

FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Improved Static Property Checking Tool (Deliverable 3.2/7)

Due date of deliverable: July 31, 2005

Actual submission date: July 31, 2005

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: IBM

Revision 1.0

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

Notices

For information, contact yorav@il.ibm.com

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.7/2 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	5.7.05	First draft by authors	Ziv Nevo Tatyana Veksler Karen Yorav	All
0.2	7.7.05	Incorporate comments by C. Eisner	Karen Yorav	All
0.3	17.7.05	IBM clearance, release for internal review by consortium	Karen Yorav	All
1.0	31.7.05	Final approval by project management	C. Eisner	Version number, footers

Authors

Ziv Nevo, Tatyana Veksler, and Karen Yorav

Executive Summary

The Achilles heel of Model Checking tools is the size problem. State-of-the-art tools, including RuleBasePE, are currently mostly used for block level verification, and only rarely for unit verification. In order for model checking to live up to its potential the tools need to scale to larger tasks.

The purpose of this deliverable is to implement enhancements to the RuleBasePE model checking tool. These enhancements are a result of research, some of which was done in the scope of the PROSYD project, and some of which comes from academic published work.

The deliverable encompasses enhancements to two of RuleBasePE's verification engines - the SAT engine, and SmartLoc - as well as a new engine called Interpolator.

Purpose

The purpose of this document is to describe the theory and implementation of enhancements made to RuleBasePE, in the framework of Deliverable 3.2/7.

Intended Audience

This document is intended for users of IBM's RuleBase PE tool.

Background

Model Checking is a verification method aimed at giving full coverage of a design. A model checker accepts the description of a design, and a formal specification of

its correct behavior, and is expected to prove that the design adheres to its specification. The result given by a model checking algorithm is either a confirmation that the specification holds on all possible computations on all possible input traces, or a counterexample showing that the design fails to meet its requirements.

However, currently model checking tools can only be applied to relatively small designs. There is much research being carried out, both in the industry and in academia, on enhancing the algorithms used for model checking, and inventing new algorithms that will scale better. This document presents some of the theory, and implementation, involved in the enhancement of RuleBasePE.

Contents

Table of Revisions.....	iii
Authors	iii
Executive Summary	iii
Purpose	iii
Intended Audience	iii
Background.....	iii
Table of Figures	vi
Table of Tables	vi
Glossary.....	vii
1 Introduction.....	1
2 SAT Based Bounded Model Checking	2
Incremental SAT Solving for BMC.....	2
Incremental SAT - User Guide Excerpt.....	4
Clause Replication	5
Background	5
Implementation	5
Results	6
Clause Replication - User Guide Excerpt	8
Adaptive Solving	8
Background on SAT solvers.....	9
Performance Metrics	9
Adaptive Solving	11
Experimental Results	12
Adaptive Solving - User Guide Excerpt.....	14
3 Reductions in SmartLoc	16
Abstraction Refinement	16
Implementation.....	17
BDD-Based Reductions	17
Flip-flop-Pool Reduction.....	17
Equivalence-Pool Reduction	17
Cone-of-influence Reduction	18
Reset Reduction	18
Over-approximation Reduction	18
Reductions before computing counterexample	18
Experimental Results	18
SmartLoc Reduction Options - User Guide Excerpt	19
4 Interpolator	21
Basic Algorithm.....	21
Optimizations	22
Frontier set simplification	22
Minimizing the Unsatisfiable Core.....	22

	Efficient construction of the interpolant	23
	Experimental Results	23
	Interpolator options - User Guide Excerpt	24
	General	24
	Model Checking Options	25
5	Conclusions.....	26
6	References.....	27

Table of Figures

	Figure 1: Graph of results for Clause Replication	8
	Figure 2: Abstraction refinement	16
	Figure 3: Interpolator main-loop.....	21
	Figure 4: Running example, including the evaluated Boolean formulas	22
	Figure 5: Interpolator vs. Discovery (denoted here as SMV).....	24

Table of Tables

	Table 1: Results for Incremental BMC.....	4
	Table 2: Summary of Results for Incremental BMC	4
	Table 3: Results for Clause Replication	7
	Table 4: Experimentation results for Adaptive Mage	14
	Table 5: Summary of Adaptive Mage results	14
	Table 6: SmartLoc runtimes with and without reductions	19

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 $\neg b$ 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.

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.

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

Short for *unsatisfiable*. A possible result of a SAT solver. See SAT solver.

1 Introduction

One of the goals of the PROSYD project is to encourage the advancement of verification tools that support PSL. The goal of this deliverable is to present improvements that were made in RuleBasePE as a part of this effort. The improvements reported in this document are a result of research performed in the context of the project, as well as implementation and integration of results from Academia.

The following sections give details of the theory and implementation of enhancements made to RuleBasePE. Some of these ideas were already detailed in previous PROSYD documents, some are presented in published papers, and some are new ideas that are presented here for the first time. The level of detail of the description is accordingly.

The RuleBasePE tool is comprised of various verification engines that can be run in parallel. All engines solve the same problem: verifying a design against a given PSL specification and producing a counterexample if the design does not comply. However, each engine is based on a totally different verification technology. Each engine on its own has areas of strength and weakness. Some are better at verifying and others at falsifying, some are better for shallow designs and some work better for deep ones, etc. Running them in parallel gives the user the best overall solution. Still, as the solution becomes faster, RuleBasePE users attempt to apply it to ever growing designs. There is a constant need for enhancements that can improve the capabilities and scalability of the tool.

The body of this document is divided into 3 sections, each describing work performed on a different verification engine. Section 2 presents enhancements made to the SAT engine, which is the Bounded Model Checking engine of RuleBasePE. Three different algorithms were incorporated into this engine, 2 are implementations of known methods, and one is totally new technology. Section 3 gives details on the introduction of model reductions into the SmartLoc engine. Finally, Section 4 presents a new verification engine called Interpolator. This engine uses new technology that enables using a SAT solver for unbounded verification.

Some of the enhancements presented here have already been given to the users of RuleBasePE, including the new Interpolator engine. Others are still being fine tuned, and will be issued to users in the next few months. For each enhancement, the presentation includes a theoretical discussion, description of implementation issues (if relevant), experimental results, and relevant excerpts from the RuleBasePE user's guide. In the case of enhancements that have not yet been issued to users, the excerpts represent text that is going to be added to the user guide.

2 SAT Based Bounded Model Checking

This section describes work on the SAT-based BMC engine of RuleBasePE. This engine is built on top of IBM's proprietary SAT solver called Mage. Mage is a DPLL-style SAT solver, incorporating non-chronological backtracking and conflict clause learning, as described, for example, in [12].

Incremental SAT Solving for BMC

In Deliverable 3.2/1[18] a method of integrating incremental SAT solving and Bounded Model Checking was introduced. For a detailed description see [18]. The current deliverable presents the results of implementing Incremental BMC in the RuleBasePE tool, and benchmarking its performance.

Incremental BMC was implemented into the SAT engine of RuleBasePE. This option is applicable only when the engine is working in "auto mode". In this mode of operation the SAT engine first checks the specification over the first k clock cycles of all computations of the design, where k is called the initial bound, and is supplied by the user. This first verification step is called the *first instance*. Assuming no bug was found within cycles $[0..k]$, the engine proceeds to check cycles $[k+1..k+j]$, where j is the "jump" constant (also supplied by the user), which determines the number of cycles to check at each subsequent iteration. The engine continues in this manner until either a bug is found, or some upper limit is reached. This could be an upper limit on the clock cycles, the run time, or the memory consumption.

In incremental mode, the SAT solver is capable of retaining information that was gathered from one instance to the next. In particular, conflict clauses are retained. These clauses significantly reduce the search space that future instances need to cover, and thus accelerate the run. Note that not all conflict clauses can be retained. At the beginning of each instance the solver is requested to delete all clauses that are no longer relevant. Deliverable 3.2/1 presents a detailed algorithm for incorporating incremental SAT solving into the Bounded Model Checking process, and using this for induction based verification. The experimentation detailed below refers to the Bounded Model Checking algorithm itself.

