

*FP6-IST-507219*

## **PROSYD:**

*Property-Based System Design*

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

### **Effective Leverage of Different Static Checking Engines (Deliverable 3.1/2)**

Due date of deliverable: March 31, 2005

Actual submission date: March 31, 2005

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: IBM

Revision 1.2

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	<b>Public</b>	<input checked="" type="checkbox"/>
PP	<b>Restricted to other programme participants (including the Commission Services)</b>	<input type="checkbox"/>
RE	<b>Restricted to a group specified by the consortium (including the Commission Services)</b>	<input type="checkbox"/>
CO	<b>Confidential, only for members of the consortium (including the Commission Services)</b>	<input type="checkbox"/>

## **Notices**

For information, contact [gadiel@il.ibm.com](mailto:gadiel@il.ibm.com)

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.1/2 described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2005. All rights reserved.

## Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	7.2.2005	First draft by authors	E. Zarpas G. Auerbach	All
0.2	22.2.2005	Second draft by authors	E. Zarpas G. Auerbach	All
0.3	13.3.2005	Editing and formatting	L. Amitai	All
0.4	20.3.2005	Introduction and experimental results added	A. Fedeli E. Zarpas G. Auerbach	Introduction, Experiments
0.5	23.3.2005	Editing	L. Amitai	Introduction, Experiments
0.6	24.3.2005	Chapters' introduction and experimental results added	A. Fedeli G. Auerbach	All
0.7	30.3.2005	Editing – minor changes	G. Auerbach	All
1.0	31.3.2005	Final approval by IBM (Project Coordinator)	C. Eisner	Version number
1.1	10.4.2005	IBM clearance and editing	G. Auerbach	All
1.2	16.4.2005	Approval of public version by project coordinator	C. Eisner	Version number

## Authors

Gadiel Auerbach  
 Andrea Fedeli  
 Emmanuel Zarpas

## Executive Summary

**Model checking (MC)** is a way to verify logic designs against certain desired properties. Model checkers exhaustively check all possible input sequences, or search all possible reachable design states. This methodology document provides guidance for the effective leverage of various model-checking and semi-formal [15] engines, based on design characteristics such as block size, sequential depth, function, and maturity. It presents a method for managing the trade-off between performance and coverage in a platform that uses among other static engines classical SMV-based [21] model checker and a **Boolean satisfiability (SAT)** [13] [24] bounded model checker [6].

## Purpose

The purpose of this document is to suggest a property-based verification methodology, which leverages different static proof engines according to design characteristics and analysis purpose (verification vs. falsification).

## Intended Audience

This methodology document is intended for individuals who work with multi-engine model checkers [2]. It is assumed that readers are familiar with the notions and terms related to model checking.

## Background

Model checkers, which applied a fixed-point computation of a given formula on an explicit state-space representation of the underlying transition structure, were first introduced in the early eighties. Binary Decision Diagrams (BDD), which allowed the **symbolic** application of functions in a set of "current-states" to provide the set of "next-states" made model checking (MC) applicable to industrial-size designs.

The growth of the state-space representation beyond manageable size has been known as the **state-explosion problem**. An improved algorithm first performed a **reachability** analysis of the state space, which started from the initial states and subsequently restricted the fixed point analysis to the reachable subset of the state space.

One result of that approach was an **on-the-fly** forward reachability computation that allowed finding failures before the completion of the reachability analysis.

In the late nineties and early twenty-first century, more techniques appeared, such as **abstraction-refinement**, **guided search**, **bounded model checking**, **Boolean satisfiability (SAT)** and **semi-formal**. The different techniques vary according to the properties that can be checked (safety vs. liveness), analysis purpose (verification vs. falsification), coverage (model checking, bounded model checking, and semi formal) and the size of the design under test. It became clear that no one single engine can win the prize for "best verification engine". Recognizing this fact, this methodology document addresses the issue of effective leverage of different static checking engines.

# Contents

Table of Revisions.....	iii
Authors .....	iii
Executive Summary .....	iii
Purpose .....	iv
Intended Audience .....	iv
Background.....	iv
Table of Figures .....	vi
Table of Tables .....	vi
Glossary.....	vii
1 Introduction.....	1
Model Checking Background.....	1
Classic Approach .....	2
Classic Engines, First Variations.....	2
New Engines.....	2
New Approaches.....	2
2 Static Checking Engines.....	6
Introduction .....	6
Classification of Verification Tasks .....	6
Model Checking vs. Bounded Model Checking.....	6
Verification vs. Falsification .....	6
Classification of Static Engines.....	6
Classical .....	7
Discovery .....	7
SmartLoc .....	7
Beelzebub .....	9
Unfolding .....	9
SAT .....	10
FormalSim.....	11
Parallelism.....	12
Summary .....	12
3 Choosing a Search Engine.....	14
Pros and Cons of Different Engines .....	14
Discovery .....	14
SmartLoc .....	14
Beelzebub .....	14
Unfolding .....	15
SAT .....	15
FormalSim.....	15
Choosing an Engine.....	15
Discovery .....	15
SmartLoc .....	16
Beelzebub .....	16
Unfolding .....	16

	SAT .....	16
	FormalSim .....	17
	Using a Tool .....	17
4	Experiments .....	18
	Introduction .....	18
	Experiments with SAT .....	18
	Benchmark Description .....	18
	Statistical Analysis .....	20
	Mage Benchmarks .....	23
	Discovery vs. SAT .....	24
5	Conclusions .....	28
6	References .....	29

## Table of Figures

Figure 1	Discovery .....	7
Figure 2	SmartLoc .....	8
Figure 3	Beelzebub .....	9
Figure 4	Unfolding .....	10
Figure 5	FormalSim .....	12
Figure 6	Coverage vs. model size .....	13
Figure 7	Histogram of binary clauses .....	19
Figure 8	Histogram of the longest clause in the new benchmark .....	20
Figure 9	Clause number/variable number vs. zChaff runtime .....	21
Figure 10	Number of variables vs. zChaff runtime .....	21
Figure 11	Bound vs. standardized zChaff runtime .....	22
Figure 12	SAT runtime vs. Discovery runtime .....	27

## Table of Tables

Table 1	Multiplicity and variance of engines .....	12
Table 2	Statistics of the old benchmarks .....	18
Table 3	Statistics of the new benchmarks .....	19
Table 4	SAT solver behaviour in cases of SAT and UNSAT .....	23
Table 5	Mage runtime vs. old SAT runtime .....	24
Table 6	Discovery runtime vs. SAT runtime .....	26

# Glossary

## **BDD**

Binary Decision Diagram. A data structure upon which some static checking engines are based.

## **Breadth First Search (BFS)**

A search algorithm of a graph which explores all nodes adjacent to the current node before moving on.

## **Block**

A group of interconnected cells. A block may contain instances of other blocks.

## **BMC**

Bounded Model Checking. A method of model checking in which a limited number of cycles is examined. Typically, a bounded model checker can falsify, but not verify, a design.

## **Boolean satisfiability**

The problem of determining whether a CNF formula has a satisfiable assignment, that is, an assignment that evaluates the formula to "true." Also referred to as SAT.

## **Clause**

A disjunction of one or more, negated or non-negated literals, for instance, ( $a$  or not( $b$ ) or  $c$ ).

## **CNF**

Conjunctive Normal Form. A Boolean formula that is a AND of one or more clauses, for instance, ( $clause1$  and  $clause2$ ).

## **Design**

A hardware netlist representing the design phase of a chip.

## **Design under test**

A design checked by a verification tool.

## **Flip-flop**

A digital logic circuit that can be switched back and forth between two states.

## **Formal verification**

A mathematical method for verification, capable in principle of returning either "true" or "false" to a verification problem. In contrast, simulation or testing is an informal method of verification, as it can prove only the existence of a bug, but not the absence of one.

**Gate**

A logic cell which is a functional group of transistors having physical attributes that support a specific semiconductor process technology.

**Logic**

The sequence of functions performed by hardware or software. Hardware logic is made up of circuits that perform an operation. Software logic is the sequence of instructions in a program.

**Model**

A functional representation of a device or system that is delivered in object code format. This software representation contains the basic structure and characteristics of a design object which is used to perform design verification.

**Model Checking (MC)**

A formal verification technique that compares the functionality of a design to a set of user-specified properties or characteristics. Determines whether a set of conditions or properties hold true or are contained within a given implementation of a design. Also referred to as property checking.

**Netlist**

A textual file representing a hardware design as a set of library-specific cells along with their interconnections.

**Property checking**

See model checking.

**PSL**

Property Specification Language, the language for specification of designs upon which PROSYD is based.

**Reduction**

A transformation of a design relative to a property or properties such that the truth value of all properties on the new design is the same as that on the old design.

**SAT**

1. A shorthand for Boolean satisfiability. See Boolean satisfiability.
2. A possible result of a SAT solver. See SAT solver.

**SAT solver**

A procedure that decides Boolean satisfiability problems. A SAT solver returns either "SAT" if a satisfying assignment exists or "UNSAT" if such assignment does not exist.

