

FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Research Report on Improved Symbolic Search Strategies and Model Reduction for Static Property Checking (Deliverable 3.2/2)

Due date of deliverable: January 31, 2005

Actual submission date: March 15, 2005

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: IBM

Revision 1.1

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"/>

Improved Symbolic Search Strategies and Model Reduction for Static Property Checking

Notices

For information, contact nevo@il.ibm.com.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.2/2 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	18.2.05	First draft by authors	Ziv Nevo	All
0.2	8.3.05	Editing and formatting	Lisa Amitai	All
1.0	13.3.05	Final approval by IBM (Project Coordinator)	Cindy Eisner	Cover page, Table of Revisions
1.1	20.3.05	Public version release	Cindy Eisner	Version number

Authors

Rachel Tzoref

Raik Brinkmann

Ziv Nevo

Executive Summary

Model checking is a way to verify logic designs against certain desired properties of the designs. Model checkers exhaustively check all possible input sequences, or search all possible reachable design states. To do this efficiently, the design is usually transformed into a logically equivalent design by a series of transformations called *reductions*. The design after reductions should be easier to explore exhaustively. This research report examines two specific reductions, the first called *retiming reduction* and the second called *quantifier reduction*. We discuss various implementation issues, and check the effectiveness of both empirically. We conclude that using the retiming reduction is effective in only a small set of problems. For other designs, the retiming reduction simply transforms a given problem into an equivalently hard problem. Quantifier reduction is very successful in reducing model checking time, but is usually very time-consuming by itself. Therefore, quantifier reduction, as described in this report, shows no significant improvement in the overall verification time.

Purpose

The purpose of this document is to examine the effectiveness of retiming reduction and quantifier reduction in improving model checking processes and to discuss various issues in their implementation.

Intended Audience

This research report is intended for individuals who work with model checkers, and are particularly interested in reductions. It is assumed that readers are familiar with the notions and terms related to model checking.

Background

Reductions are design transformations, intended for making a model checking problem easier, while keeping some sort of logical equivalence. Classical reductions include syntactic cone of influence and constant propagation.

Retiming is a design transformation that shifts registers across parts of the design. Retiming was first suggested in the context of design synthesis [LS91]. It was later suggested [B02] to use retiming as a reduction for model checking. This research report will examine the effectiveness of the latter idea.

Quantifier reduction is a selective *semantic* cone of influence reduction. The reduction attempts to reduce variables, on which the assertion does not semantically depend. Candidate variables for reduction are quantifiers occurring in PSL assertions. This reduction applies specifically to model checkers that attempt to solve a *bounded model checking* problem, represented as a Boolean function. This research report extends ideas that first appeared in [B01].

Contents

Table of Revisions.....	iii
Authors	iii
Executive Summary	iii
Purpose	iii
Intended Audience	iii
Background.....	iv
Table of Figures	vi
Glossary.....	vii
1 Introduction.....	1
2 Existing Reduction Algorithms.....	2
3 Retiming Reduction.....	3
Theoretical Concepts	3
Implementation Details	4
Solution 1	4
Solution 2	4
4 Retiming Reduction Results	6
5 Quantifier Reduction	9
Motivation	9
Example 1	9
General Solution	11
6 Quantifier Reduction Results.....	13
7 Conclusions.....	16
Retiming Reduction	16
Quantifier Reduction.....	16
8 References.....	18

Table of Figures

Figure 1 – A basic retiming step	3
Figure 2 - A simple memory block	10

Glossary

BDD

Binary Decision Diagram. A data structure upon which many model checkers are based.

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.

Design

A hardware netlist representing the design phase of a chip.

Formal verification

Any mathematical method for verification, capable in principal 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

Another name for a logic cell, which is a functional group of transistors having physical attributes that support a specific semiconductor process technology.

Implementation

The result of the design synthesis process, in which an abstract description of a design entity is converted into gates and the electrical connections between them (signals). An implementation is rendered using components from the foundry-specific design library that represents the target semiconductor process technology. Many of the physical analysis algorithms that were previously found only in standalone physical analysis tools are now being integrated into the implementation flow to help drive the synthesis process.

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. During the development of an electronic system, models are exercised along with signals entering from the outside environment (vectors) to simulate the behavior of the system in software and ensure that it will operate properly before being manufactured in hardware.

Model checking

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 an ASIC design as a set of library-specific cells along with their interconnections.