Table 1 presents the results of running the SAT based Bounded Model Checking engine of RuleBasePE on a large benchmarking suite. The "Bound" column gives the upper bound of the last instance. All examples were run in auto mode with an upper bound of 100 for unsatisfiable instances. All run times are in seconds. The speedup is computed by dividing the non-incremental run time by the incremental run time.

The benchmarking suite used here is an updated collection based on the IBM Benchmarking suite. It includes the non-trivial examples of the public suite, along with a collection of new examples that are relatively large.

Example	Non-Incremental			Incremental			Speedup
	Result	Bound	Run Time	Result	Bound	Run Time	
1	SAT	10	0.52	SAT	10	0.42	1.24
2	SAT	10	0.55	SAT	10	0.4	1.38
3	SAT	10	2.2	SAT	10	1.82	1.21
4	SAT	15	2.43	SAT	15	1.49	1.63
5	SAT	20	2.43	SAT	20	0.98	2.48
6	SAT	10	3.71	SAT	10	3.05	1.22
7	SAT	15	5.53	SAT	15	2.92	1.89
8	SAT	25	8.14	SAT	25	1.89	4.31
9	SAT	30	11.68	SAT	30	3.65	3.20
10	SAT	30	12.66	SAT	30	6.03	2.10
11	SAT	35	13.19	SAT	35	3.06	4.31
12	SAT	35	32.25	SAT	35	5.58	5.78
13	SAT	35	43.21	SAT	35	24.05	1.80
14	SAT	35	175.62	SAT	35	115.89	1.52
15	SAT	30	358.88	SAT	30	190.09	1.89
16	SAT	55	1067.94	SAT	55	587.24	1.82
17	SAT	45	1786.5	SAT	45	1468.64	1.22
18	SAT	40	2002.45	SAT	40	1735.51	1.15
19	UNSAT	100	61.14	UNSAT	100	3.71	16.48
20	UNSAT	100	66.23	UNSAT	100	2.86	23.16
21	UNSAT	100	83.7	UNSAT	100	4.4	19.02
22	UNSAT	100	106.06	UNSAT	100	65.12	1.63
23	UNSAT	100	125.36	UNSAT	100	164.69	0.76
24	UNSAT	100	161.55	UNSAT	100	69.11	2.34
25	UNSAT	100	251.11	UNSAT	100	3.53	71.14
26	UNSAT	100	366.55	UNSAT	100	21.54	17.02
27	UNSAT	100	458.68	UNSAT	100	12.61	36.37
28	UNSAT	100	579.14	UNSAT	100	182.48	3.17
29	UNSAT	100	584.8	UNSAT	100	26.39	22.16
30	UNSAT	100	746.56	UNSAT	100	1738.28	0.43
31	UNSAT	100	750.53	UNSAT	100	497.29	1.51
32	UNSAT	100	1729.77	UNSAT	100	56.08	30.84
33	UNSAT	100	1963.62	UNSAT	100	45.24	43.40
34	UNSAT	100	2013.04	TO	90	3991.94	0.50
35	UNSAT	100	2048.84	UNSAT	100	46.47	44.09
36	UNSAT	100	3528.42	UNSAT	100	1793.3	1.97
37	TO	65	3994.81	UNSAT	100	1342.53	2.98
38	TO	85	3994.98	UNSAT	100	412.96	9.67
39	TO	60	3995.83	TO	75	3997.76	1.00
40	TO	25	3995.96	TO	25	3996.84	1.00
41	TO	55	3996.49	TO	55	3997.44	1.00
42	TO	20	3997.05	SAT	30	2558.43	1.56
43	TO	10	3997.06	TO	10	3996.89	1.00
44	TO	10	3997.27	TO	10	3996.87	1.00
45	TO	75	3997.41	UNSAT	100	2726.14	1.47
46	TO	15	3997.57	TO	15	3997.23	1.00
47	TO	40	3998.08	TO	25	3998.82	1.00

48	TO	30	3998.97	TO	30	3996.05	1.00
49	TO	40	3999.9	TO	35	3995.77	1.00
50	TO	85	4068.93	UNSAT	100	348.89	11.66

Table 1: Results for Incremental BMC

In order to evaluate the impact of incremental BMC we compute both the global speedup and the geometrical mean of speedups. Table 2 presents the summary of the results. The "Total depth" columns give the sum of the bounds, while the "Total Time" columns are the sum of run times. The table analyzes the results for satisfiable (SAT), unsatisfiable (UNSAT), and timed out instances separately (according to the results achieved by the non-incremental version). The "all Done" row is the sum of all SAT and UNSAT instances. The global speedups are computed by dividing the Non-Incremental value with the Incremental value.

	Non-Incremental		Incremental		Speedup	
	Total depth	Total time	Total depth	Total time	Global	Geomean
SAT	485	5529.89	485	4152.7	1.33	1.96
UNSAT	1800	15625.1	1790	8725	1.79	7.16
all Done	2285	21154.99	2275	12878	1.64	3.75
Timed out	615	56030.31	710	43363	1.29	1.61

Table 2: Summary of Results for Incremental BMC

The global speedup is a measure that gives more weight to the larger examples. Since some of the examples run for thousands of seconds, the effect of examples that run for tens or hundreds of seconds is very small. For this reason we also look at the geometrical mean of speedups. This measure is more indicative of the robustness of the method, showing that a general speedup is achieved on all sizes of examples.

From both Table 1 and Table 2 it is clear that incremental SAT solving improves the performance of BMC. In a few cases the results were negative - either run time was increased, or the final depth in a timed out instance was lower. However, in many cases the performance boost is very significant. There were five examples in which the non-incremental version timed out, while the incremental version was able to supply an answer. The global run time speedup is 1.64, while the geometrical mean of speedups is 3.75. It is easy to see that incremental BMC is even more effective when we consider only unsatisfiable instances. In this case the global speedup is 1.79 and the geometrical mean of speedups is 7.16.

Incremental SAT - User Guide Excerpt

Quoted from the "Engines" section of the RuleBasePE User's Guide [20]. In the general presentation of the SAT engine the following description was added:

When SAT works in auto mode incrementally, information gathered in one interval is passed to the next. This increases the capability of the engine, enabling it to explore deeper cycles faster.

To use this option efficiently, it is advised to start the verification from cycle 0, because the information gathered during the first intervals can dramatically reduce the time needed to solve subsequent intervals. In other words, in some cases, verifying the interval between clock cycles 100 - 110 may be significantly

harder than verifying clock cycles 0 - 110 incrementally, in intervals of, say, 10 clock cycles each.

In the list of SAT engine options that are available in "auto mode" the following option was added:

Incremental

When this option is set and Auto Mode is used, the SAT solver saves information it gathers in each interval and uses it in the next.

Clause Replication

Background

The Boolean formula representing a BMC instance shows a kind of symmetry across cycles. The major part of this formula is the unwinding of the transition relation, which consists of k copies of the transition relation. The difference between the copies is the renaming of the literals to fit to the different cycles. A more detailed explanation of this was given in Deliverable 3.2/1.

In [15], Strichman notes that this symmetry can be exploited to artificially produce useful conflict clauses. When the conflict analysis function generates a conflict clause, the replication algorithm generates additional clauses by changing the literal names. These are clauses that would not have been generated through regular conflict analysis, but are nevertheless correct, and hopefully useful.

As an example, assume that for every variable v , v_i is its name at clock cycle i . Further, assume that the conflict clause $C = (x_1 \vee y_3)$ was generated by the conflict analysis function. The clause C represents a connection between the value of x at cycle 1 and the value of y at cycle 3. But this connection is not limited to cycles 1 and 3, the clause $(x_2 \vee y_4)$ represents the same connection, shifted by one clock cycle. The clause replication algorithm generates such shifted clauses for all applicable clock cycles.

