



PROperty based SYstem Design

FP6-IST-507219

Research Report

Improved Decision Heuristics for High Performance SAT Based Static Property Checking

Public Version

Marco Roveri, Alessandro Cimatti

Ohad Shacham, Karen Yorav

(Annexes Deliverable 3.2/1)



Notices

First edition (October 2004)

For information, contact Marco Roveri roveri@irst.itc.it or Ohad Shacham ohads@il.ibm.com.

Table of Revisions

Issue	Date	Description and reason for the modification	Affected sections
0.1	Oct, 26 2004	Creation – IBM part only	
0.2	Oct 31, 2004	Integration with document sent by ITC	
0.3	April 11, 2005	Public version	

Abstract

This document contains important aspects of SAT Solver performances and specification for improving this performance. The document looks at follow-up work done in ITC and IBM.

Purpose

This document comes to present results of activity started and explained in the initial delivery 3.2_1.

Intended Audience

This information is meant to be read by highly technical personnel that are interested in finding ways to improve their SAT solvers. It is assumed that readers are familiar with all the notions and terms related to SAT solvers, formal verification, CNFs, and so forth.

Background

Each chapter starts with a background subchapter. This subchapter explains the status before the work started. The rest of the work shows the results of the work done in the frame of this project.

Contents

Incremental BMC	3
Background and Related Works	3
Incremental BMC within the NuSMV Model Checker	11
Clause Groups	11
Integration with zChaff and MiniSAT	12
Zig-Zag Algorithm	12
Dual Algorithm.....	13
Incremental SAT-based LTL Model Checking Routines	13
Experimental Analysis	14
The IBM Formal Verification Benchmark	14
Results	15
Conclusions	16
References	19
Mage	21
Background and Related Works	21
zChaff.....	22
Siege	23
Mage	23
Decision Heuristics for Tuning SAT	25
Rewarding Short Clauses	26
VSIDS vs. VMTF.....	28
VMTF Parameters	28
References	30

Incremental BMC

Background and Related Works

BMC basis: We assume given a Finite State Machine (FSM) expressed in some high level language like e.g. Verilog or VHDL. States of the FSM are given by the assignment of truth values to the *state variables*. The FSM has a non-empty set of *initial states*, and the *reachable states* are all those states which can be reached from one of these initial states. A safety property is specified as a propositional formula over the state variables. The aim is to prove that the safety property holds in every one of the reachable states. Transitions of the FSM are represented by a propositional formula $\mathbf{T}(s, s')$ and the set of initial states by a formula $\mathbf{I}(s)$. The safety property that is to be proved is denoted by $\mathbf{P}(s)$, and the value of the state variables at time n are denoted by s_n . The shorthand notations \mathbf{I}_n , \mathbf{P}_n and \mathbf{T}_n are used in place of $\mathbf{I}(s_n)$, $\mathbf{P}(s_n)$ and $\mathbf{T}(s_n, s_{n+1})$ respectively. Furthermore, $[\varphi]^p$ is used to denote a set of clauses defining φ , such that p is the literal representing the truth-value of the whole formula. To this end, p is called the *definition literal* of φ . The expression $[\varphi]$ is used as shorthand for $[\varphi]^p \cup \{p\}$.

BMC translates a safety formula from LTL [1] into a propositional formula under bounded semantics. The general structure of a $\mathbf{G}(P)$ (*globally P*) formula, as generated in BMC [2], is as follows:

$$\Phi : I(s_0) \wedge \bigwedge_{i=0}^k T(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^k \neg P(s_i) \right)$$

I_0 is the initial state, $\rho(i, i+1)$ is the transition between cycles i and $i+1$, and P_i is the property at time instant i . If this propositional formula is proven to be satisfiable, the satisfying assignment provided by the SAT solver is a counterexample to the property $\mathbf{G}(P)$. To convert the initial propositional formula into Conjunctive Normal Form (used as the input format by most SAT solvers), extra variables are introduced to avoid combinatory explosion. Usually, these extra variables represent more than 80% of the total number of variables in the CNF formula.

SAT basis: SAT is the problem of determining the satisfiability of a Boolean formula. The problem was used by Cook to define NP-completeness [3]. Today, many implementations are available for solving the problem (zChaff [4] and MiniSAT [5]). Most of them are based on the complete DPLL algorithm [6]. DPLL-based algorithms work in a loop that chooses a variable, assigns a value to this variable (a step also called "deciding" on a variable), and then propagates this value through the formula. Whenever the current assignment can be shown not to be satisfying, the algorithm backtracks to earlier decisions and changes them. An high-level description of the algorithm as reported in [4] is as follows:

```

while(true) {
    if (!decide()) // if no unassigned vars
        return(satisfiable) ;
    while (!bcp()) {
        if (!resolveConflict ())
            return (not_satisfiable) ;
    } }

bool resolveConflict() {
    d=most recent decision not 'tried both ways';
    if (d==null) // no such d was found
        return false ;
    flip the value of d;
    mark d as tried both ways ;
    undo any invalidated implications;
    return true;
}

```

decide() is a function that chooses the next variable according to which branching will occur. There are many heuristics for choosing this next variable, such as DLIS (Dynamic Largest Individual Sum) and VSIDS (Variable State Independent Decaying Sum). For instance zChaff uses VSIDS as its decision heuristic. *bcp()* detects unit clauses (clauses in which there remains only one undefined variable) and assigns values to variables accordingly. As a result, BCP detects conflicts, suggesting that there is no satisfying assignment in the current search branch. When a conflict is detected the algorithm backtracks and removes one or many assignments. It returns true when the boolean constraint propagation (*bcp*) [4] finishes without conflict.

Incremental SAT

Standard DPLL solvers which use conflict analysis and clause reordering techniques traditionally take a complete propositional formula and state whether or not it is satisfiable. If there are a number of similar SAT instances to be solved, then the solver will potentially carry out a high number of the same inferences for each one. Incremental SAT addresses this issue by allowing new clauses to be added to the database and the solver run again, without starting the search process from the beginning. The motivation behind this functionality is that learned clauses may not only be useful in that particular problem, but in *similar* ones too. This extension is still quite restrictive as there is no provision for the removal of clauses from the database, a facility which greatly increases the range of problems that can be tackled.

A method to remove, as well as add, clauses to a modern state-of-the-art DPLL based solver (one which uses *conflict clauses* for learning purposes) was suggested by [7]. However, when clauses are to be removed, deciding which conflict clauses also have to be removed requires considerable analysis, and thus time. The reason being that the conflict clauses are dependent on other clauses in the database - removing one or more clauses may therefore invalidate some of the conflict clauses.

To avoid this problem Eén and Sörensson [5] proposed that clauses should only be able to be added but that when the solver is called it is passed a list of unit literals which are assumed to be true. These are then "forgotten" when the solver returns. The advantage of this method is that *all* the learned clauses can be kept in the database since conflict clauses are independent of the assumptions under which they are made.

Note, also, that the removal of added clauses can be achieved in the following manner:

1. augment each clause to be added with a new variable.
2. call the solver with that variable set to false.
3. to delete the clauses, call the solver with the new variable set to true.

Although this will introduce a significant number of new variables, they will all be passed as unit clauses to the solver, so will be removed by the simplification procedures before the search begins.

```

addClauses(Initial)           -- the initial clause set
for  $i \in 0..n$  do             -- for each problem instance
    addClauses( $[P_n]^n$ )       -- add clauses specific to the current problem
    solve(LiteralList)       -- solve current problem (passing necessary
                                -- assumptions)

```

Any clauses which may need to be removed at some stage should be augmented with a definition variable, as described above. The list of assumption literals passed to the solver is used to determine which clauses to keep and which to discard for that specific problem instance.

This interface has been implemented in the MiniSAT [5] boolean satisfiability solver, available from <http://www.cs.chalmers.se/~een>. A similar interface is also provided by the zChaff [4] and is available from <http://www.princeton.edu/~chaff/>.

