High-level Methods for OBDD-based Sequential Verification

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich IV
der Universität Trier

von
Dipl. Inform. Christian Stangier

2002

1. Berichterstatter: Prof. Dr. Christoph Meinel (Universität Trier)
2. Berichterstatter: Prof. Dr. Werner Damm (Carl von Ossietzky Universität Oldenburg)

Datum der Disputation: 29. November 2002

Contents

Introduction **1**

1 Preliminaries **7**

1.1 Boolean Functions 7

1.1.1 Operations on Boolean Functions 7

1.1.2 Data Structures for Boolean Functions 8

1.2 OBDDs – Ordered Binary Decision Diagrams 8

1.2.1 Manipulation of OBDDs 10

1.2.2 Efficient Synthesis of OBDDs 10

1.2.3 Influence of the Variable Order on the OBDD-Size 12

1.2.4 Optimization of the Variable Order 13

1.2.5 Sifting 14

1.3 Verification of Combinatorial Circuits 14

1.4 Formal Verification of Sequential Systems 15

1.4.1 Equivalence Check for Two Finite State Machines 15

1.4.2 Reachability Analysis 16

1.4.3 Symbolic Model Checking 18

1.5 Software Packages 20

2 A Case Study in Formal Verification **23**

2.1 Formal Verification in an Industrial Setting 23

2.1.1 Protocol Compiler 24

2.2 An Assertion Checking Methodology for Protocol Compiler 24

2.2.1 Verification Flow 25

2.2.2 Example RS232 25

2.2.3 Example Cow-Stomach 28

2.2.4 Dealing with Inputs during Verification 31

2.3 Conclusion and Open Problems 33

3 Partitioning **35**
3.1 The Partitioning Problem . . . 35
3.1.1 Common Partitioning Strategy . . . 36
3.1.2 Related Work . . . 38
3.2 RTL Partitioning . . . 39
3.2.1 Hardware Description Languages . . . 39
3.2.2 RTL based Partitioning Heuristic . . . 39
3.2.3 Discussion of the Method . . . 41
3.2.4 Experimental Results . . . 42
3.3 Group Partitioning . . . 48
3.3.1 Dependency Matrices . . . 48
3.3.2 The Grouping Algorithm . . . 48
3.3.3 Reordering of Clusters . . . 51
3.3.4 Experimental Results . . . 53

4 Hierarchization **57**
4.1 Hierarchical Image Computation . . . 57
4.1.1 Hierarchical Partitioning . . . 58
4.1.2 Hierarchical Image Computation . . . 60
4.2 Dynamic Conjunction Scheduling . . . 62
4.2.1 Dynamic Scheduling Heuristic . . . 62
4.2.2 Experimental Results . . . 64
4.3 Hierarchical Group Partitioning . . . 67
4.3.1 Discussion of the AndExist Algorithm . . . 67
4.3.2 Modular Grouping of the Transition Relation . . . 69
4.3.3 Experimental Results . . . 72
4.4 Conclusion . . . 76

5 Reordering **77**
5.1 Sample Sifting . . . 78
5.1.1 Basic Concept . . . 78
5.1.2 Sample Sifting Strategy for Symbolic Model Checking . . . 78
5.1.3 Experimental Results . . . 85
5.2 Block Restricted Sifting . . . 90
5.2.1 Basic Concept . . . 90
5.2.2 BRS Strategy for Symbolic Model Checking . . . 92
5.2.3 Experimental Results . . . 94
5.3 Conclusion . . . 94

6 Conclusion **97**

A Additional Tables **99**

References **107**

Introduction

Motivation

During the last years we are facing an enormous increase in utilization of digital circuits: Many applications like information processing, telecommunication, network computing, automotive and industrial control demand for even more powerful and faster circuits.

The computer industry keeps pace with this demand by increasing the system integration of VLSI (Very Large Scale Integration) designs. Moore's law [Moo65], which says that the number of transistors on a chip doubles every 18 month is still valid and an end of this growth is not in sight.

On the other hand there is a growing need for proving correctness of complex designs for the following two reasons:

1. Simulation is the conventional method for validation of digital designs. This method loses its ability to guarantee correctness, if designs grow larger and become more complex.
2. Economic, safety and security reasons request for mathematical (i.e. formal) proofs of the correctness of digital systems.

The famous *Pentium bug*, an error in the floating division unit of the Intel Pentium I processor, which was responsible for a $475 Million loss in Intel's revenue, shows the importance of proving correctness of digital hardware.

But, there are more important reasons to insist on the correctness of digital circuits than the profit of a company:

- The automotive control of an airplane should be correct under any circumstance.
- The control of a satellite should be at least resettable (there is no way to switch the power off – and back on).
- Hardware realizations of cryptographic protocols should be guaranteed to be functionally correct.

Virtually any aspect of our daily live is affected by digital circuits with an increasing tendency and the above list gives only a few examples of possible applications for formal verification.

A central problem in CAD (computer aided design) of digital designs is the representation of Boolean functions. Within the last decade ordered binary decision diagrams (OBDDs) [Bry86] have become one of the most popular data structures in this area.

The classical application for OBDDs is the formal verification of combinatorial circuits. Here, the equivalence of the specification and an implementation of a given circuit has to be checked. Recently, the focus of the interest in research and industry has moved to the application of OBDDs to sequential systems. Many design tools use OBDDs to represent finite state machines. The emerging need for formal verification of sequential systems requires the use of highly sophisticated data structures and algorithms.

Formal Verification of Sequential Systems

Formal verification is the mathematical proof that an implementation of a circuit C fullfils its specification S. In combinatorial verification this is done by checking the equivalence of the representations of C and S. OBDDs are well suited for this task, because they represent Boolean functions in a compact way and the equivalence check is trivial. The term *sequential verification* denotes formal verification of synchronous finite state machines (FSMs) or systems of communicating FSMs. Various sequential verification techniques exist, e. g. the functional equivalence of two FSMs can be tested. Unfortunately, often a formal specification S of a FSMs is not given. In this case, certain properties that the FSM should satisfy are tested. These properties include: Assertions, deadlocks, livelocks, etc. The main applications in sequential verification are *assertion checking* and *model checking.* [CES86]

A major problem in sequential verification is the so called *state space explosion problem*, resulting from huge state space of complex FSMs that cannot be represented explicitely. This problem can be circumvented by representing state sets symbolically by using OBDDs. Together with the efficient algorithms for manipulation of Boolean functions OBDDs enable efficient *symbolic breadth first traversal* for reachability analysis, assertion checking and symbolic model checking [McM93].

Usage of OBDDs does not solve all problems in sequential verification. The size of the OBDD representation and the efficiency of the algorithms heavily influences the performance of the applications.

Two of the main areas for optimization in OBDD-based sequential verification are

- Partitioning of the transition relation of the FSM to allow efficient reachable states computation, and
- ordering of state- and input-variables to reduce the size of the OBDD-representation.

Scope of the Thesis

This work covers practical aspects of OBDD-based sequential verification. It is structured in a top-down fashion: We start by presenting a formal verification methodology at the *user-level* of a design and verification tool. Then, we present algorithms at the *application-level* of OBDD-based formal verification, i. e. reachability and image computation. Finally, we move down to the *representation-level* and deal with improvement of the OBDD-representation size.

The mayor area of this work is *Algorithmic Engineering.*

OBDD-based sequential verification moved into the focus of research for various reasons:

- Designers realize that their simulation based validation approaches are no longer capable to find all bugs and search for more reliable techniques.
- OBDD-based techniques for combinatorial verification are well investigated and one hopes to transfer these techniques to sequential verification.
- OBDDs are already integrated in many design tools simply for function representation purposes, and thus they are ready to use for formal verification.
- The interest in formal verification has generally increased.

Some of the points mentioned above are challenges: Enabling a designer to verify complex properties of a design, requires almost *push-button-technology.* Some of the points actually are problematic: Algorithms that are good in combinatorial verification do not necessary work in sequential verification.

For these reasons we propose the utilization of

High-Level Methods

to improve the efficiency and ease the applicability of OBDD-based sequential verification.

The main contributions of this work are:

1. An assertion checking methodology, integrated in the verification flow of the high-level design and verification tool Protocol Compiler.
2. New approaches for partitioning of transition relations of complex finite state machines, that significantly improve the performance of OBDD-based sequential verification.
3. Dynamic variable reordering techniques that drastically reduce the time required for reordering without a trade-off in OBDD-size.

Overview of the Thesis

The present work is structured as follows:

We start this work in Chapter 1 with recapitulating the basic concepts, state-of-the-art data structures and algorithms and give the most common applications in formal verification in VLSI design.

In Chapter 2 we conduct a case study on formal verification in an industrial setting. We develop a methodology for assertion checking, which is a subproblem of model checking and apply it to two examples: A simplified RS232 transceiver and a pipelined FIFO-like buffer. The study shows how to integrate formal verification into a typical verification flow. The study also shows the limitations of this approach and serves as a motivation for the research presented in the following chapters.

Chapter 3 deals with the partitioning problem, i. e. building a partitioned transition relation of a finite state machine. The transition relation allows image computation, and thus reachable states computation and model checking. We start by presenting a heuristic that uses explicit high-level information of the design under consideration. The heuristic is called

RTL-method, because it utilizes register transfer level (RTL) constructs of the description language Verilog. The outcome is an increased efficiency of the formal verification algorithms.

In the second part of Chapter 3 we develop, motivated by the success of the RTL-method, a heuristic that is independent of high-level information. This heuristic imitates the effects of the RTL-method that cause the improved efficiency. This heuristic is called *Group-method.*

The heuristics presented in Chapter 3 produce a somewhat "flat" partitioning, but nowadays designs are structured – and developed – in a hierarchical fashion. Thus it seems obvious to adapt hierarchy for the partitioning problem. In Chapter 4 we present algorithms that build a hierarchical partitioned transition relation and allow hierarchical image computation. The first heuristic in Chapter 4 for partitioning is based on high-level information and is called due to its modular structure *RTLMOD-method.* We also tackle another important problem in OBDD-based image computation: The conjunction scheduling problem. We present a conjunction scheduling heuristic that is based on the hierarchical image computation. A highlight of this heuristic is that it is able to change the schedule dynamically.

In the second part of Chapter 4 we develop – similar to Chapter 3 – a heuristic that is independent of explicit high-level information. This heuristic is called *GROUPMOD-method.*

In Chapter 5 we focus on a problem on the representation layer of formal verification, i. e. the optimization of the OBDD-representation by dynamic variable reordering.

Reordering of variables is the only way to improve the size of the OBDD-representation. In the last years the dynamic *Sifting* approach turned out to produce the best average results. But, this approach can be very time consuming, and time is the most valuable resource in model checking. We present two acceleration techniques for variable reordering: *Sample Sifting for symbolic model checking* and *Block Restricted Sifting for symbolic model checking.* The first method utilizes high-level information of the represented functions and the state of the computation, while the second only is application dependent and thus easier to implement.

Finally, Chapter 6 concludes this thesis with summarizing its key results and an outlook on possible future work.

Publications

Parts of this thesis have been published in similar form:

- C. Stangier and U. Holtmann: *Applying Formal Verification with Protocol Compiler*, Proc. of EUROMICRO Digital System Design (DSD'01), 2001.
- Ch. Meinel and C. Stangier: *Speeding Up Image Computation by using RTL Information*, Proc. of Formal Methods in CAD (FMCAD'00), LNCS 1954, Springer, 2000.
- Ch. Meinel and C. Stangier: *A New Partitioning Scheme for Improvement of Image Computation*, Proc. of ASP Design Automation Conference (ASPDAC'01), 2001.
- Ch. Meinel and C. Stangier: *Hierarchical Image Computation with Dynamic Conjunction Scheduling*, Proc. of IEEE Int. Conf. on Computer Design, 2001.
- Ch. Meinel and C. Stangier: *Modular Partitioning and Dynamic Conjunction Scheduling in Image Computation*, In Proc, ACM/IEEE Int. Workshop on Logic and Synthesis, 2002.

- Ch. Meinel and C. Stangier: *Speeding Up Symbolic Model Checking by Accelerating Dynamic Variable Reordering*, Proc. of IEEE 10th Great Lakes Symopsium on VLSI, 2000.

Some parts have been published in a survey paper:

- Ch. Meinel and C. Stangier: *Data Structures for Boolean Functions. BDDs - Foundations and Applications*, In: Computational Discrete Mathematics, Ed. H. Alt, LNCS 2122, Springer, 2001.

Acknowledgments

First of all, I would like to thank my advisor Prof. Christoph Meinel for all his support and advice over the last years.

Many thanks are devoted to Ulrich Holtmann from Synopsys, who put this thesis on the right track.

I have to thank Harald Sack for always being motivating and listening to all my complaints.

Thanks to my parents for being my parents (and not someone else's parents). And, thanks to Tommy Stumpe, for being a great friend and providing me with vital doses of Rock'n'Roll (in every sense).

Furthermore, I wish to thank all the members of Prof. Meinel's working group, who helped and supported me during my PhD.

Chapter 1

Preliminaries

This chapter covers the basic concepts that underly this work. In particular we introduce *ordered binary decision diagrams* (OBDDs), the state-of-the-art data structure for representation of Boolean functions. We discuss properties of OBDDs and present the basic algorithms for manipulation of OBDDs. In the Section 1.4 we introduce the main applications of OBDD-based sequential formal verification, i. e. state space traversal, equivalence checking and symbolic model checking. The chapter is concluded by an description of the software packages we used to implement our algorithms and heuristics.

1.1 Boolean Functions

Definition 1.1

- The set $\{0, 1\}$ is denoted by $\mathbb{B}$.
- The set of all Boolean functions $\mathbb{B}^n \to \mathbb{B}, n \in \mathbb{N}$ is denoted by $\mathbb{B}_n$.

In computer-aided design, Boolean functions $f \in \mathbb{B}_n$ are of central importance for describing the switching behavior of digital circuits. Hence, those functions are also called *switching functions.* By introducing a suitable 0-1-encoding, all finite problems can – at least in principle – be modeled by means of switching functions. The great importance of switching functions stems from the possibility to obtain substantially simplified, optimized and with optional properties provided circuits by applying optimization techniques during the design process. In the area of VLSI circuits this task is performed by CAD systems. But, before optimization techniques can be applied, a way to represent the switching functions themselves uniquely and as efficiently as possible in computers has to be found.

The representation of a Boolean function is not an end in itself: On the basis of such a representation the various algorithms of CAD systems are performed. Hence, it is important to analyze the ability of the representation for supporting the basic operations of the algorithms.

1.1.1 Operations on Boolean Functions

The following list gives the basic operations on Boolean functions $(f, g \in \mathbb{B}_n)$:

Satisfiability test (SAT): Test, whether there exists an assignment $a \in \mathbb{B}^n$ of the variables of f with $f(a) = 1$.

Equivalence test: Check, whether $f \equiv g$ for two functions f, g.

Evaluation: Compute $f(a)$ for a given assignment $a \in \mathbb{B}^n$

Synthesis: Compute a representation P_h of $h = f \otimes g$ for an arbitrary binary Boolean operator $\otimes$.

Replacement by constants: Compute a representation P_h for $h = f_{|x_i=a}$ (i. e. the input x_i in f is replaced by the Boolean constant $a \in \mathbb{B}$).

Replacement by function: Compute a representation P_h for $h = f_{|x_i=g}$ (i. e. the input x_i in f is replaced by the Boolean function g).

Quantification: Compute a representation P_h for $h = \forall x_i : f$ (universal quantification) resp. $h = \exists x_i : f$ (existential quantification) of f.

Minimization: Given a representation P_f of a function f. Compute a minimized representation P'_f.

1.1.2 Data Structures for Boolean Functions

Why are special data structures for the representation of Boolean function needed? Well known representations for Boolean functions include:

Circuits: The elementary SAT problem is NP-complete for circuits and the minimization problem is NP-hard.

Boolean formulas: The equivalence test is co-NP-complete for Boolean formulas.

Truth tables: All operations can be performed efficiently, but truth tables always require exponential space.

Conjunctive Normal Form (CNF): Even simple functions often have only an exponential representation in CNF. The SAT problem is NP-complete for CNF.

Disjunctive Normal Form (DNF): The problems with DNF are similar to CNF, but SAT is in P.

Branching Programs (BPs): Many functions can be represented very efficiently using branching programs but already the SAT problem is NP-complete for BPs.

For definitions of these data structures and proofs of the lower bounds see [MT98]. All these data structures fail for use in electronic design automation either for reasons of space complexity (e. g. truth tables) or time complexity of the operations (e. g. branching programs).

1.2 OBDDs – Ordered Binary Decision Diagrams

In 1986, by introducing *ordered binary decision diagrams (OBDDs)*, Randall E. Bryant got ahead a fundamental step in the search for suitable data structures in circuit design [Bry86, Bry92]. Bryant's OBDDs combine two advantages: the new established data structure is not only quite space efficient but can also be handled efficiently from the algorithmic point of view.

Definition 2.1 An *Ordered Binary Decision Diagram* (OBDD) P for a Boolean function $f \in \mathbb{B}_n$ is a directed acyclic graph consisting of inner nodes labeled by Boolean variables and sinks labeled by the Boolean constants 1 and 0. Each inner node has two outgoing edges: the 1-edge and the 0-edge. The OBDD has a starting node called root. The computation of $f(a_1, \dots, a_n)$ follows a path from the root to a sink, where on a node labeled by x_i the input bit a_i is tested. If $a_i = 1$, the path follows the 1-edge, otherwise the 0-edge. The value of the reached sink determines the value of $f(a_1, \dots, a_n)$. On a path from the root to the sink, each variable occurs at most once. The variables on a path respect a given order, which is (possibly after renaming) $x_1, \dots, x_n$. For an edge leading from a node labeled by x_i to a node labeled by x_j it follows that $j > i$.

An OBDD with more than one root node (i. e. representing $f : \mathbb{B}^n \to \mathbb{B}^m, m \in \mathbb{N}, m > 1$) is called a *shared* OBDD [MIY90]. In practice all functions to be represented are kept in one single shared OBDD. For simplicity we stay with the term OBDD.

Figure 1.1 gives two examples for OBDDs for the Boolean function $f = bc + a\overline{b}\overline{c}$ w. r. t. the variable order $a < b < c$.

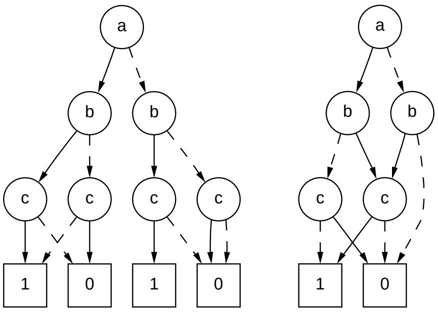


Figure 1.1: Two OBDDs representing $f = bc + a\overline{b}\overline{c}$

From the Shannon decomposition

$$f = x \cdot f_{|x=1} + \overline{x} \cdot f_{|x=0}$$

one can derive the first important property of OBDDs:

Property 2.2 Universality: Any Boolean function $f \in \mathbb{B}_n$ can be represented by an OBDD w. r. t. any predefined variable order π.

For a proof see [SW93].

If we give up the restrictions on the variable order and the read-once property we get more general *decision diagrams*. Like in many other representations general decision diagrams have difficulties in handling Boolean functions caused by the missing uniqueness. By using a surprisingly simple reduction mechanism, for OBDDs this problem can be solved very elegantly. Obviously, the following two reduction rules keep the represented function invariant:

Elimination rule: If 1- and 0-edge of a node v point to the same node u, then eliminate v, and redirect all incoming edges from v to u.

Merging rule: All terminal nodes with a given label are merged to one node, redirect all incoming edges to this node. If the non-terminal nodes u and v are labeled by the same variable, their 1-edges lead to the same node and their 0-edges lead to the same node, then eliminate one of the two nodes u, v, and redirect all incoming edges to the remaining node.

The elimination rule and the merging rule are illustrated in Figure 1.2.

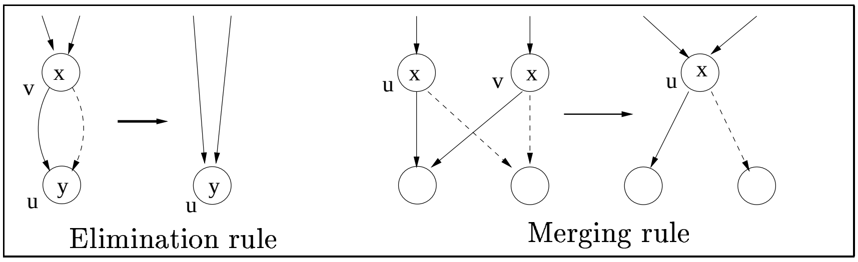


Figure 1.2: Reduction rules for OBDDs

Definition 2.3 An OBDD is called reduced, if none of the two reduction rules can be applied.

It is easy to see that the right OBDD in Figure 1.1 is reduced. Regarding the algorithmic properties of reduced OBDDs, the following property of canonicity is of basic importance:

Property 2.4 Canonicity: With respect to a fixed variable order π, the reduced OBDD of a Boolean function f is uniquely determined.

For a proof also see [SW93].

Besides universality and canonicity, OBDDs have a third fundamental property, which makes OBDDs such a successful data structure for representation of Boolean functions: It is the efficiency of algorithmic manipulation.

1.2.1 Manipulation of OBDDs

OBDDs are the only data structure for the representation of switching functions, whose representation size is not exponential in the number of variables for all functions (like truth tables) and that has deterministic polynomial time algorithms for all important operations.

In Figure 1.3, the runtime and space requirements for these operations are given ($|P_f|$ denotes the number of nodes in the OBDD P for the function f, which depends on n variables and $a \in \{0,1\}^n$). All OBDDs have to respect the fixed variable order π. (Results due to [Bry86, SW93])

It is worthwhile mentioning that most operations (except synthesis and replacement by function) have time and space requirements linear in the size of the OBDD. Together with an efficient implementation OBDDs form a powerful data structure.