However, not all conflict clauses can be replicated. If a clause from the Init or Spec parts of the formula was involved in the generation of the conflict clause, then this conflict clause is not replicable, because the formula is not symmetric with respect to the Init and Spec parts. This requires some bookkeeping to make sure that all generated clauses are correct.

Implementation

As described in [16], during the generation of conflict clauses they are marked with a "replicable" bit. Once a clause from the Init or Spec parts is involved the resulting conflict clause is marked un-replicable. In order for a generated conflict clause to be marked replicable all clauses in the implication graph involved in its generation must be marked replicable.

The implementation of clause replication in Mage differs from the description of [16], because the Strichman's work was applied to a different SAT solver. Clause replication within Mage involves keeping additional information about the minimum and maximum clock cycles that are involved in the implication graph of each conflict clause, and limiting the clock cycles over which a clause can be

duplicated. Some optimizations need to be turned off when using clause replication and this has an effect on the performance boost.

The implementation includes conditions that determine which clauses to replicate, and how far to duplicate. Duplicating all possible clauses on all possible cycles would generate too many clauses. This would hurt performance in two ways: memory consumption could exceed the limit unnecessarily, and the computational overhead in processing the redundant clauses could increase run times significantly. Instead, the implementation places a size limit L on candidate clauses. Only clauses of size less than L are duplicated. Furthermore, in some cases it is beneficial not to duplicate on all possible clock cycles, but rather only on those cycles that are new to this instance. This prevents the addition of clauses that already appear in the database.

The replication code is currently being tuned for even better performance. As mentioned earlier, a couple of optimizations within the solver were turned off because they conflict with the correctness of the clause replication algorithm. Further research is required in order to find ways to integrate these optimizations back into the solver safely. When this is done, hopefully within a month or so, the replication algorithm will be released to customers.

Results

The results of the current implementation are presented in Table 3. The examples are from the IBM Benchmark suite and include both satisfiable and unsatisfiable instances. The Speedup column gives the run time without clause replication divided by the runtime with clause replication¹. Figure 1 gives a scatter plot depicting these results. It is easy to see that in almost all cases Clause Replication is as good as or better than running without it. Note that the graph scale is logarithmic. It is clear that clause replication has a higher impact on larger examples, which shows that this method is highly scalable. The global speedup on all examples was about 2x.

Example	Result	Bound	Run Time	Speedup
batch_01	SAT	15	2.47	1.33
batch_02_2	SAT	10	0.85	0.95
batch_03	SAT	35	14.62	1.11
batch_04	SAT	25	9.4	0.97
batch_05	SAT	35	31.61	1.03
batch_06	SAT	35	26.97	1.26
batch_1_02_3	SAT	10	1.06	0.93
batch_1_11	SAT	35	88.4	2.18
batch_1_16_2	SAT	10	1.15	0.96
batch_15	SAT	10	3.19	1.00
batch_18	SAT	30	120.87	4.70
batch_19	SAT	30	15.49	1.11
batch_2_16_2	SAT	10	1.11	1.04
batch_20	SAT	45	1307.5	1.89
batch_21	SAT	30	12.33	1.06
batch_22	SAT	55	770.56	2.05
batch_27	SAT	20	2.7	1.03

¹ All runs are without the "Incremental" option.

batch_28	SAT	15	21.19	0.42
batch_3_02_3	SAT	10	1.01	1.04
batch_3_16_2	SAT	10	1.09	1.05
batch_4_16_2	SAT	10	1.43	1.03
batch_5_02_3	SAT	10	1.06	1.00
batch_5_16_2	SAT	10	1.38	1.05
batch_6_16_2	SAT	10	1.44	1.00
batch_23	SAT	40	1339.97	2.98
batch_29	SAT	30	2075.2	1.93
batch_09	UNSAT	100	72.92	0.95
batch_1_02_1	UNSAT	100	216.66	2.27
batch_1_13	UNSAT	100	133.07	0.99
batch_1_14	UNSAT	100	131.82	10.70
batch_1_17_1	UNSAT	100	87.68	0.99
batch_12	UNSAT	10	0.28	0.93
batch_16_1	UNSAT	100	64.8	1.02
batch_2_02_1	UNSAT	100	276.61	2.02
batch_2_02_3	UNSAT	100	60.65	1.04
batch_2_17_1	UNSAT	100	226.89	1.93
batch_26	UNSAT	100	595.31	1.00
batch_3_02_1	UNSAT	100	80.05	3.35
batch_4_02_1	UNSAT	100	90.97	2.12
batch_4_02_3	UNSAT	100	68.93	0.97
batch_5_02_1	UNSAT	100	84.58	2.58
batch_6_02_3	UNSAT	100	67.04	1.07
batch_7_02_3	UNSAT	100	105.38	1.02
batch_2_11	UNSAT	100	881.19	4.53
batch_1_31_1	TIME_OUT	25	3988.9	1.00
batch_14_13	TIME_OUT	0	3997.28	1.00
batch_2_14	TIME_OUT	80	3960.54	1.01
batch_2_31_1	TIME_OUT	25	3989.13	1.00
batch_21_13	TIME_OUT	75	3917.98	1.02
batch_3_11	TIME_OUT	50	3969.18	1.01
batch_3_31_1	TIME_OUT	25	3989.69	1.00
batch_30	TIME_OUT	15	3983	1.00

Table 3: Results for Clause Replication

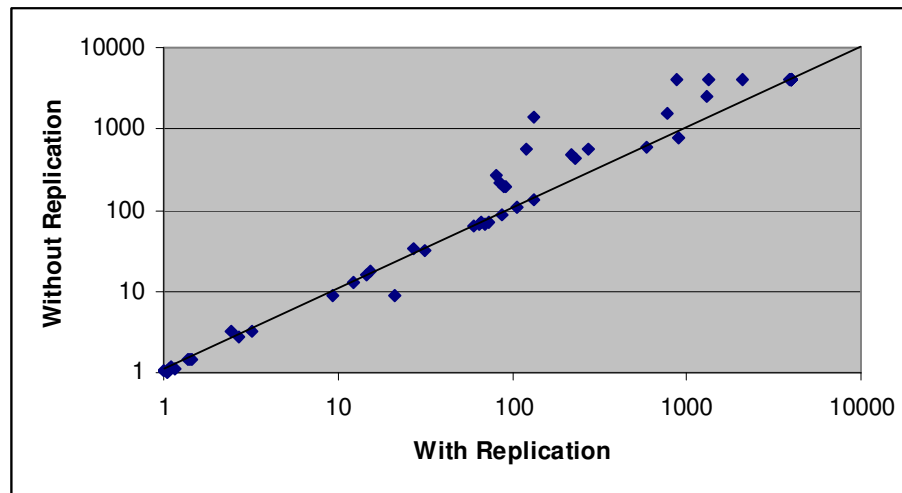


Figure 1: Graph of results for Clause Replication

Clause Replication - User Guide Excerpt

The following option will be added to the SAT engine section in the RuleBasePE user's guide [20] in the next release that incorporates replication:

Replication

When this option is turned on the SAT Solver will use a replication algorithm that generates extra conflict clauses. This algorithm can accelerate the search by duplicating conflict clauses across clock cycles.

Adaptive Solving

Adaptive Solving [14] is an innovative idea aimed at boosting the performance of the underlying SAT solver used by a verification engine. Current state-of-the-art SAT-based verification tools are based on DPLL-style SAT solvers, and adaptive solving aims at boosting the performance of this type of solvers. However, the methodology itself is general, and may easily be applied to other types of SAT solvers.

