extended to other SAT based BMC tools to achieve similar speedups.

**Key words:** BMC, Preprocessing, SAT, Model Checking, Craig Interpolation

## 1 Introduction

Bounded Model Checking (BMC) has become an important technique used to find errors in sequential circuits [4]. BMC accomplishes this by iteratively unfolding a circuit $k$ times for $k = 0, 1, \ldots$, adding the negated property, and then finally converting the BMC instance into a SAT formula for a SAT solver. If the SAT solver finds the $k$-th problem instance satisfiable, a path of length $k$ violating the property has been found.

The use of incremental SAT solvers has been shown to be very effective on BMC type problems [2, 4, 11, 18]. Regrettably, one of the most powerful new techniques used in modern SAT solvers is currently not used in BMC tools as it does not inherently support incremental solving, mainly modern preprocessing [7]. Since the introduction of SatELite, and widespread use of MiniSAT, all state-of-the-art DPLL (i.e. search) based SAT solvers include powerful preprocessors. These preprocessors reduce the overall size of a formula, and provide a significant performance increase on almost all types of industrial benchmarks. Preprocessing accomplishes this by performing variable elimination, block clause removal, and others techniques. These techniques, however, introduce complications in incremental SAT solving where new variables and clauses can be added and/or removed from the problem after each iteration.

[*] This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

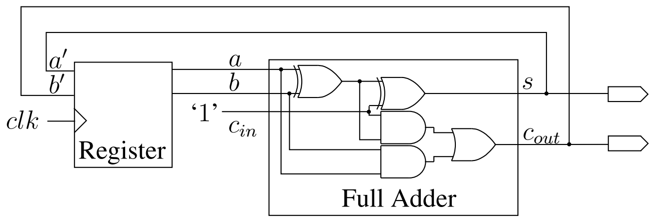

**Fig. 1.** Example BMC problem: counter circuit.

This problem manifests itself for instance if a latch variable from the circuit is eliminated during the preprocessors variable elimination routine. Now while this latch variable might not be needed in the current iteration of the BMC problem, it might be required in future iterations or unrollings.

Here, we present a technique that allows preprocessing to be used during incremental solving of BMC problems. We do this by introducing the idea of "Don't Touch" variables that restrict which variables and clauses the preprocessor is allowed to eliminate. We show that these variables can be easily calculated and are relatively few in number. Furthermore, we demonstrate that our new preprocessing technique results in a significant speedup on a wide breadth of incremental BMC benchmarks.

The rest of this paper is structured as follows. Section 2 will introduce the concepts and related work with regards to BMC using SAT, and preprocessing. Section 3 will introduce our tool, and cover how preprocessing can be used alongside incremental SAT solving. Our results showing the advantage of using preprocessing will be discussed in Section 4. Finally, Section 5 will conclude the paper.

## 2 Preliminaries

This section will give a a quick overview of Bounded Model Checking, and the use of incremental SAT solvers in this field. It will then discuss the preprocessing part of a SAT solver in more detail, and highlight the complications that arise when considering incremental SAT solving. Finally, it will discuss some related work.

### 2.1 Bounded Model Checking with Incremental SAT

BMC is the extension of Equivalence Checking (EC) over a finite time domain. Here, EC refers to functional equivalence, meaning that two circuits do not need to be exactly identical, but they must perform the same function (e.g. A carry select adder and ripple carry adder are functionally equivalent). In the EDA domain, most EC type problems either compare an unoptimized circuit to an optimized circuit to insure that they function the same, or compare the circuit to a specification. In BMC, the same can be done by iteratively unrolling a sequential circuit. Furthermore, it is common in BMC to prove safety or liveness properties (i.e. that a state is always or never reached).

Since BMC problems are normally formulated with respect to digital circuits, we need some way of representing each logic gate, and then the entire circuit, in conjunctive normal form (CNF) which is required by the SAT solver. This is done by doing a Tseitin transformation [21]. A Tseitin transformation takes a gate and transforms it into a conjunction of clauses. Each clause represents part of the relationship between the inputs of a gate, and what is expected at the output of the gate. Figure 1 shows the circuit diagram of a one bit Full Adder which is very common in most VLSI designs. By including $l$, $m$, and $n$ as intermediate variables (outputs of the first XOR and two AND gates), we can produce the following CNF representation of the circuit:

$$\begin{aligned} FA_{cnf} = \quad & (\neg l \vee a \vee b) \wedge (\neg l \vee \neg a \vee \neg b) \wedge (l \vee \neg a \vee b) \wedge (l \vee a \vee \neg b) \wedge \\ & (\neg s \vee l \vee c_{in}) \wedge (\neg s \vee \neg l \vee \neg c_{in}) \wedge (s \vee \neg l \vee c_{in}) \wedge (s \vee l \vee \neg c_{in}) \wedge \\ & (\neg m \vee c_{in}) \wedge (\neg m \vee l) \wedge (m \vee \neg c_{in} \vee \neg l) \wedge \\ & (\neg n \vee a) \wedge (\neg n \vee b) \wedge (n \vee \neg a \vee \neg b) \wedge \\ & (c_{out} \vee \neg m) \wedge (c_{out} \vee \neg n) \wedge (\neg c_{out} \vee m \vee n) \end{aligned}$$

In the definition of $FA_{cnf}$, each line represents one of the five gates in the circuit, and just as a gate consists of a conjunction of clauses, the CNF for the entire circuit consists of the conjunction of clauses that represent each gate.