Property checking

A formal verification technique that verifies that a design does or does not conform to a specified property under all possible sets of legal input conditions. 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.

Synthesis

An EDA process that reads a high-level electronic design description and implements it at a lower level of abstraction. Synthesis tools typically include algorithms for logic optimization and technology retargeting. Legacy synthesis tools produce a gate-level implementation, at which point the design netlist is handed off to the IC layout process. More recent developments have synthesis becoming more tightly integrated with the IC layout process in order to better achieve convergence of goals such as timing.

Verification

The process of falsifying or verifying the functional and performance requirements of a design, be it a chip, board, or system. 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 or all 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.

Verilog HDL

One of two standardized hardware description languages used to specify the structure and behaviour of electronic systems in textual format. Developed in the mid-1980s as a proprietary language and acquired by Cadence Design Systems, it became a de facto industry standard. In the mid-90s Cadence placed it into the public domain and it became a de jure standard promulgated by the Institute of Electric and Electronic Engineers (IEEE). Verilog is also the name of a legacy simulation tool offered by Cadence.

1 Introduction

Model checking algorithms [CGP99] verify logic designs against certain desired properties of the designs. They do this by exhaustively checking all possible input sequences, or by searching all possible reachable design states. For each property, the result of such a search is either "passed", meaning that the property holds in the design, or "failed", usually accompanied by a counterexample showing an illegal behaviour of the design.

An inherent problem in model checking algorithms is the *state explosion* problem, due to the fact that the state-space size is exponential to the number of memory elements. Therefore, it is crucial to apply reduction algorithms to transform industry-sized designs into models that can be handled by industry-strength model checkers. In general, reductions should make a verification problem easier. Usually, reducing the number of memory elements or the number of combinational gates makes a verification task easier.

Reductions must be safe, meaning they must preserve the validity of the properties. A property that is valid in the original model must also be valid in the reduced model, and vice versa. Reductions should also save enough information, so that when the model checker produces a counterexample on the reduced model, this counterexample can then be extended to include all the signals in the original model.

2 Existing Reduction Algorithms

Most industrial model checking tools include a few basic reduction algorithms. The most common reductions are:

- **Cone of influence:** This family of reductions removes logic that does not affect the value of signals appearing in the properties.
 - A *syntactic cone of influence reduction* assumes that each signal is affected by its source signals, and is usually performed by a DFS search, starting from all signals that appear in the properties.
 - A *semantic cone of influence reduction* examines precisely how a signal is affected by its source signals, and eliminates source signals, on which the output signal is independent. Such analysis is much harder than the syntactic analysis, as inspecting all source signals for their contribution to the output is prohibitively expensive in most cases. Therefore good heuristics must be used for deciding when and how such a semantic reduction should be applied.
- **Constant propagation:** This family of reductions identifies signals that are kept constant on every possible execution path. It then replaces these signals with the appropriate constant values, and propagates these constants to all signals affected by the reduced signals, using straightforward propagation rules, and possibly making the affected signals constants as well.
- **Eliminating duplicate logic:** This family of reductions identifies equivalent signals, that is, signals that behave exactly the same on every execution path. The reduction then leaves only one copy of the duplicated logic. All equivalent signals are aliased to it.

The model checking tool reconstructs counterexamples on the original model by simulating the counterexample from the reduced model on the original model, using information produced by the reductions regarding equivalent signals and constant signals.

3 Retiming Reduction

Theoretical Concepts

Retiming is a structural optimization that moves registers across portions of combinational logic, while keeping input-output equivalence. The traditional use of retiming is for synthesis purposes, with the objective of minimizing the cycle delay (by minimizing the longest combinational path) [LS91, MSB91].

For model checking purposes, we are interested in minimizing the number of registers (memory elements) [B02], since the size of the state space is exponential to the number of memory elements.

The sequential circuit is modelled by a directed acyclic graph, in which the vertices represent combinational gates and the edges represent the nets connecting them. Each edge has a corresponding weight representing the number of registers between the two vertices it connects.

A basic step in retiming (Figure 1) is to add (remove) a register to every incoming edge of a certain vertex, and remove (add) a register from every outgoing edge of that vertex. Therefore, a basic step can be performed only if, at the time it is performed, the weight on each outgoing (incoming) edge of the vertex equals at least 1. This imposes the following set of constraints on sequences of retiming steps.