BMC and Incremental BMC

Bounded Model Checking (BMC) was introduced at the end of the last decade [2] as an alternative to Binary Decision Diagrams (BDDs) for showing the presence of, or proving the absence of specific properties in a given system.

For successively increasing problem bounds (n), a formula which encodes the statement "*the property P holds for all paths of length n which begin in an initial state*" is generated. This formula is then passed to a SAT solver which states whether it is satisfiable or not. The problem bound is increased until either a maximum value is reached, or the property is found to be false. Because of the nature of BMC, unless the upper bound on possible bug lengths is known, it is not possible to prove that a property holds for *any* value of n , just that it holds for all paths of length *no greater* than n .

Typically, for a given value of n , the formula which is generated is similar to the one for $n-1$, which in turn is similar to the one for $n-2$, and so on. Each individual formula is treated as a completely new problem instance by the SAT solver, instead of it exploiting these similarities. Bounded Model Checking therefore has a great deal to gain from the incremental techniques described in the previous section.

For increasing values of n (starting at 0), the following information needs to be represented by the formula which is to be solved:

- The property to be checked should hold in all states reachable in n steps from an initial state.
- Each path must not contain any duplicated states (equivalent to forcing every state to be unique).

Clauses which represent the fact that the property has to hold in all states reachable in $n-1$ steps can be removed, as they are no longer needed for the subsequent stages. The uniqueness requirement can be encoded by including extra clauses over and above those which state that paths of length $n-1$ must have no loops.

This form of Incremental Bounded Model Checking is equivalent to just extending the base-case in Temporal Induction (see the next section). An upper bound on possible bug lengths can be found using Temporal Induction techniques too. This operates in a similar manner to what is described here except that it works backwards, finding the longest distinct state paths in which the property eventually does not hold.

Incremental Temporal Induction

Temporal Induction was introduced in [8] as a method for reasoning about safety properties over the individual time steps of a FSM. A modification to this algorithm that operated incrementally was later presented in [9].

As with a standard induction proof, a temporal induction proof consists of a base-case and an induction-step. For a given number of steps, the base-case proves that the property always holds, while the induction-step proves that no state where the property is false can be reached with one more transition.

The base-case and induction-step are defined in terms of the following formula:

$$\begin{aligned} \mathbf{Base}_n &:= \mathbf{I}_0 \wedge ((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \dots \wedge (\mathbf{P}_{n-1} \wedge \mathbf{T}_{n-1})) \wedge \neg \mathbf{P}_n \\ \mathbf{Step}_n &:= ((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \dots \wedge (\mathbf{P}_n \wedge \mathbf{T}_n)) \wedge \neg \mathbf{P}_{n+1} \end{aligned}$$

Basic Temporal Induction

Base-Case: The base-case is defined as all paths of length n starting from an initial state such that P , the property to be checked, holds in all of the states except the n -th one. Therefore, if the base case is unsatisfied then the property holds for all paths of length n , starting from an initial state (when checking a base case, it is assumed that all shorter base cases have already been proven). In other words, assuming that the property holds for the first $n-1$ states, there is no path of length n to a state in which P does not hold.

Induction-Step: The induction step is defined as a path of length $n+1$ where P holds in all states except the last one. Once again, if it is unsatisfiable then, given that the property holds for a path of length n , there exists no next state where it does not hold.

The states in the induction step must be unique to ensure that the method is complete. Without this restriction it may not be provable even though the property is true. The reason being that if a path contains a loop then it is infinite, thus the induction-step will hold for any length, even if the property is eventually false.

Algorithm

The following algorithm checks successively increasing lengths of base and step cases.

```

for  $n \in 0..∞$  do
  if (satisfiable([Base $n$ ]))
    return PROPERTY FAILS
  if ( $\neg$ satisfiable([Step $n$ ]  $\cup$  [Unique $n$ ]))
    return PROPERTY HOLDS

```

Unique _{n} is a set of constraints representing the fact that every state must be different.

This algorithm can be modified in a number of ways, for example checking only the base-case gives a pure bug hunting algorithm which will find counter-examples more quickly (it cannot however prove that no bug exists). Also, executing body of the “**for ... do**” loop for every value of n may take too long, especially if the bug or proof is deep. Therefore, starting at a higher n and taking bigger steps can reduce the time taken to return a result. This may not return the shortest counter-example though, whereas the algorithm given above is guaranteed to do so.

Exploiting Incremental Induction

The temporal induction algorithm presented in the previous section can be split in to two parts -- one to find increasing lengths of base-cases and the other to find increasing lengths of induction-steps. The incremental algorithms for these two parts and an interleaving of them, all given in [9], are shown and described below.

Extending the Base-Case

```

addClauses([I0])
for  $n \in 0..\infty$  do
    addClauses([Pn]pn)
    solve({¬pn})
    if (SATISFIABLE)
        return PROPERTY FAILS
    addClause({pn})
    addClauses([Tn])

```

The first line adds the formula which represents the set of initial states. Line 3 adds the clauses representing the safety property for the states reachable in n steps (**P** _{n}) -- this is done because it generally makes the SAT problem easier. These clauses are augmented with an additional new variable that is assumed to be false when the solver is called. If the property holds then a unit clause containing only the negation of the new literal is added to the database, thus resulting in the removal of the previously added clauses. Finally, clauses representing the transition to the next state are added and the process (from line 2) is repeated.

This algorithm terminates if the property is found not to hold, or once a desired value for n has been reached (by replacing ∞ with the desired maximum value).

Extending the Incremental-Step

```

addClauses([¬P0])
for  $n \in -1..\infty$  do
    solve({ })
    if (UNSATISFIABLE)
        return INDUCTIVE STEP HOLDS
    addClauses([Tn])
    addClauses([Pn])
    for  $i \in 0..n+1$  do
        addClauses([si ≠ sn])

```

This algorithm works backwards from the states in which the property does not hold. At any stage of the algorithm, if the clause set is satisfiable then there is a route consisting of $n+1$ steps that leads to a state where the property does not hold. In each of the initial n steps, the property is assumed to hold.

When the formula is unsatisfiable, this means that there are no paths of length greater than n (where each state is unique) that lead to a state in which the property does not hold. Therefore, the longest possible counter-example will be of length n .

At each stage, when the step is being extended, the clauses representing the transitions to the previous states are added, as well as those which say that the property should hold in every such state. The restriction to unique states ensures that the paths considered do not contain any loops and are therefore finite.

Combined Algorithm

There exist many possible ways in which to combine algorithms 2 and 3. The following is an example of how they can be interleaved, thereby allowing the solver to share conflict clauses between the two processes.

```

addClauses( $[I_0]^z$ )           --  $z$  is the definition literal for  $I_0$ 
for  $n \in 0..\infty$  do
    addClauses( $[P_n]^{p_n}$ )     --  $p_n$  is the definition literal for  $P_n$ 
    solve( $\{\neg p_n\}$ )         -- induction-step:  $I_0$  not included
    if (UNSATISFIABLE)        -- property guaranteed to hold
        return PROPERTY HOLDS
    solve( $\{z, \neg p_n\}$ )      -- base-case: include  $I_0$ 
    if (SATISFIABLE)         -- counter-example found
        return PROPERTY FAILS
    addClause( $\{p_n\}$ )         -- assert  $P_n$  from now on
    addClauses( $[T_n]$ )        -- assert transition from  $s_n$  to  $s_{n+1}$ 
for  $i \in 0..n-1$  do
    addClauses( $\{s_i \neq s_n\}$ ) -- add uniqueness constraints

```

The definition literals are used to control how their clauses are used by the solver. Not asserting the definition literal at all is equivalent to not including those clauses in the overall formulae, since they are tautological. Note that such clauses are satisfied when either the original variables satisfy them and the definition literal is true, or the original variables do not satisfy them and the definition literal is false. In both cases the definition literal will take on the required truth assignment. Asserting it to be true forces them to be satisfied by the original variables only and asserting it to be false forces them to be unsatisfied by the original variables.