To generate a BMC problem from Figure 1 which is a synchronous 2-bit counter, we can use $FA_{cnf}$ as the transfer function $T_{k-1,k}$ describing the circuits transition from one clock period to the next. This is done by adding the constraints that $a' = s$ and $b' = c_{out}$, which are the next state variables. If we assume the initial state of the registers is $(0, 0)$, represented by $\neg a_0$ and $\neg b_0$, and the property we want to verify is that the counter at time step $k$ does not equal 3 (encoded into CNF form as $(a_k) \wedge (b_k)$), we will produce the following equation:

$$\begin{aligned} I_0 &= (\neg a_0) \wedge (\neg b_0) \\ T_{k-1,k} &= FA^k_{cnf} \\ \neg P_k &= (a_k) \wedge (b_k) \end{aligned}$$

$$\begin{aligned} BMC_k &= I_0 \wedge T_{0,1} \wedge \ldots \wedge T_{k-1,k} \wedge \neg P_k \\ BMC_k &= (\neg a_0) \wedge (\neg b_0) \wedge FA^1_{cnf} \wedge \ldots \wedge FA^k_{cnf} \wedge (a_k) \wedge (b_k) \end{aligned}$$

In order to find a solution to $BMC_k$, we need $a_k$ and $b_k$ to be 1. If we evaluate $BMC_1$ and $BMC_2$, both will be unsatisfiable. However, if we check $BMC_3$, we will see that such a solution exists. In essence, BMC works by incrementally checking every value of $k$ until a solution is found, an upper bound on $k$ is reached, or a fixed point is detected. Several methods for finding fixed points exist. For instance, $k$-induction uses induction to prove that all reachable states will never violate a given property [17]. It does this by first proving that for every unrolling depth $\leq k$ the property can not be violated. Next the induction step will check whether its possible to show that all states reachable in $\leq k + 1$ steps starting from the property $P_0$ only lead to states that satisfies $P_{k+1}$. If so, the property is valid for every unroll depth of the circuit. For a more detailed account, confer [17].

Another approach is based on Craig interpolation [5] that can prove invariants [14]. Craig interpolants in BMC are used as an over-approximated forward image of reachable states in a transition system. If the computed over-approximated forward image reaches a fixed point, that is no new states are reachable, and the given invariant still holds, no counterexample is possible for any unrolling depth. The definition of a Craig interpolant is as follows:

**Theorem 1 (Craig).** *Given two propositional formulas $A$ and $B$ with the property that $A \wedge B$ is unsatisfiable, then there exists a Craig interpolant $C$ for $A$ and $B$. This Craig interpolant has the following properties:*

- *$C$ contains only variables which occur in $A$ and $B$ (*AB-common variables*).*
- *$\models A \Rightarrow C$ and $\models C \Rightarrow \neg B$*

To use Craig interpolation we define $I_k$ to be the initial state, $P_k$ the invariant to disprove, and $T_{i,i+1}$ the transition relation from a state at time step $i$ to a state at time step $i+1$. After showing that $I_0 \wedge \neg P_0$ is unsatisfiable (that is initially the property is not violated), the procedure first solves the BMC formula $\Phi = A \wedge B$, where $A := R_0 \wedge T_{0,1}$, $B := \neg P_1$ and initially $R_0 := I_0$. If $\Phi$ is unsatisfiable then a Craig interpolant $C_1$ for the formulas $A$ and $B$ is computed[1]. By $A \Rightarrow C_1$, the interpolant $C_1$ is an over-approximation of the states reachable in one step from $R_0$. If this over-approximation shifted to the zeroth instantiation of the variables (as described by $C_0$) is a subset of the so far reachable states, that is $C_0 \Rightarrow R_0$, then further transitions can only lead to states already characterized by $R_0$. As a consequence, the target states are unreachable and the verification procedure terminates. Otherwise, we expand the set of reachable states by adding reachable states given by the shifted interpolant, that is $R_0 := R_0 \vee C_0$. The procedure is iterated until the above termination criterion holds. The construction of a Craig interpolant can be done on-the-fly by the underlying SAT solver during conflict analysis. A more detailed description can be found in [14].

With respect to incremental SAT solving, this is implemented by first sending the problem $BMC_0$ to the SAT solver. This ensures that the initial state does not violate the property we are checking. Then, in incremental fashion, the clauses representing the property $P_0$ are removed from the SAT solver, and clauses for $T_{0,1}$ and $P_1$ are added to the solver. The reason why we do not generate separate problems for each value of $k$, is that modern solvers are intelligent and learn new information as they solve the problem. By incrementally adding clauses to the solver, and rerunning the same solver, we can retain all this learnt information. This is the main idea of an incremental SAT solver and one of the reasons it is so effective today. For a more detailed overview of a modern SAT solver please refer to [9].

## 2.2 Modern SAT Based Preprocessing

Preprocessing encompasses a wide range of techniques used to simplify the CNF formula before starting the incremental SAT solver. Earlier clause database minimization algorithms like NiVER [19] had been shown to significantly reduce the size of the problem, however they are too computationally time consuming for larger benchmarks. Recently, with the introduction of MiniSAT, a new efficient form of preprocessing for SAT was introduced [7]. MiniSAT took the ideas introduced by the QBF solver Quantor [1], called subsumption and variable elimination through resolution [16], and streamlined them for use in SAT.