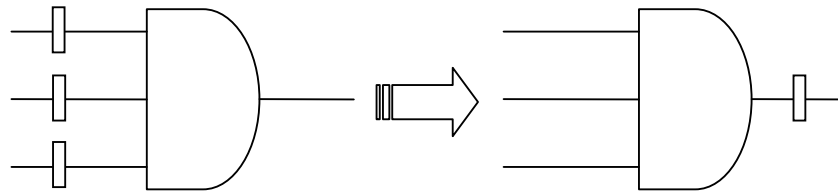


Figure 1 – A basic retiming step

We denote the number of registers moved from each outgoing edge of a vertex v to each incoming edge of v with $L(v)$ (known as the lag of v). We denote the weight of an edge before retiming with $w(e)$.

A legal retiming is an integer assignment to $L(v)$ for each vertex v in the circuit, so that for each edge e in the graph connecting vertex u to vertex v , the following constraint holds: $w(e) + L(v) - L(u) \geq 0$.

This set of constraints guarantees that, after retiming, the graph does not contain edges with negative weights. Thus, the retimed circuit can be physically implemented.

The set of constraints is then fed into an Integer Linear Programming (ILP) solver, together with the desired objective function (in the case of model checking, minimizing the number of registers in the retimed circuit). The ILP solver returns the value $L(v)$ for each vertex v , and the registers are then shifted accordingly to create the retimed circuit.

In retiming for synthesis purposes, there is a requirement to preserve input-output equivalence. To enforce this requirement, a special host vertex is added to the graph [LS91]. The host has outgoing edges to all inputs and incoming edges from all outputs.

This requirement does not exist in retiming for model checking purposes. As mentioned in [B02], registers on inputs and outputs represent temporal offsets and do not impact the validity of the properties in the circuit. These offsets can be restored by temporal shifts in the counterexample obtained on the retimed circuit.

Implementation Details

We implemented our retiming reduction as part of RuleBase/PE.

RuleBase/PE is an industrial-strength formal verification tool, developed by the IBM Haifa Research Laboratories [B96]. It is a platform from which a variety of reduction and verification algorithms are executed, possibly in parallel, over a distributed environment. RuleBase/PE includes hardware compilers and a collection of debugging aids, thus providing a complete solution for the formal verification engineer. RuleBase/PE is especially applicable for verifying the control logic of large hardware designs.

The main issue we had to face is known as the *initialization problem*. We need to initialize the registers in the circuit after retiming so that it is functionally equivalent to the circuit before retiming. We considered two existing solutions.

Solution 1

The first solution we considered was suggested in [TB93], in the context of retiming for synthesis purposes. Therefore, it adds a host vertex as described above. The basic idea is to add the following logic to the circuit before retiming:

1. A reset signal.
2. For every register, the next function becomes: "If reset is high, then init logic; otherwise, original next logic".
3. The reset signal is sourced by the host vertex.

By adding this logic, we guarantee that the set of initial states is reachable from itself. The circuit with the additional logic goes through retiming. The lag of the reset signal determines the number of cycles it takes for the retimed circuit to correct the delay that was created by moving registers backwards, and to return to an original initial state.

Solution 2

Another solution to the initialization problem was suggested in [B02]. This solution is based on decomposing the retimed circuit into two parts:

- **retiming recurrence**, which is basically the circuit after register transformation according to the retiming results, and
- **retiming stump**, which is a partial unfolding of the sequential circuit before retiming.

The purpose of the stump is to initialize the retiming recurrence so that the combined stump and recurrence circuit are functionally equivalent to the circuit

before retiming. The number of appearances of each combinational gate in the stump depends on its lag, as returned by the ILP solver. In order to enable the creation of a stump, all lag values must be non-positive, i.e., registers may only be moved forward. This is not a real limitation, since the fact that we can move registers through inputs and outputs enables any solution for the retiming constraints to be normalized to an equivalent non-negative solution, with the same final register placement.

The stump also forces a limitation on the way properties are checked. Properties must be formulated as part of the circuit. Otherwise, the different property signals may get out of sync. Having the property formulated as part of the circuit enables us to duplicate its logic in the stump, without causing a delay between its signals.

The advantage of the second solution is that, unlike the first solution, logic is duplicated only in the stump, thus it only affects the initial states. In the first solution, we add logic to the next function of all registers, thus affecting all cycles. For model checking purposes, it is common to initialize registers with non-deterministic values. Thus, adding init logic to the next state function of all registers could significantly increase the size of the model. The disadvantage of the second solution is that since there is a partial unfolding of the circuit, the properties should be checked both in the retiming recurrence, and separately in the stump. This means that only safety properties can be checked, provided that they are formulated as $AG(p)$ with an additional automaton, where p is an output of a Boolean expression [BBL98].