If the induction-step is unsatisfiable then the property is guaranteed to hold because a base-case of length n has already been proven, therefore any counter-examples are of greater length. However, recall that if the induction-step is unsatisfiable, then this means that the longest possible counter-example is of length n therefore the property holds as there are no possible counter-examples.

A graphical Notation

It is possible to have a pictorial representation of the algorithms described in previous section. In the following, the term bad state is used to mean any state in which the safety property does not hold. Counter-examples are symmetric with respect to the initial states and the bad states, that is they begin in an initial state, have a number of intermediate states and end in a bad state. As such, if the transition relation was inverted and the initial and bad states swapped then everything described so far could have been carried out in reverse.

The induction-step is therefore a method for finding an upper bound for the counter-example, and the base-case as a way to produce the counter-example. Graphically, shortest counter-examples will look like the following (where B' and I' are used to mean $\neg B$ and $\neg I$ respectively):

length 0:	IB
length 1:	IB' ← T → I'B
length 2:	IB' ← T → I'B' ← T → I'B
length 3:	IB' ← T → I'B' ← T → I'B' ← T → I'B
	...
length n :	IB' ← T → I'B' ← T → ... ← T → I'B' ← T → I'B

As can be seen from the diagram, there is a significant amount of sharing between the counter-examples of different lengths. More specifically, if the initial I or the final B is removed from the n^{th} counter-example:

or

(1)	B' ← T → I'B' ← T → ... ← T → I'B' ← T → I'B
(2)	IB' ← T → I'B' ← T → ... ← T → I'B' ← T → I'

then *any* counter-example of length n or longer will include both these sections. This means that if the clauses representing (1) and (2) are unsatisfiable then *all* shortest counter-examples of longer lengths will also be unsatisfiable. Therefore this gives an upper bound on the length of the shortest counter example.

Since it is a *shortest* counter-example that is being considered, it can also be concluded that:

- | | |
|------------------|---|
| | 1. Between no two states is there a shorter path |
| <i>or weaker</i> | 2. Between no two non-neighbours is there a transition (and the last state is unique) |
| <i>or weaker</i> | 3. No two states are the same |

The weaker versions are easier to implement, and using only the third condition is enough to make the algorithm complete.

Incremental BMC within the NuSMV Model Checker

The algorithms described in the previous sections have been integrated into the model checker NuSMV [10]. SAT-based Bounded Model Checking routines can now take advantage of incrementality, and in particular, it can now be shown that invariants (safety properties) are true, not just false, by using incremental temporal induction

For the new routines, NuSMV can make use of zChaff or MiniSAT, both of which support incrementality. The interface to these solvers (and the built-in non-incremental solver Sim) has been completely redesigned to facilitate the new routines. It has been made highly generic so as to allow for potentially easier integration of other SAT solvers at a future date.

Clause Groups

At each step of the incremental algorithms, new clauses are added to the solver that in certain cases have to be removed for future steps. Such a scenario arises when the Zig-Zag algorithm is being used to check invariants; clauses representing the initial state are only included in the formula to be solved when the base case is being extended, and have to be removed when checking the step case.

The idea of a group is used by NuSMV to facilitate the addition, and subsequent removal, of sets of related clauses. There are two types of groups used in NuSMV, permanent and temporary, and all clauses must be associated with a particular group.

Permanent Group

There is only one permanent group for any given model, and clauses added to this group cannot be removed. Clauses in this group will be included in the formula for all future attempts to find a solution.

Temporary Groups

There can be any number of temporary groups, and they are used to facilitate the removal of clauses which have been added to the solver.

This is achieved by augmenting each clause with an extra literal, unique to that group, before adding all the clauses to the solver. When the clauses in the group are to be included in the formula to be solved, the extra literal is forced to be false. When the group is to be removed, the extra literal is forced to be true.

At any stage of execution, any combination of temporary groups can be included in the formula that is being solved, as well as the permanent group which is always included.

Sub-formula Polarity

NuSMV also has the ability to specify the polarity for a particular sub-formula, for example, the clauses representing a safety property at a given time step. The sub-formula has a polarity literal associated with it which, in the example, means that it is possible to specify that the safety property holds at the time step, or that it doesn't.

Integration with zChaff and MiniSAT

In this section we describe the details of the interface of the zChaff [4] and MiniSAT [5] incremental SAT solvers that have been integrated within the NuSMV model checker.

zChaff

The latest version of zChaff,[4] which provides support for incrementality, is used by NuSMV. It incorporates the idea of groups, with the ability to add and remove them. However, it is not possible to have a group which is re-added after it has been removed. The reason for this is that removed groups are destroyed and are unable to be used again. As such, we have had to represent groups within NuSMV, although this must be done when MiniSAT is used (see the next section).

As with NuSMV, zChaff has a notion of a permanent group, as well as temporary groups. NuSMV adds all of its groups into zChaff's permanent group, and the literals that determine which of them are to be used, into a temporary group. Recall that a group can be 'disabled' by setting the common literal in the group's clauses to true.

This temporary group is subsequently destroyed once the formula has been solved.

The only drawback to this workaround is that in the case where a group is only used once. The clauses representing such a group will still remain within zChaff when their common literal is set to true. In practice however, this scenario only occurs with the incremental LTL checking routine.

The tableau that is generated at each step is only used for the current bound, and does not play a part in any future formulae to be solved.

It should be noted that, if we to use zChaff's built in grouping mechanism which destroyed a group upon removal, this problem would not occur since the clauses would be 'forgotten' by the solver.

MiniSAT

MiniSAT does not have a notion of groups, so such a mechanism has to be implemented by NuSMV. It does however have the ability to specify a list of unit literals that are to be made true for the current call to the solver. When the solver is invoked, a list of unit literals can be passed which are assumed to be true for that call only.

As with zChaff, all groups within NuSMV are added to MiniSAT, however the unit literals which determine which groups to use are passed explicitly to MiniSAT using the mechanism mentioned.

MiniSAT also suffers from the same problem that zChaff does with incremental LTL checking. Groups which will never be used again do not get removed from the solver, and unlike zChaff, there is no built-in mechanism to do so.

Zig-Zag Algorithm

The clauses representing the initial states of the model are added to a temporary group (herein referred to as group 1). For each time step k , the following procedure is carried out (recall that the step case is extended first, then the base case):

- Clauses representing the invariant at the current time step are added to the permanent group, and a second temporary group (group 2) contains a single unit clause that forces the invariant to be false for the current time step.
- The formula is solved without group 1, and if it is unsatisfiable then there are no paths of the current length to a state where the invariant does not hold, therefore the property is true.

- Otherwise, the base case has to be extended. This is done by solving all the groups, and this time a solution corresponds to a bug in the system.
- If there is no solution, then the invariant is forced to be true at the current time step permanently (by removing the clause from group 2 and adding its negation to the permanent group).
- Finally clauses representing the transitions to the next states, and clauses that force all states to be unique are added to the permanent group.

Dual Algorithm

This algorithm uses two instances of the solver, one for extending the base case and one for extending the step case. Extending the base case is handled as follows:

- Clauses representing the initial states are added to the permanent group. For each time step k , the following procedure is carried out:
- Clauses representing the invariant at time step k are added to the permanent group. These clauses are such that the invariant can be set to hold at the given time step, or not depending on the value of a specific literal.
- This literal is added to the temporary group and is set so that the invariant does not hold.
- All the groups are solved and a solution represents a bug in the system.
- If the formula is unsatisfiable, the unit clause in the temporary group is deleted, and its negation is added to the permanent group, thus making the invariant true at the current time step from this point onwards.
- Clauses representing the transitions to the next states are added to the permanent group.