MiniSAT's preprocessor consists of two main parts: subsumption, and variable elimination through resolution. Subsumption allows us to remove, or strengthen clauses that

[1] Note, that $C_1$ only contains *AB*-common variables.

look similar. For instance, if we have the two clauses $(x_1 \vee x_2)$ and $(x_1 \vee x_2 \vee x_3)$, we say that $(x_1 \vee x_2)$ subsumes $(x_1 \vee x_2 \vee x_3)$. We can therefore remove the larger clause from the problem. The second part of the algorithm, called variable elimination, removes variables through resolution. The basic idea is that if we want to remove the variable $x_1$, we resolve all the clauses containing $x_1$ with those containing $\neg x_1$. This produces many new clauses, but none contain the variable $x_1$. We have to be careful when choosing what variables to eliminate, however, as it can produce many additional clauses, and the size of the database can explode.

More recently, [12] introduced a way to further eliminate so called blocked clauses. Blocked clauses are defined as clauses that contain at least one blocked literal. A literal blocks a clause if every resolvent on that variable with other clauses containing its inverse, results in a tautology. In [12], they showed that this method works well in combination with variable elimination and subsumption.

### 2.3 Issues Associated with Combined BMC and Preprocessing Approaches

The preprocessor we use in in this paper uses the three techniques introduced in Section 2.2 (variable elimination, subsumption, and blocked clauses elimination) to minimize the size of the problem. However, when dealing with incremental SAT solving, these preprocessing techniques introduce many issues. Some of the issues associated with incremental SAT and preprocessing were discussed in [7] when MiniSAT introduced subsumption and variable elimination in their preprocecssor. In [7], they demonstrated that subsumption can be directly used in incremental SAT. However, they also showed that variable elimination introduces problems when variables are eliminated, but are then reintroduced later as new clauses for later unrollings are added.

For instance, latch variables can be eliminated from the transfer function $T_{0,1}$, when solving the problem $BMC_1$. However, this can cause problems when adding $T_{1,2}$ and solving $BMC_2$, as the variables that connect $T_{0,1}$ to $T_{1,2}$ are no longer properly represented. Similar issues appear with block clauses elimination, as clauses that were blocked, might become unblocked when new clauses are added.

In [7] they performed a small case study on one benchmark (vis.prodcell from [20]) with their incremental solver TIP [8] combined with their preprocessor SatELite. In their combination, SatELite was used to preprocess the entire formula for each incremental step. This method, however, is unfeasible on larger benchmarks that contain millions of clauses and variables as it takes too much time. Even on smaller benchmarks, the accumulated time spent preprocessing can be significant. In our method we achieve similar compaction, and only call the preprocessing routine once. This allows us to solve many large benchmarks that other solvers are unable to handle. Furthermore, we use Craig interpolation for proving safety properties (TIP uses k-induction). When using Craig interpolation, preprocessing cannot be run between stages unless the preprocessor itself can modify and generate Craig interpolants during preprocessing.

## 3 Incremental BMC with MiraXT and Craig

In this section we now present our ideas and implementation. Our tool starts by reading in a BMC problem specified in the AIGER format. It then uses MiraXT's [13] preprocessor to simplify the transition $T_{k-1,k}$ as much as possible to produce $T^{pre}_{k-1,k}$. Once preprocessing has finished, we then start our incremental BMC solver that replaces $T_{k-1,k}$ in the BMC equation with $T^{pre}_{k-1,k}$. In our implementation, assumptions are used to activate or deactivate clauses. To find a fixed point of reachable states we apply Craig interpolation presented in [14]. The Craig interpolants are cacluated with the help of the AIG package which uses MiniSAT as backend solver [15]. This combinations allows us to solve both satisfiable and unsatifiables problems. The novel aspects of our solver will now be discussed in the following subsections.

## 3.1 Using Preprocessing in an Incremental Solver

As mentioned in Section 2.3, it is possible to preprocess the entire formula for each time step $k$. This method not only consumes a lot of time as $k$ increases, but on larger benchmarks it can be completely infeasible. To deal with these shortcoming we preprocess the transition relation $T_{k-1,k}$ to produce $T^{pre}_{k-1,k}$ in such a way that insures it globally sound. This allows us to reuse $T^{pre}_{k-1,k}$ for every unrolling. This is done by preventing the elimination of certain variables and clauses from the transfer function.

To accomplish this in practice, when an input problem is read and then encoded into CNF form, we keep track of which CNF variables represent the repective latches and gates in the original circuit. Because of the way a BMC problem is constructed (as described in Section 2.1), we want to insure that the connection from $T_{k-1,k}$ to $T_{k,k+1}$ stays intact. To do this, we assign all latch variables as "Don't Touch" variables. A don't touch variable is a variable we are no longer allowed to eliminate. Furthermore, these variables cannot represent blocked literals. These restrictions prevent the preprocessing routine from destroying the global soundness of $T^{pre}_{k-1,k}$.

Note, that it can also happen that other variables need to be added to the "Don't Touch" lists. For instance, in some of the BMC benchmarks discussed in Section 4, the properties depend on variables that are not represented by latch variables. In these cases, the variables that the property is checking must also be marked to insure that the property can be correctly tested. However, as will be shown, it is normally the case that only a small portion of the variables (5% on average) are marked as "Don't Touch".