We chose to implement the second solution because it is more suitable for retiming for model checking purposes. It doesn't apply the limitations that are irrelevant for model checking, such as adding a host in order to preserve input-output equivalence. In addition, we decided that the potential model-size increase of the first solution is significantly more problematic than the second solution's limitation to only safety properties.

For the ILP solver, we used the same software package that IBM donated to the [COIN] Website. We tried several ILP algorithms and Simplex seemed to work the best. For details, see *Results* on page 6.

The reconstruction task was completed in two phases. First, we shifted each net in the trace according to the number of times it appeared in the stump. Next, we simulated the shifted trace on the circuit before retiming.

4 Retiming Reduction Results

We used seven different hardware designs for benchmarking. Designs were different in size and in functionality. For each hardware design, we chose a small set of PSL assertions, which were written by a verification engineer, for verifying the design. In other words, real-life examples were used.

We used RuleBase/PE, IBM's model checker, for checking the assertions with and without retiming. All runs were performed on a 2.4 GHz Intel Pentium 4 machine, with 2.4GB RAM, running Linux RHEL 3.

We first measured the number of registers, reduced by retiming (Table 1). We then measured the running time of two different verification engines with and without retiming (Table 2). Finally, we tested a possible improvement to the ILP solver, and compared its runtime (Table 3) to the runtime of the solution described in Section 3.

In Table 1, we describe the original number of registers for each design. We then give the number of registers left after applying the reductions described in Section 2, not including retiming. The third column shows the number of registers, further reduced by retiming reduction. From this table, it seems that retiming is successful in its goal, further reducing a significant number of registers. On average, retiming further reduced 20.57% of the registers left after basic reductions.

Design	No. of registers before reductions	No. of registers after reductions without retiming	No. of registers further reduced by retiming	% of registers reduced by retiming
sdd_ibuf	12245	785	185	24
sdl_formal	181	135	34	25
peps	6735	146	24	16
n_lr	1472	449	155	34
cr_issue	1237	302	41	14
bicntl	225	81	14	17
fwd	224	111	16	14
Average				20.57

Table 1

In Table 2, we compare the runtime of verification engines with and without retiming. We measured the runtime for two different verification engines. The first is a SAT-based engine, which translates a bounded model checking problem to a satisfiability problem, and then uses a SAT-solver to search for a satisfying assignment [B99]. The second engine is a BDD-based engine, which uses BDDs to represent the model and to perform an exhaustive search from initial states.

The SAT-based engine performs bounded model checking, and thus, for the assertions satisfied by the model (designs `cr_issue` and `fwd`), a limit of 100 cycles was used. In addition, a timeout of two hours (7200 seconds) was used for both engines.

From the results, it is clear that although the retiming reduction was successful in reducing a significant number of registers, it was usually not successful in reducing verification engines runtimes. In a few cases, it even increased runtime.

We propose three possible explanations for the above conflicting results:

- One reason for the increase in runtime seems to be the large stump needed for coherent initialization. Despite the fact that registers were reduced, inputs, state variables, and combinational logic were added to the model to compose the stump. In particular, it appears that the number of cycles in the stump greatly influences the complexity of building the set of initial states.
- Another reason may be that the registers in the retimed circuit have a more complicated transition function than in the non-retimed circuit.
- A minor reason for the increase in runtime is the additional reconstruction effort. This influences only very short examples, since the additional reconstruction phase takes only a few seconds.

Design	Engine runtime without retiming		Engine runtime with retiming	
	SAT-based	BDD-based	SAT-based	BDD-based
<code>sdd_ibuf</code>	115	timeout	148	timeout
<code>sdl_formal</code>	5	16	5	18
<code>peps</code>	46	73	32	84
<code>n_lr</code>	394	timeout	59	timeout
<code>cr_issue</code>	34	timeout	36	timeout
<code>bicntl</code>	2	3	3	2
<code>fwd</code>	15	25	20	76

Table 2

Table 3 shows the runtime of the retiming reduction itself. The runtime of the retiming reduction is dominated by the runtime of the ILP solver runtime. It can be seen that the runtime increases significantly as the design size grows. In fact, the

retiming runtime dominates the overall verification process (reduction runtime and engine runtime) in some cases.