Extending the step case is done as described in the following. This part of the algorithm only has a permanent group, and initially, clauses representing the states where the safety property does not hold are added to it. Then, for each time step k (recall that this part works backwards from a final bad state), the following is done:

- The formula is solved, and if there is no solution then the property is true. A solution means that there is a path of the current length, k , from a state in the model to a state where the property does not hold.
- If there is a solution, then clauses representing the transitions to previous states are added to the permanent group, as well as clauses that force all states to be unique. Clauses representing the fact that the invariant must hold in all previous states are added to the permanent group.

Incremental SAT-based LTL Model Checking Routines

We have also introduced the ability to check LTL specifications using incremental SAT-based bounded model checking. This operates in almost exactly the same way as the Base Step solver that is used as part of the Dual algorithm for invariant checking. The only difference is that the tableau for the specification has to be generated for each bound from scratch, as it cannot be built incrementally.

Initially, clauses representing the initial states are added to the solver, along with the tableau for the negation of the specification. At any stage, a solution to the formula corresponds to a counter-example.

For each successive bound, the following process is carried out:

- The clauses representing the tableau are removed from the solver
- Clauses specifying the next states are added to the solver
- The tableau for the negation of the specification for the new bound is added
- The new formula is solved

Unlike the invariant checking routines, however, this method is only able to show the presence of a bug, not prove that no bug exists. It does increase the speed of the checking algorithm though.

Experimental Analysis

The IBM Formal Verification Benchmark

With the increased use of formal verification, benchmarking new verification algorithms and tools against real-life test-cases is now a must in order to assess performance gains. However, industrial designs are generally highly proprietary; therefore, models generated from these designs are usually not published. This makes difficult to assess the results reported in papers from the industry, as their benchmarks are usually not available and it is not possible to compare the published results with those achieved by other engines. Additionally, formal verification algorithms described in academic papers are often difficult to assess in terms of performance, since they are usually not applied to “real-life” benchmarks. We want to stress how difficult it is to assess the real practical value of more sophisticated theoretical results without proper benchmarking. In the past, benchmarking enabled significant technology improvements, such as those for BDD packages in [11]. In the same way, many major improvements to boolean satisfiability solvers were proven useful by experimental results [12]. This may appear trivial; however there are many examples in the literature where elaborated and complex algorithms are not evaluated in a satisfactory manner. From the author’s experience, the claims of several papers could not be reproduced using the IBM Formal Verification Benchmark.

The IBM Formal Verification Benchmark library encompasses 37 declassified models, from 31 different hardware designs. The IBM Formal Verification benchmark library is available for academic users from the IBM Haifa verification projects web site [13].

The designs presented in the library are industrial designs that were verified by IBM teams. Each of the benchmark’s 37 files contains:

- A group of one or more temporal formulas collectively called “rule”. To avoid language compatibility issues related to the specification language, the original PSL/Sugar [14,15] formulas were translated into very simple $G(p)$ formulas (still written in PSL/Sugar), which most model checkers can readily address.
- A design model in PSL/Sugar environment description layer format. Some variables were renamed and some simple reductions were applied to hide the original design intent.

The models presented are of different sizes (Cf. Table 1) and varying degrees of complexity. This benchmark is available in PSL/Sugar [15] and in Sugar 1 format [14]. It was also translated to BLIF [16] format. The CNF output of BMC, applied to the benchmark for several bounds, is available from [17]. Some of these CNFs were used for the SAT2003 and SAT2004 contests [18,19].

Name	Variables	Gates	Formulas	Name	Variables	Gates	Formulas
IBM 01	94	3266	1	IBM 17-1	1582	29190	2
IBM 02-1	139	1699	5	IBM 17-2	1581	28807	2
IBM 02-2	135	1671	1	IBM 18	78	4768	1
IBM 02-3	177	1983	7	IBM 19	120	5557	1
IBM 03	109	2656	1	IBM 20	78	4805	1
IBM 04	222	5067	1	IBM 21	78	4768	1
IBM 05	309	8410	1	IBM 22	103	6451	1
IBM 06	132	3375	1	IBM 23	102	6259	1
IBM 07	438	1341	1	IBM 24-1	49048	125896	3
IBM 08	395	84886	1	IBM 24-2	44807	115151	2
IBM 09	232	2000	1	IBM 25	120	4501	1
IBM 10	218	8702	6	IBM 26	1713	9640	1
IBM 11	222	8987	3	IBM 27	42	999	1
IBM 12	224	1055	1	IBM 28	95	3303	1
IBM 13	1506	17459	27	IBM 29	90	2562	1
IBM 14	156	3066	2	IBM 30	180	6654	1
IBM 15	231	4884	1	IBM 31-1	224	2488	3
IBM 16-1	1163	21750	1	IBM 31-2	224	2488	2
IBM 16-2	1162	21674	6				

Table 1: IBM Formal Verification Benchmarks Circuits Details

The EDL [20,21] format of the benchmark was translated into the language of the NuSMV model checker to allow for the experimental analysis carried out and described in this document. The NuSMV format of such benchmarks will be made available by IBM to the whole community.

Results

Detailed analysis has been carried on the new algorithms described in previous section on using the IBM Benchmark suite with the original EDL models converted to the SMV format.

In the experimental analysis we carried out we compared the non incremental algorithms for LTL Bounded Model Checking [2] already present within the NuSMV model checker against the corresponding incremental algorithms and the inductive invariant checking algorithms described in this document and just implemented in the NuSMV model checker.

The results are reported in Table 2. Tests were carried out on a 700 MHz Pentium III, with a time-out of 3600s and a memory limit of 1Gb. Bound values in brackets are correct only if the specifications are false. For each algorithm, the left column gives the runtime in seconds, and the right column gives the memory usage in Mb. In the case of a time-out or memory-out, the bound reached is given in brackets.

Conclusions

It has become clear from the implementation of the incremental algorithms with NuSMV that a number of properties need to be provided by the underlying SAT solver.

The first, and most obvious, is the ability to add new clauses to those already in the solver. It can be safely assumed that any incremental solver will have this ability.

Secondly, there must be a way of specifying clauses that are only relevant to the next solving attempt. That is, they will be removed before any further calls are made to the solver. This is achieved in zChaff by the ability to specify groups which can be removed at a later stage, and although its mechanism is more flexible in that the groups can be removed at any future point, this extra functionality is never utilized by the algorithms presented above.

MiniSAT achieves the same effect in a slightly different way. Clauses groups that you wish to remove at a future point are augmented with a unique additional literal. The negation of this literal is passed to the solver if the group is to be included, and the literal itself is passed if the group is to be disregarded. This method also has the advantage of being able to re-add a group of clauses again, if required.

This brings us to the final ability; to be able to efficiently re-insert clause groups that have been previously removed. Recall that this is required for the zig-zag invariant checking algorithm, since the clauses representing the initial states are only included in the formula when the base case is being extended.

Although zChaff does not provide this function explicitly, it can be implemented in the same way that MiniSAT handles it. An additional literal is added to each clause in the group, and they are added to the solver permanently. Passing the negation of the extra literal to the solver in a temporary group causes the clauses to be included in the formula, and passing the literal itself (in a temporary group) causes them to be ignored.

The only major problem exists with the incremental LTL checking routine, since neither zChaff nor MiniSAT remove clauses which become permanently true. At each step the entire tableau for the previous bound has to be discarded, therefore the clauses representing it will be satisfied forever more. However, they will still be in the solver's internal database, which results in an ever increasing portion of memory that is wasted.

It would seem that the reason why such clauses do not get deleted is down to efficiency, based on non-incremental assumptions. That is, it is thought that clauses which are permanently true due to propagation of unit clauses will be spread throughout the formula, rather than in a contiguous block. However, since the clauses that are being permanently removed in the incremental routines are all part of the same group, they will reside in one large block of memory. Thus, deleting them will not result in severe fragmentation of the memory used by the solver.

This problem is actually handled by zChaff, as it provides the ability to permanently delete a group of clauses. However, this function is not currently used by NuSMV due to the generic interface that is employed. MiniSAT has no provision to delete clauses at all, so within NuSMV we have had to develop a mechanism that would work with both solvers.