Our method results in transfer function with 55% fewer clauses, and a variable reduction of almost 70%. We have experimented further by allowing the preprocessing of multiple transistion functions that can be inserted in place of the multiple copies of the minimized single transition, however, the additional compression was minimal. With respect to the method briefly mentioned in [7], preprocessing a block of roughly 40 transistion relations ($T_{k-1,k}$) would be required to see a similar further reduction in the number of clauses when compared to our method. Additionally, because preprocessing was applied after every round in [7], it accounted for a minimum of 30% of the total time. Using our method, the preprocessing time is significantly less (normally $< 1\%$).

Consequently, because $T^{pre}_{k-1,k}$ is constant, it becomes easier to apply many other BMC techniques. Remember, if the problem is being preprocessed continually after every time step, it becomes a complex problem determining which variables in each time step are equivalent. This is important not only for Craig interpolation which will be discussed next, but also for ideas such as Strichman learning [18]. Strichman learning is the idea of copying clauses learnt from $T_{k-1,k}$ with the solver and transposing them to $T_{k,k+1}$ in the next stage. Similar ideas can be implemented with variable activities and search heuristics inside the solver. We plan to explore these ideas in the future.

### 3.2 Preprocessing and Craig Interpolation

In this section we focus on the problems that arise when we compute Craig interpolants and apply preprocessing. The Craig interpolants our solver produces use the proof-based construction applying the rules McMillan presented in [14] during the conflict analysis procedure of our solver. In other words we require a resolution proof of the original (un-preprocessed) CNF to generate a correct Craig interpolant. It is easy to see that simplification via subsumption and block clauses elimination has no impact on the construction of a correct Craig interpolant as these rules lead to a equi-satisfiable CNF with the property that a resolution tree in this simplified CNF is also a correct and complete resolution tree for the original CNF. For variable elimination by means of resolution, this is not the case. Fortunately, variables that are not marked as don't touch variables have no influence on the construction of Craig interpolants. This is because a resolution proof of a simplified CNF by means of variable elimination can be modified by adding those resolution steps performed during variable elimination to achieve a correct resolution proof for the original (not simplified) problem. To do this, we must distinguish between the the set of clauses we want to over-approximate and those which we do not (i.e. we differentiate between the $T^{pre}_{k,k+1}$ in part $A$ and those in part $B$). Therefore consider:

$$\overbrace{\ldots \wedge T_{k,k+1}}^{A} \wedge \overbrace{T_{k+1,k+2} \wedge \ldots}^{B}$$

We are interested in Craig interpolant $C$ that is implied by $A$ and unsatisfiable in conjunction with $B$. Before we compute such a $C$ we start our preprocessing routine on $T_{k,k+1}$ with don't touch variables and achieve a simplified transition relation $T^{pre}_{k,k+1}$. The don't touch variables prevent the preprocessing from doing resolution steps on variables that are located in $A$ and in $B$ ($AB$-common). So only resolution steps on variables that can either be found in $A$ ($A$-local) or in $B$ ($B$-local) are applied. In both situations ($A$- and $B$-local) the partial Craig interpolants for clauses derived via variable elimination and then applying the corresponding construction rules are identical to those of the original simplified problem clauses. Since we are only applying simplification techniques that are globally sound, our incremental BMC solver still creates Craig interpolants that are correct when applied to the simplified formula. In other words, our preprocessing technique is independent of the construction of the Craig interpolants.

## 4 Experimental Results

To evaluate our methods, we used the entire 2008 Hardware Model Checking Competition benchmark set [10]. This includes the *public* and *Intel* Benchmarks. In total, this set contains 645 mixed (sat/unsat) instances that are solvable at different depths. The test machine used for all the results stated here had a Quadcore Intel Q9450 processor @ 2.66GHz. The machine contained 8GB of RAM and was running a 64 bit linux