**Semi-formal verification**

A methodology that combines simulation and testing with formal verification.

**Verification**

The process of falsifying or verifying the functional and performance requirements of a design. Many different kinds of verification tools are in use today, including simulation, formal verification, various types of physical analysis tools, emulation, and rapid prototyping. Most design verification strategies employ many of these approaches to assure the reliability of the final product prior to its manufacture.

**Verify**

To prove that a property holds on a particular design.

**UNSAT**

A possible result of a SAT solver. See SAT solver.



# 1 Introduction

Model checking has long been envisioned as a key method for dramatically enhancing verification coverage. A proposition that is proven by model checking is guaranteed to hold for all possible input sequences satisfying the given input restrictions. At the same time, model checking automatically generate traces showing at least one possible way to violate the given proposition. Hence model checking appears to be extremely appealing for two complementary reasons: it provides exhaustive verification and, if a bug is found, model checking is able to produce, by itself, a trace showing the violation.

However, the infrastructure data type and the kind of analysis performed has long confined the application of model checking to relatively small portions of design, typically with around 200 memory elements (flip-flop) after reduction. This is the case even when model checking is combined with various reduction techniques aimed at keeping the size of the problem small enough to be manageable. The application of model checking has been limited to sensitive portions of the system, which had to be checked almost exhaustively. This required the verifier to manually excise the interesting system portion, and to rewrite an abstract version of the code surrounding the excised portion.

The Background section, below, sketches the basics of "classic" model checking, briefly recalls the new approaches that have appeared in the last decade or so, and provides a scenario in which the application of model checking is extended significantly by integrating three fundamental aspects:

- collaboration
- decomposition
- degree of parallelism

The last aspect, in particular, is seen as the key to successful model checking.

The following chapters describe the characteristics of various engines, whether classical or new, provide guidance about which verification methodology to employ, and present the results of our experiments with the different engines. Other interesting aspects, such as efficient trace reconstruction when properties are found to be violated, are beyond the scope of the present deliverable.

---

## Model Checking Background

The first attempts to apply model checking occurred after the theoretical definition of the approach as an application of the fixed point computation of a given formula on the state space of the underlying transition structure (in the early eighties). These first attempts were essentially based on explicit representations of both the state space and the transition relation.

This restricted the application of model checking to systems with tens of millions of states. This is quite an impressive result, considering that these experiments date

back to the mid-eighties. However, it is not effective when a Boolean encoding of states limits the application to portions of around twenty flip-flops (as is usually the case in a real design). Reduction techniques at our disposal today would have made a difference, but the first real boost came from the introduction of Binary Decision Diagrams (BDD) as a means to represent both sets of states and Boolean functions in a single format. This enabled the "symbolic" application of functions as a set of current-states, which provides the set of next-states (or vice-versa, in pre-image computation).

## Classic Approach

The classic model checking procedure performs a backward, fixed-point computation. Starting from the whole set of states, it reduces, step by step, to the only states that satisfy the given proposition when the fixed point is reached. If that fixed point contains the initial state set, the property is proven. If at least one of the initial states is out of the fixed point, there is a way to violate the proposition.

## Classic Engines, First Variations

The largest drawback of the classic approach is that it cannot give an answer until the fixed-point is reached. In a classic approach, with a BDD-based implementation, the size of the BDD for intermediate computation can easily grow beyond a reasonable size. Therefore, there is a concrete risk of not being able to reach the fixed-point, i.e., to be unable to give a closed answer. The growth of the state space representation beyond manageable size has been imprecisely, but popularly, known as "state explosion".

Many attempts have been made to solve this problem. One method is to first perform a reachability analysis of the state space (starting from the admitted initial states) and subsequently restricting the fixed point analysis to the reachable subset of the state space. One of the first interesting results was the on-the-fly forward reachability variation, which, instead of going backward toward the fixed point, performs a forward analysis. Every given number of steps, it performs a check for property violation by applying the classic approach to the portion of the state space that was added to the reachable subset since the last check. This increases the chances of finding failures before the completion of reachability analysis, and, preferably, before the state explosion.

## New Engines

Several new techniques appeared in the late nineties and early twenty-first century, which started to change the panorama of verification techniques. Some of these were better aimed at proving properties (localization reduction, e.g., SmartLoc) and some were extremely good at finding bugs (guided search e.g., Beelzebub, and Bounded Model Checking by SAT solving). It soon became clear that no single engine could be called the "best verification engine", no matter how improvements would have enhanced any single verification technique. This naturally led to the following stage.

## New Approaches

A dramatic change, both in the size of treatable objects and in verification performance, occurred with the application of three new exploration paradigms:

- Collaborative engines
- A/G reasoning

- Distributive verification

## Collaborative Engines

The basic idea of this paradigm is that different engines are better at different state space exploration stages. Combining them, switching from one to another during the exploration phase, and sharing information among them, makes it faster to find failures and to prove properties. The ability to avoid spending time in irrelevant regions of the state space was a huge leap.

Collaboration poses some fundamental questions: when to switch between engines, in which order to schedule them. However, the idea of being able to exploit information produced by different engines to move forward in the verification process has demonstrated itself to be striking.

## A/G Reasoning

Assume/guarantee (A/G) reasoning is a compositional approach that allows the decomposition of a verification problem into a set of smaller verification problems. The verification of a single system property becomes the result of the conjunction of the verification of all the items of a property graph.

In verifying each design portion, some properties are used as *assertions*, i.e., they have to be proven to hold. Other properties are used as *assumptions*, i.e., they provide a set of conditions that the verification process can take as proven, and exploit their truth status to infer the truth of assertions. Assumptions about internal design elements will become assertions in their turn at some stage of the verification process. These will then be proven using some other assumption set. The design topology and property locations determine the property graph on which A/G reasoning is applied.

This compositional approach works especially well when the design to be verified is interspersed with a large number of properties (each of which is expected to be true). A/G reasoning has been shown to be extremely effective at proving properties. However, by its nature, it is less effective in cases where properties are going to fail, as the verification failure of even just a single element in the property graph disrupts the whole verification process.

Other practical aspects, such as the computational complexity of assumption composition induced by circuit topology, or circuit decomposition restrictions induced by property texts (see PROSYD D3.1/1), weaken the applicability of this approach to a system as a whole. Nevertheless, the A/G reasoning approach preserves its general value as a simplifier of the verification of system portions, enabling the splitting of the circuit into pieces that can be verified using different techniques.

## Distributive Verification

Rather than choosing one specific engine, the verification strategy that has proven to be the most effective is that of parallel verification, in which the same problem is attacked from different angles, by using different engines simultaneously. The decision about the engine best suited for the verification of a given property on a given design is made only after the verification has been performed. Generally, it is not possible to say whether a property will pass or fail, or which engines will work better, on the basis of observation alone.

In an ideal context with unlimited computation resources, simply trying all possible engines in parallel, would correspond to always making the right engine selection

first. This would ensure, by construction, that the verification is completed in the shortest possible time—an advantage for all those approaches meant to use a single engine at a time.

As soon as distributive verification is turned into a viable option, it is only a question of choosing how much parallelism the verification requires. There are two fundamental kinds of parallelism, coarse and fine, which act on very different aspects of the verification challenge.

This naive and simple paradigm has been shown to be the most effective. It has reduced verification times from days to minutes (when compared with previous versions of the same tool with no parallelism), in many cases in which a close answer (sure passing, or a certain failure) was available. Although naive in its nature, the effectiveness of this verification strategy depends on at least two ingredients:

- Computation resources have to be available to allow engines spawning at the required degree of parallelism
- Each engine has to be efficient in performing its task

Both ingredients are relevant. A lack of resources entails scheduling choices that can have a dramatic impact on verification time. At the same time, a low quality engine is unlikely to find a solution more effectively simply as a consequence of a large degree of parallelism.

## The Future

None of these approaches is a sure-winner. For all of them, the main issue is the balance between automatic intervention and user intervention. There are good reasons for both orientations. Automatic orchestration is unbeatable for collaborative situations in which the tool can identify the best conditions for exploiting an engine switch. However, there are verification cases that appear to be perfect for a specific engine, and the tool might not easily derive this. In such cases, direct user control can make a big difference in deciding which engine to try first. A/G reasoning, on its turn, can suffer either from an excess of property presence or a lack of properties. Again, the choice between an automatic approach to problem decomposition and full user control is not an easy one.

The natural evolution of this diverse approach is a kind of medley, in which a decomposition mechanism downsizes the verification challenge, and applies a collaborative approach to the verification of each single piece, in parallel. At the same time, other parallel runs try to attack the verification problem using different decomposition schemes.

The currently available last-generation tools take very little advantage of previous verification activities. Some kind of high-level adaptive heuristics could be tried, taking into account the location and nature of the differences between the description verified previously and the description to be verified now.