Modern SAT solvers rely heavily on various heuristics and strategies such as decision heuristics, restart strategies, learning strategies, clause deletion strategies, etc. [11][1][5][7][13][15] However, many ideas that seem appealing in theory turn out not to perform well in practice, decreasing the run time on a few examples while increasing it on most others. As a result, these ideas are discarded. Even heuristics that are considered overall successful are not useful in all cases, but it is impossible to know beforehand which heuristics are most suitable for a given example.

Adaptive solving attempts to solve this problem by optimizing the way different strategies are used. The solver is given the capability to apply different heuristics when they are useful and turn them off when they are not. The adaptive solver tracks the performance of the search and evaluates it using a *Performance Metric*. Whenever the search seems not to be progressing well enough, it changes run-time parameters by enabling or disabling heuristics. In this way the adaptive solver is capable of making use of heuristics that do not work well in all cases.

Background on SAT solvers

The basic algorithm run by modern SAT solvers was described in Deliverable 3.2/1, and will not be repeated here. Instead, we list some of the terms that are required by the following description.

- **Input:** The input to the SAT solver is Boolean Function in *Conjunctive Normal Form* (CNF), i.e. a conjunction of clauses, where each clause is a disjunction of literals, and each literal is an instance of a variable or its negation.
- **Decisions:** A decision is the choice of a variable and a value to assign into it. Each decision is associated with a decision level, starting with 1 for the first decision and incrementing by 1 with each further decision.
- **Boolean Constraint Propagation (BCP):** Following a decision, the implications of the decision are carried out by the BCP function, until no further deductions can be made. All assignments made as implications of a decision are associated with the same decision level.
- **Conflict Clauses:** When a conflict occurs, the reason for the conflict is analyzed, and a new clause is added to the database. The new clause is called a *conflict clause* and it prevents the same conflict from happening a second time.

Performance Metrics

The performance of a SAT solver is measured by the time it takes to solve a given instance. In order to evaluate whether a certain heuristic or strategy is beneficial, the overall run time on many different formulas is compared. The reality is that even for the best heuristics, there are examples on which they do not work well. The role of the performance metric is to assess the compatibility of the solver settings for a specific example during the run.

A performance metric is required to adhere to the following requirements:

- The metric can be evaluated during the solver's run.
- It can be calculated efficiently, i.e. with low overhead, and with minimal additional space consumption.
- The metric gives a score that (roughly) corresponds to the effectiveness of the search.

The nature of the SAT problem is such that it is unrealistic to expect to find a metric that will give a perfect correlation to the end result. However, based on our understanding of the way the solver operates, we have come up with several candidates. We list below the metrics that have been implemented in the adaptive version of Mage. Each metric is evaluated on a *sample*, consisting of a constant number of decisions, in this case 2048.

Decision Level

When a decision causes a conflict, the solver backtracks to a previous decision level and cancels all the assignments made in between. In this case the decision level of the next decision will be some smaller number. Otherwise, if there was no conflict, the decision level increments by one. A high decision level means that the solver has made many un-conflicting decisions. A low decision level can be the result of a conflict, or restart.

The **DL** metric looks at the average decision level in a single sample. The solver reaches high decision levels when it makes a large number of decisions without conflicts, or when the conflicts do not set it back by much. As a result many variables keep their value for long periods of time. This could mean that the solver spends significant amounts of time searching in a small part of the state space. On the other hand, a low average may indicate a high conflict rate. Since each conflict clause restricts the space that remains to be searched, a high conflict rate is a good sign. This leads us to expect that an efficient run is one in which the average decision level is relatively low.

Conflict Clause Size

As mentioned earlier, every conflict clause that is added is a constraint that reduces the state space that remains to be searched. In general, smaller conflict clauses are capable of posing a greater restriction. So, although it is possible for a specific small clause to be less useful than a very large one, in general we expect that very small conflict clauses advance the search more rapidly towards its goal.

The **CCS** metric is the average length of all the conflict clauses that were learned in a sample - the smaller the better. Note that the score does not explicitly reflect the actual number of conflict clauses.

Binary Conflict Clauses

Binary conflict clauses are especially important because they have the highest potential of generating implications (a single assignment makes the clause unit). It is therefore expected that adding many binary clauses to the database greatly advances the search. The **BIN** metric measures the percentage of binary conflict clauses out of the total number of conflict clauses learned in a given sample.

We have also considered looking at ternary clauses as a part of this metric. However, extensive experimentation revealed that the percentage of ternary conflict clauses is linearly correlated to the percentage of binary clauses. In all of our examples, these two numbers are almost equal, and they increase and decrease in the same manner from one sample to the next. We concluded that there is no added benefit in tracking ternary conflict clauses, and this is not included in the latest version of Adaptive Mage.

BCP Ratio

When a watched literal l in a clause c becomes false, the BCP process must go over the literals in c and look for a new watched literal. In the worst case scenario, all the literals in c are examined. The **BCP** metric measures the ratio between the number of literals examined (all together) and the number of clauses visited, i.e., it calculates the average number of literals examined per clause. This ratio is indicative of the speed at which implications are carried out. Since the BCP operation is the major part of the computation, it is important to keep this number low.

Unary Clauses Learning

The conflict analysis function is capable of producing unary conflict clauses. This amounts to learning the value some variable must have in order to satisfy the formula. The variable is then assigned a permanent value. When this happens, the algorithm backtracks to decision level zero and applies BCP to discover all implications of this assignment. Any assignment resulting from this BCP process is also permanent.

The **UNARY** metric tracks the rate at which permanent values are assigned. It gives the number of such values assigned in the last sample. An examination of the behaviour of this metric reveals that learning happens in bursts. Typically, there are extended periods of time with little or no learning, and then suddenly tens or even hundreds of variables are assigned a value.

Adaptive Solving

In general, an Adaptive Solver works according to the following scheme:

- Every fixed number of decisions the performance metric is computed, and a score is assigned according the search's performance since the last evaluation.
- A condition on this score is evaluated, to determine whether a change in the configuration should be made.
- If the condition for switching configurations is met, one or more parameters of the run are changed. This is called *a switch*.

The details of the implementation of an adaptive version of Mage are as follows:

- Adaptive switching is enabled only after 20,000 decisions, so that it is used only for non-trivial examples.
- The metric is evaluated every 2048 decisions.
- The switching condition is a comparison of the metric score to some upper threshold. The threshold was tuned by running on a large set of examples.
- To prevent the adaptive solver from switching too often, after each switch the switching condition is made slightly tighter (by increasing the bound).

- The parameter that is controlled by the adaptive algorithm is called the "sign" option. This option controls the way a value is chosen for a decision variable. Normally, after choosing a variable to decide upon, the variable is assigned *true* or *false* according to the scores of the corresponding literals. The default is to assign *true* to the literal with the highest score. Activating the "sign" option will assign *true* to the literal with the lower score.
- After each switch, switching is disabled for a few samples, in order to give the new configuration a chance to stabilize.
- A limit is placed on the total number of switches allowed during a single run. This takes care of difficult examples in which metric scores are inherently high, and the algorithm just switches endlessly.

Currently, the adaptive algorithm is not part of the tool that customers have. We are running tests on it in order to tune the algorithm and its parameters. We also plan to control more settings (rather than just the "sign" option). When this is done the adaptive algorithm will be released to customers.

Experimental Results

An extensive experimentation was conducted on the adaptive version of Mage, using the IBM Benchmarks Suite. A bound of 10,000 seconds was placed on each run, so that no timeouts occurred. This is done so that the time we choose to time out on will not influence the speedup results.

The analysis of the results is done by comparing the overall speedup, i.e., the speedup on the sum of the run times of all examples. This number is indicative of the effect Adaptive Solving has on a large number of examples of various sizes, such as in the case of a full verification project. This analysis places more weight on the larger examples, which is suitable for our experimentation, because our goal is to reduce the run times of large examples without hurting the small ones too much, thus reducing the overall time needed for a verification project to be completed.