| Benchmark | | | Basic Solver | | | Solver with Preprocessor | | | |
|---|---|---|---|---|---|---|---|---|---|
| Family | #Sat | #Unsat | #Vars. | #Cla. | Time | #Don't | #Vars. | #Cla. | Time |
| 139* (99) | 90 | 9 | 1,078,504 | 2,993,535 | 29.84 | 77,280 | 596,407 | 1,466,401 | 44.25 |
| ab* (5) | 5 | 0 | 5,227 | 13,215 | 39.23 | 800 | 3,184 | 5,385 | 16.56 |
| bc57 (7) | 7 | 0 | 16,048 | 40,724 | 2,892.79 | 2,568 | 9,888 | 17,051 | 463.45 |
| bj* (44) | 17 | 20 | 546,133 | 780,626 | 26.69 | 2,642 | 117,758 | 364,473 | 22.22 |
| br* (5) | 5 | 0 | 7,291 | 18,280 | 0.15 | 1,236 | 4,586 | 7,903 | 0.18 |
| cmu* (4) | 0 | 2 | 3,260 | 6,211 | 5.82 | 126 | 1,059 | 2,772 | 1.37 |
| count* (2) | 2 | 0 | 258 | 590 | 0.02 | 64 | 192 | 255 | 0.02 |
| cs* (4) | 4 | 0 | 23,946 | 65,698 | 2.32 | 2,168 | 12,422 | 29,286 | 1.46 |
| dm* (20) | 20 | 0 | 45,081 | 109,574 | 1.21 | 7,582 | 24,067 | 35,941 | 0.88 |
| eijk* (28) | 0 | 10 | 30,732 | 14,989 | 82.94 | 966 | 4,341 | 8,275 | 97.80 |
| intel*(60) | 5 | 16 | 2,418,808 | 1,682,281 | 3,131.30 | 73,869 | 422,899 | 789,911 | 2,063.18 |
| irst* (3) | 3 | 0 | 5,248 | 12,171 | 2,123.84 | 1,068 | 2,255 | 2,907 | 1,344.94 |
| ken* (16) | 2 | 14 | 39,551 | 113,003 | 0.68 | 1,634 | 10,597 | 24,200 | 0.76 |
| mutex* (2) | 2 | 0 | 418 | 1,026 | 0.01 | 80 | 276 | 462 | 0.02 |
| nec* (13) | 2 | 10 | 352,378 | 915,713 | 84.33 | 69,012 | 227,053 | 451,209 | 27.31 |
| nus* (32) | 8 | 18 | 70,248 | 155,705 | 241.26 | 6,028 | 30,954 | 64,018 | 46.15 |
| pc* (5) | 5 | 0 | 11,921 | 30,579 | 0.20 | 1,490 | 7,054 | 12,800 | 0.24 |
| pd* (235) | 28 | 157 | 647,556 | 1,326,910 | 449.23 | 31,357 | 171,826 | 478,380 | 174.62 |
| prod* (19) | 19 | 0 | 25,933 | 65,419 | 95.93 | 4,274 | 17,063 | 29,558 | 34.11 |
| ring* (2) | 2 | 0 | 418 | 962 | 0.05 | 100 | 296 | 380 | 0.03 |
| short* (2) | 2 | 0 | 222 | 492 | 0.01 | 56 | 164 | 200 | 0.01 |
| srg* (3) | 3 | 0 | 1,281 | 3,006 | 0.04 | 282 | 882 | 1,197 | 0.03 |
| texas* (19) | 9 | 10 | 107,207 | 308,411 | 27.52 | 6,190 | 35,563 | 98,435 | 9.51 |
| vis* (16) | 5 | 9 | 25,041 | 64,654 | 109.69 | 1,174 | 9,403 | 24,063 | 191.62 |
| Total (645) | 245 | 275 | **5,462,710** | **8,723,774** | **9,345.07** | **292,046** | **1,710,189** | **3,915,462** | **4,540.71** |

**Table 1.** Preprocessing and Solver Performance.

2.6.24 SMP enabled Linux kernel. Lastly, for all benchmarks a timeout (TO) value of 900 seconds was used for each instance. In our current setup, our tool runs in two seperate stages (one stage with Craig interpolation enabled, and one without). The first stage gets approximately 1/3 of the total time, with the second getting the remainder[2].

The first two tables (Tables 1 and 2) show how preprocessing reduces the size of the transfer function, and hence the entire problem, allowing the solver to achieve significant speedup. Table 1 gives an overview of the additional power preprocessing provides when related to each benchmark family. Table 2 takes a more in-depth look at specific benchmark instances. In the first few columns of Table 1 labelled *Benchmark*, the family name (and number of benchmarks in the family), followed by how many *Sat* or *Unsat* instances in the family were solved is shown. The next group of columns labelled *Basic Solver* is our solver without the use of a preprocessor, followed by *Solver with Preprocessor* which is with preprocessing enabled. For both solvers, the sum of all the variables (*#Vars.*) and clauses (*#Cla.*) for all the transfer functions in each benchmark family is shown. For *Solver with Preprocessor*, the number of don't touch variables (*#Don't*) is also given. Finally, the time required by each solver to solve all of the instances in the benchmark family is specified. Note, this table only includes benchmarks solved by both solvers (all other benchmark instances were removed). This allows for a more direct comparison showing the advantage of preprocessing.

[2] To ensure a fair comparison, MiraXT is only used in single thread mode.