1.2.2 Efficient Synthesis of OBDDs

By $\otimes$ we denote an arbitrary binary Boolean operation, e. g. the conjunction or the disjunction. In order to compute the OBDD P_h of $h = f \otimes g$ from the OBDD representations

Operation	Runtime	Space
Satisfiability test ($\exists a, f(a) = 1$)	$O(\lvert P_f\rvert)$	$O(\lvert P_f\rvert)$
Equivalence test ($f \equiv g$)	$O(min(\lvert P_f\rvert, \lvert P_g\rvert))$	$O(\lvert P_f\rvert + \lvert P_g\rvert))$
Evaluation ($f(a)$)	$O(n)$	$O(P_f)$
Composition ($f \otimes g$)	$O(\lvert P_f\rvert \cdot \lvert P_g\rvert)$	$O(\lvert P_f\rvert \cdot \lvert P_g\rvert)$
Replacement by constant ($f_{x_i=c}$)	$O(\lvert P_f\rvert)$	$O(\lvert P_f\rvert)$
Replacement by function ($f_{x_i=g}$)	$O(\lvert P_f\rvert^2 \cdot \lvert P_g\rvert)$	$O(\lvert P_f\rvert^2 \cdot \lvert P_g\rvert)$
Quantification (e. g. $\exists x_i : f = f_{x_i=1} \vee f_{x_i=0}$)	$O(\lvert P_f\rvert^2)$	$O(\lvert P_f\rvert^2)$
Reduction (minimization of P_f w. r. t. π	$O(\lvert P_f\rvert)$	$O(\lvert P_f\rvert)$

Figure 1.3: Complexity of OBDD operations

P_f and P_g of two functions f and g, one uses Shannon's decomposition w. r. t. the leading variable x in the variable order π:

$$h = f \otimes g \;=\; x\ (f|_{x=1} \otimes g|_{x=1}) \;+\; \overline{x}\ (f|_{x=0} \otimes g|_{x=0}),$$

where $f|_{x=1}$ is the subfunction that results from f after replacing the variable x by the constant 1. By repeated application of this decomposition, an OBDD representation P_h of the function h can be computed.

In an OBDD P_f every node represents a subfunction f' of f. If the node that represents f' is marked with x_i, its successors represent $f'|_{x_i=1}$ resp. $f'|_{x_i=0}$. For the representation of any subfunction in P_h two pairs of nodes from P_f and P_g have to be computed. If one would simply follow the Shannon decomposition, where the number of computations doubles on each level, 2^n pairs would be computed (for n variables). But, only $(|P_f| \cdot |P_g|)$ different pairs exist. Thus, re-computation of pairs has to be avoided, as different subfunctions may be represented by the same node.

The already computed results from earlier stages are being recalled from a *computed-table*. In this way, the originally exponential number of decompositions is now bounded by the product of the two OBDD-sizes.

To increase the usage of the computed table all synthesis operations are mapped to a single operation, the so called if-then-else operator (ITE):

$$ITE(f, g, h) = f \cdot g + \overline{f} \cdot h.$$

For example $h = f \cdot g$ maps to $h = ITE(f, g, 0)$. Because of the huge number of ITE operations during synthesis, the computed table is usually implemented as a hash based cache to reduce memory consumption. The cache exploits the locality of reference, i. e. results are usually only reused shortly after their computation.

Another helpful construction is the usage of a *unique-table* which holds in a hash-table all already represented nodes. Before a node is created it is checked in the unique-table whether an functionally equivalent node already exists. This technique implements the merging rule. Together with an immediate check for the elimination rule the constructed OBDDs are always reduced and there is no need to call the reduction operation explicitely.

Remark:

- In a reduced OBDD the complexity for the satisfiability test (SAT) reduces to: $O(1)$, because the 0-function is uniquely represented by the 0-sink.

- In a reduced shared OBDD the complexity for the equivalence test ($f \equiv g$) reduces to $O(1)$, because equivalent functions are represented by the same node, i. e. the equivalence test is a pointer comparison.

Construction of OBDDs: Symbolic Simulation

The process of constructing an OBDD is called *symbolic simulation* of the circuit to be represented. Symbolic simulation is based on the synthesis operation:

Starting with the (trivial) OBDD representations of the input nodes one constructs, in topological order, OBDDs for each gate from the OBDDs of the corresponding predecessor gates.

Of course, it may happen that the OBDDs of the circuits are quite large. However, many circuits of practical interest inherently contain much structure – hence, the reduction rules of the OBDDs cause the graphs describing the circuit to remain small.

1.2.3 Influence of the Variable Order on the OBDD-Size

The size of an OBDD and hence the complexity of its manipulation heavily depends on the underlying variable order. An example is shown in Figure 1.4. With respect to the variable order $a_1, b_1, \ldots, a_n, b_n$ the function

$$a_1b_1 + a_2b_2 + \ldots + a_nb_n$$

has an OBDD representation of linear size. For the variable order $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ however, the size of the OBDD grows exponentially in n. It can be shown that any order that separates the a-variables from the b-variables leads to an exponentially large OBDD.

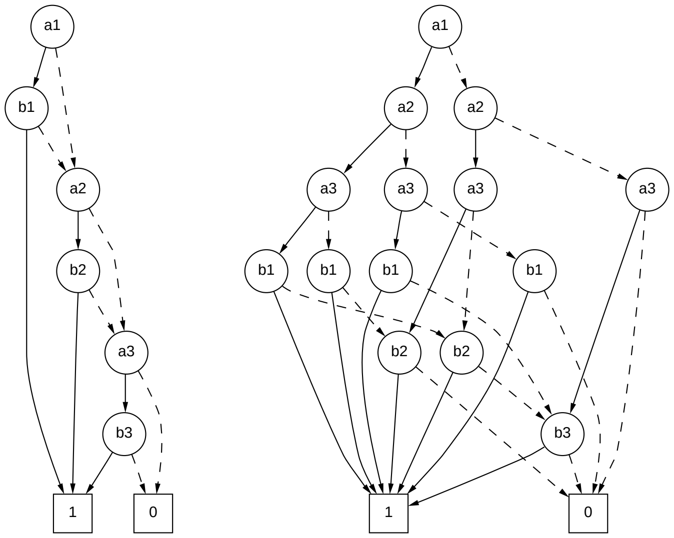


Figure 1.4: Influence of the variable order on OBDDs

The same effect occurs in the case of adder functions: Depending on the variable order, the OBDD-size varies from linear to exponential in the number of input bits. Other important functions, e. g. the multiplication of two n-bit numbers imply OBDDs of exponential size w. r. t. every variable order [Bry91, Woe01].

Due to the uniqueness of the OBDD representation of a Boolean function f w. r. t. a given variable order, the only way to optimize the size of the OBDD representation for f is to find a suited variable order.

1.2.4 Optimization of the Variable Order

Due to the strong dependency of the OBDD-size upon the chosen variable order it is one of the most important problems in the use of OBDDs to construct "good" orders, i. e. orders that fit well to the represented function. However, the problem to construct an optimal order of a given OBDD is known to be NP-hard [THY93, BW96]. The currently best known exact procedure is based on dynamic programming and has running time $O(n \cdot 3^n)$ [Weg00]. Unfortunately, for real-life applications this method is useless. To make the problem even worse, Sieling [Sie98] has shown that there is no polynomial time approximation scheme for the variable ordering problem unless P=NP.

We have to distinguish two different approaches for optimization of the variable order:

Static Techniques: There exist a variety of heuristics to determine a variable ordering before building the OBDD of a function. These heuristics utilize various information given by the netlist of the function [MWB88, MIY90, FOH93]. It turned out that these heuristics often work only for very specific functions. Nevertheless, these heuristics can be useful to determine starting orders.

Applications in OBDD-based sequential verification often require representation of state sets, as these states sets and thus their OBDD representation changes, modifications of the variable order become necessary and a dynamic approach is required

Dynamic Techniques: *Dynamic Variable Reordering* is the process of improving the variable order and hence the size of an already existing OBDD.

All dynamic variable reordering strategies currently are based on the so called *swap* operation. The swap operation exchanges the position of two neighboring variables in the variable order. The advantage of the swap operation is that it only affects the OBDD-nodes on the two levels that are exchanged. Thus, the operation remains local and its effects on the OBDD-size can be measured immediately. Also, the operation is symmetric and can be reverted at any time. Figure 1.5 shows a schematic of the swap operation, in which the number of nodes could be reduced by one.

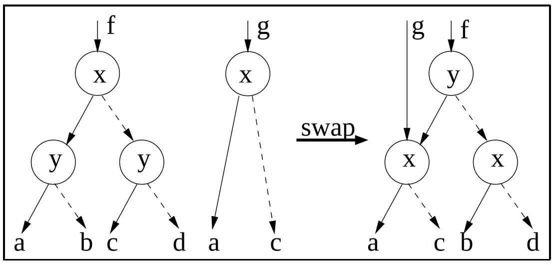


Figure 1.5: Schematic of the Swap Operation.

Virtually any optimization paradigm has been applied to variable reordering, but still one of the most successful strategies is the local search algorithm proposed by Rudell 1993.

1.2.5 Sifting

Rudell's so called *Sifting* [Rud93] algorithm is a local search algorithm, based on the swap operation. In its basic form it works as follows (see also Figure 1.6): Each variable consecutively is moved through the variable order by performing repeated swap operations. The best OBDD-size is stored and after the variable order has been moved to the bottom and the top of the variable order it is moved to the stored best position. Because the size of the OBDD may drastically grow during sifting of a variable, a factor $MAXGROWTH$ limits the maximum growth of the OBDD-size during sifting of a variable.

The sifting algorithm performs a local search on up to n^2 variable orders (on a OBDD with n variables) out of $n!$ possible variable orders and thus visits only a small fraction of the search space. Nevertheless, in practice the Sifting algorithm produces sufficient results.

```
Sifting(P) {
  foreach(i $\in \{1 \dots n\}$){
    optsize= $|P|$; optpos=startpos=i;
    for(j=startpos to n-1){
      swap(P,j+1,j);
      if($|P|$ < optsize){ optsize = $|P|$; optpos =j};
      if($|P|$ > optsize *MAXGROWTH) break;
    }
    for(k=j downto 2){
      swap(P,k-1,k);
      if($|P|$ < optsize){ optsize = $|P|$; optpos =k};
      if($|P|$ > optsize *MAXGROWTH) break;
    }
    if(optpos < k) for(j=k downto optpos+1) swap(P,j-1,j);
    else for(j=k to optpos-1) swap(P,j+1,j);
  }
}
```

Figure 1.6: Basic Sifting Algorithm in Pseudocode

1.3 Verification of Combinatorial Circuits

In our notion *combinatorial circuits* compute functions $f : \mathbb{B}_n^m$, $(n, m \in \mathbb{N})$. They do not contain memory elements like flip-flops or latches. The task in verification of combinatorial circuits is to check whether an implementation of a circuit C fulfills its specification S, i. e. both S and C produce the same functional outputs for all inputs. It can be easily seen that this is an intractable problem if all inputs variations to the circuit are tested explicitly.

OBDD-based combinatorial verification proceeds in two steps:

1. Construct the OBDDs P_C and P_S for C and S by symbolic simulation.
2. Check the equivalence of the OBDDs P_C and P_S.

In case of a strong canonical representation, i. e. when both functions are represented within the same OBDD, the equivalence test itself consists of a single pointer comparison. Each step in the iteration can be performed efficiently w. r. t. the OBDD-sizes of the predecessor gates. This shows that the difficulty of the NP-complete equivalent test [GJ78] has now been shifted into the representation size.

If C and S are not equivalent the operation $P_C \oplus P_S$ gives the inputs where implementation and specification differ. This may be used for debugging.

1.4 Formal Verification of Sequential Systems

A task that is more complex than the verification of combinatorial circuits is the verification of sequential or reactive systems. Throughout this thesis we consider only those systems that can be represented by synchronous finite state machines (FSMs) or systems of communicating FSMs.

Definition 4.1 A *Finite State Machine* (FSM) M is a six-tuple $(Q, I, O, \delta, \gamma, q_0)$, where Q is a set of states, I is the input alphabet, O the output alphabet, $\delta : Q \times I \rightarrow Q$ is the next-state function, $\gamma : Q \times I \rightarrow O$ is the output function, and q_0 is the initial state.

The most common techniques used for formal verification of sequential systems are:

- equivalence check for two finite state machines,
- reachability analysis, including assertion checking, and
- model checking.

1.4.1 Equivalence Check for Two Finite State Machines

A central task in formal verification of sequential systems controllers is the test for equivalence of two given finite state machines. This is needed if correctness of a FSM has to be checked, e. g. after an optimization process.

The equivalence test of two finite state machines M_1 and M_2 itself can be reduced to a reachability analysis by using the construction in Figure 1.7: Let M denote the so-called *product machine*, whose state space is the Cartesian product of the state spaces of M_1 and M_2. The output of M for a given state and a given input is 1, if and only if for this configuration the outputs of M_1 and M_2 agree. M_1 and M_2 have the same input/output behavior if and only if the output of M evaluates to 1 for all reachable states. Hence, reachability analysis is the key problem.

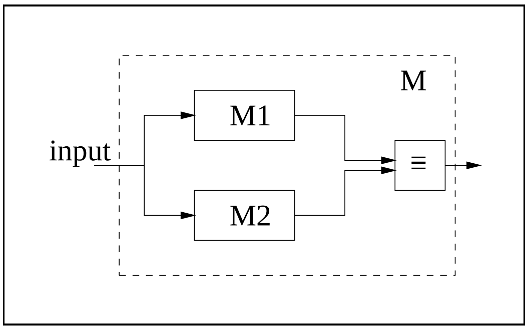


Figure 1.7: Product machine for FSM verification

1.4.2 Reachability Analysis

The computation of the reachable states is a core task for optimization and verification of sequential systems. As we have seen it is needed for equivalence checking for FSMs, also checking for *error states* or assertion checking requires the computation of the reachable states.

Since the set of reachable states can be quite large, an explicit representation of this set, e. g. in form of a list, usually suffers from the *state space explosion* problem. Coudert, Berthet and Madre have investigated the characteristic function of state sets which can be considered as a Boolean function and therefore be represented by an OBDD [CBM89, CM95]. They have shown that this representation form fits well to the operations, which have to be performed for the computation of the reachable states.

Symbolic Representation of FSMs

Typically, the components of a FSM are binary encoded. Let p be the number of input bit, n be the number of state bits and m the number of output bits. Then δ is a function $\delta : \mathbb{B}^n \times \mathbb{B}^p \rightarrow \mathbb{B}^n$, λ is a function $\lambda : \mathbb{B}^n \times \mathbb{B}^p \rightarrow \mathbb{B}^m$ and $q_0 \in \mathbb{B}^n$.

Remark:

The problem of encoding the states of an FSM is a hard problem itself, but beyond the focus of this work. For a detailed description of the state encoding problem see e. g. [The98].

If an encoding of the states with n bits is given, we can represent any set of states by an OBDD representing the characteristic function c of this set:

$$c(x) = \sum_{a \in c^1} x_1^{a_1} \cdots x_n^{a_n},$$

where $a \in \{0,1\}^n$ is an assignment and $x_i^1 = x_i$ resp. $x_i^0 = \overline{x_i}$. Obviously the set operators $\cap$ and $\cup$ equal to $c_1 \cdot c_2$ resp. $c_1 + c_2$.

The essential part of OBDD-based traversal techniques is the **transition relation** (TR), which is based on the transition functions. If the states of an FSM M defined according to Definition 1.4.1 are encoded by n bits, $\delta = (\delta_1, \ldots, \delta_n)$ and the input is encoded by p bits the transition relation of M yields to:

$$TR(x, y, e) = T(x_1, \ldots, x_n, y_1, \ldots, y_n, e_1, \ldots, e_p) = \prod_{1 \leq i \leq n} (y_i \equiv \delta_i(x, e)),$$

where x denotes the present state, y the next state, and e the input. The TR is used for traversal of the state space.

BFS traversal

If the reachable states are computed according to a breadth-first-traversal, the representation via the characteristic function allows to compute all corresponding successor states within a single step. For this reason, one also uses the term *symbolic breadth-first traversal.* Once more, the complexity of the computation depends on the OBDD-size of the occurring state sets.

Figure 1.8 gives an outline of the traversal algorithm. In the first step of the BFS traversal iteration, set To of all successor states of the set From is computed. The image computation step results in a set depending only on next-state variables y. After renaming the set To is depending on present-state variables x. The set of new states New is computed by subtracting the set of already reached states Reached from the set To. The new states are then added to the reached states. The iteration is repeated until no new states are found.

```
Traverse(TR, $S_0$) {
/* Transition Relation TR, initial set $S_0$ */
/* Output: Set of reachable states */
   Reached = From = $S_0$;
   do {
      To = Img(TR, From);
      To = $\text{To}_{Y \mapsto X}$;
      New = To \ Reached;
      From = New;
      Reached = Reached ∪ New;
   } while (New $\neq \emptyset$);
   return Reached;
}
```

Figure 1.8: Basic algorithm for reachability analysis based on breadth-first traversal

Image computation

The symbolic breadth first reachable states computation mainly consists of repeated image computations $Img(\text{TR}, From)$ of a set of already reached states $From$:

$$Img(\text{TR}, From) = \exists_{x,e}(\text{TR}(x, y, e) \cdot From)$$

The transition relation is *monolithically* represented as a single OBDD and such a monolithic representation is usually much too large to allow an efficient computation of the reachable states. Therefore, more sophisticated reachable states computation methods make use of a **partitioned transition relation** [BCL91], i. e. a cluster of OBDDs each of them representing the TR of a subgroup of latches. A transition relation partitioned over sets of latches $P_1, \ldots, P_j$ can be described as follows:

$$\text{TR}(x, y, e) = \prod_j \text{TR}_j(x, y, e) \ , \text{ where}$$

$$\text{TR}_j(x, y, e) = \prod_{i \in P_j} \delta_i(x, e) \equiv y_i .$$

Image computation using AndExist

With the use of a partitioned transition relation the image computation can be iterated over P_i and the $\exists$ operation can be applied during the product computation *(early quantification)*:

$$Img(\text{TR}, R) = \exists_{v^j}(\text{TR}_j \cdot \ldots \cdot \exists_{v^2}(\text{TR}_2 \cdot \exists_{v^1}(\text{TR}_1 \cdot From) \ldots),$$

where v^i are those variables in $(x \cup e)$ that do not appear in the following TR_k, $(i < k \leq j)$. The so called *AndExist* [BCL91] or *AndAbstract* operation performs the AND operation of two functions (here partitions) while simultaneously applying existential quantification

($\exists_{x_i} f = (f_{x_i=1} \vee f_{x_i=0})$) on a given set of variables. The variables to be quantified out are those that are not in the support of the remaining partitions. Unlike the conventional AND operation the AndExist operation only has an exponential upper bound for the size of the resulting OBDD, but for many practical applications it prevents a blow-up of OBDD-size during the image computation.

Another important problem in the context of image computation is finding an optimal schedule of the partitions for the AndExist operation. Geist and Beer [GB94] presented a heuristic for the ordering of partitions each representing a single state variable. The goal of this heuristic is to keep the support variable set of the intermediate products as small as possible. This heuristic was broadened by Ranjan et. al. [Ran95] to allow partitions including more than one state variable.

An insight into the complexity of the partition problem was given by Hojati et. al. [HKB96]: They have shown that finding a tree of conjunctions s. t. the support of the largest intermediate product is less than a given constant is NP-complete even under the simplifying assumption that the support of $f \wedge g$ is the union of the supports of f and g.

For a more detailed description of OBDD-based reachability analysis see [MT98].

1.4.3 Symbolic Model Checking

Since a complete formal verification of a sequential system is often too complex, methods are of interest that guarantee at least correctness of certain properties. One of them is the so called *model checking.*

Model checking is the problem to decide whether an implementation satisfies its specification given in terms of a temporal logic, e. g. the so called *computation tree logic* (CTL). The formulas of CTL describe properties of infinite paths of states that are traversed during the computation.

The idea to use OBDDs for a symbolic representation of state sets during model checking was first introduced by Burch et. al. [Bur90].

Using this way of *symbolic* model checking, real-life systems up to 10^{100} states can be verified.

The Temporal Logic CTL

The logic that is used in symbolic model checking is a modal logic and it is called *computation tree logic* (CTL). CTL is based on the concept of *branching time.* In branching time the temporal order "<" defines a tree, which branches toward the future. The past of each event is uniquely defined, but its future is not. This corresponds to the dynamic behaviour of FSMs, where the the past sequence of traversed states is unique, but the future states are not known.

The reason to use CTL is that its operators can be easily expressed by fixed point computations. CTL consist only of present and future operators, past operators are not allowed.

In addition to the Boolean operators $\wedge, \vee$ and $\neg$, CTL has four temporal operators:

X The *next* operator describes a condition that is true in the next state of the computation.

G The *global* operator describes a condition that is true for all states of a path.

F The *future* operator describes a condition that is true on a path sometimes in the future.

U The *until* operator $a\mathbf{U}b$ is true on a path if a is true until b is true.

All temporal operators are quantified with either an universal quantifier **A** ("on all paths it holds...") or an existential quantifier **E** ("there exists a path, where...").

Typical examples of CTL formulas of practical relevance include:

- AG(req $\mapsto$ AF ack): Each request is eventually acknowledged.
- AG $\neg$(p $\wedge$ q): Mutual exclusion: p and q are never satisfied at the same time.
- AG EF (s_0): There is always a sequence of paths that allow to return to the initial state s_0. This property checks for deadlocks.