The experimentation was conducted on an Intel Pentium 4, with a single 2GHz CPU, 1GB RAM, running Linux. Table 4 gives the run time results for all examples, in seconds. It compares the run times between seven versions:

- **Native** is Mage running with its default parameters and no adaptive algorithms. In particular, the "sign" option is not used, so in all decisions the sign with the highest score is chosen."
- **Sign** is Mage running with the "sign" option all of the time.
- Versions **DL**, **CCS**, **BIN**, **BCP**, and **UNARY** correspond to adaptive versions each using the metric implied by its name.

The Native and Sign versions do not calculate any of the metrics. Initial experimentation showed that the overhead incurred by the computation of the metric scores is negligible (only a few seconds even for the largest examples). Because the difference is so small we omit the results for a version that computes all metrics but does not apply adaptive solving.

Table 5 gives a summary of the results. In this table the "Time" rows display the sum of run times on all the examples in the benchmark (in seconds). The "Speedup" for each version is the runtime of Native divided by the runtime of that

version. The "Min" and "Max" rows show the minimum and maximum speedups achieved by each version on a single example. The results for satisfiable and unsatisfiable examples are given separately, and the "ALL" section summarizes all the examples.

Table 5 shows that BIN and UNARY are the best metrics, giving speedups of 1.6 and 1.5 respectively. The BCP and CCS versions give modest speedups, and DL has a negligible speedup. For the CCS, BIN, and UNARY versions, the speedup is better on SAT instances than on UNSAT. On SAT instances alone, UNARY gives a 2x speedup, while on the UNSAT instances, there is hardly any gain. On the other hand, the BCP version works better on UNSAT instances. The "sign" option is, indeed, not recommended as a default option, since it significantly increases the overall run time.

Examining the minimum and maximum speedups reveals how the overall speedup is achieved - by significantly reducing run times on some examples and only slightly increasing run times on others. For example, the worst damage the BIN version causes is a speedup of 0.75 (which equals to an increase of about a third), while its best performance is almost 12x faster. The best speedup was achieved on an example that runs 3821 seconds on the Native version, and 325 seconds with the BIN version.

A phenomenon we encountered, and can be seen in Table 4, is that in many cases the adaptive version performs better than either Native or Sign. From this we learn that different sub-spaces of the search space require different settings, suggesting that Adaptive Solving has great potential.

	Native	Sign	DL	CCS	BIN	BCP	UNARY
IBM_02_1_1_cycle_45	26.67	29.11	23.67	24.79	26.84	26.76	26.62
IBM_02_1_1_cycle_50	26.87	23.51	24.8	27.11	27.11	27.17	26.98
IBM_04_cycle_45	39.95	30.15	38.73	31.08	40.01	30.98	39.82
IBM_05_cycle_45	19.28	32.71	28.25	19.21	19.13	23.06	19.1
IBM_05_cycle_50	23.58	42.47	30.48	23.56	23.53	30.88	23.53
IBM_06_cycle_45	47.62	11.32	47.24	41.52	47.5	47.42	47.43
IBM_07_cycle_10	62.35	341.05	40.29	44.42	43.65	46.62	42.59
IBM_07_cycle_15	24.28	18.5	19.63	21.81	16.4	16.69	19.49
IBM_07_cycle_20	24.57	30.14	20.17	20.18	19.19	19.28	20.64
IBM_07_cycle_25	24.74	48.81	22.02	26.55	21.42	23.67	18.41
IBM_07_cycle_30	25.14	50.95	17.43	22.63	18.91	18.32	15.42
IBM_07_cycle_35	25.53	22.8	28.7	22.02	14.01	15.57	17.45
IBM_07_cycle_40	25.9	27.19	20.88	19.99	19.68	21.96	17.8
IBM_07_cycle_45	26.02	192.98	15.86	22.02	28.04	29.16	26.58
IBM_07_cycle_50	26.33	19.4	25.45	22.9	16.45	16.41	24
IBM_11_1_cycle_45	1140.99	1232.29	679.36	1175.36	952.89	1054.36	894.69
IBM_14_2_cycle_25	25.62	22.42	25.88	19.43	25.87	20.7	25.68
IBM_14_2_cycle_30	51.18	58.86	66.39	46.96	43.52	37.62	50.96
IBM_14_2_cycle_50	669.84	6881.23	668.32	535.39	618.71	593.9	684.73
IBM_17_1_2_cycle_40	16.23	8.89	19.14	16.25	16.19	16.17	16.3
IBM_17_1_2_cycle_45	19.38	7.79	24.37	19.39	19.92	19.37	19.32
IBM_17_1_2_cycle_50	15.98	11.64	15.77	16.15	19.23	16.03	15.29
IBM_19_cycle_35	37.96	32.8	30.09	51.42	43.29	37.8	37.71
IBM_19_cycle_40	69.01	120.82	91.02	60.07	84.79	70.69	70.41
IBM_19_cycle_45	112.26	120.35	124.43	220.46	144.64	112.29	155.17

IBM_19_cycle_50	283.95	194.38	401.87	215.21	310.72	341.76	542.8
IBM_20_cycle_25	73.59	75.19	73.32	67.84	73.88	88.68	73.86
IBM_20_cycle_30	378.49	313.49	378.6	304.09	378.23	366.73	377.86
IBM_20_cycle_40	2915.99	2193.85	2913.52	3338.03	2098.47	2612.28	3193.65
IBM_20_cycle_45	2262.22	1384.75	1584.61	4376.08	1495.49	1793.88	2104.08
IBM_20_cycle_50	5688.42	1281.89	1458.45	1075.95	3400.8	4287.91	964.08
IBM_21_cycle_35	30.56	24.51	28.6	35.64	30.45	30.47	30.38
IBM_21_cycle_40	97.71	73.52	90.42	92.78	97.43	97.39	97.09
IBM_21_cycle_45	321.15	221.58	230.48	198.86	235.65	322.9	320.48
IBM_21_cycle_50	207.1	335.21	216.49	211.32	275.71	207.17	270.92
IBM_22_cycle_35	38.34	41.92	47.13	40.96	38.24	37.13	38.18
IBM_22_cycle_40	115.97	155.49	124.63	113.16	116.41	121.43	116.21
IBM_22_cycle_45	368.1	612.17	400.4	378.86	366.91	443.05	371.93
IBM_27_cycle_45	12.83	21.72	19.23	18.31	12.73	19.67	12.68
IBM_28_cycle_30	21.22	18.12	21.11	31.1	21.21	21.31	21.19
IBM_28_cycle_40	21.72	21.41	27.08	39.31	21.74	52.79	21.73
IBM_28_cycle_45	22.25	55.86	28.96	25.27	22.29	22.26	22.14
IBM_29_cycle_15	305.51	149.21	283.99	176.02	173.99	183.16	194.88
IBM_29_cycle_20	1828.63	1792.38	1817.36	1745.12	1948.38	1542.37	1764.72
IBM_29_cycle_30	3821.98	10000	3875.55	1013.73	325.29	3505.24	859.39
IBM_29_cycle_50	673.73	10000	4015.01	647.92	663.63	815.04	758.6
IBM_new_2_cycle_20	214.05	152.21	213.85	127.8	69.35	92.03	207.56
IBM_new_2_cycle_25	916.42	1020.32	906.14	236.31	226.7	310.5	228.15
IBM_new_5_cycle_20	137.81	31.65	118.62	122.9	75.56	124.26	80.63
IBM_new_6_cycle_20	252.53	245.97	252.3	146.51	140.69	170.39	217.72