| Benchmark | | | Transistion Relation | | | | | Solve Time | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Depth | S/U | #Don't | #V Orig. | #V Simp. | #C Orig. | #C Simp. | woPre | wPre | S$_p$ |
| 139442p0 | 6 | Unsat | 462 | 4,069 | 2,465 | 10,775 | 4,890 | 0.08 | 0.11 | 0.68 |
| 139442p0neg | 4 | Sat | 462 | 4,102 | 2,507 | 10,874 | 4,962 | 0.08 | 0.11 | 0.71 |
| 139464p0 | 6 | Unsat | 1,134 | 21,497 | 11,593 | 60,917 | 30,202 | 0.51 | 0.91 | 0.56 |
| 139464p0neg | 4 | Sat | 1,134 | 21,594 | 11,717 | 61,208 | 30,419 | 0.70 | 1.16 | 0.61 |
| bc57sensorsp2 | 104 | Sat | 380 | 2,369 | 1,465 | 6,007 | 2,524 | 325.57 | 93.86 | **3.47** |
| bc57sensorsp2neg | 104 | Sat | 380 | 2,369 | 1,467 | 6,007 | 2,524 | 745.72 | 45.17 | **16.51** |
| bc57sensorsp3 | 104 | Sat | 380 | 2,328 | 1,430 | 5,884 | 2,439 | 747.44 | 65.62 | **11.39** |
| bj08aut5 | 4 | Unsat | 6 | 387 | 216 | 1,146 | 561 | 0.01 | 0.01 | 0.67 |
| intel003 | **22**(32) | Unsat | 132 | 1,060 | 653 | 2,711 | 1,150 | 0.32 | 0.13 | 2.50 |
| intel006 | **38**(40) | Unsat | 530 | 3,611 | 2,441 | 8,907 | 4,060 | 9.19 | 6.32 | **1.45** |
| intel009 | 113 | Sat | 8,322 | 91,644 | 56,819 | 244,587 | 116,605 | TO | 573.78 | >**1.57** |
| intel020 | **68**(72) | Unsat | 586 | 6,181 | 4,041 | 16,445 | 7,982 | 20.17 | 6.62 | 3.05 |
| intel023 | **74**(106) | Unsat | 597 | 6,174 | 4,041 | 16,387 | 7,920 | 325.84 | 17.65 | **18.47** |
| intel024 | **86**(88) | Unsat | 595 | 6,157 | 4,037 | 16,343 | 7,907 | 131.68 | 33.01 | **3.99** |
| intel041 | 35 | Sat | 14,292 | 125,377 | 81,416 | 324,013 | 152,173 | 385.77 | 376.26 | **1.03** |
| intel042 | 47 | Sat | 13,888 | 122,375 | 79,405 | 316,488 | 148,448 | TO | 423.18 | >**2.13** |
| intel043 | 83 | Sat | 11,052 | 104,349 | 67,644 | 272,697 | 128,065 | TO | 624.94 | >**1.44** |
| intel048 | — | — | 26,689 | 261,275 | 167,397 | 685,929 | 337,346 | TO | TO | — |
| neclaftp1002 | 24 | Unsat | 15,760 | 79,164 | 50,826 | 205,876 | 100,521 | 2.98 | 3.26 | 0.91 |
| neclaftp3002 | 16 | Sat | 5,652 | 33,166 | 21,314 | 88,098 | 43,506 | 40.15 | 7.21 | **5.57** |
| nusmvtcasp5 | 24 | Sat | 344 | 3,143 | 1,492 | 8,304 | 3,045 | 20.15 | 4.39 | **4.59** |
| pdtpmsmiim | **152**(-) | Unsat | 399 | 1,259 | 1,110 | 2,844 | 1,590 | TO | 219.51 | >**4.10** |
| pdtviseisenberg1 | **50**(-) | Unsat | 58 | 2,682 | 906 | 7,923 | 3,479 | TO | 281.76 | >**3.19** |
| pdtviseisenberg2 | **54**(-) | Unsat | 58 | 2,682 | 906 | 7,923 | 3,479 | TO | 329.03 | >**2.74** |
| pdtvismiim1 | **6**(8) | Unsat | 160 | 1,088 | 472 | 2,833 | 866 | 0.03 | 0.02 | **1.75** |
| pdtvisns2p2 | **34**(-) | Unsat | 142 | 2,671 | 848 | 7,689 | 2,526 | TO | 215.16 | >**4.18** |
| pdtvissfeistel | 30 | Unsat | 722 | 10,041 | 3,074 | 28,475 | 9,329 | 46.69 | 4.51 | **10.36** |
| texasPImainp05 | **26**(40) | Unsat | 473 | 8,473 | 3,067 | 24,435 | 8,339 | 21.41 | 0.32 | **66.08** |
| texasPImainp08 | 10 | Sat | 473 | 8,473 | 3,067 | 24,435 | 8,339 | 0.76 | 0.22 | **3.38** |
| vis4arbitp1 | **36**(38) | Unsat | 46 | 371 | 182 | 985 | 331 | 67.19 | 90.13 | 0.75 |
| visbakery | 59 | Sat | 46 | 780 | 262 | 2,230 | 756 | TO | 634.02 | >**1.42** |

**Table 2.** Instance Specific Preprocessing Results.

The results in Table 1 first show that both the number of variables and clauses are reduced significantly using preprocessing. On average, variables were reduced by almost 70% when compared to the basic Tseitin transformation. Additionally, the number of clauses was reduced by over 55%. This was accomplished as the number of don't touch variables was on average only about 5% of the Teistin variables, thereby not overly restricting the preprocessor. Using an efficient implementation, our preprocessor was able to preprocess each benchmark instance on average in 0.163 seconds. For most benchmarks, this time is insignificant. Even on the largest Intel benchmark, which contains 261,275 variables and 685,929 clauses, preprocessing time was only 5.8 seconds. In total, the reduction of the transfer function reduced the solve time by half. However, our technique does more than just reduce the solving time.

To provide a better picture of where the speedup is coming from, Table 2 takes a closer look at specific benchmark instances. In this table, the first columns labelled *Benchmark* give the instance name, solution or fixed point depth (*Depth*), and if the instance was satisfiable (*S/U*). For the depth column, if the problem is unsatisfiable, the depth at which the fixed point was found using Craig interpolation does not need to be