Computation of the operands

The operands EX, EU and EG are computed by the routines CheckEX, CheckEU and CheckEG, where the latter two are fixed point computations:

CheckEX consists of an pre-image computation:

$$CheckEX(p) = PreImg(TR,p) = \exists y(TR(x,y) \wedge p).$$

CheckEU computes the least fixed point for EU and can be described inductively as:

$$\begin{aligned} CheckEU_0(p,q,TR) &= q \\ CheckEU_{i+1}(p,q,TR) &= CheckEU_i(p,q,TR) \vee \\ &\quad (p \wedge CheckEX(CheckEU_i(p,q,TR),TR)) \end{aligned}$$

The number of states represented by the sequence of the $CheckEU_i$ **increases** monotonously. The sequence converges and at a certain iteration k, because the number of states is finite:

$$CheckEU_{k+1} = CheckEU_k = CheckEU.$$

CheckEG computes the greatest fixed point for EG and can be described inductively as:

$$\begin{aligned} CheckEG_0(p,TR) &= p \\ CheckEG_{i+1}(p,TR) &= CheckEG_i(p,TR) \wedge \\ &\quad (p \wedge CheckEX(CheckEG_i(p,TR),TR)) \end{aligned}$$

The number of states represented by the sequence of the $CheckEG_i$ **decreases** monotonously. The sequence converges and at a certain iteration k, because the number of states is finite:

$$CheckEG_{k+1} = CheckEG_k = CheckEU.$$

The remaining operators EFp, AXp, AGp and A(p U q) are derived from the above operators by the following rules:

$$\begin{aligned} EFp &= E(trueUp) \\ AXp &= \neg EX\neg p \\ A(qUp) &= \neg(E(\neg pU\neg q \wedge \neg p) \vee EG\neg p) \end{aligned}$$

Figure 1.9 gives an algorithmic description of the operators CheckEX, CheckEU and CheckEG that form the basic symbolic model checking algorithm *Eval(f)*.

For a more detailed introduction to symbolic model checking see [McM93].

```
Eval(f) {
   case{
      (f== atomic proposition): return f;
      (f== $\neg p$): return $\neg$Eval(p);
      (f== $p \vee q$): return Eval(p) $\vee$ Eval(q);
      (f== EXp): return EvalEX(p);
      (f== E(p U q)): return EvalEU(Eval(p),Eval(q),0);
      (f== EGp): return EvalEG(eval(p),1);
   }
}
EvalEX(p){
   return PreImg(TR,p);
}
EvalEU(p,q,y){
   y'= q $\vee$(p $\wedge$ evalEX(y));
   if (y == y') return y;
   else return EvalEU(p,q,y');
}
EvalEG(p,y){
   y'=(p $\wedge$ EvalEX(y));
   if (y == y') return y;
   else return EvalEG(p,y');
}
```

Figure 1.9: Basic Symbolic Model Checking Algorithm

1.5 Software Packages

In the following we will briefly describe the OBDD package and the formal verification package, in which we implemented our algorithms and heuristics. The Protocol Compiler synthesis and verification package is described in Chapter 2.

The OBDD Package

The *Colorado University Decision Diagram Package* (CUDD) [Som] is a BDD-package written by Fabio Somenzi and his working group at the University of Colorado at Boulder. It inherits algorithms for OBDDs, zero-suppressed OBDDs (ZDDs) [Min93] and algebraic DDs (ADDs) [BFG93+]. CUDD is written in C, allowing easy programming and fast access of graph data-structures. CUDD includes various algorithms for dynamic reordering like Sifting or Group-sifting. Alternative techniques like simulated annealing [BLW95] or genetic algorithms [DBG95] are implemented as well. CUDD includes sophisticated memory management techniques. We are using version 2.3.0 of CUDD.

The Verification Package

The *Verification Interacting with Synthesis Package* (VIS) [VIS] is a cooperation of the Carnegie Mellon University, University of California Berkeley and University of Colorado at Boulder. It integrates synthesis, simulation and formal verification of finite state machines. The algorithms for these tasks are based on OBDDs. The interface to the OBDD package is transparent, i. e. independent of the chosen OBDD package. Currently three packages are supported: CMU [Lon92], CAL [CAL] and CUDD. We are using solely the CUDD package.

Figure 1.10 shows the architecture of the VIS system. The VIS front-end is able to read in gate level BLIF (Berkeley Logic Interchange Format) descriptions or low-level RTL descriptions in BLIF-MV [Kuk96]. Together with the vl2mv compiler, which compiles Verilog [TM91] into BLIF-MV, a subset of Verilog at RT-level can be read in. VIS itself implements algorithms for representation, simulation and verification of FSMs. For optimization of FSMs, VIS interacts with SIS (*Sequential Interactive Synthesis*) [SIS]. We are working with version 1.3 of VIS

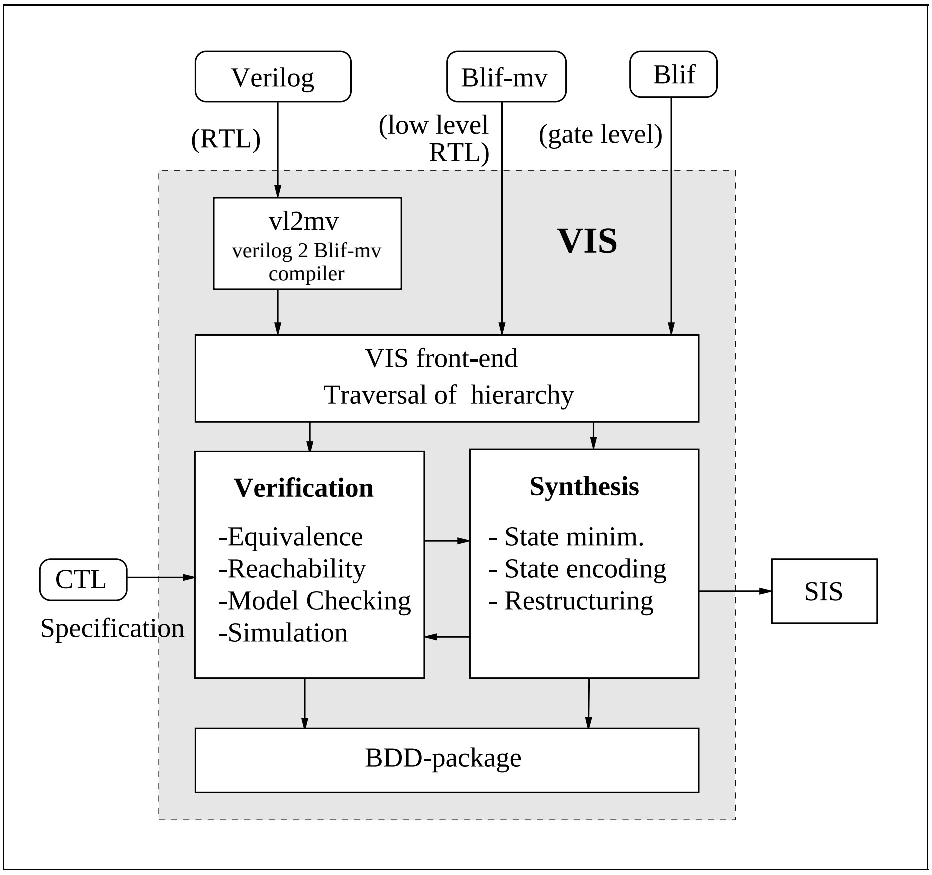


Figure 1.10: Architecture of the VIS-system

Chapter 2

A Case Study in Formal Verification

In the previous chapter we have learned about two important techniques for the verification of sequential systems in CAD, which are reachability analysis and model checking. In this chapter we will conduct a case study, which presents the application of formal verification methods in an industrial surrounding. We develop a methodology which allows to integrate formal verification in the design flow.

The abilities of formal verification, as well as its limitations become visible. The limitations of the approach motivate the research that is presented in the remaining chapters.

This chapter is structured as follows: In the next section we discuss the application of formal verification in industrial settings and briefly present the synthesis and verification tool Protocol Compiler. The next section presents our formal verification methodology, which is based on assertion checking. The methodology is applied to two examples that are taken from the networking arena. The first is a simplified RS232 transceiver, the second a pipelined FIFO-like buffer written in Verilog. Because the examples represent different types of protocols, two different techniques are given. At the end of the section the general problem of the handling of inputs during verification is discussed. The last section concludes the chapter and gives an outlook on challenges and open problems.

2.1 Formal Verification in an Industrial Setting

Recently, usage of complex controller and protocol designs like ATM, Sonet, SDH, etc. as well as their complexity has increased and due to Moore's law [Moo65] we can expect this to continue.

Much design time is spent verifying the design and in practice nearly all of it is done by simulation. But, as the complexity of the design increases, simulation loses its sufficiency. To guarantee correctness of complex designs, formal verification techniques like model checking or theorem proving are a useful supplement to simulation. Unfortunately, formal verification currently does not fit well into the design process for multiple reasons:

- Fairness constraints and specifications must be given in languages such as CTL, that are not related to any hardware description language (HDL), hence unknown by most designers. Also, they are not very intuitive.

- formal verification is an inherently complex task.
- Expert guidance is required.
- When formal verification finds a bug, it is difficult to determine where the bug originates from.

In this case study we present two examples how to apply formal verification methods in Protocol Compiler. We use a simulation-like approach and re-use the testbench for simulation as well as assertion checking.

This case study demonstrates the possibility of integrating formal verification in the design process of real life controllers. Nevertheless, its limitations and challenges become visible. This chapter serves as a motivation for the problems that are addressed in the following chapters.

2.1.1 Protocol Compiler

Protocol Compiler is a design environment for the processing of structured data streams such as the ATM, SDH/SONET, or MPEG protocols. It is based on previous work by Seawright, Brewer, and Crews on logic synthesis from grammatical productions [SB94, CB96]. Seawright gave an overview of the tool [Sea96] and Holtmann and Meyer application examples for SPDIF [HB98] and SDH/SONET [MST97a] protocols. The design environment consists of entry, debugging support, protocol synthesis, and HDL generation (both VHDL [HJ96] and Verilog [TM91] are supported). The Protocol Compiler language is conceptually higher than register transfer level (RTL).

Protocol Compiler provides a sophisticated debugging environment for simulation as well as formal verification based on the following features:

- The high-level language of Protocol Compiler makes it simple to describe stimuli or to check for errors. The language is applicable to the design as well as the testbench.
- Interfaces to Verilog and VHDL simulators are provided.
- Protocol Compiler provides formal verification capabilities like image computation, reachable states computation and model checking [CBM89, Bur90] based on OBDDs [Bry86]. These formal verification capabilites are set up on the internal representation of the controller, which is also based on OBDDs.
- Results from simulation or formal verification are mapped back onto the source by highlighting frames.

In this case study, we focus on applying assertion checking that is a subproblem of model checking.

2.2 An Assertion Checking Methodology for Protocol Compiler

We start the description of our verification methodology by outlining our general verification flow, which closely combines simulation and formal verification. Then, we present two different assertion checking strategies for the verification of two different controllers.

2.2.1 Verification Flow

Our design flow (see Figure 2.1) starts with the conventional steps to enter the design, write a testbench and then simulate it and fix bugs. We first check typical cases and then enhance the stimuli generators to cover corner cases as well. During the first cycles of the verification flow it is very likely to have obvious errors that can easily be detected by simulation and do not need the more time consuming formal verification approach.

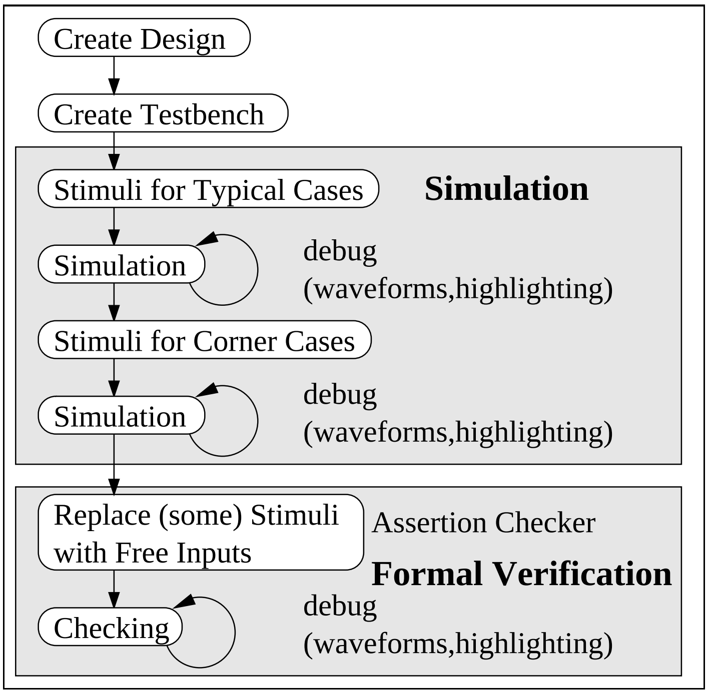


Figure 2.1: Verification Flow.

In order to cover *all* corner cases we finally apply formal verification in form of assertion checking. Some parts of the stimuli generator are replaced by free inputs but most of the testbench is re-used. An assertion checker is added. Then, the formal verification engine of Protocol Compiler is used.

Assertion checkers are part of the testbench and prove whether the design under test behaves correct, i. e. the correct result is returned. We use the same or similar assertion checkers for simulation and formal verification.

During formal verification, we perform assertion checking by computing all reachable states and determining whether an assertion is violated in any of the states. If a bug is found, i. e. an assertion is violated, we generate a trace file and review it with a waveform viewer. The trace shows the shortest explicit sequence of input values and state transitions leading to the bug.

2.2.2 Example RS232

As the first example, we show how to verify a simplified RS232 receiver with simulation and formal verification methods.

The RS232 is a simple and well-known serial communication interface. Transmission is done through a single wire (we ignore all control signals). A byte is transmitted as a sequence of

8 data bits plus some start, stop and parity bits. For the sake of simplicity, we assume that receiver and transmitter are already synchronized at the bit-level. Also, we always use two stop bits and even parity.

Figure 2.2 shows the implementation of the receiver with Protocol Compiler. Thick outlined boxes are *terminal frames* and describe a delay of 1 clock cycle plus a condition like Rx==0 to be met. Other frames describe sequence { }, repetition $[\ ]^+$ or concurrency $\overset{\top}{\bot}$. Ovals are *actions* which are executed whenever the corresponding frame successfully finishes. If the incoming bit stream is invalid, e. g. a wrong parity bit, then the corresponding terminal frame will not accept and the *exception frame* executes the `action set(invalid)` for one cycle before the receiver resumes.

Please see the overview [Sea96] or application papers [MST97a, HB98] of Protocol Compiler for a more detailed description of the language.

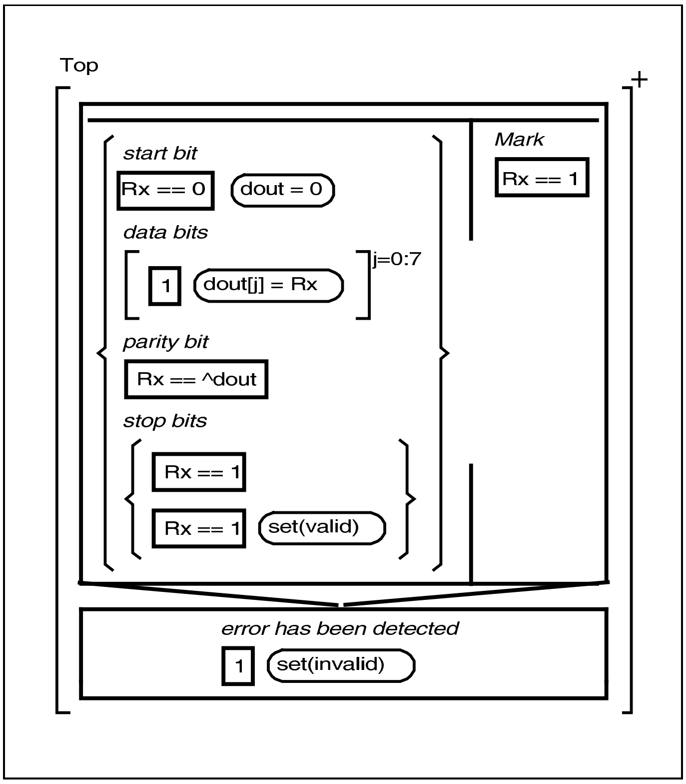


Figure 2.2: Protocol Compiler Implementation of the RS232 Controller.

All frames of the controller put together work as follows: The Top frame first repeats executing the terminal frame `Rx==1` on the right-hand side of the alternative frame until `Rx` goes to 0. Then execution shifts to the left side and a full word is read, beginning with the start bit `Rx==0`, 8 data bits, parity and two stop bits. If the parity bit has the wrong value, then the terminal frame `Rx==!dout` will not accept, thus raising an error and execution jumps into the exception handler at the bottom where the invalid flag is set. Ditto both stop bits are checked. Then, the receiver starts searching for the first start bit (`Rx==0`) again.

In this example we show how to apply formal verification methods by reusing a major part of the simulation testbench. In other words, how to apply formal verification without leaving the design-simulation track.

We start the verification task with simulation. The simulation is driven by a testbench that creates inputs (e.g., using a counter) and sends them to the Transmitter (Tx) of the RS232.

Furthermore, these messages are buffered in a circular register. The register is needed to store the messages for the latency period between sending and receiving. In this example a register with two slots is sufficient, since only one message is pending at a time. One register is keeping the actual message, the other one is ready for writing. The circular register also allows simultaneous reading and writing. Each time a message is delivered from the Receiver (Rx) it is compared to the actual message in the circular register. The error flag is set, if these messages differ. See Figure 2.3 for a schematic view of the testbench.

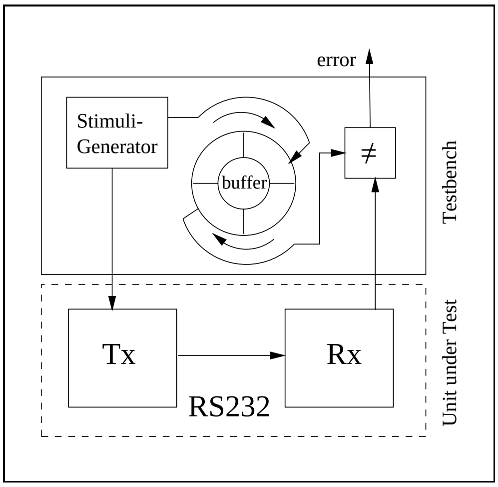


Figure 2.3: Schematic View of the Testbench for the RS232.

This setup provides us with a sound testing environment, e. g. we can test the correct transmission of all 256 possible input values. But, this simulation cannot assure full correctness of the controller, since it is not only required to test all possible values, but all possible *sequences* of input values including any arbitrary number of `ready=0` signals in between.

A simulation testbench is not able to process this infinite input stream. If we want to assure correctness over all input sequences we have to use formal verification methods.

As part of the transformation from simulation to formal verification we build an assertion checker (AC) by reusing most of the testbench. The only change for assertion checking is to remove the stimuli generator from the testbench and to declare the inputs to be free. Free inputs are treated like external inputs, i. e. no special value is assigned and thus the inputs can take any possible value in any clock-cycle. In this context a free input is like a stimuli generator that exhausts all possible sequences of all possible values. After doing this we lose the ability to perform simulation in return for the possibility of assertion checking. Figure 2.4 shows the assertion checker for the RS232.

Assertion checking amounts to computing the set of reachable states. After this set has been computed, checking the correctness property is easy: The situation when an error occurs, i. e. , the *error* flag is set, marks a state or a set of states. We only have to check whether the error states are within the reachable state set.

The testbench describes the fairness constraints and specifications necessary for assertion checking. Normally, they would be written in languages such as CTL (see [Bur90]), which are not related to HDLs and unknown by most designers. This is one of the reasons why application of formal verification is currently difficult. Instead of CTL, we use the Protocol Compiler language which is similar to an HDL but at an higher abstraction level. Because the language is fully synthesizable, all constructs can be turned into OBDDs making formal verification possible.

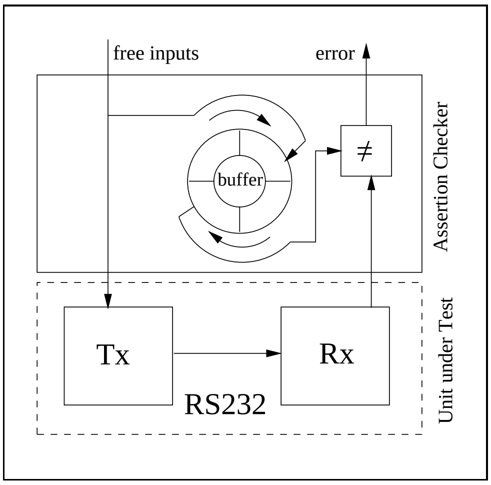


Figure 2.4: Schematic View of the Assertion Checker for the RS232

Experimental Results

The verification of the RS232 required 34 seconds on a 336 MHz UltraSPARC-II, resulting in 268,545 reachable states within a sequential depth of 29 shells. The design was proven to be correct.

In the next example we present a more sophisticated assertion checking method and show how to combine assertion checking and debugging at the source level.

2.2.3 Example Cow-Stomach

The second example is a "cow-stomach": a small FIFO-like buffer inserted between two modules forming a source and sink. In networking designs, pairs of modules often form a source and sink. In general, data simply flows from source to sink, however the sink is sometimes not able to accept data and indicates this by clearing a `ready` flag. Common design practice is to register all flags (ports) between modules to improve clock speed. This introduces a one-clock-cycle delay per register, so the sink must indicate with `ready` that it *will be ready in the next cycle* to accept data.

The sink essentially has to predict its ability to accept data a cycle later, however, in some designs this is not possible. The best thing the sink can do is to provide a `taken` flag indicating that is was able to accept data in the current clock cycle. In that case we need to insert a FIFO-like buffer between source and sink. We call this buffer a *cow stomach* and its goal is to provide maximum throughput using only few resources (see Figure 2.5).

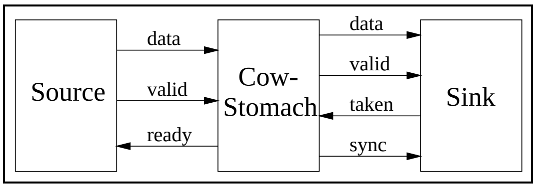


Figure 2.5: Cow-Stomach.

The cow stomach also has registered outputs, so there is now a four cycle delay to send a data token from source to sink and an acknowledgment back. The cow stomach consists of a buffer, organized as a FIFO, and control logic. We found that designing the control logic, although not too complex, was easy to get wrong. The difficulty arises from the fact that data has to be speculatively send from the source several cycles before it is known whether the sink will accept the data. One can view this as a 2-stage pipeline. The cow-stomach must store up to 4 data tokens, so the controller must also keep track of the buffer status as well as the pipeline stages.