In an attempt to avoid the large retiming runtimes, we translated the original ILP problem into a network max flow problem, which is usually easier to solve. As a result, the retiming runtimes were reduced by an order of magnitude. These results are preliminary.

Design	Retiming runtime with standard modelling	Retiming runtime with network max flow
sdd_ibuf	324	36
sdl_formal	2	0.16
PEPS	1	0.1
n_lr	8	1
cr_issue	2	0.24
bicntl	1	0.04
fwd	1	0.07

Table 3

5 Quantifier Reduction

Motivation

Using a Bounded-Model-Checking (BMC) approach [B99], the construction of the prove goal consists of constructing an appropriate circuit with one Boolean output signal, and either checking whether there is an input assignment that produces a logic value of one (respectively zero) at this output, or checking whether this output is constantly one (respectively zero), depending on the verification task at hand. In any case, the cone of influence reduction can be applied. A syntactic cone of influence reduction is a straightforward method that is commonly applied. Applying a semantic cone of influence reduction on top of it is more difficult, as inspecting all source signals for their contribution to the output is prohibitively expensive in most cases. Therefore a good heuristic is needed for deciding when and how such a semantic reduction should be applied.

We tried to develop a heuristic and method that utilizes the regular structure of the design and the property commonly observed in the presence of quantifiers, which is reflected in the problem circuit constructed, i.e., in the Bounded-Model-Checking problem.

Regular designs form a large and important class of hardware designs. They are usually constructed by repetitive use of some building blocks. Often many of these building blocks are treated uniformly. Take, for example, the cells of a memory, participants in a communication network, or masters on a bus waiting for a slave device to become available. If this uniformity is reflected by the internal structure of the design, it is called 'regular'. Regular designs can contain many instances of similar building blocks, such as memories, bus-systems, multiplexers, and arbiters and thus have many state variables. For example, even a small memory with 8 address lines and 16 data lines has at least 2^{4096} states. These state variables are then part of the syntactic cone of influence of the BMC-problem. This often leads to a very large search space for the search procedure and to a high degree of symmetry. It can make BMC impractical, as it often leads to exponential runtime behaviour, and consequently to inefficiency.

Example 1

Suppose you want to verify a property for some piece of hardware, which has a very regular structure such as a memory. This regularity is likely to be found in the property as well. Consider, for example, the following simple 256x16-bit memory block in Verilog in Figure 2.

```

`define MEM_DEPTH 8
`define MEM_CELLS 256
`define MEM_WIDTH 16

module memory8 (cs, rw, rs, a, di, do, clk);

input cs;
input rw;
input rs;
input [`MEM_DEPTH-1:0] a;
input [`MEM_WIDTH-1:0] di;
input clk;
output [`MEM_WIDTH-1:0] do;
reg [`MEM_WIDTH-1:0] m [`MEM_CELLS-1:0];
reg [`MEM_WIDTH-1:0] o;
integer i;
always @(posedge clk)
begin
    if (rs)
    begin
        for (i = 0; i<=`MEM_CELLS-1 ; i = i+1)
        begin
            m[i]=`MEM_WIDTH'd0;
        end // for (i = 0; i<=`MEM_CELLS-1 ; i = i+1)
        o = `MEM_WIDTH'd0;
    end
    else // if (clk)
    if (cs)
    begin
        if (rw)
            m[a]=di;
        else
            o = m[a];
        end
    else
        if (!rw)
            o = `MEM_WIDTH'd0;
    end // always @ (posedge clk or negedge reset)
    assign do = o;
endmodule // memory

```

Figure 2 - A simple memory block

To verify that writing data to the memory is correctly implemented, the following PSL-property should be asserted:

```

assert forall ax in {0, MEM_CELLS-1}
{ (rs == 1'b0) &&
  (cs == 1'b1) &&
  ( rw == 1'b1) &&
  (a == ax) }
|=> m[ax] == prev(di)

```

This property states that, independent of which memory cell is selected, the data is available in this memory cell in the next clock cycle. Here the regularity of the

memory reflects within the property as quantifier $\exists x$. Obviously, it should not matter how big the memory is, i.e., how many memory cells are present, because the design is scalable. Unfortunately, size does matter in terms of verification time. In this example, it is easy to scale down the design manually, by reducing the number of memory cells or the width of the data path, and we can be pretty sure that the original design will be correct if the scaled down version is correct. However, for more complicated designs than the one above, this may be different.

In order to automate the scaling process, we reduce the resulting BMC-problem as described below.