Table 4: Experimentation results for Adaptive Mage

		Native	Sign	DL	CCS	BIN	BCP	UNARY
SAT	Time	8662	14579	8609	7726	6702	7057	7933
	Speedup	-	0.59	1.00	1.12	1.29	1.23	1.09
	Min	-	0.10	0.77	0.87	0.83	0.83	0.91
	Max	-	4.35	1.64	3.88	4.04	2.95	4.01
UNSAT	Time	14955	25256	13067	9228	8269	12637	7313
	Speedup	-	0.59	1.14	1.62	1.80	1.16	2.04
	Min	-	0.07	0.17	0.51	0.75	0.41	0.52
	Max	-	4.44	3.90	5.29	11.75	1.33	5.90
ALL	Time	23618	39835	21676	16954	14971	19695	15247
	Speedup	-	0.59	1.09	1.39	1.58	1.18	1.55
	Min	-	0.07	0.17	0.51	0.75	0.41	0.52
	Max	-	4.44	3.90	5.29	11.75	2.95	5.90

Table 5: Summary of Adaptive Mage results

Adaptive Solving - User Guide Excerpt

The following option will be added to the SAT engine section in the RuleBasePE user's guide [20] of the first release that incorporates the adaptive solving algorithm:

Adaptive Solving

This option, when turned on, instructs the SAT Solver to use adaptive solving techniques. This means that during the search the SAT solver will periodically attempt to alter its internal settings in order to accelerate the search. This

option is relevant only for relatively large examples. Even if the option is turned on, there will be no adaptive changing of settings for short runs.

3 Reductions in SmartLoc

SmartLoc is a BDD-based abstraction-refinement engine, used as yet another RuleBase PE engine. SmartLoc is intended mainly for proving the validity of properties, rather than for finding bugs in the design. This section describes improvements made to SmartLoc by incorporating a few model-reduction algorithms into it.

Abstraction Refinement

Abstraction refinement [7] is a process, in which a model is first grossly abstracted, to form a much smaller (and easier to verify) model, and then iteratively refined. At each iteration, a model checker checks the current abstract model. If it can come up with a definite answer (either pass or fail), the process terminates. Otherwise, a refined model is built for the next iteration.

The abstract models are constructed such that a proof by the model checker will also hold on the concrete model. A falsification, on the other hand, must be checked on the concrete model to determine its validity. This however may be easier than model checking the concrete model, as we only have to check whether a falsifying counterexample is also valid on the concrete model.

Whenever a counterexample is found to be spurious (cannot be reconstructed on the concrete model) a refinement process takes place, yielding a less abstract model. The whole abstraction refinement process is demonstrated in Figure 2.

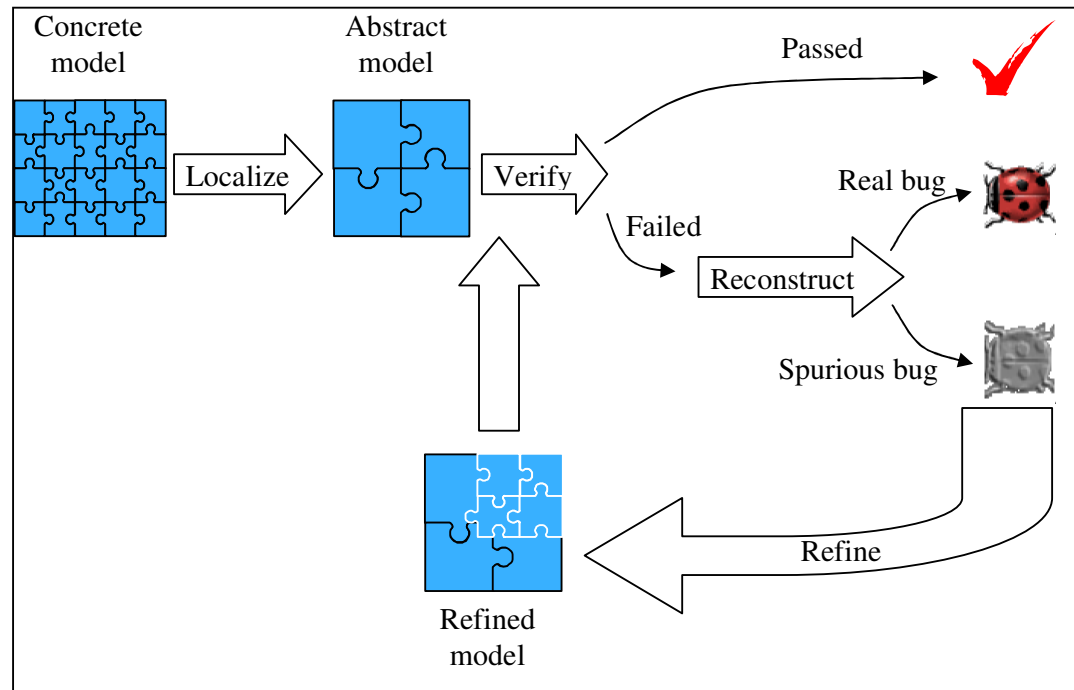


Figure 2: Abstraction refinement

Implementation

SmartLoc forms abstract models by means of *Localization*: The real behaviour of state-variables (usually flip-flops) is replaced by a purely non-deterministic behaviour. This ensures that SmartLoc only **adds** possible behaviours to state variables (and to the model), while also simplifying the model. Since behaviours are only added, the abstract model still exhibits all behaviours of the concrete model. Thus, proving that no bug exists under all possible behaviours of the abstract model also proves that no bug exists under all possible behaviours of the concrete model.

Initially SmartLoc builds the abstract model by combining:

1. The exact behavior of state variables that directly effect the checked assertion
2. Non deterministic behavior for all other variables

Then, whenever a spurious counterexample is found, the model is refined by replacing an abstract non-deterministic behavior of a few state variables with their real behavior. The set of refined state variables is determined by analyzing the spurious counterexample, and finding those variables whose real behavior must be taken into account for this spurious counterexample to disappear.

This process is called “counterexample-guided abstraction refinement”, and was first described in [3].

BDD-Based Reductions

The following model reductions were added to SmartLoc in order to improve performance. All of the reductions are executed at every iteration (on each abstract model).

Flip-flop-Pool Reduction

This reduction finds and reduces flip-flops which are constant throughout the reachable state space. Those signals are saved in a Flip-flop Pool. The Flip-flop Pool is a database containing lists of constant flip-flops for different models, together with signatures of those models. For every abstract model, SmartLoc checks in the Flip-flop Pool which constants are relevant for its current model, and updates the model accordingly.

Equivalence-Pool Reduction

This reduction finds pairs of flip-flops, whose behaviour was found to be equivalent (or negated) throughout the reachable state space. For every such pair, it reduces the model by substituting one signal in a pair by the other one (or its negation) throughout the model description. Those pairs of signals are saved in an Equivalence Pool. Similarly to the Flip-flop Pool, the Equivalence Pool is a database containing lists of pairs of equivalent (or negated) flip-flops, together with signatures of the models in which this information is correct. For every abstract model, SmartLoc checks in the Equivalence Pool which pairs of flip-flops are relevant for its current model, and updates the model accordingly.

Cone-of-influence Reduction

This reduction calculates the *cone-of-influence* of signals appearing in the formula – all the signals that impact the formula – and reduces signals that are out of cone. Intuitively it may appear that this reduction is redundant, because SmartLoc does not include out-of-cone signals in its abstract models. However, in combination with other reductions, this reduction may be very useful (for example, after some signals were found constant, it may reduce parts of the cone of those signals).

Reset Reduction

The reset reduction tries to find signals that behave like a reset, meaning that they are high for several cycles and then remain low forever. If such signals can be found, they are reduced in later cycles.

Over-approximation Reduction

For every abstract model, SmartLoc calculates a few very strong over-approximations of the reachable states of that model, and applies the above reductions (constants, equivalence, cone-of-influence, and reset reduction) on those over-approximations.