Merits of SAT-Based Incremental BMC

From the experimental results presented in the previous section, it can be seen that solving a problem incrementally is significantly faster than generating the complete formula for each problem bound. Not only does the non-incremental method have to generate a new formula at each step, but it is unable to make use of any learned clauses from previous bounds.

The inductive algorithms for checking invariants are generally slower than the corresponding LTL checking methods, but they are doing twice as much work. Not only are they checking for bugs of ever-increasing length, but they are also trying to find an upper bound on the path length of such bugs. This additional part also usually requires longer for a particular bound, since it has the added restriction that any path must not contain duplicated states.

Therefore, if a bug exists then the LTL checking routines will be the fastest way to find it. However, they are unable to prove that invariants are true. For this, the inductive algorithms must be used. An extreme example of the differences can be seen with the IBM Benchmark model `IBM_FV_2002_10` which has six true specifications. The inductive methods show that no bug exists in any of them in less than 30 seconds.

Model (Spec)	True/ False	Bound	Inductive Invariant Checking (Zig-Zag)		LTL Checking Routines			
					Non-Incremental		Incremental	
IBM_FV_2002_01	FALSE	14	1146,38	194,79	478,43	237,54	322,20	171,34
IBM_FV_2002_02_2	FALSE	5	2,13	10,25	3,50	13,59	1,37	9,77
IBM_FV_2002_02_3 (1)	FALSE	4	2,02	10,83	2,72	13,64	1,28	10,42
IBM_FV_2002_02_3 (2)	Unknown	> 75 (> 93)	> 3600 (75)	> 785.75	> 3600 (50)	> 159.49	> 3600 (89)	> 834.43
IBM_FV_2002_02_3 (3)	FALSE	4	2,08	10,84	2,66	13,64	1,20	10,42
IBM_FV_2002_02_3 (4)	Unknown	> 75 (> 95)	> 3600 (75)	> 744.36	> 3600 (50)	> 157.69	> 3600 (86)	> 714.94
IBM_FV_2002_02_3 (5)	FALSE	4	1,93	10,82	3,05	13,64	1,38	10,45
IBM_FV_2002_02_3 (6)	Unknown	> 75 (> 92)	> 3600 (75)	> 773.02	> 3600 (56)	> 180.27	> 3600 (89)	> 721.48
IBM_FV_2002_02_3 (7)	Unknown	> 51 (> 80)	> 3600 (51)	465,39	> 3600 (38)	> 158.85	> 3600 (80)	> 948.41
IBM_FV_2002_03	FALSE	32	1455,06	157,66	133,85	47,71	33,91	42,50
IBM_FV_2002_04	FALSE	24	531,79	127,16	95,98	50,94	18,93	38,59
IBM_FV_2002_05	FALSE	31	510,29	223,58	414,25	95,03	27,83	64,17
IBM_FV_2002_06	FALSE	31	705,32	162,91	723,30	68,45	132,38	71,08
IBM_FV_2002_09	Unknown	> 85	> 2534.2	> 1024 (85)	> 3600 (51)	> 148.13	> 3449.45	> 1024 (75)
IBM_FV_2002_10 (1)	TRUE	1	3,98	26,34	> 3600 (40)	> 449.86	> 3600 (59)	> 511.77
IBM_FV_2002_10 (2)	TRUE	1	4,09	26,35	> 3600 (37)	> 423.37	> 3600 (61)	> 592.11
IBM_FV_2002_10 (3)	TRUE	1	3,98	26,35	> 3600 (28)	> 353.66	> 3600 (36)	> 518.99
IBM_FV_2002_10 (4)	TRUE	1	4,06	26,35	> 3600 (32)	> 390.82	> 3600 (42)	> 512.77
IBM_FV_2002_10 (5)	TRUE	2	6,41	34,57	> 3600 (17)	> 227.59	> 3600 (35)	> 513.11
IBM_FV_2002_10 (6)	TRUE	2	6,41	34,56	> 3600 (18)	> 238.75	> 3600 (33)	> 473.96
IBM_FV_2002_11 (1)	Unknown	31 -35	> 3600 (14)	> 217.39	> 3600 (16)	> 222.35	> 3600 (30)	> 459.06
IBM_FV_2002_11 (2)	Unknown	> 50	> 3600 (13)	> 193.36	> 3600 (15)	> 213.63	> 3600 (24)	> 317.70
IBM_FV_2002_11 (3)	Unknown	> 40	> 3600 (11)	> 151.75	> 3600 (15)	> 213.89	> 3600 (25)	> 446.20
IBM_FV_2002_14 (1)	Unknown	> 50	> 3600 (32)	> 300.74	> 3600 (41)	> 135.74	> 3600 (49)	> 676.37
IBM_FV_2002_14 (2)	Unknown	> 50	> 3600 (30)	> 266.71	> 3600 (30)	> 156.81	> 3600 (41)	> 473.90
IBM_FV_2002_15	FALSE	9	37,18	36,87	18,17	39,19	4,65	26,56
IBM_FV_2002_16_1	Unknown	> 0 (> 1)	> 624.09	> 1024 (0)	> 853.02	> 1024 (2)	> 524.77	> 1024 (1)
IBM_FV_2002_16_2 (1)	FALSE	5	16,70	61,49	9,55	57,25	3,71	49,34
IBM_FV_2002_16_2 (2)	FALSE	5	17,39	62,13	10,89	58,54	4,37	50,45
IBM_FV_2002_16_2 (3)	FALSE	5	17,43	62,08	10,49	58,51	4,51	50,43
IBM_FV_2002_16_2 (4)	FALSE	5	17,04	62,20	10,41	58,75	4,51	50,66
IBM_FV_2002_16_2 (5)	FALSE	0	7,79	29,90	0,13	31,01	0,12	30,99
IBM_FV_2002_16_2 (6)	FALSE	0	8,00	29,97	0,18	31,31	0,16	31,14
IBM_FV_2002_17_1 (1)	Unknown	> 50 (> 175)	> 1388.91	> 1024 (29)	> 3600 (51)	> 402.37	> 3600 (175)	> 834.21
IBM_FV_2002_17_1 (2)	Unknown	> 50	> 3600 (22)	> 540.86	> 3600 (37)	> 352.67	> 1839.61	> 1024 (48)
IBM_FV_2002_17_2 (1)	Unknown	> 50 (> 176)	> 1273.71	> 1024 (30)	> 3600 (61)	> 479.52	> 3292.15	> 1024 (176)
IBM_FV_2002_17_2 (2)	Unknown	> 50 (> 180)	> 1357.3	> 1024 (30)	> 3600 (61)	> 479.21	> 3166.92	> 1024 (180)
IBM_FV_2002_18	Unknown	26 (28) - 30	> 3600 (25)	> 335.35	> 3600 (24)	> 119.50	> 3600 (28)	> 640.66
IBM_FV_2002_19	FALSE	29	1042,80	143,22	> 3600 (27)	> 109.59	277,98	116,38
IBM_FV_2002_20	Unknown	36 - 45	> 3600 (26)	> 341.51			> 3600 (32)	> 644.59
IBM_FV_2002_21	FALSE	29	840,50	102,45			161,94	79,77
IBM_FV_2002_22	Unknown	> 50	> 3600 (32)	> 286.91			> 3600 (31)	> 459.72
IBM_FV_2002_23	Unknown	31 - 40	> 3600 (22)	> 228.94			> 3600 (28)	> 646.18
IBM_FV_2002_27	FALSE	25	14,42	10,88			44,46	45,06
IBM_FV_2002_28	FALSE	14	1075,52	97,84			1327,37	158,37
IBM_FV_2002_29	Unknown	21 - 30	> 3600 (13)	> 302.73			> 3600 (12)	> 189.24
IBM_FV_2002_31_1 (1)	Unknown	> 21 (> 22)	> 3600 (21)	> 347.03			> 3600 (22)	> 608.12
IBM_FV_2002_31_1 (2)	Unknown	> 20 (> 22)	> 3600 (21)	> 353.22			> 3600 (22)	> 415.03
IBM_FV_2002_31_1 (3)	Unknown	> 20 (> 23)	> 3600 (21)	> 216.90			> 3600 (23)	> 609.11
IBM_FV_2002_31_2 (1)	Unknown	> 20 (> 23)					> 3600 (23)	> 424.21
IBM_FV_2002_31_2 (2)	Unknown	> 20					> 3439.63	> 1024 (20)