We envision the performance of verification in years to come as involving:

- more engines
- more collaboration
- better ways to perform problem decomposition
- all combined in a framework that empowers formal verification as much as possible<sup>7</sup>

- seamlessly integrating it with more traditional dynamic verification (i.e., simulation)
- focusing the formal verification activities on those design portions that have the lowest coverage figures or that have structural characteristics that make them well suited for static analysis
- acting in parallel with simulations locally constrained by the same properties utilized by the formal engines

# 2 Static Checking Engines

---

## Introduction

This chapter starts with the analysis of different engines utilized in a parallel context. A verification task varies according to the analysis purpose (verification or falsification), the desired coverage (bounded search, guided search or full state-space exploration), the design size and diameter, and the estimated depth of the "bad" states. The chapter describes the methods and capabilities of static engines such that the user can choose the appropriate set of engines that suits best the verification task.

---

## Classification of Verification Tasks

### Model Checking vs. Bounded Model Checking

The traditional methods of model checking (MC) [12], in which given properties are checked on all legal paths of a given design, can not cope with modern large-scale designs. As formal verification becomes more common [1][3], the "state explosion" problem inherent in BDD-based model-checking engines becomes more important to solve. Bounded model checking (BMC) [6] engines such as Boolean Satisfiability (SAT) [14][24] can check larger design as they evaluate the given properties on truncated paths of the design.

Unlike the model checking problem that, given a model  $M$  and a property  $\phi$ , tries to determine if  $M \models \phi$ , the BMC methodology restricts the verification problem to determining whether  $M \models \phi$  on the first  $k$  iterations of  $M$ . The class of properties that can be checked this way is smaller than the one handled by model checking.

### Verification vs. Falsification

The "bug hunting" methodology is aimed at showing that given properties does not hold in the design under test. An example of this is the "as long as effective" methodology, which does not focus on verifying the design under test, but rather on maximizing the bug finding rate over a designated period of time and then letting simulation take it from there. A different methodology is to pursue actual verification of the design-under-test at hand, thus trying to prove the block works correctly.

Some engines are better for proving assertions while others are better for falsifying assertions. Some engines can only falsify assertions

---

## Classification of Static Engines

This section describes different static checking engines. The engine names refer to the engine names of RuleBase Parallel Edition.

## Classical

The Classical engine is an SMV-based engine [20] that checks whether finite-state systems satisfy specifications written in the temporal logic CTL [10]. The system described to the tool can be either synchronous or asynchronous, detailed or abstract. A variety of temporal properties such as safety, liveness, and fairness may be described in CTL.

Classical uses BDD-based [8] symbolic model-checking [9][21] algorithms to efficiently evaluate assertions. The engine computes which states satisfy an assertion by recursively employing fix-point algorithm. The engine performs pre-image calculations (backward steps). It then checks whether initial states of the system satisfy an assertion.

The BDD data structure represents the entire set of reachable states. Hence, the bottleneck of Classical and other BDD-based engines is typically the memory requirements that may grow exponentially.

## Discovery

Discovery, like Classical, uses BDD-based algorithms to prove that CTL properties hold on a system. The main difference between them is that Discovery performs a reachability analysis of the system before evaluating properties with the Classical algorithm. Discovery uses a BFS search from initial states until a fixed point is reached. During reachability analysis, Discovery starts with the initial states and steps "forward" to the next states until all the reachable states are discovered. The main advantage of the reachability analysis is that it checks safety properties "on the fly" [4].

Another advantage is that with knowledge of reachable states, the model may be significantly simplified, which speeds up the further run of the Classical algorithm. Therefore, the reachability analysis runs even if there are no safety properties to evaluate.

Moreover, it is possible to check liveness properties during reachability ("liveness on the fly"). Liveness properties are checked by the Classical algorithm, but they may be falsified on the partial state space, explored so far during reachability. This may be faster than checking liveness properties on the full reachable state space after reachability analysis.

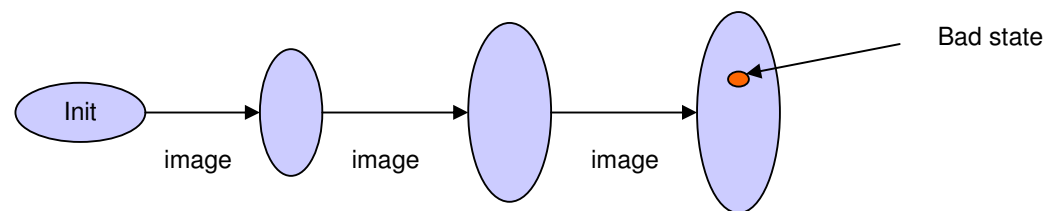


Figure 1 Discovery

## SmartLoc

SmartLoc is an abstraction-refinement engine [11]. The goal of SmartLoc is to evaluate properties over large designs, causing Discovery to explode in memory. SmartLoc tries to save space at a price of time, by successively running the Discovery algorithm on a series of abstractions of the real model. It calculates an

abstraction by releasing the behaviour of a subset of the signals in the model, that is, making them free signals. In the first abstraction, only the signals that are close to those used in the evaluated property keep their original behaviour, and the rest of the signals are made free. Each subsequent abstraction refines the previous one by means of giving more signals their original behaviour, thus making a model more precise.

Given an abstraction, SmartLoc evaluates the property over the abstract model. Safety properties may be either evaluated on the fly during a reachability analysis or by the Classical algorithm. For non-safety properties SmartLoc may use the Classical algorithm only.

If the property passes, it is deduced to be true in the concrete model, which finishes the engine run. If a property fails, an abstract counterexample is built over the abstract model. However, an abstract counterexample does not necessarily reflect a real bug. In order to determine whether the counterexample is a real or a spurious one, SmartLoc tries to reconstruct values for the free signals, that is, those signals that are not included in the current abstraction. Those signal values must be consistent with both the transition relation and the counterexample found.

If SmartLoc succeeds in reconstructing values for all the signals in the model all along the counterexample trace, the property is deduced to be false, and the counterexample is presented. Otherwise, the counterexample is spurious and SmartLoc refines the abstraction by adding more signals to the set of signals keeping their original behaviour.

The SmartLoc algorithm is implemented using BDDs but it can be used in conjunction with SAT or ATPG-based implementations.

SmartLoc can evaluate both safety and liveness assertions that can be falsified by a single trace. That is, SmartLoc can evaluate assertions whose counterexample can be presented in a single trace. SmartLoc is used in both falsification and verification. However, it is better for verification rather than falsification, because if a property fails, reconstruction of the free signals may be computationally difficult.

Unlike Classical and Discovery, SmartLoc can only evaluate one assertion at a time.

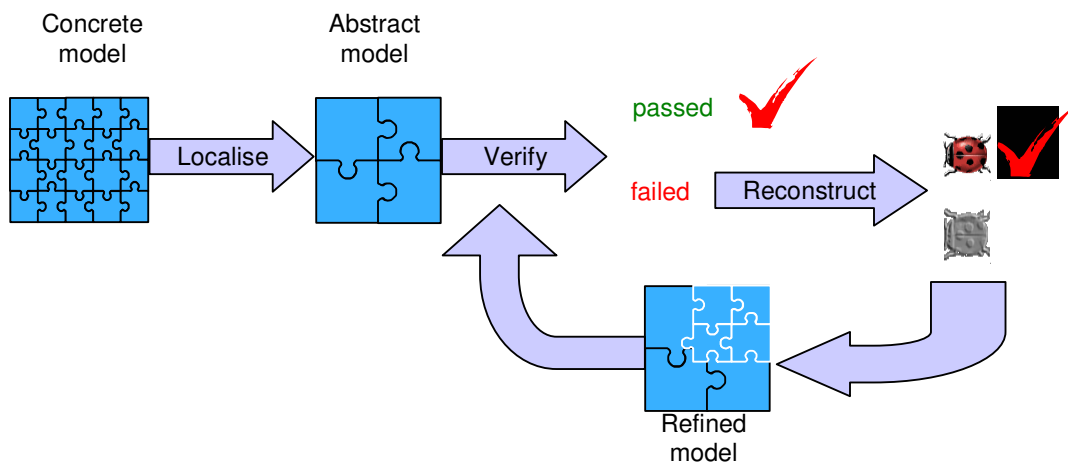


Figure 2 SmartLoc

## Beelzebub

Beelzebub is a guided-search [17][18] verification engine. Like Discovery, Beelzebub is a BDD-based engine, which can be used for both falsifying properties and proving them. Unlike Discovery, Beelzebub adopts the methodology of guided-prioritised search, rather than full state-space exploration. Thus, it is usually able to maintain compact BDDs and to avoid memory explosion.

Beelzebub's search is performed by taking a few approximated backward steps from the set of buggy states. It then uses these steps to direct an exact forward partial search from the set of initial states, until an exact buggy path is found. If no such path is found, it takes more approximated backward steps, and uses them to better direct the next forward search.

Beelzebub can only check safety properties and it can only run one assertion at a time.