| | | Our Solver | | | | ABC | | | | TIP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench. | # | #S | #U | Total | Time | #S | #U | Total | Time | #S | #U | Total | Time |
| 139* | 99 | 90 | 9 | 99 | **44.25** | 90 | 9 | 99 | 1,208.66 | 90 | 9 | 99 | 1,345.00 |
| ab* | 5 | 5 | 0 | 5 | 16.56 | 5 | 0 | 5 | **8.22** | 5 | 0 | 5 | 13.14 |
| bc57 | 7 | 7 | 0 | 7 | 463.45 | 7 | 0 | 7 | 513.65 | 7 | 0 | 7 | **309.29** |
| bj* | 44 | 17 | 20 | 37 | 6,322.22 | 17 | 25 | 42 | **2,159.47** | 17 | 24 | 41 | 3,138.61 |
| br* | 5 | 5 | 0 | 5 | **0.18** | 5 | 0 | 5 | 1.23 | 5 | 0 | 5 | 0.88 |
| cmu* | 4 | 0 | 2 | 2 | **1,801.37** | 0 | 2 | 2 | 1,807.23 | 0 | 1 | 1 | 2,701.89 |
| count* | 2 | 2 | 0 | 2 | **0.02** | 2 | 0 | 2 | 0.05 | 2 | 0 | 2 | 0.06 |
| cs* | 4 | 4 | 0 | 4 | **1.46** | 4 | 0 | 4 | 13.58 | 4 | 0 | 4 | 3.65 |
| dm* | 20 | 20 | 0 | 20 | **0.88** | 20 | 0 | 20 | 42.05 | 20 | 0 | 20 | 11.78 |
| eijk* | 28 | 0 | 12 | 12 | 14,763.48 | 0 | 28 | 28 | **65.14** | 0 | 24 | 24 | 3,605.04 |
| intel* | 60 | 12 | 16 | 28 | **34,877.55** | 0 | 6 | 6 | 48,633.19 | 5 | 2 | 7 | 48,368.96 |
| irst* | 3 | 3 | 0 | 3 | 1,344.94 | 0 | 0 | 0 | 2,700.00 | 3 | 0 | 3 | **421.42** |
| ken* | 16 | 2 | 14 | 16 | **0.76** | 2 | 14 | 16 | 13.38 | 2 | 14 | 16 | 2.56 |
| mutex* | 2 | 2 | 0 | 2 | **0.02** | 2 | 0 | 2 | 0.06 | 2 | 0 | 2 | 0.03 |
| nec* | 13 | 2 | 10 | 12 | **927.31** | 2 | 4 | 6 | 6,310.14 | 2 | 10 | 12 | 2,979.94 |
| nus* | 32 | 8 | 18 | 26 | **5,446.15** | 8 | 15 | 23 | 8,182.97 | 8 | 14 | 22 | 10,233.06 |
| pc* | 5 | 5 | 0 | 5 | **0.24** | 5 | 0 | 5 | 5.86 | 5 | 0 | 5 | 1.31 |
| pd* | 235 | 28 | 161 | 189 | 42,620.07 | 28 | 191 | 219 | **15,059.02** | 28 | 178 | 206 | 27,581.31 |
| prod* | 19 | 19 | 0 | 19 | **34.11** | 19 | 0 | 19 | 858.40 | 19 | 0 | 19 | 23.73 |
| ring* | 2 | 2 | 0 | 2 | 0.03 | 2 | 0 | 2 | 0.11 | 2 | 0 | 2 | **0.02** |
| short* | 2 | 2 | 0 | 2 | **0.01** | 2 | 0 | 2 | 0.05 | 2 | 0 | 2 | **0.01** |
| srg* | 3 | 3 | 0 | 3 | **0.03** | 3 | 0 | 3 | 0.18 | 3 | 0 | 3 | 0.08 |
| texas* | 19 | 9 | 10 | 19 | 9.51 | 9 | 10 | 19 | 3.79 | 9 | 10 | 19 | **1.75** |
| vis* | 16 | 6 | 10 | 16 | 1,055.64 | 6 | 10 | 16 | **36.40** | 6 | 8 | 14 | 2,099.85 |
| Total | 645 | **253** | 282 | 535 | 109,730.24 | 238 | **314** | **552** | **87,622.84** | 246 | 294 | 540 | 102,843.37 |

**Table 3.** Overall Comparison to Other Solvers.

the same for both solvers due to preprocessing. If the depths were different, the depth for the basic solver is given in brackets. In the majority of cases, the use of preprocessing reduced the depth at which the fixed point was found. In one case by over 30 iterations. This alone has a dramatic effect on the performance of the solver.

In the next part of Table 2 labelled *Transistion Relation*, the number of variables and clauses in the original transfer function and simplified preprocessed version are given (*#V Orig.*, *#C Orig.*, *#V Simp.*, and *#C Simp.*). Again, the number of don't touch variables is also provided (*#Don't*). Finally, the last part of this table, gives the solve time without the preprocessor (*woPre*) and with the preprocessor (*wPre*). The last column of Table 2 then presents the speedup preprocessing provides on each instance. As can be seen, on just about every large or very deep benchmark, preprocessing provides a significant advantage. Only on benchmarks that are very small and easily solved, does the time for preprocessing negatively effect the speedup. Furthermore, speedup was obtained on both satisfiable and unsatisfiable instances. On many of the larger benchmarks, the calculated speedup was also limited by the fact that the solver without preprocessing was unable to solve the formula in 900 seconds (TO = Timeout).