General Solution

Let f be a BMC-instance (basically a representation of a Boolean function) structurally depending on some quantifier variable $\mathbf{x} = (x_1, \dots, x_n)$ of size n , where x_i are Boolean variables. (In the example above, we have $\mathbf{ax} = (ax_1, \dots, ax_8)$.) Suppose we want to verify that f is unsatisfiable, i.e., f is constantly 0 (the case of satisfiability can be treated analogously). The following algorithm examines all possible values for \mathbf{x} in order to prove that f does not depend on \mathbf{x} and thus does not belong to the semantic cone of influence.

For each pair of values \mathbf{a} and \mathbf{b} for \mathbf{x} we can either prove or disprove that the cofactors $f_a = f|_{\mathbf{x}=\mathbf{a}}$ and $f_b = f|_{\mathbf{x}=\mathbf{b}}$ are equivalent. Let \mathbf{x}' denote all the remaining Boolean variables that f_a and f_b (syntactically) depend on. As both functions only differ in the values for \mathbf{x} , and both depend on \mathbf{x}' , they should have a similar structure. Therefore, this check should be easy for a combinational equivalence checker.

If f_a and f_b are not equivalent, there is a variable assignment \mathbf{c} for the remaining variables \mathbf{x}' for which $f_a(\mathbf{c})$ and $f_b(\mathbf{c})$ are different, i.e., where either $f_a(\mathbf{c})$ or $f_b(\mathbf{c})$ is 1. This directly leads to a counterexample to the proposition that $f=0$, and the procedure stops.

If $f_a = f_b$, no conclusion can be drawn yet, and another combination of values for \mathbf{x} is examined. Performing this check for all possible values for \mathbf{x} , requires $n-1$ equivalence checks, resulting either in a counterexample or in the fact that all cofactors are equivalent.

If f does not depend on \mathbf{x} , an arbitrary value for \mathbf{x} can be chosen, i.e., it can be removed from the cone of influence of f . This means, it is sufficient to check either one of the n cofactors for unsatisfiability (alternatively, the design and the property can safely be scaled down manually).

In the presence of more than one quantifier, this method is repeated for each of them. The order in which quantifiers are removed does not affect the validity of the process. It may have effect though on how beneficial the reductions are, and on how difficult the individual proofs may be. The size of the cofactors and their number can give some guidance about which quantifiers should be preferred. Also, the reduction can be interleaved for different quantifiers, as the independence of some variable is globally valid. To do so, the possible values of a quantifier are put into a partition, initially containing singleton cells only, one for each value. If some cofactors are equivalent, their classes are merged. If a partition becomes a unit, the problem can be reduced while all other partitions (for other) variables remain valid.

Our solution applies an idea proposed in [B01]; however, with some key differences. They use syntactic measures for equivalence and symmetry, whereas we use a purely semantic approach.

6 Quantifier Reduction Results

We implemented the reduction scheme above within Infineon's property checker gateprop. We used the combinatorial equivalence checking engine of Infineon's formal verification platform CVE. The experiments were conducted on a standard PC with 2,6 GHz clock frequency and 2 Gigabytes of main memory under Linux RedHat 9.

In the experimental setup, we used a simple heuristic for quantifier selection in the presence of more than one quantifier, which was to take the smallest domain quantifiers first. If two quantifiers had the same size, the order of appearance in the assertion was used.

We applied this reduction scheme to two handcrafted memory designs (memory, memory2) as well as to two industrial BMC-examples, an arbiter (arbiter_1111) and a tag ram (tsram256x55). The arbiter was scalable in the number of masters, which defined the size of the domain of a quantifier (2, 4, 8, 16 or 32) in the assertions. The number of timeframes ranged between 1 and 7. The tag ram was not scalable; therefore, only the domain size of the quantifiers could be restricted. The number of quantifiers appearing in the assertions ranged from 1 to 5, and the window of the assertions ranged between 4 and 8.