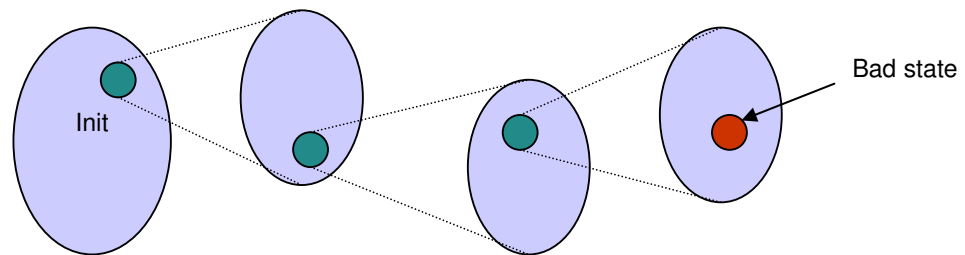


Figure 3 Beelzebub

## Unfolding

Unfolding is a symbolic, BDD-based bounded model-checking parallel engine, based on the unfolded structure [25][5]. It proves safety properties bounded to the first  $k$  cycles, using BDD-based procedures.

The engine unfolds the circuit into  $k$  combinational circuits, one for each cycle. The flip-flops of the original design become wires in the Unfolding internal representation—flip-flop inputs of cycle  $i$  are connected to the flip-flops outputs of cycle  $i+1$ . Non-deterministic primary inputs of the original design are the variables of the Unfolding model.

Unfolding eliminates false resource sharing along time. For example, if a flip-flop performs some task in even cycles and another task in odd cycles, Unfolding will have a different function for each task, while Discovery will have one big function that represents all of its tasks.

Unfolding is unable, in most cases, to prove the correctness of assertions for all cycles. Unfolding checks safety formulas only.

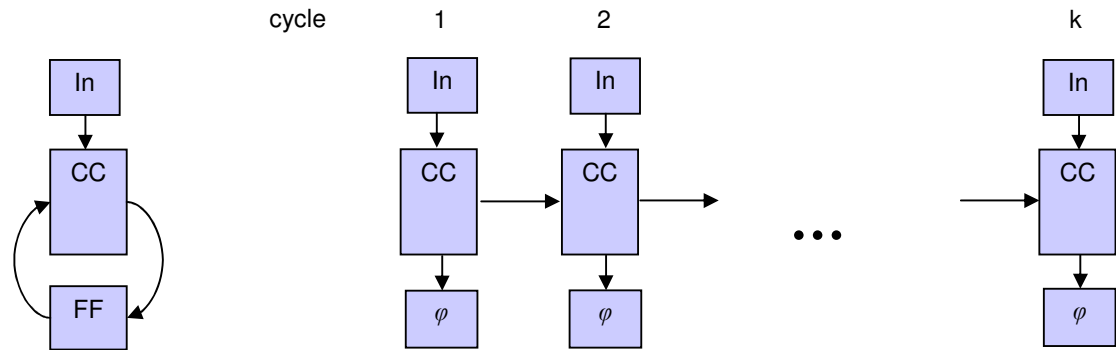


Figure 4 Unfolding

## SAT

SAT is a bounded model-checking engine. SAT checks a finite range of cycles in which to search for the bug. Since, typically, the larger the range, the longer the search process takes, bounded model checking with SAT is normally conducted in a gradual manner.

First, SAT searches for bugs starting from the initial state up to some user-defined bound  $k$ . If no bug is found, the next problem instance will be from  $k+1$  to  $k+(\text{interval size})$ , and so forth. There are three possible terminations of this process:

- A bug is found. In this case, SAT presents a counterexample to the user.
- The problem becomes too difficult to solve in a reasonable amount of time. This means that the property was not proven to hold globally. The only guarantee in such cases is that the property holds up to the last cycle that SAT was able to prove. For example, if SAT proved that there is no bug in the range 0..20, and then it timed-out while attempting to prove that there is no bug from cycle 21 to 30, then the only guarantee you have is that there is no bug up to cycle 20. In this case, you may wish to try to look at cycles 21.. 25, which is an easier problem.
- No bug is found up to the diameter  $D$  of the design (in this context "the design" is the combination of the hardware design and its specification). Assume that the shortest path from an initial state to each of the reachable states of the system is calculated, then the diameter is defined to be the longest of these paths. Since hardware designs have a finite number of reachable states, it is guaranteed that such a finite diameter indeed exists (finding the diameter is a hard problem by itself). Thus, if there is no bug up to cycle  $D$ , it means that there is no bug at all, and the property is verified. Note that before  $D$  is reached, there is no guarantee that the property holds globally.

SAT builds a Boolean formula, which is logically satisfiable if and only if there is a bug in the given range. Each signal in the design, in each cycle, is represented by a different variable in this formula. As a second step, it tries to satisfy the formula, i.e., to find a single assignment that evaluates to *true*. If it finds one, then this is the counterexample. The satisfiability problem also requires exponential time to solve, but there are no exponential memory requirements. Thus, a fast CPU is the key for obtaining fast results with SAT rather than a large memory. The size of the formula that is generated in the first step has a great effect on SAT's speed. The formula grows linearly with the distance from the initial state. Thus, the distance from the

initial state, rather than the range itself, has the greatest effect on the difficulty of the problem. For example, searching for cycles between 20 and 30 takes far more time than searching for cycles between 10 and 20.

SAT is commonly used for "bug hunting", especially if the bugs can be reached in early cycles. Proving that no bug exists (verification) is much harder for SAT, because this requires, reaching the diameter of the design.

SAT has an incremental mode, in which information gathered in one interval is passed to the next. This increases the capability of the engine, enabling it to explore deeper cycles.

SAT can only check safety properties and it can only check one assertion at a time.

## **Mage**

Mage is a SAT solver that is being developed at IBM Haifa Laboratory. Mage is an enhanced version of zChaff [22]. Some enhancements are based on known techniques [23]. The Mage solver is accompanied by a generator of CNF formulas that employs incremental techniques. The Mage solver and the CNF generator cooperate in a new SAT engine that reaches deeper and faster into the design under test. In this methodology document Mage refers to both the solver and the engine.

## **FormalSim**

FormalSim is a bug-oriented explicit model checker, which represents states as bit-vectors. Alternately, FormalSim is a guided simulator with memory for all visited states (to prevent re-visit) and for promising states (for back-tracking). FormalSim provides guidance using estimated distance to the target state.

FormalSim is intended for finding delicate bugs in big hardware blocks. It is a fail-oriented explicit model checker, which evaluates reachable system states one by one. FormalSim uses for its search heuristics techniques that originate from both simulation-based methods and formal methods. Such engines are called "semi-formal".

By performing a partial search of the state-space, FormalSim can cope with hardware blocks with tens of thousands of flip-flops (after reductions), which cannot be checked by current formal-verification engines. Yet the search is guided well enough to cover most interesting parts of the state-space. Combining this with the ability to backtrack dead-end paths, this engine also outperforms simulation-based methods in checking for a specific bug.

FormalSim uses a variety of heuristics to assess the priority of each state, which is its probable distance to a failure state. It also allows user guidance using the guide construct. FormalSim makes an attempt to keep counterexamples short and it keeps track of paths leading to top-priority states

FormalSim checks safety properties only.

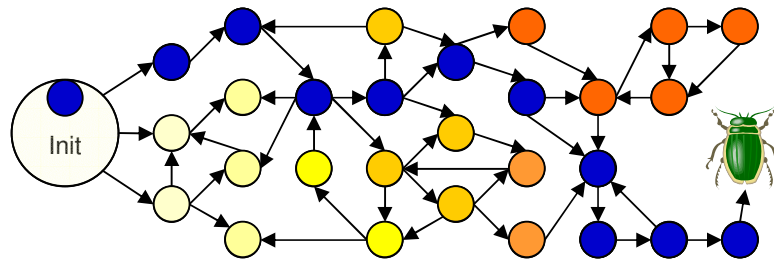


Figure 5 FormalSim

## Parallelism

Checking engine may run in a parallel platform. In **coarse-grain parallelism** several checking engines check simultaneously a given property. The engine share runtime information, e.g., no bug is found until a certain cycle. In **fine-grain parallelism** engines can request “slaves”: additional processes for distributed tasks. This method of parallelism distributes memory consumption among various machines, reduces the total computation time and handles larger verification tasks. Fine-grain parallelism also enables decomposition of a large verification task into smaller, tractable subtasks, which can be run simultaneously on different machines and reducing both runtime and memory consumption.

## Summary

Static checking engines differ in the verification method and capabilities. Table 1 summarizes the main engines' characteristics. Figure 6 shows the trade off between the size of the design under test and coverage. The next chapter draws guidelines for choosing one or more checking engines for a given verification task.