Table 2: Results of running NuSMV on the IBM benchmarks

References

- [1] A. Pnueli. "A Temporal Logic of Concurrent Programs". In Theoretical Computer Science, Vol 13, pp 45-60, 1981.
- [2] A. Biere, A. Cimatti, E. M. Clarke and Y. Zhu "Symbolic Model Checking Without BDDs" in TACAS 1999, LNCS:1579, Springer.
- [3] S. Cook. "The Complexity of Theorem Proving Procedures". In Proceeding, Third Annual ACM Symp. on the Theory of Computing, 1971.
- [4] M. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In 38th Design Automation Conference, page 530-535. ACM/IEEE, 2001.
- [5] N. Eén, N. Sörensson "An Extensible SAT-solver" short version to appear in SAT2003 formal proceedings. Full paper available from http://www.cs.chalmers.se/~een/Satzoo/An_Extensible_SAT-solver.ps.gz.
- [6] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving". In Journal of the ACM, 5(7), 1962.
- [7] Whittemore, J. Kim, K. Sakallah "SATIRE: a new incremental satisfiability engine" in Proc. of the 38th Conference on Design Automation, pages 542 - 545, ACM Press 2001.
- [8] M. Sheeran, S. Singh, G Stålmarck "Checking Safety Properties Using Induction and a SAT-solver" in FMCAD 2000, LNCS:1954.
- [9] N. Eén, N. Sörensson "Temporal Induction by Incremental SAT Solving" First International Workshop on Bounded Model Checking, 2003. ENTCS issue 4 volume 89.
- [10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella "NuSMV 2: An OpenSource Tool for Symbolic Model Checking" In Proceeding of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark, July 27-31, 2002.
- [11] B. Yang et al. "A Performance Study of BDD-Bases Model Checking". In Formal Methods in Computer-Aided Design: 2nd International Conference, Lecture Notes in Computer Science, 1522. Springer-Verlag, 1998.
- [12] L. Zhang and S. Malik. "The Quest for Efficient Boolean Satisfiability Solvers". In Proceedings of 14th Conference on Computer Aided Verification (CAV2002), Copenhagen, Denmark, July 2002.
- [13] IBM Formal Verification Benchmark Library. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html

- [14] I. Beer et al. “The Temporal Logic Sugar”. In Computer Aided Verification, Proceedings of the 13th International Conference, CAV 2001. LNCS 2102, July 2001.
- [15] Accelerera. PSL/Sugar LRM. <http://www.eda.org/vfv/>
- [16] Berkley Logic Interechange Format (BLIF). University of California, Berkley, 1992. <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/usrDoc.html>
- [17] IBM CNF Benchmark Illustration: Berkmin561 vs zChaff. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks_illustrations.html
- [18] SAT2003 Contest. <http://www.satlive.org/SATCompetition/2003/>
- [19] SAT2004 Contest. <http://www.satlive.org/SATCompetition/2004/>
- [20] Web version of the RuleBase User Manual. http://www.haifa.il.ibm.com/projects/verification_RB_Homepage/
- [21] I. Beer et al. “RuleBase: An Industry Oriented Formal Verification Tool”. In 33rd Design Automation Conference, DAC 96. ACM/IEEE, 1996.

Mage

Background and Related Works

Most current complete SAT solvers are based on a backtracking search algorithm called DPLL.¹ A detailed explanation of this subject is beyond the scope of this document. We therefore give a short overview of the algorithms and terms used.

DPLL-based algorithms work in a loop that chooses a variable, assigns a value to this variable (a step also called "deciding" on a variable), and then propagates this value through the formula. Whenever the current assignment can be shown not to be satisfying, the algorithm backtracks to earlier decisions and changes them.

The process of identifying all variable values that can be inferred from a decision is called Boolean Constraint Propagation (BCP). This procedure detects unit clauses (clauses in which there remains only one undefined variable) and assigns values to variables accordingly. As a result, BCP detects conflicts, suggesting that there is no satisfying assignment in the current search branch. When a conflict is detected the algorithm backtracks and removes one or many assignments.

The variation between SAT solvers stems from different interpretations and implementations of the following strategies.

1. **Decision Heuristic.** The decision heuristic is the function that determines the next variable to be assigned a value. The most profound advancement made in this subject in recent years is the idea of locality, which means the decision heuristic guides the solver to prefer deciding on variables that were involved in recent conflict analysis (see below). This idea is what guides decision heuristics such as VSIDS in Chaff and VMTF in Siege, both of which have shown great speedups in running times.

For more information about DPLL, see [1]¹

2. **Conflict Analysis.** Modern SAT solvers apply learning during the state space search in order to prune unexamined branches of the search tree, which can be proven to be unsatisfiable. The best-known learning technique is *Conflict Driven Clause Learning* – each conflict is analyzed and a constraint (a conflict clause) is generated and added to the formula so that this conflict cannot be encountered again. This restricts the search space that remains to be searched. For a given conflict there can be many different conflict clauses that are appropriate. The different strategies differ in the way they choose the conflict clause (or clauses) to be added for each conflict.
3. **Restarts.** Some SAT solvers use restarts, which periodically abandon the current search tree (without completing it) and start a new one. This technique may increase the robustness of the solver.
4. **Clause deletion.** Eventually, addition of conflict clauses to the solver's database may cause memory explosion. Therefore, solvers incorporate different clause deletion policies.

Our work is closely related and based upon the following solvers, which are considered state-of-the-art in the field.

zChaff

zChaff [2] was initially released in 2001 by Princeton University as a new implementation of the Chaff algorithm. Its main features are:

- Two-literal watching scheme for BCP – only two literals in every clause are "watched". The watched literals can be either satisfied or unassigned. Hence, the clause is checked for constraint propagation only when one of the two literals gets an assignment of 0. Furthermore, the clause needs not be accessed upon backtracking because a watched literal can be unassigned.
- Variable State Independent Decaying Sum (VSIDS) – a decision heuristic oriented towards satisfying the variable that occurs the largest number of times in recently generated conflict clauses. It does so by scoring variables based on their occurrence in recently learned conflict clauses and choosing to branch on the variable with the highest score.
- Conflict analysis and learning – for every conflict encountered during the search process, a reason for the conflict is found and recorded as a conflict clause. A single clause is added to the database.
- Non-chronological backtracking – based on the results of conflict analysis, the solver finds the assignments and implications it needs to undo in order to resolve the conflict.
- Periodic restarts – every so often the solver clears the state of all the variables and starts the search from scratch. However, conflict clauses are not deleted and, therefore, conflict analysis done before the restart will prune the search space in such a way that the solver will not search spaces that were covered before. Restarts drastically increase the robustness of zChaff because they help the solver to escape from local minima.

- Clause deletion – conflict clauses help the solver to prune the search space. However, adding too many clauses to the database increases the overhead of BCP and can degrade performance. In order to prevent this, during the solving procedure, the SAT solver omits clauses from its database. The omitted clauses are clauses that were learned and not original clauses of the formula. This procedure does not hurt the semantics of the formula. There are several heuristics for clause deletion.

In 2004, Princeton University released an updated version of zChaff [3]. The new version includes many improvements and is considered to be superior.