The results are summarized in Table 4. The first column shows the number of the experiment. Missing numbers indicate runs of the Verilog compiler that were not considered for this analysis. The second column shows the class of the design considered, for example, arbiter_1111_4 is the arbiter design configured for four masters. The following three columns show the results of the property checker without applying the reduction (time in seconds, memory usage in Megabytes, and a remark on the result). The next three columns show the results with reduction in the same way, where the complete time and memory for the reduction and solving the remaining problem is shown. The remarks can be read as follows:

ok = the property holds

fail = the property fails

GivenUp = the prove engine gave up

TimeLimit = the experiment has been killed because the time limit of 1800 seconds has been reached

MemoryLimit = the main memory has been exhausted

#	Class	Without Reduction			With Reduction		
		Time in sec.	Memory in MB	Remark	Time in sec.	Memory in MB	Remark
	Memory	33.11	75	ok	1.21	7	ok
1		1.25	7	ok	1.21	7	ok

Improved Symbolic Search Strategies and Model Reduction for Static Property Checking

2	31.86	63 ok	0.00	0 ok
Memory 2	870.87	91 fail	3602.12	15 fail
4	0.12	7 ok	0.00	0 ok
5	0.10	7 ok	0.04	7 ok
6	6.53	8 ok	0.00	0 ok
7	6.07	13 ok	0.00	0 ok
8	9.05	7 ok	2.08	8 ok
9	323.00	49 ok	1800.00	0 Time Limit
10	526.00	0 Given up	1800.00	0 Time Limit
arbiter_ 1111_2	0.03	30 ok	0.04	30 ok
12	0.01	6 ok	0.01	6 ok
13	0.01	6 ok	0.01	6 ok
14	0.01	6 ok	0.00	6 ok
15	0.00	6 ok	0.01	6 ok
16	0.00	6 ok	0.01	6 ok
arbiter_ 1111_4	0.39	42 ok	0.42	36 ok
18	0.09	6 ok	0.09	6 ok
19	0.06	6 ok	0.06	6 ok
20	0.01	6 ok	0.01	6 ok
21	0.09	6 ok	0.09	6 ok
22	0.09	6 ok	0.09	6 ok
23	0.01	6 ok	0.00	0 ok
24	0.04	6 ok	0.08	6 ok
arbiter_ 1111_16	0.94	7 ok	0.97	7 ok
26	0.94	7 ok	0.97	7 ok
arbiter_ 1111_32	20.56	28 ok	20.55	28 ok
28	5.65	7 ok	4.74	7 ok
29	4.63	7 ok	4.98	7 ok
30	6.02	7 ok	6.56	7 ok
31	4.26	7 ok	4.27	7 ok
arbiter_ 1111_8	0.32	11 ok	0.30	6 ok
33	0.32	6 ok	0.30	6 ok
tsram25 6x55	9936.80	8791 fail	33783.17	2656 fail
35	38.23	236 ok	37.21	15 ok
36	6.18	21 ok	6.25	21 ok
37	37.20	15 ok	43.14	0 ok
38	34.39	15 ok	55.09	0 ok
39	33.28	16 ok	82.00	0 ok
40	33.25	16 ok	138.00	0 ok
41	34.02	15 ok	251.00	0 ok
42	37.55	16 ok	478.00	0 ok
43	6.23	21 ok	12.88	0 ok
44	37.68	15 ok	631.00	0 ok
45	38.71	236 ok	648.00	0 ok
46	56.59	57 ok	1678.00	0 ok
47	5.97	23 ok	18.83	0 ok
48	37.50	16 ok	18.83	15 ok
49	853.00	1625 ok	37.44	2048 Out of

Improved Symbolic Search Strategies and Model Reduction for Static Property Checking

				Memory
50	14.02	14 ok	18.49	13 ok
51	95.00	142 ok	222.00	176 ok
52	115.00	273 ok	1800.00	0 Time Limit
53	142.00	510 ok	1800.00	0 Time Limit
54	192.00	351 ok	1800.00	0 Time Limit
55	245.00	376 ok	1800.00	0 Time Limit
56	271.00	520 ok	1800.00	0 Time Limit
57	546.00	820 ok	1800.00	0 Time Limit
58	912.00	1630 ok	1800.00	0 Time Limit
59	763.00	0 Given up	1800.00	0 Time Limit
60	0.00	12 ok	0.01	12 ok
61	609.00	322 ok	1800.00	0 Time Limit
62	662.00	325 ok	1800.00	0 Time Limit
63	819.00	325 ok	1800.00	0 Time Limit
64	826.00	334 ok	1800.00	0 Time Limit
65	158.00	0 Given up	1800.00	0 Time Limit
66	629.00	0 Given up	1800.00	0 Time Limit
67	697.00	0 Given up	1800.00	0 Time Limit
68	758.00	0 Given up	1800.00	0 Time Limit
69	194.00	494 ok	607.00	356 ok