Engine	Allowed properties	Verification/ falsification	MC / BMC	Single/multiple assertions
Classical	All PSL	Both	MC	Multiple
Discovery	All PSL	Both	MC	Multiple
SmartLoc	W-ACTL	Better for verification	MC	Single
Beelzebub	Safety	Better for falsification	MC	Single
Unfolding	Safety	Falsification only	BMC	Multiple
SAT	Safety	Falsification only	BMC	Single
FormalSim	Safety	Falsification only	MC	Multiple

Table 1 Multiplicity and variance of engines

## Coverage vs. Model Size

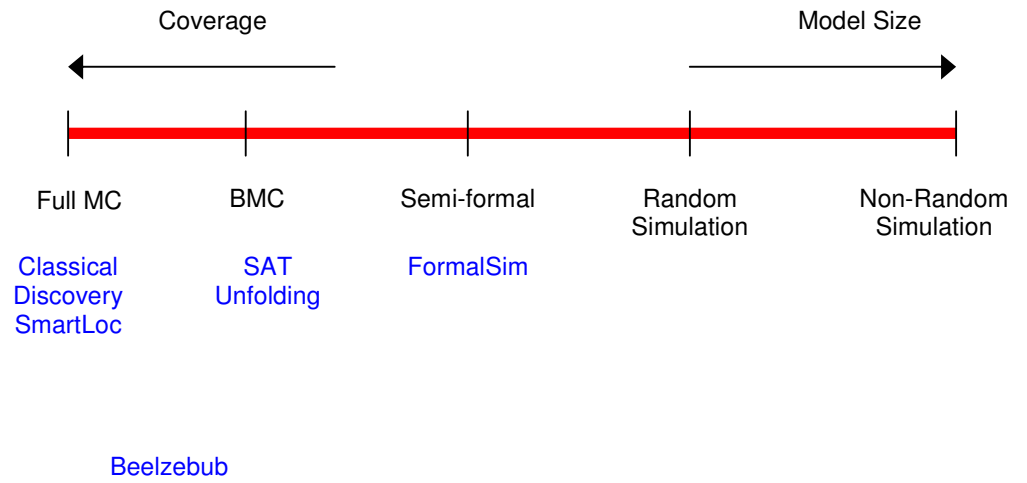


Figure 6 Coverage vs. model size

# 3 Choosing a Search Engine

Choosing which search engine to use in order to solve a specific problem is a difficult question. Some search engines are "proof oriented" and thus, better able to prove a property true, and some are mainly falsifiers, and better at finding bugs. This chapter provides some guidelines about the best way to leverage the different static checking engines.

---

## Pros and Cons of Different Engines

There are advantages and disadvantages to the different engines.

### Discovery

Discovery is a complete engine: if given sufficient amount of time and memory, it will determine if a PSL formula holds regarding a model. However, there are many cases where BDD explosion will prevent the solution of the problem within a reasonable amount of time and memory.

Independent of its performance, Discovery has two unique features:

- An expert user can sometime predict from the log if a specific Discovery run will run to completion (using the fact that in most cases, the graph number of BDD nodes vs. cycles will have an asymptotic behaviour).
- It can calculate the "depth" of a design (how many cycles needed to perform reachability analysis), which is very useful. This gives hints about which engines to use and reveals the number of cycles needed to run SAT in order to prove a property true. However, Discovery's performance is very sensitive to BDD order. So it is usually necessary to monitor the Discovery run and eventually restart with a better BDD order. Discovery's performance is also very sensitive to its numerous options.

### SmartLoc

SmartLoc's main use is to prove a property true regarding a model. SmartLoc tends to be moderately robust.

### Beelzebub

Beelzebub is an incomplete engine, used mainly for falsification. It is particularly powerful at finding 'deep' bugs. This is due to its ability to limit the forward search to promising directions. Nevertheless, it is quite common to see Beelzebub outperform all other engines in finding 'shallow' bugs, or even in proving properties. Also, when the assertion describes a series of events, Beelzebub can use these events as hints, and limit its search even more.

## Unfolding

Unfolding is a Bounded Model Checking engine. It is complete but only up to a certain bound; thus, it is mainly used as a falsifier.

Unfolding transforms state variables to wires and non-deterministic inputs to BDD variables. Hence, it is not sensitive to the number of flip-flops in the design, and has an advantage in complex and shallow designs with many flip-flops and few inputs.

## SAT

SAT is another Bounded Model Checking engine and, as Unfolding, it is mainly used as a falsifier.

SAT is able to tackle far bigger models than Discovery. Recently-released Mage enables an exploration with bigger bounds. One of the strong points of SAT is the very small time needed to generate a counterexample.

## FormalSim

FormalSim is an incomplete engine, used for falsification. Its main strength is its ability to tackle very large models.

---

# Choosing an Engine

Unfortunately, it is very difficult to predict which engine under which configuration will be the most efficient for a specific model without actually running the various engines. Experience tends to prove that if an engine behaves well for a specific rule in a design, it is likely that this engine will also perform well for other rules on the same design.

It is necessary to experiment with different engines under different configurations in order to see which will perform the best.

The following section briefly describes the different configurations that may be tried for different engines

## Discovery

Discovery appears to be the engine that needs the most experimentation with the various options.

The speed and efficiency of the different reduction techniques can vary according to the design. Experimentation is the only way to find out which reduction level is optimal for a specific design. For example, a high reduction level may be very time consuming and inefficient; however, for some designs, this reduction can cut the size of the model by two.

Discovery performances are very sensitive to BDD order. Therefore, the majority of the time is spent on killing and re-running the property verification in order to obtain a good order. The choice of reordering algorithm is important. The two most popular choices are Cheetah + light Rudell and Cheetah + Rudell. If you are still investing time in getting a good order, the second option is recommended. However, later on, you should revert to the first one, unless you are facing BDD

explosion. Note that a good order for Discovery is not necessarily good for the other BDD-based engines.

The "saving donuts" interval can have a significant influence on performance. Disabling it or to setting it to 10 will ease the task of reachability analysis; however, this will eventually make counterexample generation much more time consuming. It may be good idea to start without saving donuts and if the property fails, re-run with another configuration.

Light proof can sometimes enable substantial performance gain, although, if it does not work, it wastes time. However, if your properties are likely to be too long to be solved, you don't risk a lot by investing 30 seconds on light proof.

## SmartLoc

In addition to the options available in Discovery (reduction, BDD reordering and light proof), SmartLoc has two alternative refinement algorithms:

- Counter-example based algorithm adds a set of signals which cause spurious counterexamples to disappear [19]. This refinement algorithm suits the "bug hunting" methodology and it is commonly used.
- Cone-of-influence based algorithm adds signals, that influence signals that already exist in the model. This refinement algorithm is "pass oriented" in a sense that a property is deduced to fail only if it fails on the concrete model, i.e., when all the signals are given their original behaviour.

## Beelzebub

Beelzebub has the same reduction and BDD reordering as Discovery. Other options (such as "skip target enlargement") may be used in specific cases (for example, if the target enlargement step is too long).

## Unfolding

Unfolding appears to be the engine with the most complex options and configurations. The memory option should be set according to the hardware configuration. At least two set of options should be tried: one with reordering and full search and second without reordering and with partial search.

## SAT

The SAT default options usually give a good enough indication of SAT performance on a specific design. SAT can only prove a property true up to a certain bound. Of course, if this bound is greater than the diameter of the design, then the property is proven. Unfortunately, there is no easy way to identify this value for a specific property and a specific design. The incremental algorithm of the new engine—Mage is usually faster than the non-incremental version. However the non-incremental version is especially useful for checking deep models.

Empirical studies have shown that many designs that can not be verified with the standard Discovery engine can be efficiently solved with SAT, and vice-versa [6][7]. Hence, the methodology that combines these two static engines is very productive. SAT is usually better at finding "shallow" bugs while Discovery better proves properties and finds "deep" bugs.

## FormalSim

You should first find out the level of compiler optimisation that you need for your design. The higher the compiler optimisation level, the better FormalSim will perform in the later stages. However, highly optimised compilation can be very slow. The main option for a FormalSim run is the exploration-step depth. If you have an idea about how deep the bugs can hide, you can define the depth accordingly. If not, it is a safe policy to start with a small depth and to increase it if necessary.

---

## Using a Tool

You can perform these experimentations in a systematic way by using a tool. Such a tool would run different options and engines on several CPUs (for example, overnight or over the weekend) for a few significant rules. It would then extract a statistic ranking for the various cases. You could use these statistics as guidelines in order to select the most relevant engines with the best configuration.

It is important to realise that FormalSim, Unfolding, and Beelzebub are almost ineffective when the properties hold against the model and that SmartLoc is mostly ineffective for falsification (SAT can prove property up to some bound). So, if you have an idea about the status of the property, you can give the different engines different priorities.

# 4 Experiments

---

## Introduction

We made several experiments in order to try to find correlations which could help us to predict statically which engine would perform the best on a specific model.

---

## Experiments with SAT

Since benchmarking is easier with SAT engines than with BDD-based engines, we experimented with SAT solvers. We experimented on the IBM 2004 CNF benchmark (available for academic users on [http://www.haifa.il.ibm.com/projects/verification/RB\\_Homepage/benchmarks.html](http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html)).