Reductions before computing counterexample

If a formula fails during the reachability analysis, then before searching for a counterexample, SmartLoc applies the above reductions (constants, equivalence, cone-of-influence, and reset) on the under-approximation of reachable states computed so far. The results of these reductions are not saved in the Flip-flop Pool or the Equivalence Pool, since they are calculated over the under-approximated model and cannot be applied to the full model.

Experimental Results

We compared the performance of SmartLoc with and without the reductions described above. We chose only properties which should pass, as the main use of SmartLoc is proving properties.

We used a 2.40GHz Intel Xeon machine with 0.5MB cache and 2GB RAM. The timeout was set to 82800 seconds. The results appear in Table 6.

Example	Without Reductions	With Reductions	Speedup
1	44	37	1.18
2	44	44	1
3	53	55	0.96
4	90	76	1.18
5	448	349	1.28
6	59,640	11,595	5.14

7	34994	Timeout	-
8	Timeout	958	-
9	Timeout	Timeout	-
10	Timeout	Timeout	-

Table 6: SmartLoc runtimes with and without reductions

The results show that the reductions do not really save time on easy examples. However, on most time-consuming cases, reductions do improve SmartLoc runtimes. In some cases the improvement is dramatic, while in others the improvement is small or none at all. There is one case, in which SmartLoc with reductions hits the timeout, although without the reductions it was able to solve the problem within the timeout. This is probably due to lengthy but ineffective reductions.

SmartLoc Reduction Options - User Guide Excerpt

The following is a quote from the SmartLoc section in the RuleBasePE user's guide [20].

Use of Over-Approximation Reductions

Before starting model checking, it is possible to compute various over-approximations of the reachable states of the model and to apply reductions on those over-approximated models. These reductions remain sound on the original model, which may, therefore, be simplified, making further model checking more efficient. This option specifies whether to enable these reductions, the default being 'Enabled'.

Note that the more over-approximations are computed, more variables are potentially reduced. This is controlled by the intensity option described next.

Intensity of Over-Approximation Reductions

If you choose to enable over-approximation reductions, this option specifies the intensity level of these reductions (from 1 to 9). The default value is '3'.

Flip-Flop Pool Reductions

If, during model checking or during reductions, a flip-flop was found to have a fixed value throughout the reachable state space, this flip-flop together with its value is written into a Flip-Flop Pool. This pool contains a list of flip-flops that are constant in a given model, together with a signature of that model. This information is used in further runs by flip-flop pool reductions. If a flip-flop has a fixed value throughout the reachable space, it is reduced.

Similar reductions may be applied if two flip-flops are proven to be equivalent. In this case the information is kept in an Equivalence Pool.

This option specifies whether to enable flip-flop pool reductions. The default is 'Enabled'.

Equivalence in Flip-Flip Pool

This option specifies whether flip-flop pool reductions include a signal equivalence check. The default value is 'Disabled', because in some cases, equivalence reductions may make the model more complex.

Reductions before computing a counterexample

If a bug is found during reachability analysis, then a counterexample is searched for within the state space explored so far. As a result, the parts of the model that lie outside the explored state space can be reduced, making the search for a counterexample more efficient. When the counterexample is found, the reduced variables are returned to the model. This option controls whether to enable this reduction, and the default is 'Enabled'.

4 Interpolator

Interpolator is a new SAT-based unbounded model checking engine, capable of both verifying and falsifying safety properties (though far more effective in verifying than in falsifying). The engine is based on the work of McMillan [11], and uses Mage as the underlying SAT solver. In addition, various optimizations were added to the engine, enabling the successful application of the method to many non-trivial examples, as demonstrated in the Experimental Results section below. Interpolator was added to RuleBasePE as yet another engine in March 2005.

Basic Algorithm

The interpolation method exploits the ability of a SAT solver to produce refutations. In bounded model checking, a refutation is a proof showing there is no counterexample of k steps or fewer. Such a proof implies nothing about the truth of the property in general, but does contain information about the reachable states of the model within this bound. In particular, given a proof that there is no counterexample of k steps or fewer, we can generate an interpolant P , which is an over-approximation of the (single) forward image from the initial states, and from which no bad state can be reached within $k-1$ steps. Formally,

$$P(s_1) \supseteq \{s_1 \mid \exists s \text{ INIT}(s) \wedge \text{TR}(s, s_1)\}$$

This over-approximate image operation can then be iterated to compute over-approximations of the sets of states reachable in 2, 3, 4, ... steps. If the property holds on all these over-approximated sets, and if the last set did not introduce new states (fix-point), we can conclude that the property passes.

The algorithm is detailed in Figure 3. A running example, along with the Boolean expression given to the SAT solver, is given in Figure 4.

1. If $\text{INIT}(s) \wedge \text{BAD}(s)$ is satisfiable \rightarrow a bug found!
 2. $R_{\text{new}} := \text{INIT}$
 3. Do
 4. $R := R_{\text{new}}$
 5. Check $R(s) \wedge \text{TR}(s, s_1) \wedge \text{TR}(s_1, s_2) \wedge \dots \wedge \text{TR}(s_{k-1}, s_k) \wedge \bigvee \text{Bad}(s_i)$
 6. If Satisfiable
 7. If $(R == \text{INIT}) \rightarrow$ a bug found!
 8. else \rightarrow Abort loop – go to 2 and restart with a larger k .
 9. else (Unsatisfiable) \rightarrow
 10. Compute interpolant $P(s_1)$
 11. $R_{\text{new}} := R \vee P$
 12. while($R \neq R_{\text{new}}$)
 13. Fix-point \rightarrow the formula passes!

Figure 3: Interpolator main-loop

Iteration 1.1	$\text{Init}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_9, s_{10}) \wedge \bigvee \text{Bad}(s_i)$	UNSAT
Iteration 1.2	$\underbrace{\text{Init}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_9, s_{10})}_{P_{1.2}(s)} \wedge \bigvee \text{Bad}(s_i)$	UNSAT
Iteration 1.3	$\underbrace{P_{1.2}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_9, s_{10})}_{P_{1.3}(s)} \wedge \bigvee \text{Bad}(s_i)$	SAT
Iteration 2.1	$\text{Init}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_{10}, s_{15}) \wedge \bigvee \text{Bad}(s_i)$	UNSAT
Iteration 2.2	$\underbrace{\text{Init}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_{10}, s_{15})}_{P_{2.2}(s)} \wedge \bigvee \text{Bad}(s_i)$	UNSAT
Iteration 2.3	$\underbrace{P_{2.2}(s) \wedge \text{TR}(s, s_1) \wedge \dots \wedge \text{TR}(s_{10}, s_{15})}_{P_{2.3}(s)} \wedge \bigvee \text{Bad}(s_i)$	UNSAT
Iteration 2.4	$\underbrace{P_{2.3}(s)}_{P_{2.4}(s)}$	
Fix-point: $\text{Init} \cup P_{2.2} \cup P_{2.3} = \text{Init} \cup P_{2.2} \cup P_{2.3} \cup P_{2.4}$ – Formula passed		

Figure 4: Running example, including the evaluated Boolean formulas

Optimizations

The following optimizations to the basic algorithm described above were implemented in the Interpolator engine of RuleBasePE.

Frontier set simplification

When computing reachability, it suffices to compute the forward image approximation of the “new” states, rather than the entire reachable set R . Thus, we use the interpolant B in place of R in the unfolded formula.

Minimizing the Unsatisfiable Core

When a SAT solver finds that a given set of clauses is unsatisfiable, it can calculate an *unsatisfiable core* – a subset of the set of clauses, which is unsatisfiable.

The unsatisfiable core is not unique. Every superset of an unsatisfiable core, included in the original set of clauses, is an unsatisfiable core by itself. It is computationally hard to calculate the minimal unsat core.

A simple way to build an unsatisfiable core is similar to building the proof of unsatisfiability (actually, the leaves of the proof tree are the unsat core). We use this method to build the unsatisfiable core, and minimize it by iteratively applying

the SAT solver on the result. Successive applications of the SAT solver each result in a new (smaller) unsatisfiable core. This, potentially, results in a smaller interpolant.