We implemented the cow-stomach as an explicit FSM (Figure 2.6: for simplicity all actions and conditions have been removed) in Verilog and imported it to Protocol Compiler using a prototype Verilog parser.

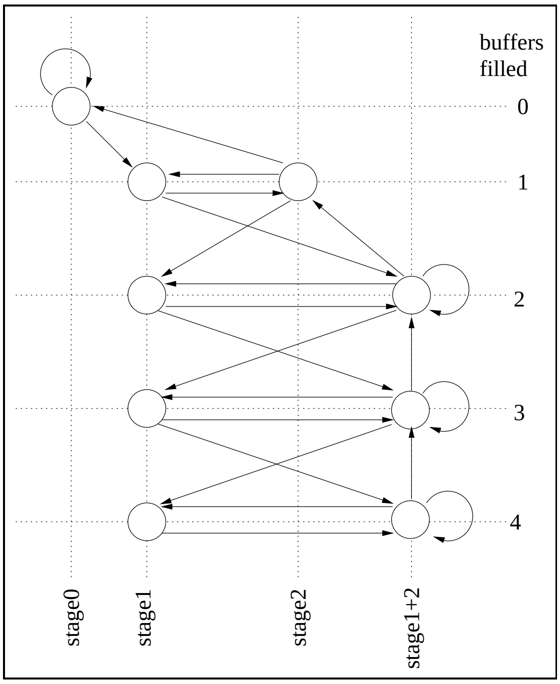


Figure 2.6: Cow-Stomach Controller.

The cow-stomach represents a common type of design where data is transferred and/or modified. The control is unstructured and depends on many flags. Errors that may occur in this context include:

- Buffer underflow or overflow,
- lost or duplicated messages, and
- out-of-order messages.

Although the control logic has only nine states, the design had initially several errors. For example, the `ready` flag was cleared too late with the result that the source kept on sending data and the internal buffer overflowed later.

To check the controller for these errors, we start with a simulation using a testbench similar to the one used in the previous example. In this case, the circular register has to be larger (at least five slots) due to the higher number of messages buffered.

After a detailed simulation phase, we may assure correctness of the controller for many situations, but again, we cannot assure correctness for all possible situations. At this stage we could turn the testbench into an assertion checker as in the previous example to check the controller for the errors described above, instead we propose a different method for two reasons:

1. The larger circular register and the higher bit-width of the messages may make formal verification of the controller too complex.
2. The Cow-Stomach is a *data-independent* protocol which allows us to use a more efficient method for the verification.

A data-independent protocol is a protocol whose control only depends on control inputs (flags like `valid` or `ready`), but not on the data itself. The RS232, for example, is data-dependent, since it computes the parity bit from the input. For a detailed analysis of data-independency refer to Wolper [Wol86]. For the verification of data independent protocols no circular register is needed. Since the data does not influence the behavior of a data-independent protocol, the data input width can be reduced to one bit. Wolper has also shown that this bit is sufficient to distinguish messages and we take advantage of this fact.

The assertion checker for the verification is built in the following way: The generator part of the assertion checker generates an arbitrary number of 0-messages. Then two consecutive 1-messages are sent, followed by 0-messages. The receiving part of the assertion checker checks whether the output of the protocol fulfills the following properties:

- Exactly two 1-messages are sent, and
- the 1-messages arrive consecutively.

The assertion checker does this symbolically for all streams of 0-messages with two consecutive 1-messages embedded into it. Errors in the design are detected in the following way:

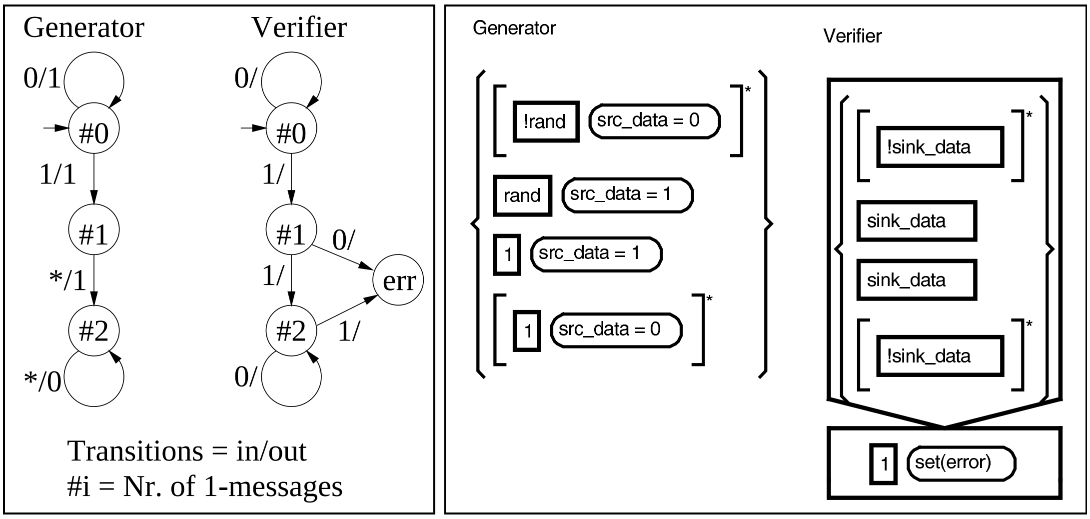
- If a message is lost, there exists a stream for which one 1-message is lost.
- If a message is duplicated, there exists a stream for which one 1-message is duplicated, resulting in three 1-messages.
- If the order of messages is disturbed, there exists a stream where the two 1-messages are no longer consecutively.

If data is corrupted, then there is a stream with either only one or three 1-messages. See Figure 2.7 for a representation of the Assertion Checker.

After the cow-stomach passed all simulation tests we still found one more bug with formal verification using this assertion checker.

Usually, formal verification confirms that your design works correct but what do you do when it still has bugs? What is needed is a way to explicitly find the conditions under which the error occurs and why. We use Protocol Compiler's facility to trace to a certain state (in this case the error state) to get a shortest path of state transitions to the error state. A trace file is generated and examined with a waveform viewer as shown in Figure 2.8.

We found that being able to see a specific trace leading to an error is very helpful. Protocol Compiler allows to move the marker (the vertical line in the middle of Figure 2.8) backward



a) b)

Figure 2.7: Assertion Checker a) as FSM b) in Protocol Compiler.

and see which states were executed in the source. We did this and found the cause of the error happened 7 cycles earlier in the state shown in Figure 2.9. The shading indicates which frames and actions were executed in the current clock cycle.

The problem was that the `src_ready` flag was incorrectly set high too early. Consequently, the source sent more data than the cow-stomach could store and a token was lost.

Using this kind of a assertion checker we add only a small overhead to the design. This allows a fast computation and little memory requirements. Nevertheless, since only one bit of the input data is required for the verification one either should reduce the bit-width of the input to one or at least set the unused bits to default values. This keeps the state space and the sequential depth of the FSM to a minimum.

Experimental Results

We performed the assertion checking of the (correct) cow-stomach protocol on a 336 MHz UltraSPARC-II. It takes 7 seconds to compute the protocol's 1763 reachable states in a sequential depth of 15 shells (includes testbench and assertion checker).

2.2.4 Dealing with Inputs during Verification

In preparation of the assertion checker the inputs (i. e. ports) to the design have to be chosen carefully. Each input may belong to one of the three types given below:

- Inputs driven by the assertion checker,
- free inputs, or
- fixed inputs.

All inputs affecting the property that is checked by the assertion checker process have to be driven by the assertion checker. This includes the data-path of the controller as well as timeouts etc.. Assertion checking is conservative regarding the choice of inputs. This means,

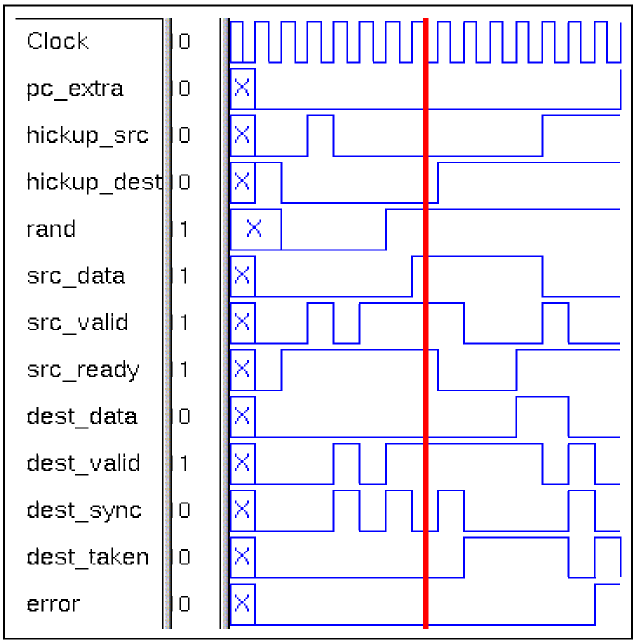
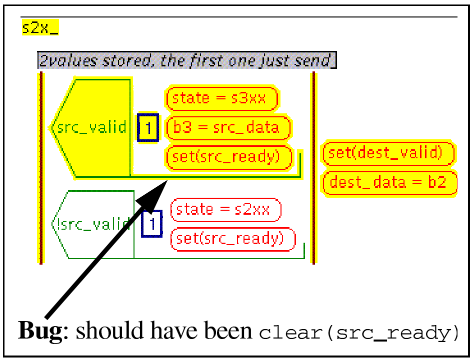


Figure 2.8: Diagnostic Trace



Bug: should have been `clear(src_ready)`

Figure 2.9: Discovered Bug

choosing the wrong inputs to be free will introduce false negatives, i. e. probably non-existing errors are notified, but it will never ignore real errors. For example, setting the data-input of the RS232-receiver to a free input will produce an error, due to the arbitrary input values like missing stop bits, although the receiver is error-free.

Any input not affecting the assertion checking process may be set to a free input. The problem that arises from this strategy is that too many free inputs increase the number of reachable states and the sequential depth of the system, making a successful assertion checking problematic. This problem may be avoided by setting inputs to fixed values. But these variables have to be chosen carefully. For example: The cow-stomach has an input *hick-up* that is set, if the sink has a congestion while processing data. Setting this input in to a fixed value will prevent a meaningful assertion checking. Setting assertion checking-related variables to fixed values may be detected by the fact that fractions of the controller that should be checked are no longer reachable at all (but this is not guaranteed).

Another way to lower the complexity of the assertion checking is to reduce the bandwidth of the data-path-variables. Once a controller has been judged data-independent the width of the data-path may be set to 1. This may be done by fixing the other bits of the data-path or – preferably – by reducing the bit-width within the protocol's declaration. Within data-dependent protocols reducing the bit-width of the data-path will influence the computation, and thus has to be done very carefully.

2.3 Conclusion and Open Problems

This case study presented two examples how to verify designs in Protocol Compiler. We use simulation and formal verification in form of assertion checking. We entered designs and testbenches in Verilog or the Protocol Compiler language and found that a very similar setup can be used for simulation and formal verification. Actually, the assertion checker can be derived from the simulation testbench (See Figure 2.3 and Figure 2.4).

Applying formal verification methods with Protocol Compiler turned out to be straightforward, because it is easy to describe stimuli as well as assertion checker and formal verification is tightly integrated in the tool.

We also found that data independent protocols can be efficiently handled by formal verification using a special type of assertion checker with reduced complexity.

Although the usage of an assertion checker is easy and formal verification is well integrated into Protocol Compiler, the limitations of the approach that are due to increased complexity become visible.

The examples presented here are realistic, but not too complex. As soon as the protocols become more complex the effort for formal verification increases drastically. The formal verification is still feasible, but it requires some "expert knowledge in formal verification technology. This knowledge usually is not given on the user side.

Protocol Compiler is a high-level design and verification tool, it follows a modular design approach and is capable of high-level design languages as e. g. Verilog. This high-level information can be used as "knowledge to drastically improve the performance of the integrated formal verification techniques.

The main areas for optimization of OBDD-based formal verification are the partitioning of the transition relation of the design under test and the dynamic variable reordering of the OBDDs.

In the remaining chapters we present approaches that

- utilize high-level information to build a modular structured partitioned transition relation,
- generate a modular structured partitioned transition relation under the assumption that the design is structured, but without explicitely using external high-level information, or
- use specialized adaptions of general approaches for dynamic variable reordering

to improve the performance of formal verification.

All these methods increase the efficiency of formal verification and reduce expert knowledge required to verify designs.

Chapter 3

Partitioning

Image computation is the core operation in any sequential verification application like model checking or reachability analysis. As an example, the basis of the Assertion Checking strategy described in the previous chapter is a reachability analysis of the combined design and assertion checker system. OBDD-based state space exploration algorithms use a partitioned representation of the transition relation of the design under consideration. Modern designs are modularized and hierarchically structured, but the conventional partitioning strategies lack the ability to incorporate additional information provided by the designer or a high-level design tool.

Most partitioning strategies are based on finding an *ordering* of the latches for clustering. In this chapter we will describe partitioning techniques that change this paradigm to *grouping* the latches for clustering.

This chapter is structured as follows: We start with an description of the partitioning problem, give the standard partitioning method and briefly outline other partitioning approaches. In the next section we describe our *RTL-heuristic* that utilizes *high-level information* given by the design to group latches and improve the partitioning. We then analyze these *grouping effects*. The section is concluded by benchmark experiments. In Section 3.3 we present our second heuristic called *group heuristic*. An analysis of the RTL heuristic shows how the grouping of latches improves the quality of the partitioning. In the group heuristic we simulate this kind of grouping, but without using explicit high-level information, making this approach more general. Also, this section is concluded by benchmark experiments.

3.1 The Partitioning Problem

The computation of an image using OBDDs as shown in Section 1.4.2 requires the OBDD-representation of the transition relation (TR) of the system. Most TRs of realistic systems are much too complex to be represented by a single OBDD. A so-called *monolithic* transition relation usually blows up in size and exceeds any space limitation. For this reason one uses a *partitioned* transition relation. This technique introduces two optimization problems:

1. Finding an optimal clustering of the latches (clustering problem), and
2. finding an optimal schedule for the conjunction of the clusters (scheduling problem).

Typically, both problems interfere with each other. The possibilities for clustering range from the monolithic TR, which as mentioned above in most cases cannot be represented in terms of OBDDs, to the single latch clustering, which has a small TR, but includes many very costly operations, whose intermediate results may also exceed space limitations.

The difficulty of the partitioning problem increases as it is not known

- how many images will be computed until a fixed point is reached,
- how the OBDD-representation of the reached state set will develop, and
- how the variable order will change due to dynamic variable reordering.

All these points have to be taken into account by a good partitioning strategy.

3.1.1 Common Partitioning Strategy

The common strategy for clustering and conjunction scheduling is the so called IWLS95 method [Ran95] that is used e. g. by VIS [VIS]. (The name stems from the first presentation of this method at the IWLS workshop.)

The AndExist-based image computation step using a partitioned TR is done by iteratively taking clusters into the product and quantifying out variables that are not needed in the following iterations. Clusters are built by conjoining bit-relations to a cluster until a given *partition-size threshold* is exceeded. Then, a new cluster is introduced (clustering problem). A more subtle decision has to be made for the conjunction scheduling of clusters (or bit-relations) to allow an efficient AndExist operation (scheduling problem).

The IWLS95 method partitions a transition relation as follows:

1. **Order latches:** First, the latches are ordered by using a benefit heuristic.
2. **Cluster latches:** The single latch relations are clustered by following a simple strategy. OBDDs representing latch transition functions are added to a cluster OBDD (i. e. by performing AND) in the order computed by the benefit heuristic until the size of the OBDD exceeds the partition-size threshold. Then, a new cluster OBDD is created.
3. **Order clusters:** In the last step the clusters are ordered similarly to the latches by again using a benefit heuristic.

For a description of the algorithm in pseudocode see Figure 3.1.

The heuristic argument of the IWLS95 method is that the number of variables being quantified out of the product and the number of additional variables being introduced into the product determines the efficiency of the operation. Thus, the goal of the heuristic is to keep the number of variables in the support of the product as small as possible.

It is worthwhile mentioning that already a subproblem of the scheduling problem is algorithmically intractable: Hojati et. al. . [HKB96] have shown that finding a tree of conjunctions s. t. the support of the largest intermediate product is less than a given constant is NP-complete even under the simplifying assumption that the support of $f \wedge g$ is the union of the supports of f and g.

The ordering of bit-relations or clusters for the conjunction schedule is done by a *benefit heuristic* [GB94] that performs a structural analysis of the transition functions of the latches. The heuristic chooses greedily from the set of unordered relations the one that fits best to the following criteria:

```
IWLS95Partition(latches){
    ordered_latches = BenefitOrder(latches);
    tmp_cluster=cluster= 1-Sink;
    foreach(latch in ordered_latches){
        tmp_cluster = cluster $\wedge$ latch;
        if(BDDsize(tmp_cluster) > THRESHOLD){
            append(cluster,cluster_list);
            cluster = latch;
        } else{
            cluster = tmp_cluster;
        }
    }
    ordered_cluster = BenefitOrder(cluster_list);
    return ordered_cluster;
}
```

Figure 3.1: Algorithm for the IWLS95 Heuristic in Pseudocode.

1. The relation having the maximum number of variables quantified out, when chosen for the next product.
2. The relation having the maximum number of present-state variables to be introduced in the product.
3. The relation having the maximum number of next-state variables to be introduced in the product.
4. The relation having the maximum level of a variable in the OBDD to quantified out.

The formula which computes the benefit according to the above criteria when latch resp. cluster i is chosen:

$$W_i = w_1 \cdot \frac{||q||}{||x||} + w_2 \cdot \frac{||x||}{||\bar{q}||} + w_3 \cdot \frac{||y||}{||\bar{y}||} + w_4 \cdot \frac{b_i}{\bar{b}},$$

where $|| \quad ||$ denotes the cardinality of the following sets:

q Variables which can be quantified out if i is chosen

$\bar{q}$ Variables which have not been quantified out yet

x Present state and input variables in i

y Next state variables in i

$\bar{y}$ Next state variables not yet introduced

b_i Maximum OBDD level of a variable to be quantified out in i

$\bar{b}$ Maximum OBDD level of a variable not yet quantified out.

According to [Ran95] the weights are:

$$w_1 = 2, w_2 = 1, w_3 = 1, w_4 = 1.$$

The version of VIS that we are using (VIS-1.3) has the weights by default set to:

$$w_1 = 6, w_2 = 1, w_3 = 1, w_4 = 2.$$

This is done to put more weight on those choices that reduce the number of variables in the next product.

Remark:

The 4th criterion is not useful if dynamic variable reordering is activated during the partitioning process, because after reordering the ranking computed on the previous variable order becomes – at least partially – obsolete.

Figure 3.2 summarizes the partitioning process for the IWLS95 strategy.

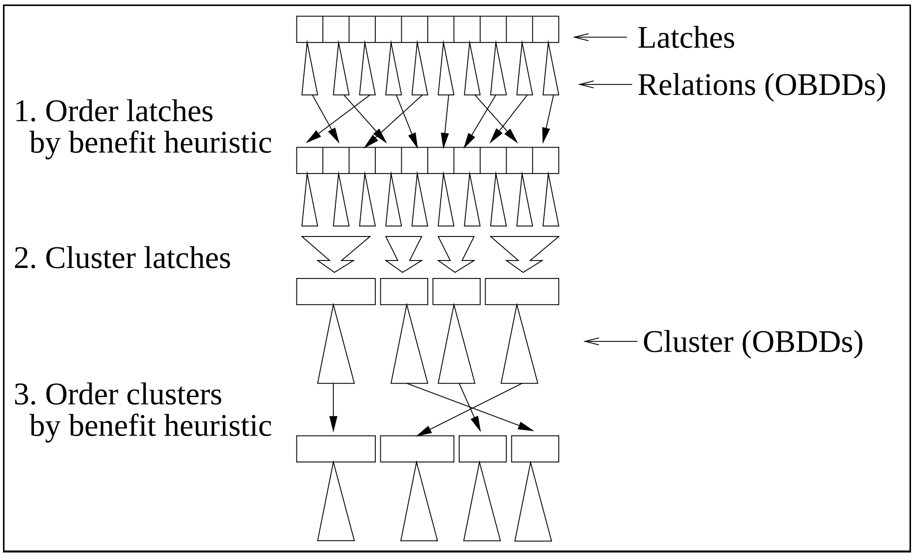


Figure 3.2: Schematic of IWLS95 Partitioning Strategy.

3.1.2 Related Work

After the IWLS95 method has been the state-of-the-art method for partitioning for several years, recently new approaches (beyond those present here) have been published:

Hachtel et. al. [MHS00] presented a heuristic that minimizes *active lifetime* of the variables in the product to gain a good conjunction schedule. The active lifetime is the number of conjunctions in which the variable is involved schedule and it is computed from a dependency matrix, which describes the dependency between the different latches. Additionally, the authors give a blocking strategy for the clustering, which forbids clustering across certain borders, which turned out to produce too large clusters.

Chauhan et. al. [CCJ+01] presented a heuristic that creates a parse tree resulting in a non-linear quantification schedule. The heuristic argument is to perform the cheapest conjunctions first. This strategy allows a dynamic conjunction scheduling, although experiments are given only for the fixed scheduling heuristics.

Cabodi et. al. [CCQ02] presented an improved benefit heuristic, which avoids the reordering problem mentioned in the section above and an adaptive clustering approach also allowing dynamic conjunction scheduling and clustering.

Except for [CCJ+01], the basic idea of these approaches still is ordering of bit-relations.

3.2 RTL Partitioning

In this section we will discuss the basic concept of using high-level information to improve image computation. The heuristic described here is named *RTL-heuristic*, because we utilize constructs typical for descriptions on register transfer level (RTL). After a description of the heuristic, we will discuss its benefits and the underlying mechanisms. Experiments prove the effectiveness of this approach.

3.2.1 Hardware Description Languages

Since modern complex designs require a structured hierarchical description to be feasible they are currently written in a hardware description language (HDL) at register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of *data flow* and *behavioral constructs*. Logic synthesis tools take the RTL description to produce an optimized gate level netlist and high level synthesis tools at the behavioral level output RTL descriptions. Verilog [TM91] and VHDL [HJ96] are the most popular HDLs used for describing the functionality at RT level. Within the design cycle of optimization and verification the RTL level is an important and frequently used part.