Table 4

In all examples where the reduction succeeded, the proof time for the reduced problems was significantly smaller than for the original problem. However, the reduction time was much longer than the original time for the whole problem in many cases. Unfortunately, we were not able to tell the reason for this unexpected behaviour. Possible reasons include the following:

- The supposition of structural similarity between the cofactors is wrong. In particular, it might be possible that their variable support becomes almost disjoint after they were optimized by syntactic cone reduction on the individual instances (cofactors). This could be tackled by finding a variable correspondence.
- The heuristics for determining a good variable order is insufficient. Here, a more thorough analysis of the effect of the reduction prior to the semantic cone analysis could help.
- The equivalence-checking engine is not robust enough to deal with partially matched functions.

7 Conclusions

In this research report, we examined the impact of retiming reduction and quantifier reduction on assertion based formal verification. Both reductions were implemented within industrial model checking platforms, and their empirical effectiveness was tested.

Retiming Reduction

We implemented retiming within RuleBase/PE, IBM's model checking platform, as yet another reduction, among other reductions that are used in the verification process. We used an ILP solver, developed by IBM, to solve the retiming problem.

We measured different runtimes, including the runtime of solving the retiming problem and the runtime of different verification engines on a retimed circuit and on a non-retimed circuit. We also measured the number of registers reduced by the retiming reduction.

The results show a significant reduction in the number of registers by the retiming reduction. However, the verification engines runtime does not improve significantly, and in some cases even deteriorates. In addition, the retiming reduction itself may take a significant time to complete, though this can probably be solved by using different ILP algorithms.

We conclude that if we use the implementation described in this report, retiming can only be helpful for limited types of designs (probably pipelines and such). In most other cases, the retiming reduction simply transforms the problem to what seems to be an equivalently hard problem.

Future work should concentrate on changing the minimization target, as it seems that reducing the number of registers by itself is not a sufficient measure. The minimization target should take into account the size of the stump as well as the number of reduced registers. In addition, other ILP algorithms should be tested, in order to find a more suitable algorithm for this specific family of ILP problems. Identifying designs for which the retiming effort is worthwhile is also a desired direction. Finally, there are several retiming-oriented optimizations [LS91, B02], such as fan-out register sharing and fan-in register sharing, which may further reduce the number of registers in the retimed circuit. These were not implemented in the described solution.

Quantifier Reduction

We implemented quantifier reduction within gateprop, Infineon's property checker. We measured the runtimes for the property checker with and without reduction for different verification problems.

The results show no significant reduction in the overall verification time. However, the time for proving the reduced problems decreased significantly. Therefore, improving the reduction procedure may improve the overall result.

Future work should concentrate on improving the equivalence check for cofactors and on improving the order in which the quantifiers are considered. In addition, syntactic approaches like proposed in [B01] should be applied.

8 References

- [B96] I Beer, et al, *RuleBase: An Industry-oriented Formal Verification Tool*, Proceedings of the 33rd annual conference on Design automation, pp. 655-660, 1996.
- [B99] A. Biere, et al, *Symbolic Model Checking without BDDs*, Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, March 1999.
- [B01] Brinkmann, R. *Using Symmetry for Problem Reduction in Bounded-Model-Checking on the Register-Transfer-Level*. Proc. of SymCon'01 – Symmetry in Constraint Satisfaction Problems, CP'01 Post-Conference Workshop, 2001.
- [B02] J. Baumgartner, *Automatic Structural Abstraction Techniques for Enhanced Verification*, PhD Thesis, University of Texas, Austin, TX, December 2002.
- [BBL98] I Beer, S. Ben-David, A. Landver, *On The Fly Model Checking of RCTL Formulas*, Proceedings of the 10th International Conference on Computer Aided Verification, pp. 184-194, 1998.
- [CGP99] E.M. Clarke, O. Grumberg, D. Peled, *Model Checking*, The MIT Press, 1999.
- [COIN] <http://www-128.ibm.com/developerworks/opensource/library/os-coin.html>
- [LS91] C.E. Leiserson, J.B. Saxe, *Retiming Synchronous Circuitry*, Algorithmica, 1991.
- [MSB91] S. Malik, E. Sentovich, R. Brayton, *Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques*, IEEE Transactions on Computer-Aided Design, vol. 10, No. 1, January 1991.
- [TB93] H. Touati, R. Brayton, *Computing the Initial States of Retimed Circuits*, IEEE Transactions on Computer-Aided Design, vol. 12, No. 1, January 1993.