Since this minimization algorithm involves iteratively running a SAT solver, it may be time consuming. There is a tradeoff between the proof size and the run time, and the user can control how much effort to invest in this process.

Efficient construction of the interpolant

In [11] the method of building interpolants requires building the proof of unsatisfiability first. However, the proof might be huge, or even untractable, while the resulting interpolant may be much smaller. We implemented an efficient construction of the interpolant, without holding the whole proof of unsatisfiability in memory. This enabled to prove properties, on which the initial implementation exploded.

Experimental Results

We chose to compare Interpolator to the BDD-based engine Discovery. Discovery implements a BFS search on the state-space by performing a series of *image-computations*. That is, finding all “next states” for a given set of states. Image computation is first performed on the set of initial states, and then iteratively on the sets of newly-found states. If the set of newly-found states contains a buggy state, then a bug was found. If the set of newly-found states becomes empty, then a fix-point was found, meaning the property holds. Discovery is based on McMillan’s SMV [9].

We ran the 2 engines on 35 examples from the benchmarks, on which the checked property holds. This is since Interpolator is expected to perform poorly as a bug hunter. We used a 2.40GHz Intel Xeon machine with 0.5MB cache and 2GB RAM. Timeout was set to 100,000 seconds.

Results are shown in Figure 5. The geometrical mean of the speedup is 5.85, meaning that interpolator proves a property almost 6 times faster (on average) than Discovery. The median of the series of speedups is 2.71, meaning that on 50% of the examples Interpolator performs more than 2.7 times faster than Discovery. The global speedup is 1.61, and the correlation between the two runtimes is -0.09, meaning there is hardly any correlation.

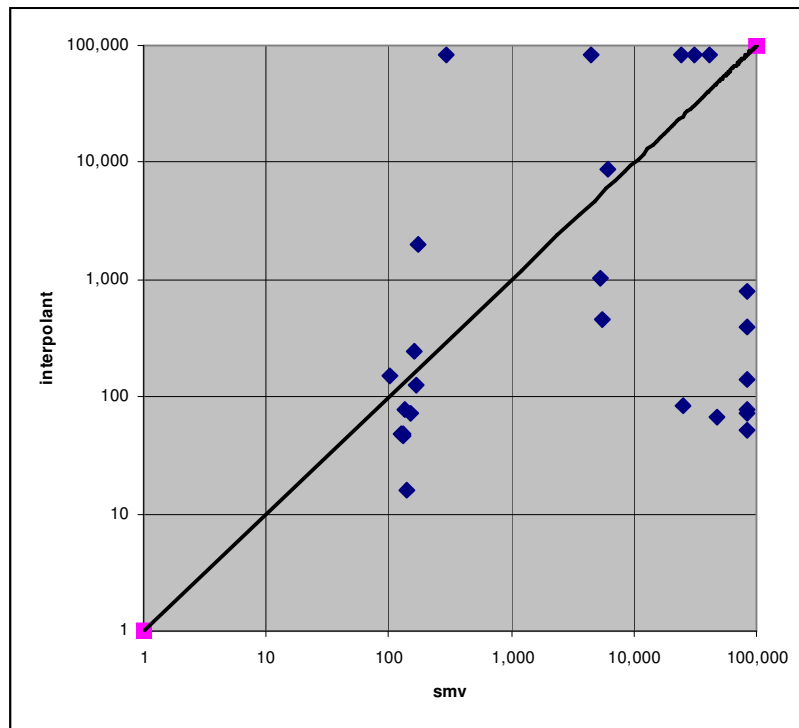


Figure 5: Interpolator vs. Discovery (denoted here as SMV)

It is clear from the results that Interpolator significantly outperforms Discovery, (and probably also any other BDD-based engine) in proving properties.

Interpolator options - User Guide Excerpt

The following is a quote from the Interpolator section in the RuleBasePE user's guide [20].

General

Use time limit for a single SAT solver instance

Whether to activate a time limit for each SAT solver instance.

Time limit for a single instance

In case the previous option is TRUE, this option specifies the time limit in seconds for a single SAT solver instance.

Directory for temp files

This option specifies the directory where the temporary files are saved. If no directory specified, the temporary files are saved in the engine directory.

Model Checking Options

Initial bound

This option specifies the bound with which to run the first over-approximation loop.

Bound growth interval

This option specifies the interval by which to increase the bound for consequent over-approximation loops.

Reduce interpolant

This option specifies whether to apply BDD reductions in an attempt to decrease interpolant sizes.

Decrement unsat core delta

Before building the proof of unsatisfiability, there is a possibility to try to reduce the number of original clauses that appear in the proof (the unsat core). This is done by iteratively applying SAT solver on the unsat core, which results in a new (smaller) unsat core. This loop may be executed until a fixpoint, when the unsat core does not decrease anymore. However, sometimes it requires many iterations, which might be time consuming. Moreover, often the unsat core decreases fast only during the few initial iterations, and then continues decreasing slowly until convergence. Therefore, there is a possibility to specify when to stop the above loop, thus making an optimal tradeoff between time and proof size.

This option specifies the minimal delta (in percents) between two consequent unsat cores sizes, after which we stop decrementing unsat core. Delta=0 means decrementing until the fixpoint. Delta=100 means not decrementing unsat core at all.

5 Conclusions

The purpose of the effort described here was to enhance the capabilities of RuleBasePE. This is measured mainly in terms of run times - have we been able to reduce the run time of RuleBasePE on a significant set of examples? However, there are other measures that need to be considered, such as memory consumption, and, in the case of bounded model checking, the depth that was covered.

All of the enhancements presented in this deliverable have been evaluated using benchmark suites from IBM. We have presented the results for each enhancement separately. Overall, the results are encouraging. The most impact was made by the introduction of the Interpolator engine. This engine was capable of proving the correctness of very large designs, designs that were beyond the capability of other RuleBasePE proving engines.

6 References

- [1] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Conf. on AI*, pages 203-208, 1997.
- [2] 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.
- [3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement, Computer Aided Verification (CAV), July 2000.
- [4] E.M. Clarke, O. Grumberg, D.Peled, *Model Checking*, The MIT Press, 1999.
- [5] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT solver. In *DATE'02*, pages 142-149, 2002.
- [6] A. Kuehlmann, F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference (DAC'97)*
- [7] R. P. Kurshan. Computer-Aided-Verification of Coordinating Processes. Princeton University Press, 1994.
- [8] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5): 506-521, 1999.
- [9] K. L. McMillan. The SMV System DRAFT. Carnegie Mellon University, Pittsburgh, PA, 1992.
- [10] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Press, Norwell, MA, 1993.
- [11] K.L. McMillan. Interpolation and SAT-based model checking. Computer-Aided Verification, LNCS 2725, pages 1–13. Springer, 2003.
- [12] 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.
- [13] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2002.
- [14] O. Shacham, K. Yorav. Adaptive application of SAT solving techniques. In *3rd Intl. Workshop on Bounded Model Checking*, Edinburgh, 2005.
- [15] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *12th International Conference on Computer Aided Verification*, 2000.
- [16] O. Shtrichman. Pruning Techniques for the SAT-based bounded model checking problem. In *13th International Conference on Computer Aided Verification*, 2001.
- [17] H. Zhang. SATO: an efficient propositional prover. In *Conf. on Automatic Deduction*, pages 272-275, 1997.
- [18] Improved Decision Heuristics for High Performance SAT Based Static Property Checking. PROSYD Deliverable 3.2/1.

- [19] Research Report on Improved Symbolic Search Strategies and Model Reduction for Static Property Checking. PROSYD Deliverable 3.2/2.
- [20] RuleBasePE User's Guide, Part 3 - Model Checking with RuleBase Parallel Edition. Available as part of the RuleBasePE product.