Siege

Siege [4] is a SAT solver provided by Simon Fraser University. Siege was proven to be one of the best SAT solvers currently available. Only a few techniques of its implementation are known; the rest were not published. We mention Siege here because of its decision heuristic, called VMTF (variable move to front). Siege maintains a variables list, and chooses the first unsatisfied variable in this list. The initial order of the list is sorted by the occurrence of variables in the formula. Every time a new clause is added to the database (by the learning procedure), a constant number of variables from the clause are moved to the front of the list. This list is resorted according to the occurrence of variables in the clauses database every time that a restart occurs.

Mage

Mage is a SAT solver that is being developed at IBM Haifa labs. Mage has higher solving capabilities than zChaff, even though it is still a prototype implementation. Our experimental results show that Mage is able to solve hard CNF instances that zChaff is unable to solve. In addition, in most of the instances for which zChaff returns a solution, Mage returns a solution much faster. Mage is still under development and there are many algorithms and techniques that we plan to implement. We believe that these enhancements will increase the robustness of Mage, as well as its performance.

Table 1 contains a comparison between the current version of Mage, zChaff 2004, and the first version of zChaff. The comparison was obtained on an Intel Xeon dual CPU 2.4GHz, 2.5GB RAM platform running Linux. The results are explicit and show that Mage is significantly better than both versions of zChaff. Mage easily solves CNF instances on which zChaff times out after 20000 seconds. In addition, in most of the instances that zChaff is able to solve, Mage's runtime is significantly better.

CNF Formula	Mage	zChaff	zChaff 2004
1	66.69	2648.63	95.16
2	14549.8	12800.29	2272.53
3	13.53	1849.3	982.15
4	32.55	8.47	64.02
5	30.1	51.92	253.47
6	137.5	318.91	238.67
7	1157.49	Timeout	12090.3
8	7.98	19.38	17.86
9	96.84	111.74	49.36
10	410.52	592.75	3461.43
11	441.9	Timeout	11933.5
12	1.22	1.82	1.31
13	10.09	6.46	14.3
14	77.12	461.08	131.61
15	107.67	347.3	363.78
16	688.34	Timeout	7856.03
17	12.96	3.86	21.5
18	45.22	196.98	1069.26
19	78.79	28.04	239.62
20	324.93	18132.02	1781.41
21	0.73	1.36	0.5
22	8.43	3.92	18.12
23	52.66	10.58	210.51
24	201.14	371.32	3657.02
25	282.91	365.22	1575.31
26	547.04	5599.64	Timeout
27	0.72	2.5	0.68
28	65.81	4.37	18.95
29	112.99	8.61	157.11
30	178.37	25.46	2247.83
31	367.47	10461.52	498.11
32	727.89	Timeout	2440.14
Total runtime	20839.31	134435.6	53761.55

Table 1: Comparison of Mage, zChaff 2004, and zChaff on real life BMC instances.

Decision Heuristics for Tuning SAT

In this section, we present a few ideas we tried out in an attempt to improve the performance of our SAT solver when used for Bounded Model Checking (BMC). We used subsets of our benchmark suite in order to check whether each change made was, in fact, an improvement. At first, we tested our ideas on zChaff 2001. Later on, we switched to our new solver, Mage. Mage is an ongoing work in development, described in the previous chapter. However, we believe that the information we gained from working with zChaff 2001 is interesting in itself, even though it was conducted on an inferior solver.

When dealing with SAT solvers, it is often the case that a small change can cause significant reductions in performance, even though, intuitively, we expect to see better results. This is mainly due to two reasons:

- An SAT solver attempts to solve an NP-complete problem in polynomial time. This implies heavy use of heuristics, which perform well on some examples and very badly on others. Any change in the heuristic causes unpredictable results.
- A significant part of the remarkable advancement made in state-of-the-art modern solvers is due to software engineering innovations rather than to actual algorithmic advances. The impressive performance shown by zChaff (and subsequent solvers) is mostly due to the fact that it is tuned to cause a low rate of cache misses. Any implementation change, even a minor one, can break this balance. For example, adding a variable to the structure that represents a literal causes an increase in running time of more than 30%, even if this variable is never used. Therefore, many ideas that seem to make sense turn out to be unproductive because of this phenomenon.

For this reason, it is important to take the results presented here with a grain of salt. In the future, we plan to invest in an even more efficient implementation of Mage. Whatever changes we make are likely to affect the efficiency of the heuristics that we check on our current prototype implementation. Thus, any idea that does not work well now will need to be checked again at a later date.

The good news, however, is that our interest lies in a very specific domain of problems. It is unlikely that we would be able to come up with a decision heuristic that would give better results on any possible CNF. However, we are only interested in applying our solver to the problems that are generated from BMC. Furthermore, it makes sense to limit ourselves to instances that arise from applying BMC to hardware designs, since this is the major goal of our model checker. Thus, we expect that there is some underlying structure that is common to the examples we run on. This gives us hope that we will be able to fine tune our heuristics accordingly.

Apart from implementation issues, there are two points at which modern solvers differentiate—the decision heuristic and the conflict-clause learning mechanism. Any improvement in these procedures has the potential to vastly improve the performance of the solver. In the following subsections, we suggest changes to the decision heuristic and then use experimentation to assess their impact. In the final implementation of Mage, we plan to integrate innovations in the clause learning mechanism as well.

Rewarding Short Clauses

Different works about conflict clause learning seem to agree that it is best to learn short clauses. Taking this idea a step further, we suggest that it may be beneficial to reward literals that appear in short conflict clauses. A similar idea was described in [5]. The idea is to prefer decisions that have the potential to generate many implications. We tried out many strategies for rewarding literals in short clauses. We implemented this on top of zChaff, which uses the VSIDS decision heuristic. Whenever a conflict clause is learned, we check its size, and if it fits into the specified criterion, we increment the scores of the literals in the clause (or their negations). Table 2 presents the different versions we tried. The versions differ in size criterion, the amount by which scores are incremented, and whether the literals or their negation are rewarded.

Version	Clause size	Literal rewarded	Rewarded by
1	<10	same	1000
2	<5	same	1000
3	binary	same	1000
4	<10	same	100
5	<5	same	100
6	binary	same	100
7	<10	same	50
8	<5	same	50
9	binary	same	50
10	<10	same	$50 + (10-n) \ll 3$
11	<5	same	$50 + (5-n) \ll 3$
12	≤ 5	opposite	$50 + (5-n) \ll 3$
13	≤ 10	opposite	$50 + (11-n) \ll 3$
14	>10	same	50
15	>10	same	$50+n$

Table 2: Different strategies for rewarding short clauses

The results of experimentation on a limited benchmark suite suggest that none of these are worthwhile implementing on top of VSIDS. Although we found many instances for which performance was improved, sometimes by an order of magnitude, we found that this happens mostly on short instances. If we focus on the significantly large examples, we see that all versions have a negative effect on performance.

Example	Native	1	2	3	4	5
r_2_3_2.k45	110.29	10.01	28.46	606.1	14.4	52.37
r_2_3_4.k40	66.74	22.21	33.8	95.4	38.58	42.83
r_11_2.k35	662.68	1496.31	3204.03	2039.08	768.79	820.63
r_11_3.k30	385.76	3156.84	1117.21	1385.96	779.62	667.01
r_18.k25	879.97	2049.82	1842.75	1357.32	1033.72	1846.5
r_22.k45	931.98	3561.16	5712.16	1531.91	2053.17	1594.26
r_23.k25	345.55	838.92	1175.37	392.23	383.3	332.64
r_28.k50	118.65	124.59	121.15	100.45	126.11	327.13
r_29.k15	357.73	3046.05	2376.63	1112.82	713.91	1022.16
t_1_14.k14	44.4	37.17	130.86	13.54	55.54	82.55
Sum	3903.75	14343.08	15742.42	8634.81	5967.14	6788.08
Improvement on Sum		3.67418	4.03264	2.211927	1.528566	1.738861