### Benchmark Description

Bounded Model Checking translation technology is continuously evolving. Therefore, we re-generated the IBM CNF Benchmark from the IBM Model Benchmark using a state-of-the-art BMC tool (benchmarks used for Deliverable 3.2/1 were generated with an older BMC tool). We ran the new translation for the following bounds  $k=0..10$ ,  $k=11..15$ ,  $k=16..20$ ,  $k=21..25$ , ...,  $k=95..100$ .

We will refer to the 2002 and 2004 versions as the old and the new benchmarks, respectively (so the benchmark used for Deliverable 3.2/1 is the old benchmark).

The two next tables present statistical descriptions of, respectively, the old and the new benchmarks.

	Var	clauses	clause/ var	unit (%)	bin (%)	ter (%)	l=4 (%)	l>4 (%)
<b>Average</b>	80,167	343,826	4.12	0.3	75.5	14.2	3.5	6.5
<b>Median</b>	54,857	220,180	4.08	0.2	77.0	12.2	3.5	6.6
<b>STDEV</b>	81,924	388,156	0.52	0.3	6.4	7.7	1.0	1.3
<b>Max</b>	636,089	3,172,107	5.42	1.6	84.5	55.3	5.2	9.1
<b>Min</b>	3,645	14,681	2.48	0.0	40.8	4.3	0.1	0.1

Table 2 Statistics of the old benchmarks

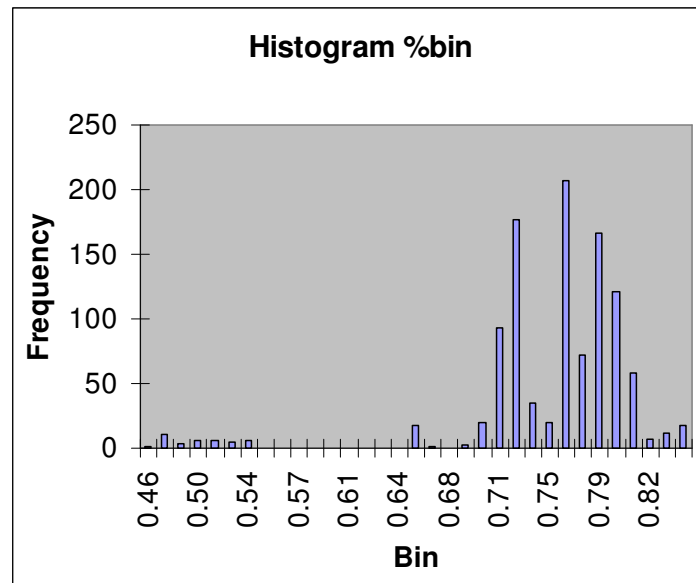
	var	clauses	clauses/ var	unit (%)	bin (%)	ter (%)	l=4 (%)	l>4 (%)
<b>Average</b>	73,414	305,301	3.99	0.6	74.5	15.0	3.6	6.3
<b>Median</b>	50,897	195,612	3.98	0.4	76.1	13.1	3.6	6.6
<b>STDEV</b>	75,553	349,248	0.45	0.6	6.1	7.3	0.9	1.2
<b>Max</b>	565,889	2,760,502	5.48	4.6	84.5	5.16	4.9	9.1
<b>Min</b>	3,606	14,104	2.55	0.0	46.5	4.4	0.1	0.1

**Table 3 Statistics of the new benchmarks**

In these two tables, **average** is the arithmetical mean, **STDEV** is the standard deviation, **clauses/var** is the ratio between the number of clauses and the number of variables, **%unit** is the percentage of a unit (i.e., of length one) of clauses in a CNF, **%bin** is the percentage of clauses of length two, **%ter** is the percentage of clauses of length three, **%l=4** is the percentage of clauses of length 4 and **%l>4** is the percentage of clauses longer than 4.

These two tables tell us that the new benchmark CNFs are about 10% smaller than the old benchmark CNFs. However, they encompass roughly the same proportion of two, three, and four length clauses. The higher proportion of unit clauses in the new benchmark is due to specific optimisations. We also see that the **clauses/var** ratio is slightly lower in the new benchmark (see *Statistical Analysis* on page 20 for a discussion about the relevance of this ratio).

Figure 7 displays the histogram of **%bin**, where two models (07 and 13\_1) have a "non standard" percentage of binary clauses and are in the 45%-55% range, rather than in the 70%-85% range.

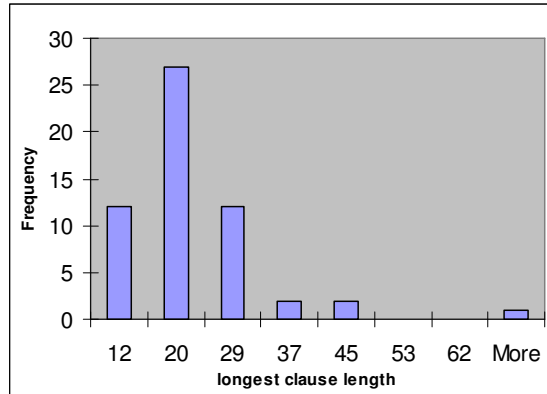


**Figure 7 Histogram of binary clauses**

We looked at some statistics for the structure of the CNFs of the new benchmark. We found that for any model of the benchmark, there are four real numbers  $a$ ,  $b$ ,  $c$ ,  $d$  such that for all the CNFs of the model,  $\#(variables) \sim ak + b$  and  $\#(clauses) \sim$

$ck+d$ , with  $k$  being the bound used to generate the CNFs. In a more general way, for any model and for any integer  $n$  strictly greater than 0, there are two real numbers  $e$  and  $f$  such that for all CNFs of the model  $\#(\text{clauses of length } n) \sim ek+f$ . Of course,  $e$  and  $f$  can be null, for example, in most models, the number of unit clauses is a constant.

The correlations between the series from the benchmark and the series predicted with the previous equations are about 0.99. This is not surprising since the CNFs are generated from the model in a way that is essentially linear to the bound. In addition, we found out that, in our benchmark, the length of the longest clause is the same for all CNFs generated from the same model. Figure 8 displays the histogram of the longest clause in the new benchmark.



**Figure 8 Histogram of the longest clause in the new benchmark**

Note that two models, 07 and 13\_1, which have the longest clauses, are also those with a “non standard” percentage of binary clauses.

## Statistical Analysis

In this section, we try to find statistical metrics to help us to predict if a CNF will be hard to solve with a SAT solver and therefore if the SAT engine will have a hard or easy job with a specific model.

### Can the "hardness" of a CNF be predicted from the value of the ratio (number of clauses) / (number of variables)?

We could not find any relevant correlation between this ratio and the zChaff I and Siege performances. In fact, for each model,  $\#clauses$  and  $\#vars$  are strongly correlated with the bound  $k$ . Therefore, for each model, for a given bound  $k$ , there are  $a, b, c, d$  four real numbers, such that, the ratio  $\#clauses/\#vars$  is also strongly correlated with  $(ak+b)/(ck+d)$ . This explains the appearance of Figure 9:

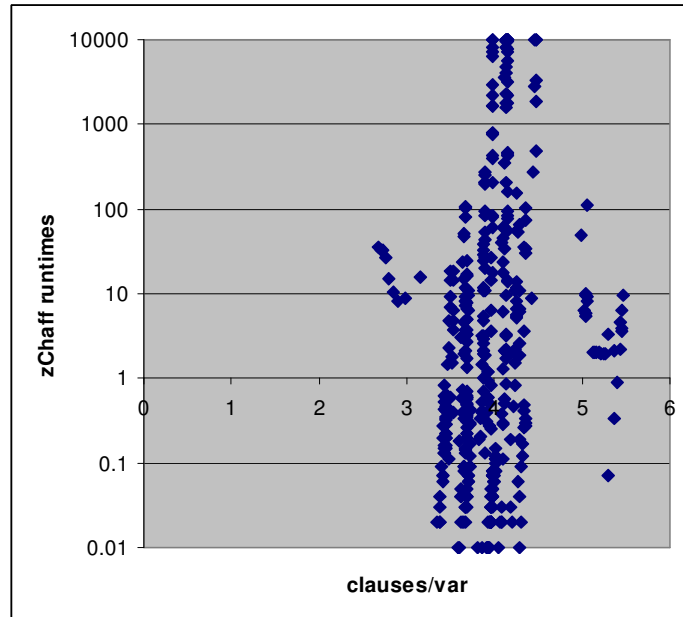


Figure 9 Clause number/variable number vs. zChaff runtime

**Can we predict the "hardness" of a CNF from the number of variables?**

This could also be the number of clauses, since both numbers are strongly correlated. In our experiments, we found a relatively low correlation (about 0.3) between the number of variables and the runtimes for zChaff I and Siege. See for illustration:

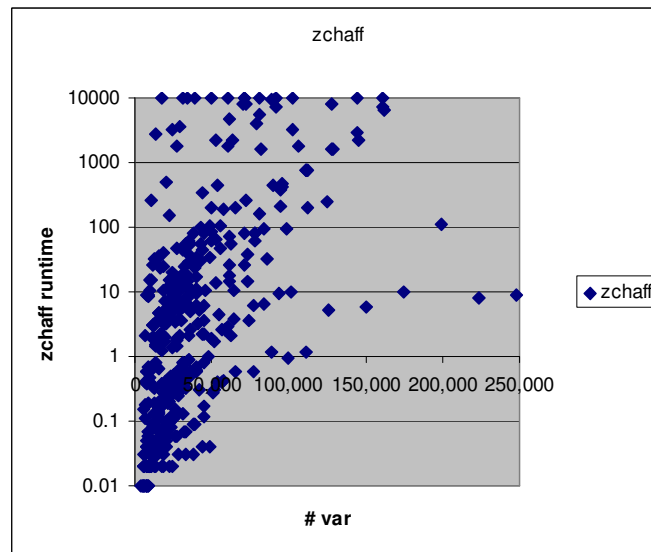


Figure 10 Number of variables vs. zChaff runtime

However, when we "standardised"<sup>1</sup>, we got a higher correlation (about 0.7). See Figure 11 for illustration:

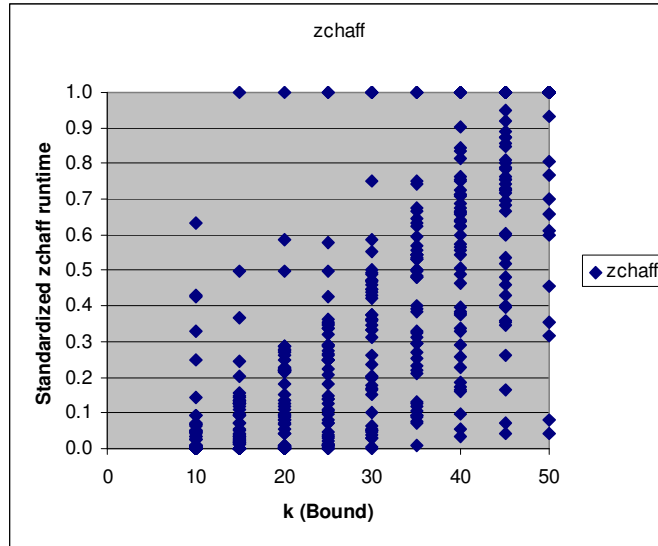


Figure 11 Bound vs. standardized zChaff runtime

**Can we predict the "hardness" of a CNF from the proportion of clauses of length one, two, three, and four?**

We did not find any relevant correlation between the proportion of clauses of length two, three, four, and more and the zChaff I and Siege runtimes. We did find a low negative correlation (about -0.4) between the standardised proportion of unit clauses and the runtimes. We think this can be explained by the fact that in most cases, for a given model, the number of unit clauses is constant, so, if we let the bound  $k$  to grow, the ratio of the unit clauses is proportional to  $\frac{c}{ak+b}$ , which explains the correlation.

**Are "SAT" and "UNSAT" CNFs as hard?**

This is difficult to say, since it is impossible to compare performance on the same CNFs. It is not relevant to compare the geometrical mean of the runtime on the SAT and on the UNSAT CNFs, as displayed in Table 4. However, it is easier to compare the behaviour of two SAT solvers on SAT and UNSAT CNFs. Table 4 shows how Siege behaves slower than zChaff I on SAT CNFs as opposed to UNSAT CNFs.

---

<sup>1</sup> For a given SAT solver, we standardised the runtimes in the following way. For each model, we divided each of the nine runtimes (corresponding to the nine bounds  $k=0..10, k=11..15, k=16..20, \dots, k=46..50$ ) by the greatest of these nine runtimes. The idea is to be able to compare results between different models.

		zChaff	Siege	Speedup
<b>All CNFs</b>	Median	30	22	2.28
	Geomean	56.52	31.80	1.80
	Total	362,980	44,632	8.13
	STDEV	10,154	837	18.76
<b>Unsat</b> (109 CNFs)	Median	18	12	2.46
	Geomean	41	20	2.03
	Total	247,777	14,642	16.92
	STDEV	12,662	296	12.31
<b>Sat</b> (67 CNFs)	median	55	59	1.84
	geomean	98	66	1.48
	total	115,203	29,989	3.84
	STDEV	3,134	1,279	26.03

Table 4 SAT solver behaviour in cases of SAT and UNSAT

## Mage Benchmarks

We performed a regression test to compare performances of two different SAT engine versions: the old non-incremental one and the new engine—Mage. Results have been collected from a set of seven different industrial test cases, on each of which a restricted but significant (for complexity, size, reduction effectiveness measurement reasons) group of properties was selected and checked. Both runs, new and old, had a timeout of one hour per property. For each test case we show the number of gates, the number of flip-flops before reduction, and the numbers of assertions that pass, fail, and ended with a bounded proof. The number of cycles is the grand total of bounded-proof depths. The time is the grand total of the time spent in computation of bounded proofs. Numbers with asterisks reflect result updates, that is, promotion from bounded proof to either pass or fail, following the improvement between old SAT and Mage. Unfortunately it hasn't been possible to reproduce the verification under exactly the same operating conditions: the old runs were executed on dedicated machines, whereas Mage runs were executed on machines shared with other users. This has to be taken into account when comparing results, as it suggest that the difference between old and new could have been even wider, if favor of the latter. The most-left column shows the ratio between cycles and times has been added (the ratio was put in parentheses when just times were compared). This ratio provides an immediate perception of Mage effectiveness

Table 5 Mage runtime vs. old SAT shows the comparison results:

TC	KGates/ FF	Properties (p/f/b)	Old SAT			Mage			MageCT/ OldCT
			Cycles	Time	C/T	Cycles	Time	C/T	
1	14/629	12*/4/48	1270	23:45:09	.015	<b>5380</b>	<b>22:29:28</b>	<b>.070</b>	4.70
2	21/275	3/1*/12	1975	05:21:59	.100	<b>6500</b>	<b>00:24:32</b>	<b>4.42</b>	44.20

TC	KGates/ FF	Properties (p/f/b)	Old SAT			Mage			MageCT/ OldCT
			Cycles	Time	C/T	Cycles	Time	C/T	
3	9/129	22/0/4	55	<b>02:07:06</b>	.001	<b>620</b>	03:53:14	<b>.040</b>	40.00
4	42/602	0/4/8	33	<b>00:52:39</b>	.010	<b>110</b>	01:00:29	<b>.030</b>	3.00
5	80/1315	1/0/8	40	<b>00:07:55</b>	<b>.080</b>	40	00:40:22	.017	0.20
6	80/1315	0/0/3	30	<b>00:01:24</b>	<b>.357</b>	<b>35</b>	01:00:49	.010	0.02
7	138/2158	8/7/0	-	00:38:43	-	-	<b>00:12:01</b>	-	(3.2)

**Table 5 Mage runtime vs. old SAT runtime**

Better results are reported in bold font. Relevant aspects of this experiment are the promotion of one property from bounded proof to failing in test case 2 and the general great superiority of reached depth for almost all test cases; the few diverging results (test case 5 and test case 6) were linked to dramatic performance decay on single properties, which is very likely due to the usage of multi-user machines. Notably, on test case 7, in which all properties have a closed status (either they pass or fail, there are no bounded proofs), Mage offered an improvement in verification time of a factor above 3. Mage property set was checked using the same set of engines adopted for the corresponding old SAT run. In general the verification setup for properties with bounded proofs comprises a verifying engine (like SmartLoc and Discovery) to be run in parallel. This is the reason of the promotion from bounded to passing in test case 1, which shows that more than just the SAT engine has been improved moving from the older RuleBase PE version to the newer one.

---

## Discovery vs. SAT

We ran both SAT and Discovery on the benchmark. For this experiment, we used the model variables order as an initial BDD order. We used either the size of the smallest counterexample or the number of cycles needed for completing reachability analysis as the greater bound for SAT, when these were available.