The design methodology in Verilog is a top down hierarchical modeling concept based on modules, which are the basic building block. Our experimental work is based on designs written in this language, but our approaches can easily be extended to other HDLs or any hierarchical finite state machine representation as it is, e. g. provided by state space decomposition algorithms (see. e. g. [MJH+98]).

3.2.2 RTL based Partitioning Heuristic

The way to build a complex design is to break it into modules, each with a dedicated functionality and a smaller complexity. For example communication protocols contain transmitter and receiver that represent independent modules. These modules are usually not too complex, thus the complexity of their TRs will be small. If a partition contains state variables of several modules, we need to represent the Cartesian product of these modules leading to a much more complex TR. The main reason for the efficiency of the partitioned TR approach is that state variables not appearing in other partitions are quantified out during the AndExist operation. This leads to much smaller OBDD-sizes and a faster computation. If the state variables of a module are spread over several partitions, the quantification does take effect only late in image computation. Therefore, most of the computation has to be done with large OBDDs.

RTL level description languages like Verilog support a hierarchical design methodology by providing module constructs. As it can be seen this modularization has effects on the image computation (see discussion below) that should not be neglected.

Although the IWLS95 method optimizes the partitioning twice (clustering and scheduling), its main disadvantage is that it only uses structural information about the latches to optimize the partitioning for an efficient order for the AndExist operation during the image computation.

We propose a method that changes the paradigm of ordering the bit-relations as it is used by other methods to a grouping paradigm. Therefore, additional semantical information about the represented functions is included in the partitioning process. As an analysis and experimental results show, there is a close connection between the RTL description and an efficient image computation.

We call this heuristic *RTL heuristic*. It proceeds in three steps:

1. **Group latches:** Each Verilog RTL description contains a main module. The modules that are instantiated in this main module form the groups of our approach. Usually, only a few, but largely independent, modules are instantiated in the main module, leading to a reasonable granularity of the groups. There is no special ordering strategy for the latches within a group. Just to avoid a random ordering we order the latches lexicographically by their internal names (names of latches from submodules are prefixed by the submodule name). Also, the bits of a certain register are named by the register and the bit number. A side effect of this sorting is, that latches of a submodule within the group stay adjacent, without being grouped explicitly. The same holds for the bits of a register.

2. **Cluster groups:** The groups represent borders for the clusters. It is not allowed for a cluster to contain latches from different groups. To control the OBDD-size of the clusters, the partition-size threshold from the IWLS95 method is applied *within* the groups. The clustering given by the groups lowers the influence of the arbitrary clustering produced by the OBDD-size threshold. Thus, resulting in a more *natural* partitioning, i. e. a new cluster is created with any new top-level semantical entity.

3. **Order clusters:** (optional) In the last step the clusters may be ordered by e. g. using the benefit heuristic from the standard method.

Figure 3.3 gives an overview of this strategy.

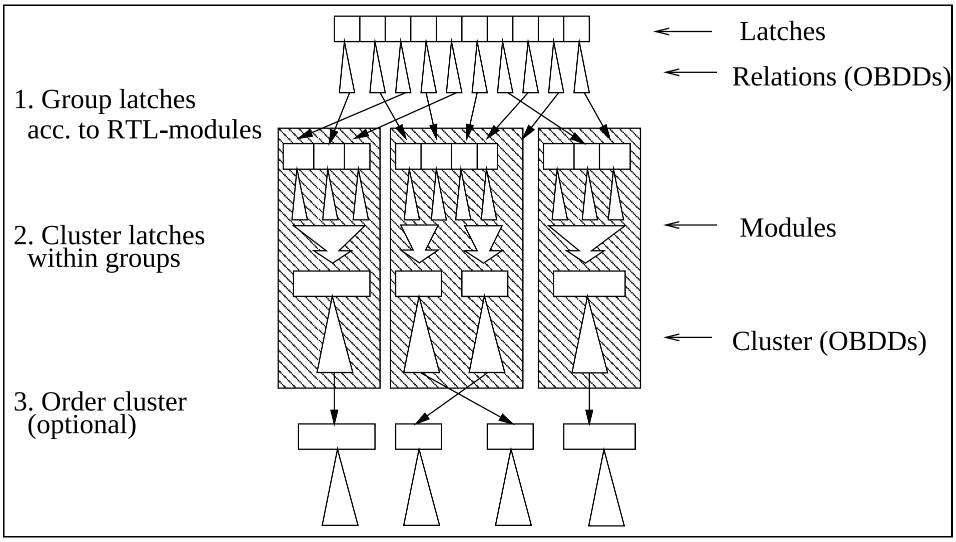


Figure 3.3: Schematic of RTL-based Partitioning Strategy.

Modifications of this strategy could be:

- **Step 1a)** As an additional step the benefit heuristic of the standard method may be applied to order the latches within the single groups. It emerged that in our case the lexicographic order of the latches – especially those that form registers – preserves more of the structure of the design and leads to better results.

- **Step 2a)** One may allow to create clusters that cross a group border. This will lead to a more compact representation of the TR with fewer clusters. Although the representation is more efficient the image computation does not perform as efficient as with the strict group borders. An explanation for this behavior is given below.
- **Step 3)** It turned out that the benefit heuristic does not help improving our results, thus Step 3 is omitted in our experiments.

3.2.3 Discussion of the Method

In the following we will discuss the influence of different partitioning schemes on the image computation of designs that are inherently modularized. Then, we will take a look how the RTL heuristic performs on the primary target of the IWLS heuristic that is minimization of the number of variables involved in the product computation.

Grouping Effects

For the ease of understanding we consider a hypothetical communication protocol consisting of a transmitter (Tx) and a receiver (Rx) (see Figure 3.4). The state variables of Tx are $t_0, \ldots, t_m$ and the corresponding transition functions are $\delta_{t_0}, \ldots, \delta_{t_m}$. The state variables of Rx are $r_0, \ldots, r_n$, the corresponding transition functions are $\delta_{r_0}, \ldots, \delta_{r_n}$.

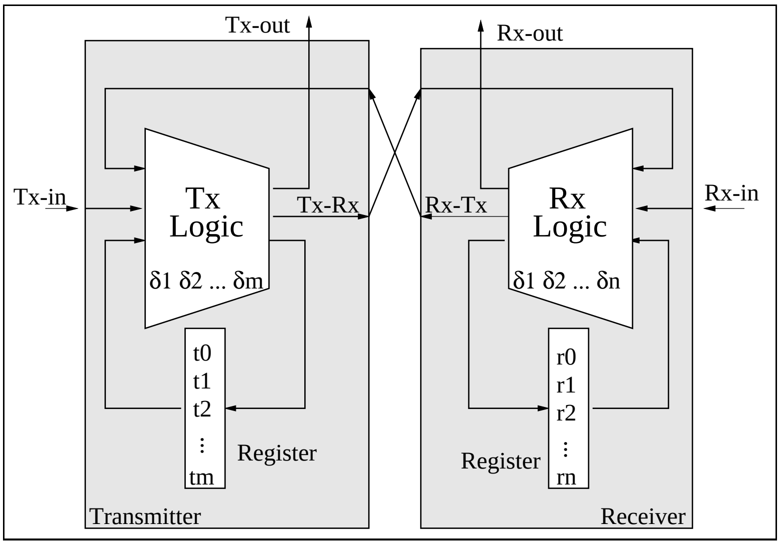


Figure 3.4: Schematic of the Communication Protocol.

A common greedy partitioning strategy merges δ_ts and δ_rs until the threshold for the OBDD-size of the partition is exceeded. We can expect that δ_ts and δ_rs appear in every partition. Hence, every partition depends on many of the variables $t_0, \ldots, t_m, r_0, \ldots, r_n$. This kind of "interleaved" partitioning will not have negative effects on monolithic controllers, but it will have negative effects on modularized controllers:

1. The quantification operation ($\exists$) included in the AndExist operation takes effect lately, resulting in large OBDDs in the preceeding computations.
2. If the partitions depend on all variables, the AND part within the AndExist is a complete AND operation with a worst case complexity of $O(|I_i| \cdot |P_i|)$, where P_i is the

OBDD of the ith partition and I_i is the intermediate OBDD of the $(i-1)$th iteration (The AndExist operation does not have polynomial runtime).

Even if the partitions are almost separated, but e. g. δ_{r_0} is represented in a partition of only δ_ts, the OBDD for δ_{r_0} cannot share any nodes with the OBDDs for the other functions. This results in an unnecessary large partition that is again depending on all variables.

We have a different situation, if we use a modularized partitioning: The set *From* of already reached states and the resulting Image (*TO*) of every iteration of the reachable states computation are independent of the chosen partitioning scheme. The OBDDs of *From* and *To* may be different due to variable reordering, but the main reason for the better performance is the different partitioning scheme: Only transition functions of one type (δ_t resp. δ_r) are merged into one partition and iterated consecutively. This scheme performs better for the following reasons:

1. The AND part of the AndExist operation is performed on a smaller number of variables. During the ith iteration the partition P_i and the intermediate result I_i share only a fraction of variables. The complexity for the AND operation of two OBDDs A and B with totally disjoint variable sets is $O(|A| + |B|)$. The AND part in the modularized scheme is much closer to this complexity than the AND part in the interleaved scheme.
2. If the end of a module is reached, the abstract operation will quantify out most variables of this module, resulting in smaller OBDDs.

Variable Balances

During the iterated image computation next-state variables (y) are introduced while present-state variables (x) are quantified out. The goal of the standard heuristic is to minimize the sum of these variables The diagram in Figure 3.5 shows this balance for a typical example of our benchmarks (p62_L_L_V02). The solid line (Orig) shows the result for the IWLS95-method. The balance of variables quickly reaches a maximum of 73 additional variables and then, decreases down to a balance of 0 additional variables after 22 clusters.

The RTL heuristic (dashed line) produces a totally different result. Although, the heuristic is not tuned to this target the balance is smaller (max. 49). Furthermore, the number of additional variables decreases several times, i. e. when crossing a group border.

The obvious differences in the number of variables is related to the improvement in the quality of the partitioning for the RTL method. Remarkably, the RTL heuristic outperforms the standard method even in a structural analysis, giving a good argument for the effectiveness of this approach.

3.2.4 Experimental Results

In this section we give a proof of our concept by performing model checking with public-domain benchmarks.

Implementation

We implemented our algorithms in the VIS-package [VIS] (version 1.3) using the underlying CUDD-package [Som] (version 2.3.0). VIS is a popular verification and synthesis package

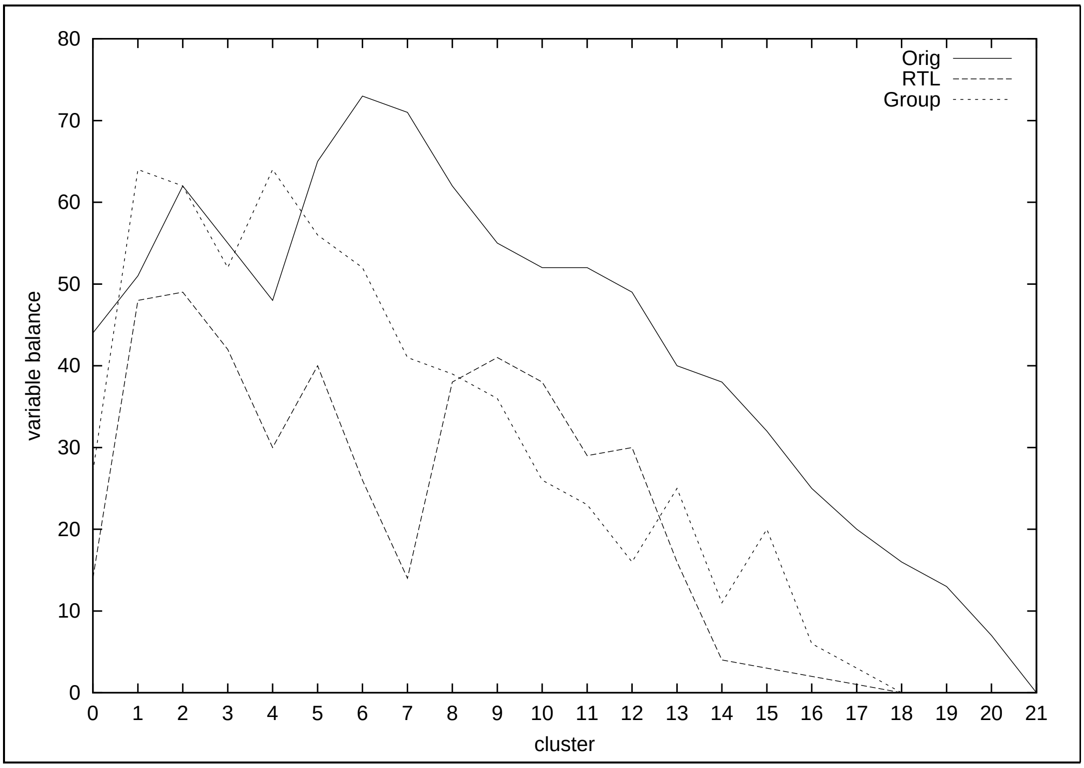


Figure 3.5: Variable Balances for different Heuristics.

in academic research. It inherits state of the art techniques for OBDD manipulation, image and reachable states computation as well as formal verification techniques. Together with the vl2mv translator VIS provides a Verilog front-end.

Benchmarks

For our experiments we used Verilog designs from the Texas97 benchmark suite [Azi97] and examples from the VIS distribution. This publicly available benchmark suite contains real life designs including:

- MSI Cache Coherence Protocol,
- PCI Local BUS,
- PI BUS Protocol,
- MESI Cache Coherence Protocol,
- MPEG System Decoder,
- DLX and
- PowerPC 60x Bus Interface.

The benchmark suite also contains properties given in CTL formulas for verification.

We chose those designs that represent RTL rather than gate level descriptions. Only designs were considered, whose transition relation could be build respecting our system limitations.

We computed 24 different benchmarks for which one or two sets of properties have been checked (resulting in 36 experiments). The runtime heavily depends on the chosen set of properties and is not proportional to the number of image computations. Therefore, it is reasonable to check more than one set of properties per benchmark. Some very small examples (CPU time < 5s) are not shown.

Experimental Setup

We left all parameters of VIS and CUDD unchanged. The most important default values are:

- Partition cluster size = 5000,
- partition method for MDDs = frontier,
- OBDD variable reordering method = sifting and
- first reordering threshold = 4004 nodes.

The model checking was preceeded by an explicit call to variable reordering (i. e. Sifting). The CPU time was limited to 6 CPU hours and memory usage was limited to 200MB. For runtime comparison, all experiments were performed on Linux PentiumIII 500MHz workstations. These settings are valid for all experiments we performed using VIS.

Results

The results of the experiments are shown in Table 3.1 and Table 3.2 We compared our RTL-method with the standard partitioning method of VIS, which is the IWLS95 method.

In Table 3.1 the runtimes of the two methods are compared. Icmp gives the number of image computations performed during reachable states computation and model checking. The CPU time is given in seconds. For statistics the benchmarks exceeding the time limit (>6h) have been calculated as 6h+1s. The improvement is given in % ($-100 < \% < 100$) and is calculated as follows:

$$a \geq b : \% = (1 - b/a) \cdot 100,$$

$$a < b : \% = (a/b - 1) \cdot 100.$$

The RTL-method wins 25 out of the 36 cases. The improvements range up to 91%. The RTL-method finishes three benchmarks more than the IWLS-method. This results in an overall improvement in time of 27%. The RTL-method shows significant losses only for three benchmarks: For TWO, ethernet and the somewhat pathological (because of 65326 image computations) packet.

For a comparison of OBDD-sizes see Table 3.2. The maximum number of live peak-nodes is given by Peakn, The number of nodes in the transition relation is given by TRn and the number of clusters in the transition relation is given by Parts.

The overall peak node size of the RTL-method basically remains unchanged in comparison to the IWLS95-method. The RTL-method requires 10% less nodes for the representation of the transition relation, while it uses 15% more clusters for its representation. This demonstrates that the RTL-method introduces additional but meaningful borders for clusters as the improvements in CPU time show.

Concluding it can be said that the RTL-method performs stable, improving the computation time and even enlarges the applicability of model checking.

	IWLS95		RTL-method	
	lcmp	time/s	time/s	%
`ONE.mv.PPCliveness.c`	40	7	9	-22
`ONE.mv.contention.ct`	19	6	8	-25
`PCIabnorm.mv.PCI.ctl`	304	253	167	33
`PCInorm.mv.PCI.ctl.o`	206	56	39	30
`TWO.mv.PPCliveness.c`	74	303	9364	-96
`TWO.mv.contention.ct`	37	47	125	-62
`ethernet.define.213.`	303	3321	14441	-77
`gcd.mv.gcd.ctl.out`	37	20	37	-45
`minMax30.mv.minMax30`	8	21	12	42
`multi_main.mv.multim`	45	34	18	47
`p62_LS_LS_V01.mv.ccp`	64	200	140	30
`p62_LS_LS_V01.mv.p6l`	99	831	444	46
`p62_LS_L_V01.mv.ccp.`	64	210	151	28
`p62_LS_L_V01.mv.p6li`	99	3511	335	90
`p62_LS_S_V01.mv.ccp.`	64	210	153	27
`p62_LS_S_V01.mv.p6li`	99	3601	318	91
`p62_L_L_V01.mv.ccp.c`	52	189	127	32
`p62_L_L_V01.mv.p6liv`	87	934	483	48
`p62_L_S_V01.mv.ccp.c`	75	121	122	0
`p62_L_S_V01.mv.p6liv`	118	231	278	-16
`p62_ND_LS_V01.mv.ccp`	83	830	779	6
`p62_ND_LS_V01.mv.p6l`	128	21039	13370	36
`p62_ND_L_V01.mv.ccp.`	75	781	788	0
`p62_S_S_V01.mv.ccp.c`	43	101	61	39
`p62_S_S_V01.mv.p6liv`	80	106	112	-5
`p62_V_LS_V01.mv.ccp.`	108	587	368	37
`p62_V_LS_V01.mv.p6li`	153	>6h	16270	24
`p62_V_S_V01.mv.ccp.c`	82	245	169	31
`p62_V_S_V01.mv.p6liv`	127	2168	595	72
`packet.mv.packet.ctl`	65326	5122	7258	-29
`single_main.mv.singl`	108	13	8	38
`single_main.mv.singl`	52	13	8	38
`three_processor.mv.p`	244	>6h	8276	61
`three_processor_bin.`	122	>6h	5708	73
`two_processor.mv.pro`	264	676	75	88
`two_processor_bin.mv`	140	150	33	78
Total	69119	110740	80649	
Improvement			27%	

Table 3.1: Comparison of CPU time for IWLS95 and RTL method

	IWLS95			RTL-method					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
`ONE.mv.PPCliveness.c`	19185	3	4456	21229	-9	4	-25	2671	40
`ONE.mv.contention.ct`	17784	3	4456	21229	-16	4	-25	2671	40
`PCIabnorm.mv.PCI.ctl`	176276	14	28613	128310	27	10	28	24482	14
`PCInorm.mv.PCI.ctl.o`	81123	15	35124	69291	14	10	33	20646	41
`TWO.mv.PPCliveness.c`	263756	8	13118	5363742	-95	9	-11	12828	2
`TWO.mv.contention.ct`	97622	7	11865	223656	-56	10	-30	12040	-1
`ethernet.define.213.`	980429	6	14907	6239111	-84	17	-64	1859	87
`gcd.mv.gcd.ctl.out`	215222	2	7470	215222	0	3	-33	6466	13
`minMax30.mv.minMax30`	101042	3	7355	101042	0	2	33	6170	16
`multi_main.mv.multim`	38694	5	14700	33423	13	4	19	3092	78
`p62_LS_LS_V01.mv.ccp`	166074	23	49952	136280	17	20	13	56040	-10
`p62_LS_LS_V01.mv.p6l`	452267	23	49952	463209	-2	20	13	56040	-10
`p62_LS_L_V01.mv.ccp.`	176540	23	49684	174275	1	18	21	53012	-6
`p62_LS_L_V01.mv.p6li`	1617162	23	49684	293189	81	18	21	53012	-6
`p62_LS_S_V01.mv.ccp.`	176540	23	49684	174275	1	18	21	53012	-6
`p62_LS_S_V01.mv.p6li`	1614473	23	49684	293189	81	18	21	53012	-6
`p62_L_L_V01.mv.ccp.c`	164244	23	48961	177614	-7	18	21	54445	-10
`p62_L_L_V01.mv.p6liv`	477543	23	48961	507110	-5	18	21	54445	-10
`p62_L_S_V01.mv.ccp.c`	168782	22	62479	144687	14	18	18	55820	10
`p62_L_S_V01.mv.p6liv`	192410	22	62479	255775	-24	18	18	55820	10
`p62_ND_LS_V01.mv.ccp`	396642	24	63506	430404	-7	20	16	54629	13
`p62_ND_LS_V01.mv.p6l`	5583160	24	63506	4091039	26	20	16	54629	13
`p62_ND_L_V01.mv.ccp.`	356794	25	65964	436916	-18	21	16	59711	9
`p62_S_S_V01.mv.ccp.c`	147063	23	62209	133192	9	18	21	56727	8
`p62_S_S_V01.mv.p6liv`	153012	23	62209	163585	-6	18	21	56727	8
`p62_V_LS_V01.mv.ccp.`	283494	24	58415	307382	-7	19	20	58007	0
`p62_V_LS_V01.mv.p6li`	4483034	24	58415	5450872	-17	19	20	58007	0
`p62_V_S_V01.mv.ccp.c`	213245	23	61795	157448	26	19	17	53163	13
`p62_V_S_V01.mv.p6liv`	964988	23	61795	345774	64	19	17	53163	13
`packet.mv.packet.ctl`	53790	3	9704	62574	-14	4	-25	5191	46
`single_main.mv.singl`	14936	2	6352	9360	37	3	-33	2227	64
`single_main.mv.singl`	14936	2	6352	9360	37	3	-33	2227	64
`three_processor.mv.p`	4621235	9	19750	4588114	0	4	55	5362	72
`three_processor_bin.`	8779857	7	20387	3257556	62	4	42	5358	73
`two_processor.mv.pro`	903917	4	12311	86871	90	3	25	2838	76
`two_processor_bin.mv`	252974	4	11610	63296	74	3	25	2831	75
Total	34420245	538	1307864	34629601		454		1168380	
Improvement				0%		15%		10%	

Table 3.2: Comparison of OBDD-sizes for IWLS95 and RTL method

3.3 Group Partitioning

The RTL heuristic significantly outperforms the standard method for partitioning. The reason for this lies in the usage of information about the system's main modules. In this chapter we will introduce the notion of a *cluster dependency matrix* (CDM) that describes the interaction between the clusters of the transition relation. We will see that the CDM of the RTL method is more structured and sparser than the CDM of the IWLS95 method. Then, We will develop a heuristic that is targeted towards a structured and small CDM. The idea of this heuristic is to find groups of strongly related latches. This heuristic is more general than the RTL heuristic as it does not require RTL information and thus can be applied to designs that are not given in an HDL or where the RTL information is not given (e. g. after optimization). Nevertheless, this heuristic also assumes a structured modular design.