Example	6	7	8	9	10
r_2_3_2.k45	74.08	45.02	89.23	56.26	12.98
r_2_3_4.k40	46.67	36.33	47.03	46.87	11.64
r_11_2.k35	1212.53	661.87	738.66	903.04	527.12
r_11_3.k30	623.09	456.6	556.44	493.17	1445.73
r_18.k25	1251.99	612.11	940.18	604.02	5071.73
r_22.k45	1417.32	1939.09	1144.51	845.2	6274.11
r_23.k25	223.56	320.36	276.34	218.69	688.27
r_28.k50	190.64	277.19	634.24	85.97	108.26
r_29.k15	547.28	619.57	302.22	574.79	2140.59
t_1_14.k14	5.7	44.3	35.46	4.13	89.95
Sum	5592.86	5012.44	4764.31	3832.14	16370.38
Improvement on Sum	1.432689	1.284006	1.220444	0.981656	4.193501

Example	11	12	13	14	15
r_2_3_2.k45	36.09	103.92	41.86	21.74	18.9
r_2_3_4.k40	18.73	16.44	19.41	27.9	66.44
r_11_2.k35	980.8	1451.15	3092.51	905.11	826.31
r_11_3.k30	1597.88	3110.39	3413.36	358.25	477.68
r_18.k25	3486.96	3365.68	1702.12	1022.61	1569.19
r_22.k45	3039.07	5913.32	19388.82	1781.2	1988.81
r_23.k25	852.4	862.48	885.14	404.7	284.51
r_28.k50	225.77	354.95	623.3	391.29	235.87
r_29.k15	2191.35	4691.31	1751.33	1007.46	1135.34
t_1_14.k14	12.94	116.56	148.13	40.96	55.14
Sum	12441.99	19986.2	31065.98	5961.22	6658.19

Improvement on Sum	3.187189	5.119744	7.957984	1.52705	1.705588
--------------------	----------	----------	----------	---------	----------

Table 3: Experimental results for the rewarding strategies from table 2.

VSIDS vs. VMTF

We implemented both VSIDS and VMTF within Mage and compared their performance. We ran the comparison on a large benchmark, which showed that VMTF is by far superior.

VMTF Parameters

We experimented with several flavors of VMTF. First, we tried adding rewarding of short clauses to the decision procedure. Recall that VMTF chooses the first undefined variable in its decision queue, without consulting the score. The scores given to literals are used only to decide on the value given to the chosen variable. The native algorithm for VMTF chooses to make true the literal that has a higher score. Thus, incrementing the score of variables that appear in short conflict clauses will influence the value given to those variables if and when they are chosen. Experimentation with rewarding literals that appear in binary clauses shows that it can improve performance by around 30%. However, the experimentation was conducted on a very small benchmark suite and more experimentation is needed in order to determine whether this method is worthwhile to implement.

Next, we looked at the way VMTF operates. For every learned conflict clause, only a subset of its variables is pushed to the top of the queue. Choosing more variables will make the solver more "localized", in the sense that it will tend to decide on variables from the very last few conflict clauses. Choosing fewer variables will give the solver a chance to explore variables from older clauses. We experimented with two settings for this option: the native setting and a shallow version that chooses only a small number of variables. The impact was not as profound as we had expected; on average the performance is about the same although on single instances, one can see larger differences.

Finally, we turned to the question of choosing the value when deciding on a variable. The scores of the two corresponding literals gave us an indication as to the amount of clauses that will be satisfied by each value. This score is biased towards conflict clauses because whenever a conflict clause is generated, we increment the score of each literal that appears in the generated clause or in the implication graph. The native VMTF algorithm chooses the value for a variable by making the literal with the higher score become true. This serves towards satisfying as many clauses as possible. However, one could argue that it may be beneficial to falsify as many clauses as possible in order to generate more implications (see [4]). We do not yet have experimental results for this option.

CNF instance	zChaff 2001	VSIDS	Reward by 4	VMTF	VMTF + Reward by 4	Shallow VMTF
r_2_3_2.k45	110.29	5.4	2.83	1.47		
r_2_3_4.k40	66.74	8.95	7.63	2.47		

r_11_1.k35	405.6	335.2	999.67	453.3		
r_11_2.k35	662.68	657.54	421.62	162.67		
r_11_3.k30	385.76	225.8	202.2	224.6		
r_18.k25	879.97	202.05	142.01	291.85		
r_22.k45	931.98	585.66	619.85	369.33		
r_23.k25	345.55	176.3	165.07	105.63		
r_28.k50	118.65	41.44	34.54	52.69		
r_29.k15	357.73	152.81	175.82	160.76		
i_1.k35	128.87	30.31	70.84	6.84		
i_8.k40	578.34	286.13	185.43	21.41		
t_1_14.k14	44.4	24.13	8.05	18.36		
cc_31		6.07	1.8	4.09	2.4	
cc_32		9.76	6.77	11.53	5.23	
e_6_1		9.11	4.89	8.28	8.87	
e_6_n_1		146.24	68.11	194.65	83.31	
e_6_n_2		57.66	16.17	58.01	36.53	
is_1_28		823.9	353.05	577.62	290.09	
is_1_29		1008.88	390.58	1064.98	421.71	
is_2_28		22.96	12.18	30.21	11.93	
is_2_29		59.12	15.38	25.94	14.67	
is_3_35		1525.45	1417.5	1318.3	1557.79	
is_3_45		10668.9	8105.14	11166.4	6245.75	
r_1_1_30		313.76	294.08	135.47	182.65	163.71
r_11_1_40		1389.94	1343.89	972.68	757.49	905.67
r_11_2_40		887.88	1273.25	216.98	311.48	323.59
r_11_2_45		2171.11	1121.46	418.59	405.94	446.6
r_14_2_40		236.44	188.02	159.89	180.39	232.43
r_14_2_45		420.86	374.94	244.1	368.95	286.68
r_14_2_50		630.88	646.06	486.69	303.08	587.98
r_18_30		1301.71	2279.85	312.38	582.82	577.38
r_18_35		2358.7	4285.28	1082.12	856.54	694.22
r_19_45		412.33	425.71	207.52	386.48	229.36
r_19_50		866.39	763.03	331.99	320.72	421.48
r_20_20		270.68	285.48	301.4	256.92	212.2

r_20_35		925.88	1074.48	906.97	770.1	1576.43
r_20_50		2756.93	4782.69	3236.44	3227.09	2223.17
r_21_45		456.31	480.33	254.35	201.97	253.87
r_21_50		498.7	1018.24	280.09	205.4	279.23
r_22_40		234.13	80.07	190.3	153.92	190.25
r_22_50		1608.27	1614.19	1250.59	1312.94	1223.72
r_23_50		1213.32	1446.49	851.33	987.3	964.59

Table 4: Experimental results of using different decision heuristics on top of Mage.

References

- [1] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394-397, July 1962.
- [2] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: engineering an efficient SAT solver. In *38th Design Automation Conference*, page 530-535. ACM/IEEE, 2001.
- [3] SAT Research at Princeton. <http://www.princeton.edu/~chaff/>
- [4] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master Thesis. Simon-Fraser University 2002.
- [5] E. Carvalho and J. P. Marques-Silva. Using rewarding mechanisms for improving branching heuristics. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, May 2004.

INDEX

BCP	4, 21, 22, 23, 24
BMCi, 3, 5, 6, 10, 14, 16, 25, 26
Conflict21, 22
DPLL4, 21
Dual	i, 12, 13
Group	
Permanent11
Heuristic21
Incrementali, 3, 4, 5, 6, 7, 8, 10, 13, 16, 19
Induction	
Temporal6, 7, 19
Magei, 21, 23, 24, 25, 26, 29, 31
MiniSATi, 4, 5, 10, 11, 12, 16
NuSMVi, 10, 11, 12, 15, 16, 18, 19
Short Clausesi, 27
Siegei, 21, 23, 24
Tuningi, 26
VMTFi, 21, 23, 24, 29
VSIDSi, 4, 21, 22, 24, 27, 29
zChaffi, 4, 5, 10, 11, 12, 16, 20, 22, 23, 24, 25, 26, 27, 29