Table 3 compares our solver to TIP [8] and ABC [3]. These are two of the best solvers from the 2008 Hardware Model Checking Competition. As can be seen, our solver performs quite well to other state of the art solvers. Our solver is the fastest on 15 of the 24 families of benchmarks, and solves the most satisfiable instances. Furthermore, if we compare TIP (which is thought to use the preprocessing method introduced

| Benchmark | S/U | #Vars. | #Cla. | Our Solver | ABC | TIP |
|---|---|---|---|---|---|---|
| intel048 | — | 261,275 | 685,929 | TO | TO | TO |
| intel013 | — | 193,730 | 506,572 | TO | TO | TO |
| intel039 | Sat | 127,308 | 328,436 | **370.83** | TO | TO |
| intel040 | Sat | 125,386 | 322,616 | **379.48** | TO | TO |
| intel041 | Sat | 125,377 | 324,013 | **376.26** | TO | TO |
| intel038 | Sat | 122,600 | 317,149 | **371.68** | TO | TO |
| intel042 | Sat | 122,375 | 316,488 | **423.18** | TO | TO |
| intel028 | — | 107,502 | 280,941 | TO | TO | TO |
| intel043 | Sat | 104,349 | 272,697 | **624.94** | TO | TO |
| intel036 | Sat | 98,327 | 262,244 | **590.42** | TO | TO |
| bjrb07amba10andenv | — | 98,148 | 294,138 | TO | TO | TO |
| intel009 | Sat | 91,644 | 244,587 | **573.78** | TO | TO |
| intel030 | Sat | 91,631 | 244,556 | **612.22** | TO | TO |
| intel037 | Unsat | 87,376 | 228,882 | **2.07** | TO | TO |
| intel012 | — | 86,016 | 225,085 | TO | TO | TO |
| neclaftp1001 | Unsat | 79,164 | 205,876 | **3.44** | TO | 753.21 |
| neclaftp1002 | Unsat | 79,164 | 205,876 | **3.26** | TO | 759.23 |
| intel027 | — | 78,321 | 206,092 | TO | TO | TO |
| intel033 | Sat | 76,328 | 204,095 | **607.56** | TO | TO |
| intel035 | Sat | 75,620 | 202,019 | **582.27** | TO | TO |
| bjrb07amba9andenv | — | 73,003 | 218,731 | TO | TO | TO |
| intel014 | — | 62,017 | 161,737 | TO | TO | TO |
| neclaftp2001 | Unsat | 48,831 | 126,273 | **1.46** | TO | 59.78 |
| neclaftp2002 | Unsat | 48,831 | 126,273 | **1.53** | TO | 61.97 |

**Table 4.** Solver Comparision on Largest Benchmarks.

in [7] but not published) to our work on satisfiable instance, we are 34% faster and solve 7 more benchmarks. Comparison on unsatisfiable instances cannot be directly done because TIP uses k-induction and we use Craig interpolation, however, our Craig interpolation does not seam to be as powerful as the k-induction used in TIP.

Lastly, Table 4 compares the solvers on the 24 largest benchmarks. These benchmarks are sorted by the number of variables in the unpreprocessed transition function. Our solver completely dominates this table solving 16 of the largest 24 benchmarks. TIP only manages to solve 4, and ABC nothing. Furthermore, even when TIP is able to solve a specific benchmark, we are an order of magnitude (or 2) faster. This really shows the promise that our preprocessing holds as it makes large benchmarks manageable.

## 5 Conclusion

In this paper we have presented new ideas showing how preprocessing can be used effectively in incremental SAT solving on large BMC benchmarks. We have shown that our methods provide speedup by allow the solver to find SAT instance faster, and fixed points sooner (less unrolling). Furthermore, we have shown that our methods lead to unmatched performance on satisfiable instances. Also, since our techniques can be used independently of any SAT based BMC tool, it should be straightforward for others to achieve similar results. In the the future, we plan to improve our Craig interpolation through strengthening and minimization [6]. This should allow us to be more competitive on unsatisfiable instances. In addition, we plan to extend the solver with ideas such as Strichman learning [18] which is not possible with other preprocessing techniques.

## References

1. A. Biere. Resolve and Expand. In *International Conference on Theory and Applications of Satisfiability Testing*, 2004.
2. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *IEEE/ACM Design Automation Conference*, 1999.
3. R. Brayton, M. Case, A. Hurst, and A. Mishchenko. ABC and ABmC Entering HWMCC'08. In *Hardware Model Checking Competition Solver Description*, 2008.
4. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Journal of Formal Methods in System Design*, 2001.
5. W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 1957.
6. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant Strength. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2010.
7. N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *International Conference on Theory and Applications of Satisfiability Testing*, 2005.
8. N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *International Workshop on Bounded Model Checking*, 2003.
9. N. Eén and N. Srensson. An Extensible SAT-Solver. In *International Conference on Theory and Applications of Satisfiability Testing*, 2003.
10. Hardware Model Checking Competition. 2008. http://fmv.jku.at/hwmcc08/.
11. M. Herbstritt, B. Becker, and C. Scholl. Advanced SAT-Techniques for Bounded Model Checking of Blackbox Designs. In *Microprocessor Test and Verification Workshop*, 2006.
12. M. Järvisalo, A. Biere, and M. Heule. Blocked Clause Elimination. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2010.
13. M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *Asia and South Pacific Design Automation Conference*, 2007.
14. K. L. McMillan. Interpolation and SAT-based model checking. In *International Conference Computer Aided Verification*, 2003.
15. F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking based on aigs, bdd sweeping, and quantifier scheduling. In *Conference on Formal Methods in Computer Aided Design*, 2006.
16. A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*. 2001.
17. M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *International Conference on Formal Methods in Computer-Aided Design*, 2000.
18. O. Strichman. Accelerating Bounded Model Checking of Safety Properties. *Journal of Formal Methods in System Design*, 2004.
19. S. Subbarayan and D. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT Instances. In *International Conference on Theory and Applications of Satisfiability Testing*, 2004.
20. The VIS Group. VIS: A system for verification and synthesis. In *International Conference on Computer Aided Verification*, 1996.
21. G. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.