3.3.1 Dependency Matrices

The quality of the partitioning can be described by a cluster dependency matrix CDM. Entry (i, j) of the CDM contains the number of support variables that cluster i and cluster j have in common. As the number of common variables gets higher the dependency increases.

Figure 3.6 gives the CDM of one of the benchmark examples (p62_L_L_V02), when the standard method is used (the CDM is symmetric, so the lower triangle part of the matrix has been omitted). The TR resulting from the standard method has 24 clusters. It can be seen that dependencies between all clusters exist and many of them are quite high. The maximum number is 62 common variables. There is no structure observable in this CDM.

Figure 3.7 shows the resulting CDM for the RTL method, given the same example. It is easy to see that:

- Some clusters are not connected (shown by empty fields). The ordering of clusters that are not connected does not have any influence on the performance of the image computation.
- The dependency is small on module borders (cluster 0-1, 9-10, 17-18). This means that at the end of a module most variables have been quantified out, resulting in a decrease in OBDD-size.
- The overall dependency is smaller (maximum at 37 common variables). A generally smaller number of common variables indicates smaller OBDD-sizes throughout the whole computation.
- The TR resulting from the RTL approach is very structured. This results in smaller OBDDs and thus fewer clusters. The number of clusters reduced to 21.

All the points mentioned above indicate a better *early quantification*, and thus a more efficient AndExist operation.

3.3.2 The Grouping Algorithm

The CDM is the result of the partitioning process. As we have seen as structured CDM will lead to a more efficient image computation. The usage of additional RTL information

o	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0		53	50	21	45	12	12	38	31	41	17	13	10	27	34	16	26	20	17	17	17	13	3	16
1			62	21	47	14	20	34	35	48	18	7	4	23	27	12	21	18	15	21	19	12	7	21
2				21	47	14	14	34	35	48	13	7	4	23	27	12	21	18	15	20	17	12	7	21
3					16	14	14	20	22	21	18	20	12	13	9	11	9	19	19	19	22	13	12	15
4						6	6	25	26	38	19	12	4	33	23	6	16	18	15	17	19	8	3	11
5							57	55	38	7	13	34	45	24	19	25	21	18	15	13	17	22	27	45
6								52	32	7	18	29	40	24	19	25	21	18	15	14	19	22	22	40
7									57	28	11	32	45	22	20	21	22	14	11	10	11	18	21	40
8										36	5	30	26	5	4	7	4	7	7	8	9	11	27	31
9											9	7	1	15	17	5	17	11	11	21	15	8	7	17
10												23	12	29	21	15	17	27	27	22	28	14	6	15
11													34	18	11	9	11	19	19	20	20	17	20	33
12														22	14	16	16	11	8	8	12	10	16	33
13															41	21	31	23	21	23	29	15	8	27
14																36	46	32	24	25	21	15	6	28
15																	34	29	22	13	17	14	10	25
16																		29	22	25	20	15	6	30
17																			39	25	29	17	7	21
18																				25	29	17	7	21
19																					5	14	7	22
20																						18	10	21
21																							12	22
22																								25
23																								

Figure 3.6: Dependency Matrix for IWLS95 Heuristic.

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0		6	6	17	11	8				6	6	17	9	8	3						
1			26	26	31	37	3	2		10	7	8	6	8	7			1	1		
2				22	26	25				7	9	10	6	9	7			1	1		
3					37	37	5	3	4	8	10	21	9	12	7			1	1		
4						35		1	2	6	7	11	8	8	3			1	1		
5							1	1	1	8	9	12	8	14	9			1	1		
6							27	27	27	7	7	7	3	9	11			1	1		
7								28	28												
8									28												
9																					
10											26	26	31	37	12	3	2			1	1
11												22	25	25	11					1	1
12													35	37	12	5	5			1	1
13														35	7		3			1	1
14															13	1	1			1	1
15																27	27			1	1
16																	28				
17																					
18																					
19																					
20																					

Figure 3.7: Dependency Matrix for RTL Heuristic.

helped creating a structured CDM for the RTL heuristic. We will now develop a heuristic that also results in a sparse and structured CDM. We will utilize the dependencies between the latches to define a reasonable number of groups, which will play a similar role as the modules in the RTL heuristic. Therefore, we call this heuristic *Group Heuristic*.

The group heuristic proceeds in two phases:

LDM phase: In the first phase a *latch dependency matrix* (LDM) is created. An entry of the matrix LDM contains the number of variables that both latch l_i and latch l_j are depending on:

$$LDM(l_i, l_j) = |supp(l_i) \cap supp(l_j)|.$$

The matrix contains numbers of variables and not the variables itself. The runtime of this phase of the heuristic is $O(l^2 \cdot n)$ for a $l \times l$ LDM of a system with n boolean variables.

Grouping phase: During the second phase the groups of latches are determined with the help of the latch dependencies matrix LDM. The idea is to put latches that have a high number of variables in common into one group. We start with the highest dependency. All pairs of latches that have this dependency are placed in one group G_{max}:

$$l_i, l_j \to G_{max} : LDM(l_i, L_j) = max.$$

While lowering the dependency threshold t all remaining latches are put into groups G_t using the same strategy. In a second step groups with similar dependencies are merged to reduce the number of groups:

$$G_i \to G_j : |i - j| < \varepsilon.$$

The merging is done to keep a certain granularity of the groups. Too many groups would result in a high number of clusters and a decrease in performance. As we have seen from the RTL method a reasonable number of groups improves the efficiency.

The problem using this very basic approach is that there always exists a certain dependency between all latches (e. g. reset signals). There are also latches that are only very loosely coupled to other latches. Thus, in this form the heuristic would result in a single large group after all latches have been grouped.

To avoid this effect, we introduced an additional criterion for the separation of groups. The separation is realized by avoiding merges of groups, whose dependencies differ too much. Groups are no longer numbered by their dependency, but by the order in which they have been created. The difference of two group numbers now gives a better criterion for the difference in the amount of dependency. If the numbers of the group differ too much, a merge is forbidden. The runtime of this phase of the heuristic is $O(l^3 \cdot n)$. Although the runtime of this algorithm seems quite high, it can be neglected in comparison to other expensive OBDD operations like variable reordering or AndExist.

For a sketch of the grouping heuristic see Figure 3.8.

After computing the groups of latches the partitioning is computed by applying the clustering strategy of the RTL heuristic outlined in Figure 3.3.

Figure 3.9 shows the application of the group heuristic to our benchmark example. The following can be seen from this CDM:

- The overall dependency compared to the standard method is much smaller (maximum 42 variables), resulting in a smaller TR size.
- Many clusters are not connected. Thus, the relative order of these clusters does not influence the image computation.

```
ComputeGroups(LDM){
  for (dep = maxdep downto 1){ /* decrease dependency */
    forall (i,j; j<i){
      if(depmatrix[i][j] == dep){
        if(!is_grouped(i) && is_grouped(j)) add_to_group(j,i);
        if(is_grouped(i) && !is_grouped(j)) add_to_group(i,j);
        if(!is_grouped(i) && !is_grouped(j)){ /* create a new group */
          create_new_group_with(i,j);
        }
        if(is_grouped(i) && is_grouped(j) &&
          abs(group_of(i) - group_of(j))< 3){ /* merge groups */
            merge_groups_of(i,j);
        }
      }
    }
  }
}
```

Figure 3.8: Algorithm for the Group Heuristic in Pseudocode.

- The CDM is not as structured as for the RTL method, but it is comparable.

The CDM of the group method is not as structured as the CDM of the RTL method, but this result is not surprising: The RTL method is a high-level method, while the group method simulates high-level effects with low-level information, i. e. it works much more heuristically than the RTL method.

3.3.3 Reordering of Clusters

To improve the quality of the group partitioning method we will reorder the clusters i. e. change the permutation of the rows and columns of the CDM. The "worst" position in a CDM is the upper right corner of the matrix. Here, the dependency of the first and the last cluster in the schedule is given. This entry gives the number of variables that stay in the product from the first to the last cluster. The surrounding entries of the matrix describe dependencies of clusters with similar large distances. Large numbers in this area result in bad performance as the OBDD for the intermediate image will remain large during the AndExist iterations. For the RTL method this area is sparse and only small dependencies occur.

We can achieve a similar effect for the group heuristic by reordering the clusters. Since rows and columns of the CDM need to have the same permutation we cannot move empty entries to arbitrary positions. Every time we change the order of some clusters we change rows and columns of the CDM, but OBDD-size and the dependencies between the clusters are not changed.

Our algorithm works as follows:

1. First, we compute the number of very small dependencies (i. e. 0 and 1) for each cluster.
2. According to these numbers we compute a ranking of clusters. A high ranking means a hight number of empty or very small entries.
3. Then, the clusters are ordered from low to high ranking.

g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0								33	42	24	15			15	13	16	9	9	11	1	11	5		
1			34	25				3	2			18					1		5				3	
2				25				3	2			18					1		5				3	
3									1			15					2						4	
4						28	23		1	3	1		18					1	5					3
5							23			1	1		18					1	2					4
6										1			13				2							1
7									38	24	24	3	3	16	13	16	14	15	15	5	7	3		
8										30	22	4		18	15	22	17	16	17	3	13	7		
9											30		3	12	9	16	13	21	13	7	9	7		
10													4	21	9	25	13	32	20	14	5	10		
11																	1		2				2	
12																		1	2					2
13															25	40	21	27	20	9	10	7		
14																25	17	11	16	3	14	11		
15																	17	28	20	9	8	7		
16																		16	14	7	7	5		
17																			19	14	5	10		
18																				15	13	11		
19																					3	5		
20																						11		
21																								
22																								
23																								

Figure 3.9: Dependency Matrix for Group Heuristic.

For a sketch of the cluster reordering algorithm see Figure 3.10.

```
ReorderCluster(CDM){
  init group_perm to 0...m-1;
  init rank to 0;
    forall (i,j; j<i){
      if(CDM[i][j] <2) rank[i]++;
    }
  sort_increasing(group_perm,rank);
  return group_perm;
}
```

Figure 3.10: Algorithm for Cluster Reordering in Pseudocode.

Clusters with the same ranking remain in their original order. The result is a CDM, where the number of empty entries increases from left to right. We also get a very compact area for the first clusters, but this is acceptable, since during the first computational steps in the AndExist the OBDDs are not too large.

For our example the reordering results in the CDM of Figure 3.11: As mentioned above, the reordering does not change the values of the entries of the CDM. The upper left corner of the CDM is now a dense area, but the upper right corner is now almost empty. It is acceptable to have higher dependencies in the beginning of the image computation, if there are smaller dependencies at the end of the image computation.

Let us take a look at the variable balance during the image computation. In the diagram of

rg	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0		15	17	13	14	20	20	16	20	19	13	11	11	15	2	5	5	2	5		2			
1			38	24	14	24	16	13	16	15	7	3	33	5	3	3	3	3						
2				30	17	22	18	15	22	16	13	7	42	3		2	2	4	1	1				
3					13	30	12	9	16	21	9	7	24	7	3				3		1	1		
4						13	21	17	17	16	7	5	9	7		1	1	1		2		2		
5							21	9	25	32	5	10	15	14	4				1		1			
6								25	40	27	10	7	15	9										
7									25	11	14	11	13	3										
8										28	8	7	16	9										
9											5	10	9	14	1				1		1			
10												11	11	3										
11													5	5										
12														1										
13																								
14																			18		18	13		2
15																	34	18		25			3	
16																		18		25			3	
17																				15			2	
18																					28	23		3
19																							4	
20																						23		4
21																								1
22																								
23																								

Figure 3.11: Reordered Dependency Matrix of Group Heuristic.

Figure 3.5 the balance for the group heuristic (dotted line) is positioned between the RTL and the standard heuristic. More variables are introduced in comparison to the RTL method but less in comparison to the standard method. We cannot detect sharp borders as for the RTL method, but the number of variables decreases several times during the iteration.

3.3.4 Experimental Results

We implemented this heuristic in VIS. We used the same experimental setup as in Section 3.2 for our experiments. Experimental results are shown in Table 3.3 and Table 3.4 (For an description of the column headers refer to Section 3.2).

We compare the results of the Group-method with the IWLS95-method. The Group-method wins 26 of the 36 cases, resulting in a 28% overall time improvement. Unfortunately, the Group-method is not able to finish the three_processor and the three_processor_bin benchmark. For the packet benchmarks only 55803 of the 65326 image computations (i. e. 85%) could be finished within the time limit.

Otherwise, the Group-method was able to achieve remarkable improvements, e. g. the p62_V_LS_V01.p6live benchmarks could be improved from more than 6h to 413s (98%). Other large benchmarks show similar improvements. The Group-method lost on the same benchmarks as the RTL-method, i. e. TWO and ethernet.

Table 3.4 compares the OBDD-size of the IWLS-method with the Group-method. Remarkably, the Group-method was able to reduce the overall amount of live peak-nodes by 45%, while only using slightly more (2%) clusters for the representation of the transition relation. At the same time, the size of the transition relation could be reduced by 21%. This shows that the Group-method produces a compact – but nonetheless effective – partitioned transition relation.

In comparison with the RTL-method, the Group-method achieves better results on single benchmarks, but the RTL-method performs more stable overall. For a direct comparison of the RTL-method and the Group-method see Table A.1 and Table A.2.

	IWLS95		Group-method	
	lcmp	time/s	time/s	%
`ONE.mv.PPCliveness.c`	40	7	9	-22
`ONE.mv.contention.ct`	19	6	8	-25
`PCIabnorm.mv.PCI.ctl`	304	253	222	12
`PCInorm.mv.PCI.ctl.o`	206	56	57	-1
`TWO.mv.PPCliveness.c`	74	303	2139	-85
`TWO.mv.contention.ct`	37	47	106	-55
`ethernet.define.213.`	303	3321	6088	-45
`gcd.mv.gcd.ctl.out`	37	20	23	-13
`minMax30.mv.minMax30`	8	21	7	66
`multi_main.mv.multim`	45	34	24	29
`p62_LS_LS_V01.mv.ccp`	64	200	129	35
`p62_LS_LS_V01.mv.p6l`	99	831	130	84
`p62_LS_L_V01.mv.ccp.`	64	210	126	40
`p62_LS_L_V01.mv.p6li`	99	3511	226	93
`p62_LS_S_V01.mv.ccp.`	64	210	126	40
`p62_LS_S_V01.mv.p6li`	99	3601	235	93
`p62_L_L_V01.mv.ccp.c`	52	189	71	62
`p62_L_L_V01.mv.p6liv`	87	934	121	87
`p62_L_S_V01.mv.ccp.c`	75	121	69	42
`p62_L_S_V01.mv.p6liv`	118	231	116	49
`p62_ND_LS_V01.mv.ccp`	83	830	572	31
`p62_ND_LS_V01.mv.p6l`	128	21039	1798	91
`p62_ND_L_V01.mv.ccp.`	75	781	715	8
`p62_S_S_V01.mv.ccp.c`	43	101	66	34
`p62_S_S_V01.mv.p6liv`	80	106	67	36
`p62_V_LS_V01.mv.ccp.`	108	587	354	39
`p62_V_LS_V01.mv.p6li`	153	>6h	413	98
`p62_V_S_V01.mv.ccp.c`	82	245	146	40
`p62_V_S_V01.mv.p6liv`	127	2168	279	87
`packet.mv.packet.ctl`	65326	5122	>6h	-76
`single_main.mv.singl`	108	13	10	23
`single_main.mv.singl`	52	13	10	23
`three_processor.mv.p`	244	>6h	>6h	0
`three_processor_bin.`	222	>6h	>6h	0
`two_processor.mv.pro`	264	676	84	87
`two_processor_bin.mv`	140	150	80	46
Total	69119	110740	79429	
Improvement			28%	

Table 3.3: Comparison of CPU time for IWLS95 and Group method

	IWLS95			Group-method					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
`ONE.mv.PPCliveness.c`	19185	3	4456	21362	-10	5	-40	4971	-10
`ONE.mv.contention.ct`	17784	3	4456	21362	-16	5	-40	4971	-10
`PCIabnorm.mv.PCI.ctl`	176276	14	28613	184722	-4	8	42	18676	34
`PCInorm.mv.PCI.ctl.o`	81123	15	35124	64961	19	11	26	29081	17
`TWO.mv.PPCliveness.c`	263756	8	13118	2317020	-88	10	-19	11730	10
`TWO.mv.contention.ct`	97622	7	11865	220979	-55	10	-30	15056	-21
`ethernet.define.213.`	980429	6	14907	1614232	-39	14	-57	6471	56
`gcd.mv.gcd.ctl.out`	215222	2	7470	215222	0	3	-33	5366	28
`minMax30.mv.minMax30`	101042	3	7355	101042	0	2	33	3151	57
`multi_main.mv.multim`	38694	5	14700	33423	13	5	0	4837	67
`p62_LS_LS_V01.mv.ccp`	166074	23	49952	119996	27	23	0	44697	10
`p62_LS_LS_V01.mv.p6l`	452267	23	49952	127833	71	23	0	44697	10
`p62_LS_L_V01.mv.ccp.`	176540	23	49684	118284	32	23	0	43117	13
`p62_LS_L_V01.mv.p6li`	1617162	23	49684	197326	87	23	0	43117	13
`p62_LS_S_V01.mv.ccp.`	176540	23	49684	118284	32	23	0	43117	13
`p62_LS_S_V01.mv.p6li`	1614473	23	49684	197326	87	23	0	43117	13
`p62_L_L_V01.mv.ccp.c`	164244	23	48961	121246	26	23	0	44027	10
`p62_L_L_V01.mv.p6liv`	477543	23	48961	121246	74	23	0	44027	10
`p62_L_S_V01.mv.ccp.c`	168782	22	62479	107565	36	23	-4	45494	27
`p62_L_S_V01.mv.p6liv`	192410	22	62479	117042	39	23	-4	45494	27
`p62_ND_LS_V01.mv.ccp`	396642	24	63506	291233	26	24	0	51339	19
`p62_ND_LS_V01.mv.p6l`	5583160	24	63506	821661	85	25	-4	49484	22
`p62_ND_L_V01.mv.ccp.`	356794	25	65964	306682	14	24	4	52423	20
`p62_S_S_V01.mv.ccp.c`	147063	23	62209	97647	33	23	0	44041	29
`p62_S_S_V01.mv.p6liv`	153012	23	62209	97647	36	23	0	44041	29
`p62_V_LS_V01.mv.ccp.`	283494	24	58415	211863	25	24	0	46795	19
`p62_V_LS_V01.mv.p6li`	4483034	24	58415	242396	94	24	0	46795	19
`p62_V_S_V01.mv.ccp.c`	213245	23	61795	145859	31	24	-4	47214	23
`p62_V_S_V01.mv.p6liv`	964988	23	61795	204216	78	24	-4	47214	23
`packet.mv.packet.ctl`	53790	3	9704	51592	4	5	-40	10906	-11
`single_main.mv.singl`	14936	2	6352	9668	35	3	-33	2238	64
`single_main.mv.singl`	14936	2	6352	9668	35	3	-33	2238	64
`three_processor.mv.p`	4621235	9	19750	4722353	-2	9	0	13522	31
`three_processor_bin.`	8779857	7	20387	5086098	42	9	-22	13902	31
`two_processor.mv.pro`	903917	4	12311	75205	91	3	25	5027	59
`two_processor_bin.mv`	252974	4	11610	131104	48	3	25	6305	45
Total	34420245	538	1307864	18645365		553		1028698	
Improvement				45%		-2%		21%	

Table 3.4: Comparison of OBDD-sizes for IWLS95 and Group method

Chapter 4

Hierarchization

In the last chapter we have shown how to improve the partitioning process by grouping latches. We have done this by either utilizing RTL information given by the design or by building groups from the latch dependencies. Nevertheless, both strategies result in a somewhat *flat* partitioning, i. e. all groups are equal in the way that no group can be favored. This makes it hard to find an order for the conjunction of the groups.

In this chapter we will introduce the concept of *hierarchical partitions* that extends the grouping techniques of the last chapter. With the use of a hierarchical partitioning we are able to perform *hierarchical image computation*, which is more efficient than the linear image computation and has a larger freedom for a choice of the conjunction scheduling. Almost naturally a *dynamic conjunction scheduling strategy* results for the hierarchical partitioning. We will develop a strategy that improves the performance of the AndExist algorithm.

The chapter is structured as follows: First, we describe how to build a hierarchical transition relation from a hierarchical FSM description, i. e. designs given in the Verilog description language. We also introduce an algorithm called *HierarchicalAndSmooth* that computes the image based on a hierarchical transition relation. In the second section we describe our dynamic conjunction scheduling heuristic. Therefore, we will take a closer look at the AndExist algorithm. Benchmark experiments conclude this section. In the last section we develop – similar to the last chapter – a heuristic for building a hierarchical transition relation that is independent of an explicit hierarchical FSM description. The *hierarchical grouping heuristic* analyzes the latch dependencies and exploits the advantages of a hierarchical transition relation. For this heuristic we perform benchmark experiments, too.