Table 6 shows the results:

Model	Rule	Formula	Discovery runtime	Timeout	# var after reduction	Results	Cycle fail/reachability analysis	SAT result	SAT runtime
1			00:03:23		93	fail	14	sat 15	00:00:40
2	1	1	00:02:48		75	pass	40	unsat 40	00:00:40
2	1	2	00:02:39		69	pass	40	unsat 40	00:00:41
2	1	3	00:02:12		66	pass	37	unsat 40	00:00:41
2	1	4	00:02:11		67	pass	37	unsat 40	00:00:38
2	1	5	00:02:08		66	pass	37	unsat 40	00:00:38
2	2		00:01:54		68	fail	5	sat 10	00:00:39
2	3	1	00:01:51		76	fail	4	sat 10	00:00:38
2	3	2	00:02:17		76	pass	37	unsat 40	00:00:41
2	3	3	00:01:53		74	fail	4	unsat 10	00:00:37
2	3	4	00:02:22		74	pass	39	unsat 40	00:01:09
2	3	5	00:01:50		74	fail	4	unsat 10	00:01:07
2	3	6	00:02:28		74	pass	39	unsat 40	00:01:18
2	3	7	00:01:41		74	pass	37	unsat 40	00:01:20
3			00:02:07		75	fail	32	sat 35	00:01:01
4			00:03:09		209	fail	24	sat 25	00:00:44
5			00:06:12		308	fail	31	sat 35	00:00:55
6			00:06:54		126	fail	31	sat 35	00:01:12
7			00:02:52		167	pass	2	sat 35	00:18:28
9			18:00:00	timeout	127			sat 55	00:00:42
10			01:33:07		218	Pass (6)	109	sat 110	00:04:46
11		1	00:05:52		202	fail	31	sat 35	00:02:01
11		2	01:29:01		201	pass	119	sat 120	00:07:28
11		3	01:14:17		201	pass	120	timeout	23:00:00
12			13:00:00	timeout		timeout		contradiction	00:00:40
13			00:04:57		80	pass	2	unsat 10	00:01:16
14		1	06:55:27		157	pass	60	unsat 60	00:00:53
14		2	01:42:26		157	pass	60	unsat 60	00:05:20
15			01:52:09		232	fail	9	unsat 10	00:00:45
16	1		13:12:41		129	pass	27	unsat 30	00:00:44

Model	Rule	Formula	Discovery runtime	Timeout	# var after reduction	Results	Cycle fail/reachability analysis	SAT result	SAT runtime
16	2	1	00:02:02		91	fail	5	unsat 10	00:00:45
16	2	2	00:02:01		91	fail	5	unsat 10	00:00:43
16	2	3	00:01:59		91	fail	5	unsat 10	00:00:44
16	2	4	00:02:31		129	fail	5	unsat 10	00:00:45
16	2	5	00:02:05		128	fail	0	unsat 10	00:00:55
16	2	6	00:02:06		129	fail	0	unsat 10	00:00:46
17	1	1	20:00:00	timeout				unsat 50	00:00:52
17	1	2	20:00:00	timeout				unsat 50	00:01:02
17	2	1	20:00:00	timeout				unsat 50	00:00:56
17	2	2	20:00:00	timeout				unsat 50	00:00:57
18			00:03:01		76	fail	29	sat 30	00:06:03
19			00:03:57		110	fail	29	unsat 30	00:00:52
20			00:05:22		76	fail	44	sat 45	00:54:25
21			00:03:19		76	fail	29	sat 30	00:00:47
22			00:16:55		100	fail	52	sat 55	00:24:40
23			00:09:13		99	fail	36	sat 40	00:46:15
25			11:36:39		118	pass	35	timeout	23:00:00
26			05:40:44	timeout				unsat 100	00:02:06
27			00:01:49		40	fail	18	sat 20	00:01:00
28			00:43:37		94	fail	14	sat 15	00:00:59
29			00:04:36		82	fail	26	sat 30	00:43:40
30			16:37:59		174	fail	25	timeout	23:00:00
31	1		08:47:56		93	pass	87	timeout	23:00:00
Total			184:30:39						96:07:34

**Table 6 Discovery runtime vs. SAT runtime**

The correlation between SAT and Discovery runtimes is quite low (0.26). So there are few relationships between runtime with SAT and Discovery, as Figure 12 illustrates.

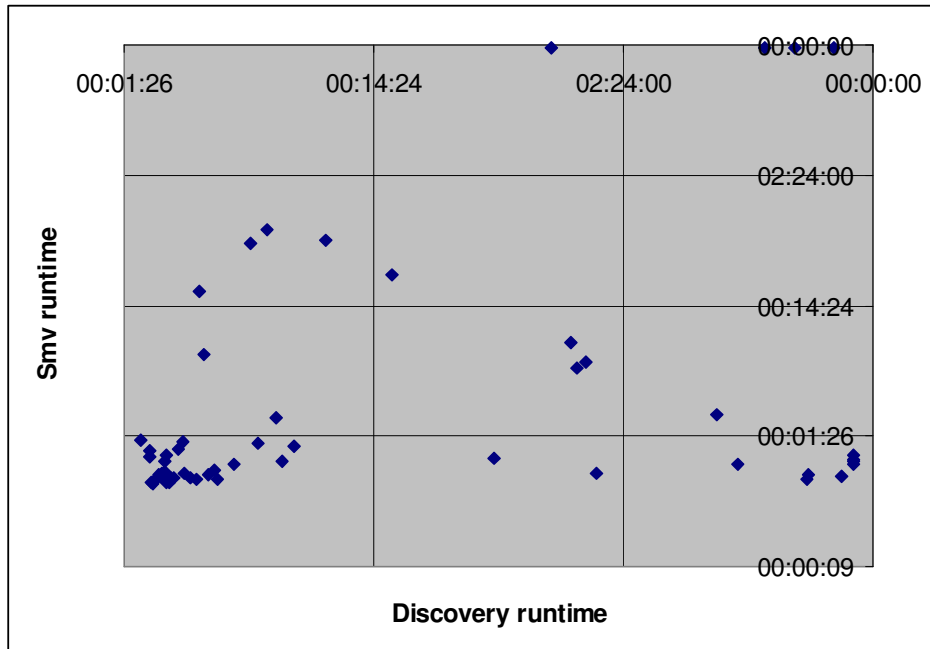


Figure 12 SAT runtime vs. Discovery runtime

# 5 Conclusions

We discussed the specialities of each engine with regard to the characteristics of the design under test, the verification method and the available resources. The experiments showed that we can not predict whether an engine can successfully perform a given verification task. For SAT, we could not find way to predict if a CNF formula will be easier to solve with some SAT solver or another. Therefore we cannot predict for a specific model if SAT engine will behave better or worst than another engine.

No one engine solves all problems, therefore in this methodology document we have provided guidance on how best to leverage different static checking engines. We proposed methods to choose an appropriate verification strategy. And we recommended about the most effective verification tactic, including, engine selection and algorithm fine tuning.

# 6 References

- [1] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In Design Automation Conference, pages 655–660, June 1996.
- [2] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Yoav Rodeh, G. Ronin, and Y. Wolfsthal. Rulebase: Model checking at IBM. In Computer Aided Verification, pages 480–483, 1997.
- [3] B. Bentley. Validating the Intel Pentium 4 microprocessor. In Design Automation Conference, pages 244–248, 2001.
- [4] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In Computer Aided Verification, pages 184–194, 1998.
- [5] V. Bertacco and K. Olukotun. Efficient state representation for symbolic simulation. In 39th ACM/IEEE Design Automation Conference, 2002.
- [6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. Lecture Notes in Computer Science, Vol. 1579, pages 193–207, 1999.
- [7] A. Biere, E. Clarke, R. Raimi and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. Proceedings of the 36th ACM/IEEE conference on Design automation, pages 317–320, 1999.
- [8] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In IEEE Trans. on Computers, Vol. C-35, No. 8, pages 677-691, 1986.
- [9] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking:  $10^{20}$  states and beyond. In Information and Computation, Vol. 98, pages 142–170, 1992.
- [10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In proc. Workshop on Logic of Programs, Lecture Notes in Computer Science, Vol. 131. pages 52–71, 1981
- [11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu., H. Veith. Counterexample-guided abstraction refinement. In Computer Aided Verification I, pages 154–169, 2000.
- [12] E. M. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 2000.
- [13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. Communications of the ACM, Vol. 5, pages 394-397, 1962.
- [14] M. Davis and H. Putnam. A computing procedure for quantification theory. Journal of the Association for Computing Machinery, Vol. 7, pages 201–215, 1960.
- [15] M. K. Ganai, A. Aziz and A. Kuehlmann. Enhancing simulation with BDDs and ATPG. In proceedings of the 36th ACM/IEEE conference on Design automation, pages 385–390, 1999.
- [16] S. G. Govindaraju and D. L. Dill. Verification by approximate forward and backward reachability. In Inter. Conf.on Computer Aided Design, 1998.

- [17] G. Cabodi, S. Nocco, and S. Quer. Mixing forward and backward traversals in guided-prioritized BDD-based verification. In *proc. Lecture Notes in Computer Science*, Vol. 2404, pages 471-484, 2002.
- [18] S. Keidar and Y. Rodeh. Searching for Counter-Examples Adaptively. In *proc. of the 6th International Workshop on Formal Methods*, 2003.
- [19] R. P. Kurshan. *Computer-aided-verification of coordinating processes*. Princeton University Press, 1994.
- [20] K. L. McMillan. *The SMV System DRAFT*. Carnegie Mellon University, Pittsburgh, PA, 1992.
- [21] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, Norwell, MA, 1993.
- [22] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, page 530-535, 2001.
- [23] L. Ryan. *Efficient algorithms for clause-learning SAT solvers*. Master Thesis. Simon-Fraser University 2002.
- [24] G. P. M. Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, pages 506–516, 1999.
- [25] R. Tzoref, M. Matusevich, E. Berger and I. Beer, An optimized symbolic bounded model checking engine CHARME'03, pages 141-149, 2003.