4.1 Hierarchical Image Computation

In this section we will develop a strategy for the construction of a hierarchical transition relation. The strategy generalizes the modular partitioning presented in the last chapter. In contrast to the RTL-method that stopped after separating the main modules of a design, the new strategy will exploit the complete hierarchy of the design. As a main result we obtain more *natural* partitions, i. e. the influence of the partition size threshold is drastically reduced. To complete this framework, we describe the algorithm for computing an image based on a hierarchical transition relation.

This hierarchical partitioning concept is based, also for the ease of understanding, on Verilog, but other types of hierarchical descriptions are possible.

Figure 4.1a shows the structure of a hierarchical FSM description typical for a hardware description at RT level. Circles represent modules i. e. subFSMs. An arrow pointing from module A to module B denotes the instantiation of a submodule B within the module A. Boxes denote latches controlling the state of the respective (sub)FSM. Multiple instantiations of the same module definition appear as different (sub)trees in the structure.

Figure 4.1b shows the concept of a hierarchical TR. The TR itself is solely represented by the OBDD-clusters (shown as triangles). The tree connecting the OBDD-clusters has been adopted from the hierarchical FSM description and serves only for structuring the TR.

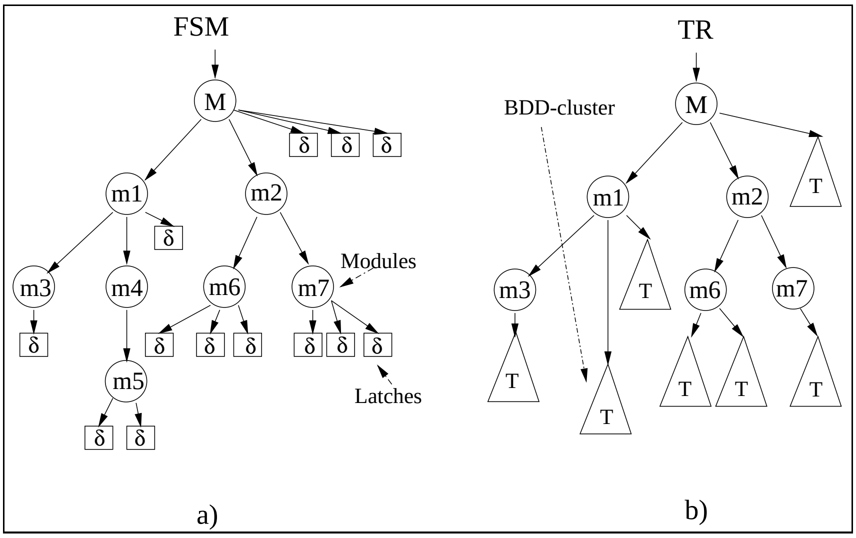


Figure 4.1: Hierarchical FSM and Transition Relation.

4.1.1 Hierarchical Partitioning

In the following we will describe the hierarchical partitioning and clustering algorithm.

The *HierarchicalCluster* algorithm takes the tree of modules and submodules of a hierarchical FSM as it is described e. g. at RT-level as input and computes a hierarchical partitioned transition relation. For an outline of the algorithm see Figure 4.2.

```
HierarchicalCluster(module,threshold){
   /* First, cluster the modules own latches */
   mv_relations =
      PreclusterMVLatches(module→latches,threshold*2 );
   module→latch_cluster =
      CreateClusters(mv_relations,threshold);

   /* Then, cluster the children of the module */
   ForEachItem(module→children, child){
       HierarchicalCluster(child,threshold);
   }
}
```

Figure 4.2: Algorithm for Hierarchical Partitioning in Pseudocode.

The algorithm traverses the modules in breadth first style, beginning with the main module. The algorithm consists of two major steps:

1. The latches of the current modules are clustered. This is done by the same scheme as in the RTL-heuristic (see Section 3.2), i. e. latches are conjuncted to a cluster until a given threshold for the OBDD-size is exceeded.

2. The children (submodules) of the current module are visited recursively.

The *CreateCluster* routine is preceeded by a routine called *PreclusterMVLatches*. This routine clusters latches that represent a multivalued (MV) register (i. e. more than one bit). During this routine different MV-registers are not merged. Also, single bit latches remain as they are. The preclustering routine puts additional structure in the partitioning, because in the *CreateCluster* routine no ordering or grouping of latches of the module takes place. *PreclusterMVLatches* assures that latches of a MV-register stay in adjacent clusters or in the best case in one single cluster. We think that grouping MV-registers is very important and thus, we allow a clustering OBDD-size threshold that is double the size of the regular threshold. If a cluster resulting from preclustering a MV-register has fewer OBDD-nodes than the regular partition-size threshold, other latches are conjuncted to it during the *CreateCluster* routine.

In the second step the whole process is repeated in the recursion for the submodules of the design. There is no merging of clusters between different modules and between different levels of the hierarchy. For a schematic of the hierarchical clustering see Figure 4.3

Remark:

The *HierarchicalCluster* algorithm works for any tree-like representation of the latches. It is not necessarily required to have a HDL description like Verilog. Only the PreclusterMVLatches routine requires an explicit description of the MV-latches. If this information is missing, the routine can be skipped.

Discussion of the Partitioning Algorithm

Figure 4.4 shows the CDM of our standard example when the hierarchical partitioning method is applied. The numbers of the clusters give the order, in which they have been created.

Additionally to the CDM, the clustering algorithm outputs information about how the modules have been clustered. By definition no cluster contains latches from different (sub)modules. Thus, modules can be clearly identified in the CDM:

- The main module is represented by cluster 0.
- Module 0 includes cluster 1–10 with submodules 1–3, 4–9 and 10.
- Module 1, which is structurally the same as Module 0, includes cluster 11–21 with submodules 11–14, 15–20 and 21.
- The last module is represented by clusters 22 and 23.

See also Figure 4.3

We can clearly see that there is almost no communication between module borders, i. e. no dependency between clusters. Even at submodule borders the dependency is very small. The dependency is relatively high (up to 42) within a module for directly neighbored clusters (the diagonal above the main diagonal). This is due to splits between semantically related latches

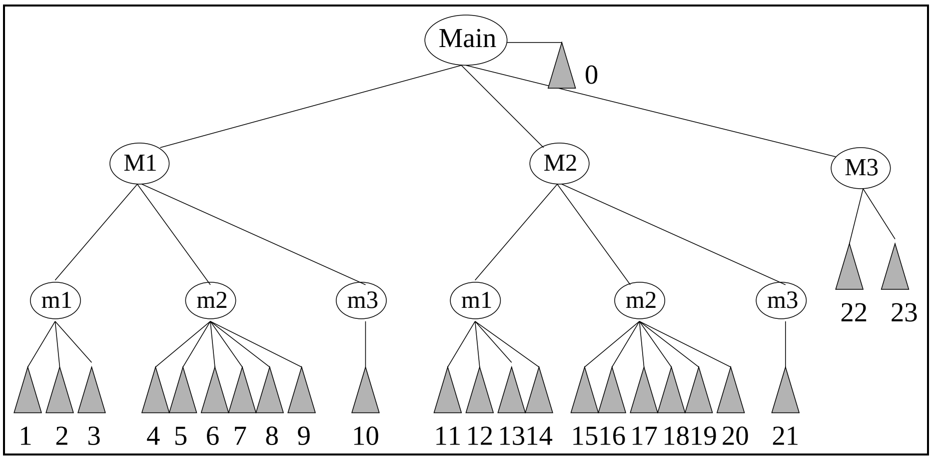


Figure 4.3: Result of the Modular Clustering.

that appear when the clustering threshold for the OBDD-size is exceeded. Although the CDM gives not a schedule for the partitioning, the order of the creation of the clusters, i. e. the cluster number indicates a good quality of the partitioning: Generally, the dependency decreases from left to right, resp. from bottom to top. There is no dependency between the last clusters and the very first clusters (the upper right corner of the CDM).

Using the hierarchical partitioning strategy the arbitrary influence of the partition-size threshold can be drastically reduced. The hierarchy produces more natural and more meaningful cluster borders than just an OBDD-size threshold. It might happen that the hierarchical partitioning produces clusters that are too small and could have been merged with other clusters without affecting the efficiency. Anyway, we left the strategy in this pure form for not to dilute the hierarchical concept.

4.1.2 Hierarchical Image Computation

The algorithm that performs the iteration of AndExist operations to compute an image $Img(TR, From)$ of a given state set $From$ with a transition relation TR is called *LinearAndSmooth*, because the clusters of the transition relation are iterated in a *linear* list order (see Figure 4.5a).

The algorithm that computes the image based on an hierarchical partitioning is called *HierarchicalAndSmooth* and replaces the *LinearAndSmooth* (see Figure 4.5b). The *HierarchicalAndSmooth* is structured similarly to the *HierarchicalCluster* algorithm (for an overview see Figure 4.6).

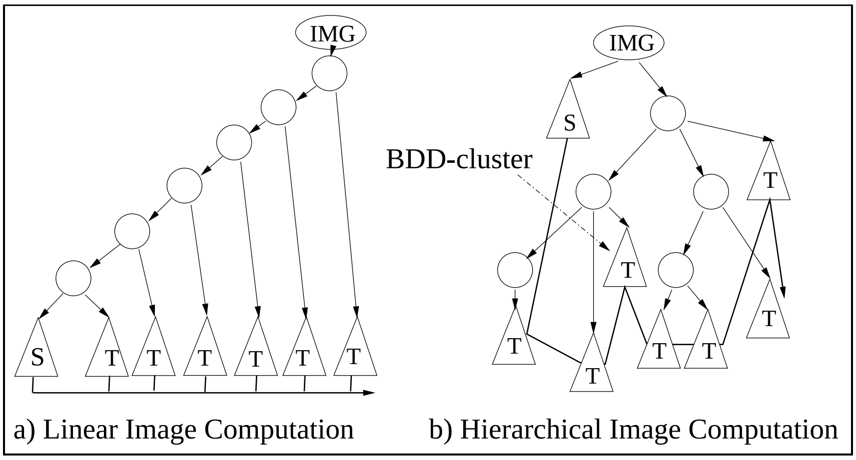
The hierarchical partitioning tree, which is isomorph to the module tree is traversed in breadth first style:

The image computation starts with the main module. First, the modules own clusters are processed. The order in which the clusters are taken into the product is determined by the routine *ChooseBestCluster*, which can result simply the list order or an order computed by a heuristic. After a cluster has been chosen for the next product, its smoothing variables have to be computed, i. e. those variables that do not occur in any later product and thus can be quantified out. This is done by the routine *ComputeSmoothVars* that traverses the complete partitioning tree to compute the variables to be quantified out.

The *ComputeSmoothVars* operation needs to be executed before every AndExist operation,

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0					2	2	2	2	2	2						2	2	2	2	2	2			
1			28	23	3	4	5	3	2		7													
2				23	3	1	5			3	7													
3							1				5													
4						42	34	42	18	38						18	10	9	12	5	9		8	5
5							34	41	13	29						12	10	8	12	3	6		9	6
6								37	22	32						9	6	18	7	8	10		12	4
7									18	29						14	10	10	12	5	7		9	5
8										17						5	3	7	3	5	5		5	1
9																9	5	10	5	5	10		6	4
10																								
11													28	28	28	3	3	1	3			7		
12														28	28	3	2	1	2		2	7		
13															30	3	1	4	1		3	7		
14																3	1	2	1		1	7		
15																	39	32	37	17	39		8	5
16																		26	38	12	27		9	6
17																			26	23	35		12	4
18																				11	23		8	5
19																					20		6	1
20																							6	4
21																								
22																								18
23																								

Figure 4.4: Dependency Matrix of the Hierarchy Heuristic.



a) Linear Image Computation b) Hierarchical Image Computation

Figure 4.5: Linear and Hierarchical Image Computation.

since the order of conjunctions may change with every image computation. Only if the order is fixed, e. g. when the list order is chosen, the quantification variables can be computed once and then be stored with the according cluster.

The chosen cluster is then taken into the product by AndExist (resp. Boolean AND if there are no quantify variables in the cluster).

After all clusters of the module have been processed, the image computation continues with the submodules (children) of the current module in the same way as described above. Again, the order in which the children are processed can either be determined by a heuristic (*ChooseBestSubmodule*) or simply be the list order.

```
HierarchicalAndSmooth(fromSet,module){
  product = fromSet;
  clustersremaining = module→cluster
  while(clustersremaining){
    cluster = ChooseBestCluster(clustersremaining);
    smoothVars = ComputeSmoothVars(cluster);
    if (smoothVars)
      tmpProduct = bdd_AndExist(product,cluster,smoothVars);
    else
      tmpProduct = bdd_And(product, cluster);
    product = tmpProduct;
    remove_from(clustersremaining,cluster);
  }
  childrenremaining = module→children;
  while(childrenremaining){
    child = ChooseBestSubmodule(childrenremaining);
    tmpProduct = HierarchicalAndSmooth(product,child);
    product = tmpProduct;
    remove_from(childrenremaining,child);
  }
  return product;
}
```

Figure 4.6: Algorithm for Hierarchical Image Computation in Pseudocode.

4.2 Dynamic Conjunction Scheduling

In the previous section we have discussed how to compute an image from a hierarchical transition relation. But, the most important problem has not been solved yet: Finding a schedule for the conjunction of the clusters. In this section we will present a heuristic that solves the conjunction scheduling problem for hierarchical transition relations. The heuristic utilizes the structure that is generated by the hierarchical transition relation to improve the AndExist operation. Also, the heuristic works dynamically, i. e. for any image computation the schedule is computed on-the-fly, resulting in a more efficient image computation.

4.2.1 Dynamic Scheduling Heuristic

The conjunction schedule for the image computation is determined in the *HierarchicalAndSmooth* by *ChooseBestSubmodule* resp. *ChooseBestCluster*, which can be computed statically or dynamically (the simplest solution would be the list order). Ordering heuristics like [Ran95, MHS00] may be applied as well, but they are not useful for dynamic scheduling, because they only take structural information of the transition relation into account and will always result in the same schedule. Nevertheless, adjusting the conjunction schedule to changing state sets, OBDD-sizes, or variable orders might be very profitable.

We describe a strategy that improves the performance of the AndExist operation twofold: The AndExist operation generally profits from a hierarchical partitioning, and we can use the hierarchy structure to improve the conjunction schedule dynamically.

The AndExist operation profits from a *compact* cube of smooth variables. The cube of smooth variables describes the set of variables that are quantified out during the AndExist operation. We call this cube compact, if the variables that appear in the cube are adjacent and do not spread over the variable order of the OBDD. During a step of the AndExist recursion the following three cases are possible:

1. The current variable is contained in the smooth variable set: Then, the recursion continues and the two resulting OBDDs are combined by an OR operation.
2. The current variable is not contained in the smooth variable set: The result is a new node labeled with the current index and the successors of the new node are the results of the two recursions.
3. The cube has reached the sink node: The recursion reduces to an AND operation.

If the smooth variable cube is compact, the third case will appear earlier, improving the efficiency of the operation. If the clusters are separated, i. e. do not share many variables, the third case may reduce to the identity function, because the cube and the cluster reach the sink node simultaneously.

This leads us to the following strategy for *ChooseBestSubmodule* and *ChooseBestCluster*:

1. Compute the maximum level (maxlevel) in the OBDD of a variable to be quantified out of all clusters (or submodules) of a given submodule.
2. Choose the clusters (or submodules) of the current module in *increasing* order of their maxlevels, i. e. choose the *highest* cluster first.

Figure 4.7 gives a sketch of the algorithm for *ChooseBestCluster*. The algorithm for *ChooseBestSubmodule* looks similar to *ChooseBestCluster*, except that the maximal level for a quantify variable needs to be computed recursively for all the submodules.

This strategy gives us a good conjunction schedule, because we expect from the hierarchical partitioning that the clusters of the modules have highly separated variable sets resulting in compact cubes. Also, the schedule is changed dynamically, if the variable order changes during the computation as a result of increasing state sets etc.

Applying the scheduling heuristic to our standard example results in the following order for the conjunction of the clusters (For comparison see Figure 4.3 and Figure 4.4):

0 1 2 3 10 8 9 6 7 4 5 11 12 13 14 21 16 18 17 19 20 15 23 22

The order for the conjunction of the clusters follows the module structure, but for the order of submodules as well as for the clusters itself a heuristic choice is made. These heuristic choices improve the efficiency of the AndExist operation.

```
ChooseBestCluster(module){
    maxlevel = —vars— ; bestcluster = 0;
    foreach(cluster ∈ module→cluster){
        smoothVars = ComputeSmoothVars(cluster);
        mlevel = compute_maximum_level(smoothVars);
        if(mlevel < maxlevel){
            maxlevel = mlevel;
            bestcluster = cluster;
        }
    }
    return bestcluster;
}
```

Figure 4.7: Algorithm for ChooseBestCluster in Pseudocode.

Changes in the variable order, due to dynamic reordering, affect the AndExist operation. The heuristic then changes the conjunction schedule dynamically. Changes in the schedule may occur even during a single image computation step.

Remark:

The obvious strategy for *ChooseBestCluster* would be to take the clusters by *decreasing* maxlevel to reduce the complexity of the OR operation inside the AndExist. Nevertheless, the choice of the increasing order for transition relations with compact cubes has been proven empirically to produce better results than the decreasing choice. In the case of *non-compact* cubes a dynamic scheduling heuristic taking the decreasing choice might be preferable.

The heuristic for ChooseBestCluster and ChooseBestSubmodule is only one heuristic out of many possible strategies, but experiments have shown its effectiveness.

4.2.2 Experimental Results

We implemented this heuristic in VIS, too. We used the same experimental setup as in Section 3.2 for our experiments. Experimental results are shown on Table 4.1 and Table 4.2 (For an description of the column headers refer to Section 3.2).

We compare the results of the RTLMOD-method with the IWLS95-method. The RTLMOD-method wins 32 of the 36 cases and is able to reduce the CPU-time to almost 1/4 of the original CPU-time (74%). The RTLMOD-method is not only able to reduce the CPU time of large circuits up to 97%, it is also the first of our heuristics that is able to improve the ethernet and the packet benchmark. The first one largely by 75% and the second one slightly by 1%. The only losses worth mentioning appear at the TWO benchmarks.

Table 4.2 compares the OBDD-size of the IWLS-method with the RTLMOD-method. The overall peak node size has been reduced by more than half (58%), adding to the increased performance of the RTLMOD-method.

The size for the partitioned transition relation could be reduced by 30%, while almost the same number of clusters (-1%) is used. This clearly shows that the RTLMOD-method computes a compact and efficient partitioning and scheduling for image computation.

	IWLS95		RTLMOD-method	
	lcmp	time/s	time/s	%
ONE.mv.PPCliveness.c	40	7	9	-22
ONE.mv.contention.ct	19	6	8	-25
PCIabnorm.mv.PCI.ctl	304	253	152	39
PCInorm.mv.PCI.ctl.o	206	56	47	16
TWO.mv.PPCliveness.c	74	303	3436	-91
TWO.mv.contention.ct	37	47	150	-68
ethernet.define.213.	303	3321	818	75
gcd.mv.gcd.ctl.out	37	20	11	44
minMax30.mv.minMax30	8	21	12	42
multi_main.mv.multim	45	34	18	47
p62_LS_LS_V01.mv.ccp	64	200	84	58
p62_LS_LS_V01.mv.p6l	99	831	173	79
p62_LS_L_V01.mv.ccp.	64	210	103	50
p62_LS_L_V01.mv.p6li	99	3511	252	92
p62_LS_S_V01.mv.ccp.	64	210	103	50
p62_LS_S_V01.mv.p6li	99	3601	260	92
p62_L_L_V01.mv.ccp.c	52	189	78	58
p62_L_L_V01.mv.p6liv	87	934	159	82
p62_L_S_V01.mv.ccp.c	75	121	93	23
p62_L_S_V01.mv.p6liv	118	231	166	28
p62_ND_LS_V01.mv.ccp	83	830	559	32
p62_ND_LS_V01.mv.p6l	128	21039	4544	78
p62_ND_L_V01.mv.ccp.	75	781	577	26
p62_S_S_V01.mv.ccp.c	43	101	62	38
p62_S_S_V01.mv.p6liv	80	106	67	36
p62_V_LS_V01.mv.ccp.	108	587	362	38
p62_V_LS_V01.mv.p6li	153	>6h	6236	71
p62_V_S_V01.mv.ccp.c	82	245	173	29
p62_V_S_V01.mv.p6liv	127	2168	890	58
packet.mv.packet.ctl	65326	5122	5068	1
single_main.mv.singl	108	13	8	38
single_main.mv.singl	52	13	7	46
three_processor.mv.p	244	>6h	2959	86
three_processor_bin.	222	>6h	522	97
two_processor.mv.pro	264	676	72	89
two_processor_bin.mv	140	150	42	72
Total	69119	110740	28280	
Improvement			74%	

Table 4.1: Comparison of CPU time for IWLS95 and RTLMOD method

	IWLS95			RTLMOD-method					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
`ONE.mv.PPCliveness.c`	19185	3	4456	21558	-11	4	-25	3220	27
`ONE.mv.contention.ct`	17784	3	4456	21558	-17	4	-25	3220	27
`PCIabnorm.mv.PCI.ctl`	176276	14	28613	116345	33	10	28	18061	36
`PCInorm.mv.PCI.ctl.o`	81123	15	35124	69291	14	11	26	16993	51
`TWO.mv.PPCliveness.c`	263756	8	13118	2083768	-87	10	-19	13881	-5
`TWO.mv.contention.ct`	97622	7	11865	215686	-54	10	-30	10508	11
`ethernet.define.213.`	980429	6	14907	516625	47	17	-64	2191	85
`gcd.mv.gcd.ctl.out`	215222	2	7470	215222	0	2	0	6327	15
`minMax30.mv.minMax30`	101042	3	7355	101042	0	2	33	6180	15
`multi_main.mv.multim`	38694	5	14700	33423	13	6	-16	1578	89
`p62_LS_LS_V01.mv.ccp`	166074	23	49952	124430	25	23	0	41246	17
`p62_LS_LS_V01.mv.p6l`	452267	23	49952	158021	65	23	0	41246	17
`p62_LS_L_V01.mv.ccp.`	176540	23	49684	132128	25	22	4	41091	17
`p62_LS_L_V01.mv.p6li`	1617162	23	49684	183674	88	22	4	41091	17
`p62_LS_S_V01.mv.ccp.`	176540	23	49684	132128	25	22	4	41091	17
`p62_LS_S_V01.mv.p6li`	1614473	23	49684	183674	88	22	4	41091	17
`p62_L_L_V01.mv.ccp.c`	164244	23	48961	117269	28	23	0	41165	15
`p62_L_L_V01.mv.p6liv`	477543	23	48961	189172	60	23	0	41165	15
`p62_L_S_V01.mv.ccp.c`	168782	22	62479	123551	26	23	-4	42294	32
`p62_L_S_V01.mv.p6liv`	192410	22	62479	137714	28	23	-4	42294	32
`p62_ND_LS_V01.mv.ccp`	396642	24	63506	299289	24	24	0	46550	26
`p62_ND_LS_V01.mv.p6l`	5583160	24	63506	1747044	68	24	0	46550	26
`p62_ND_L_V01.mv.ccp.`	356794	25	65964	380121	-6	24	4	45076	31
`p62_S_S_V01.mv.ccp.c`	147063	23	62209	97360	33	23	0	39901	35
`p62_S_S_V01.mv.p6liv`	153012	23	62209	97360	36	23	0	39901	35
`p62_V_LS_V01.mv.ccp.`	283494	24	58415	210147	25	23	4	45928	21
`p62_V_LS_V01.mv.p6li`	4483034	24	58415	2126524	52	23	4	45928	21
`p62_V_S_V01.mv.ccp.c`	213245	23	61795	142930	32	23	0	43513	29
`p62_V_S_V01.mv.p6liv`	964988	23	61795	442596	54	23	0	43513	29
`packet.mv.packet.ctl`	53790	3	9704	68473	-21	4	-25	4742	51
`single_main.mv.singl`	14936	2	6352	9360	37	4	-50	884	86
`single_main.mv.singl`	14936	2	6352	9360	37	4	-50	884	86
`three_processor.mv.p`	4621235	9	19750	3062696	33	7	22	4838	75
`three_processor_bin.`	8779857	7	20387	560970	93	7	0	5140	74
`two_processor.mv.pro`	903917	4	12311	88215	90	5	-19	1810	85
`two_processor_bin.mv`	252974	4	11610	64924	74	5	-19	2623	77
Total	34420245	538	1307864	14283648		548		913714	
Improvement				58%		-1%		30%	

Table 4.2: Comparison of OBDD-sizes for IWLS95 and RTLMOD method

4.3 Hierarchical Group Partitioning

In the previous two sections we have developed a heuristic that generates a hierarchical transition relation from a design description given at RT-level. The concept has been proven to significantly improve the image computation process

In this section we will generalize this approach to be independent of an explicit hierarchical description of the design to allow a broader application of the concept.

We start this section with a more detailed analysis of the AndExist algorithm. Then, we present our so-called *modular grouping heuristic* (GROUPMOD). This heuristic is based on the analysis of the latch dependency matrix induced by the design. Benchmark experiments conclude this section.

4.3.1 Discussion of the AndExist Algorithm

The efficiency of the dynamic conjunction scheduling algorithm described in the previous section relies on the *compact clustering* produced by the hierarchical partitioning. Compact clustering means that there are fewer support variables and a higher number of quantify variables in each cluster. As a side effect we can expect that variables of a compact cluster will stay close in the variable order due to their high interaction even during dynamic variable reordering.

To understand the effect of compact clustering on the AndExist operation, let us take a closer look at its algorithm (see Figure 4.8). The AndExist operation performs the quantification of variables ($\exists_{x_i} f = (f_{x_i=1} \vee f_{x_i=0})$) while performing a recursive AND. Thus, it follows the general scheme of any recursive OBDD algorithm like ITE:

- First, terminal cases and the computed table are checked (lines 1, 2).
- If the actual variable (*top*) is not to be quantified, the recursion is called recursively with the successors of the current nodes (10, 11) and a node is created from the results t, e of the recursion.
- If no more variables are to be quantified out, the recursion switches to the regular AND (3, 8).
- If the actual variable is to be quantified out, the recursion continues with the successors of f and g (14, 15) as well, but afterwards, an OR operation on the results t, e of the recursion is executed (16).

Let us recall: The OR operation is called within the recursive step of the AndExist. This actually is the reason why AndExist is not a polynomial operation. To make things even worse the results t and e of the recursive step are obsolete later and are dereferenced (17), i. e. if not part of another OBDD they will be deleted. This means that the AndExist operation produces – in contrast to any other OBDD operation – temporary nodes. These temporary nodes are the main reason for a possible blow-up in OBDD-size during image computation.

We think that the problem of temporary nodes is more profound than the problem of having the OR operation inside the recursion. We try to reach shortcuts (3, 4, 8) in the computation earlier, to keep the lifetime of temporary nodes shorter. For this reason we will even accept a slightly more complex OR operation, due to larger operand sizes.

```
    AndExistRecur(f,g,smoothvars){
1)    Check_terminal_cases;
2)    /*sink-nodes, recomputation, etc.*/
3)    if(!smoothvars) return BddAnd(f, g);
4)    if(g == ONE) return BddExist(f, smoothvars);
5)    top = topmost variable in f and g;
6)    top_q = topvar in smoothvars;
7)    while(top_q < top) top_q = top_q→T;
8)    if(top_q == ONE) return BddAnd(f, g);
9)    if(top_q > top){ /* no quantify */
10)       t = AndExistRecur(f→T, g→T, smoothvars);
11)       e = AndExistRecur(f→E, g→E, smoothvars);
12)       return makeBDDnode(index of top, t,e);
13)     }else{ /* quantify */
14)       t = AndExistRecur(f→T, g→T, smoothvars→T);
15)       e = AndExistRecur(f→E, g→E, smoothvars→T);
16)       result = BddOr(t, e);
17)       Deref(t); Deref(e);
18)       return result;
      }
    }
```

Figure 4.8: Sketch of the AndExist Algorithm.

In a compact clustering as shown (simplified) in Figure 4.9a we have fewer variables in a cluster (T_i) of the transition relation than in a regular clustering as shown in Figure 4.9b. If the clusters are ordered by choosing the cluster with the highest maximum level of a quantify variable first, we can expect earlier shortcuts in the AndExist operation. As the clusters have fewer variables, the AND operation (3, 8) terminates earlier (e. g. AND$(f, 1) = f$) leaving the bottom part of R untouched.

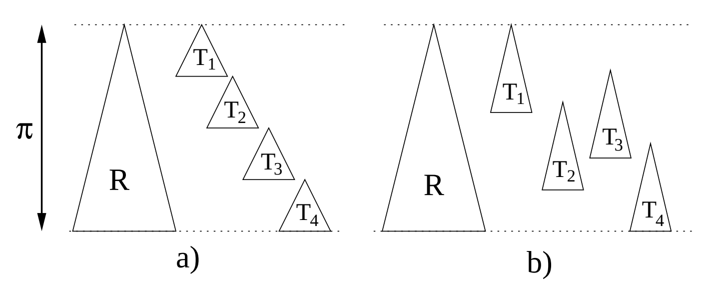


Figure 4.9: Schematic of Compact Clustering (a) vs. Regular Clustering (b).

We can conclude that the dynamic conjunction scheduling does not only try to improve single AndExist operations (e. g. by performing the cheapest operations in terms of OBDD-size first), instead the whole series of AndExist operations will be optimized. This heuristic optimizes the complete image computation process, because the output of the AndExist operation is (except for the last cluster) always an input to the next AndExist operation,

4.3.2 Modular Grouping of the Transition Relation

As we have seen in Section 4.1 a hierarchical partitioning improves the quality of the TR, is more flexible and allows dynamic conjunction scheduling. In the following we will develop a heuristic that groups latches hierarchically, but without using explicit hierarchical information.

The Grouping Algorithm

Our algorithm follows a *separate and group* strategy. Therefore, we utilize the *latch dependency matrix* (LDM) introduced in Section 3.3. An entry of the LDM gives the number of variables that latch l_i and latch l_j have in common, i. e.

$$LDM(l_i, l_j) = |supp(l_i) \cap supp(l_j)|.$$

A higher number denotes a high dependency and thus a strong interaction of the latches. A low number reflects a weaker relation of the latches.

The grouping algorithm proceeds in three phases, where in the first two phases *separated* modules of latches are build, while in the third phase *grouping* inside the modules takes place. The phases in detail:

1. **Module definition phase:** During this phase we create completely independent modules M_i. In this phase the modules contain only a single latch thats serves as a representative for the module. A latch will be a representative for a new module, whenever there is no dependency with the representatives of the other modules: $\forall_j : LDM(l_i, M_j) = 0$. This operation is performed for all latches. The threshold for the dependency is 0 to keep a reasonably small number of modules. The latches are checked consecutively. The latches that do not form an own module are put in the set R of the remaining latches. The modules M_i and the remaining latches R serve as a basis for the 2nd phase.

2. **Module assignment phase:** The remaining latches of R are assigned to the modules using a best-fit strategy, i. e. a latch l_i is put into a module M_k containing the latch l_j, which has the highest common dependency with latch l_i:

   $$l_i \rightarrow M_k : \quad LDM(l_i, l_j) = max, \quad l_j \in M_k.$$

   A minimum dependency is required, otherwise this latch is a representative for a new module. At the end of this phase all latches are assigned to modules. The value of the minimum dependency controls the number of additional modules.

3. **In-module grouping phase:** Now, we have medium to strongly related latches inside the modules. To build groups within the modules we have to apply a more complex heuristic than those before. The heuristic has to be able to group latches that are strongly related, but also has to avoid building one large group containing all latches. Let us keep in mind: All latches in a single module have a certain dependency at least to one other latch in the module.

   We propose a grouping algorithm based on merging of groups that utilizes a *group dependency matrix* (GDM). An entry of the GDM denotes the number of shared variables from the *common support* of group G_i and group G_j. The common support of a group is defined as the intersection of the support of all members of the group:

   $$GDM(G_i, G_j) = |supp(G_i) \cap supp(G_j)|.$$

> The algorithm works as follows for a module M:
>
> 1. Initialize the groups, s. t. each group contains a single latch $G_i = l_i, l_i \in M$ and compute the initial GDM.
> 2. Compute the maximum dependency in the GDM $maxdep = \max_{i,j}(GDM(G_i, G_j))$.
> 3. Pairwise merge groups G_i and G_j, whose dependency equals *maxdep*.
> 4. Update the GDM. The support of a merged group is the intersection of the support of the former groups: $supp(G_i) = supp(G_i) \cap supp(G_j)$.
> 5. Repeat 2-4 until *maxdep* is below a certain threshold or the number of allowed runs is exceeded.
>
> Step 2 of this phase guarantees that strongly related latches are grouped. Performing an intersection of the support of the groups that are merged (Step 4) and limiting the number of runs avoids construction of groups with latches that are related only very loosely.
>
> The latches that could not be grouped after termination of the algorithm are considered being the modules own latches.

The runtime of this algorithm is cubic in the number of latches, but it is negligible in comparison to other operations during construction of the transition relation (OBDD-AND, variable reordering, etc.).

In our implementation we set the minimal required dependency for the 2nd phase to 3 variables. In phase 3 we set the threshold for *maxdep* to 5 and limited the number of runs to 10% of the number of latches. For an overview of the algorithm see Figure 4.10.

Clustering

After the relations have been grouped, the clusters of the partitioned transition relation can be computed. A cluster of the transition relation is only built from latches of one group. If the OBDD-size of a cluster exceeds a certain threshold, an additional cluster for this group is created. The OBDD-size is a rather artificial indicator for the separation of clusters, but it is used in all partitioning approaches to avoid size explosion of the transition relation. Fortunately, the grouping approach drastically limits the influence of this threshold on the efficiency of the image computation by providing more meaningful borders for the clusters.

Figure 4.11 shows a schematic of the result of clustering the hierarchical partitioned transition relation, when applying the modular grouping to our standard example. It represents a typical clustering result. Starting from the root node (*Main*) representing the whole design one reaches the *modules* (M1, M2, etc.) of the design. These modules hardly interact and can be seen as almost independent FSMs. Attached to the main node, we find relations that hardly interact with any of the modules, but do not form an own module. Those relations may be seen as global state variables.

Each module has a small number of children the so-called *groups* (G1, G2, etc.). Relations within a group strongly interact and should stay as close as possible, in the best case within one cluster. Attached to each module, we find those relations that do not interact strong enough with any of the groups to be attached to one of them. Keeping those loosely coupled relations separate in the module gives us more freedom in the choice of the conjunction schedule.

```
ComputeGroups(relations){
  /* — 1. phase — */
  remaining_relations=relations;
  foreach(rel $\in$ relations){
    dependent=0;
    foreach( mod $\in$ modules){
      if( supp(rel) $\subseteq$ supp(mod)) dependent=1;
    }
    if(!dependent){
      remove(rel, remaining_relations);
      append(rel, modules);
    }
  }
  /* — 2. phase — */
  foreach(rel $\in$ remaining_relations){
    best_fit=0; best_mod=0;
    foreach( mod $\in$ modules){
      tmp_supp = $|$supp(rel) $\cup$ supp(mod)$|$ ;
      if( tmp_supp > best_fit){
        best_fit = tmp_supp; best_mod = mod;
      }
    }
    if(best_fit $\geq$ MINIMUM_FIT)
      append(rel, best_mod);
    else
      append(rel, modules);
  }
  merge_single_relation(modules);
  /* — 3. phase — */
  foreach( mod $\in$ modules){
    foreach( rel $\in$ mod) append(rel,groups);
    do{
      GDM = computeGDM(groups);
      maxdep = computeMaxdep(GDM);
      foreach( group $\in$ groups){
        foreach( group2 $\in$ groups){
          if(GDM(group,group2) == maxdep)
            merge(group,group2);
        }
      }
      runs++;
    } while(maxdep > MINDEP && runs < MAXRUN)
    merge_single_relation_groups(groups);
  }
  return modules;
}
```

Figure 4.10: Sketch of the Modular Grouping Algorithm.

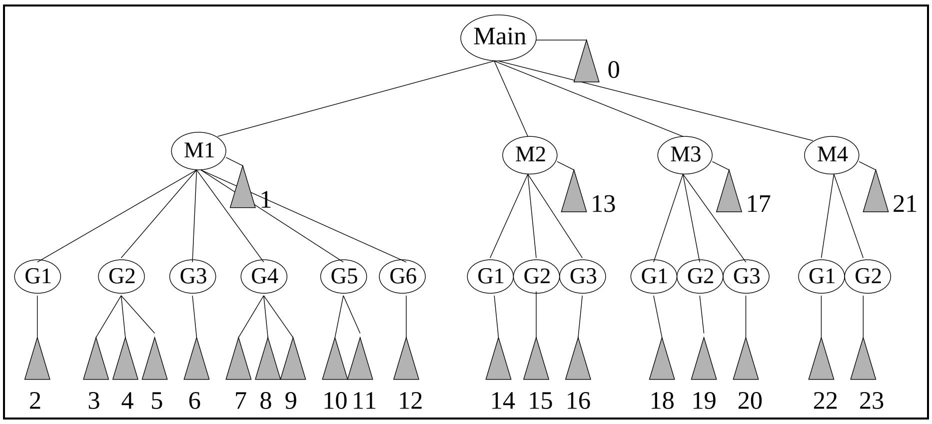


Figure 4.11: Result of the Modular Clustering.

Summarized, the modular grouping approach results in a hierarchical partitioning consisting of a small number of separate modules each consisting of a reasonable number of groups, whose relations strongly interact. The depth of the hierarchy is always limited to three layers, because the grouping algorithm only allows the creation of one main node, one module layer and one group layer. The algorithm is not recursive.

The computation of the image based on this hierarchical partitioning uses the algorithm for *hierarchical image computation* presented in Section 4.1.

Application of the modular grouping heuristic to our standard benchmark example results to the CDM shown in Figure 4.12. The structure of the hierarchical TR that underlies the CDM is shown in Figure 4.11. The numbers denote the cluster ID. The CDM is well structured. We can easily see the structure that has been induced by hierarchical partitioning. There is almost no communication between the modules. The highest dependencies appear within and between groups of a module. The CDM shows many empty entries giving us a higher freedom in the choice for the conjunction schedule.

The modular group heuristic produces a hierarchical TR similar to the heuristic presented in Section 4.1. It seems reasonable to apply the dynamic conjunction scheduling heuristic from Section 4.2, because we expect to have compact clusters for the modular group heuristic as well.

Applying the conjunction scheduling heuristic to our example results in the following starting order for the clusters:

$$0\ 21\ 22\ 23\ 17\ 18\ 19\ 20\ 13\ 14\ 15\ 16\ 1\ 2\ 12\ 11\ 10\ 5\ 3\ 4\ 8\ 7\ 9\ 6.$$

We have the same situation as in Section 4.2: The conjunction schedule follows the modular structure, but a heuristic choice is made on the order of submodules and clusters. Also, the schedule is subject to changes due to variations in the variable order caused by dynamic variable reordering.

4.3.3 Experimental Results

We implemented this heuristic in VIS, as well. We used the same experimental setup as in Section 3.2 for our experiments. Experimental results are shown on Table 4.3 and Table 4.4 (For an description of the column headers refer to Section 3.2).

gh	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0		1		2	2	2	5	4	5	8	6	2		2	2		2	2	2	1	2			
1			1	4	4	4	1	4	4	4	5	4	4									4	2	2
2				2	4	9					2					2								
3					25	9		15	9	3	21	3	3	3	5	2	3					11	8	8
4						15		9	9	5	13	4	4	1	1	1	1					7	5	5
5								3	6	7	14	5	5									3	1	1
6								2	6	5	2	2								1				
7									25	9	16	11	3					3	5	2	3	11	8	8
8										16	17	8	6					1	1	1	1	7	5	5
9											11	5	3									3	1	1
10												11	9	2	5	2	2	2	5	2	2	3	1	1
11													9									3	1	1
12																						8	6	6
13															28	26	18							
14																26	18							
15																	16							
16																								
17																			28	26	18			
18																				26	18			
19																					16			
20																								
21																							21	18
22																								15
23																								

Figure 4.12: Dependency Matrix of the Group Hierarchy Heuristic.

We compare the results of the GROUPMOD-method with the IWLS95-method. The GROUPMOD-method wins 27 of the 36 cases, resulting in an 42% overall decrease in computation time. Many of the larger benchmarks have been significantly improved (up to 93%). Similar to the Group-Method, the GROUPMOD-method was not able to finish the three_processor and the three_processor_bin benchmark. But, it was able to finish the packet benchmark and improved one of the TWO benchmarks and the ethernet benchmark.

Table 4.4 compares the OBDD-size of the IWLS95-method with the GROUPMOD-method. The GROUPMOD method produces the most compact transition relation representation of all our methods. In comparison to the IWLS95-method the overall size for the transition relation has been reduced by 50%. At the same time the number of clusters of the transition relation has only increased slightly (5%). The GROUPMOD-method is also able to reduce the overall size of peak-nodes significantly by 44%.

The GROUPMOD-method does not perform as powerful as the RTLMOD-method, but respecting the more heuristic behavior of the GROUPMOD-method, it represents a good alternative, if no external information is available. For a direct comparison of the RTLMOD-method with the GROUPMOD-method see Table A.3 and Table A.4.

For a comparison of all the partitioning heuristics presented in the last two chapters see Table A.5 and Table A.6.

	IWLS95		GROUPMOD-method	
	lcmp	time/s	time/s	%
`ONE.mv.PPCliveness.c`	40	7	7	0
`ONE.mv.contention.ct`	19	6	6	0
`PCIabnorm.mv.PCI.ctl`	304	253	237	6
`PCInorm.mv.PCI.ctl.o`	206	56	62	-9
`TWO.mv.PPCliveness.c`	74	303	237	21
`TWO.mv.contention.ct`	37	47	85	-44
`ethernet.define.213.`	303	3321	572	82
`gcd.mv.gcd.ctl.out`	37	20	44	-54
`minMax30.mv.minMax30`	8	21	13	38
`multi_main.mv.multim`	45	34	19	44
`p62_LS_LS_V01.mv.ccp`	64	200	111	44
`p62_LS_LS_V01.mv.p61`	99	831	218	73
`p62_LS_L_V01.mv.ccp.`	64	210	121	42
`p62_LS_L_V01.mv.p61i`	99	3511	244	93
`p62_LS_S_V01.mv.ccp.`	64	210	120	42
`p62_LS_S_V01.mv.p61i`	99	3601	299	91
`p62_L_L_V01.mv.ccp.c`	52	189	102	46
`p62_L_L_V01.mv.p61iv`	87	934	186	80
`p62_L_S_V01.mv.ccp.c`	75	121	69	42
`p62_L_S_V01.mv.p61iv`	118	231	114	50
`p62_ND_LS_V01.mv.ccp`	83	830	527	36
`p62_ND_LS_V01.mv.p61`	128	21039	2082	90
`p62_ND_L_V01.mv.ccp.`	75	781	794	-1
`p62_S_S_V01.mv.ccp.c`	43	101	41	59
`p62_S_S_V01.mv.p61iv`	80	106	67	36
`p62_V_LS_V01.mv.ccp.`	108	587	380	35
`p62_V_LS_V01.mv.p61i`	153	>6h	1772	91
`p62_V_S_V01.mv.ccp.c`	82	245	146	40
`p62_V_S_V01.mv.p61iv`	127	2168	675	68
`packet.mv.packet.ctl`	65326	5122	10540	-51
`single_main.mv.singl`	108	13	7	46
`single_main.mv.singl`	52	13	7	46
`three_processor.mv.p`	244	>6h	>6h	0
`three_processor_bin.`	222	>6h	>6h	0
`two_processor.mv.pro`	264	676	170	74
`two_processor_bin.mv`	140	150	35	76
Total	69119	110740	63311	
Improvement			42%	

Table 4.3: Comparison of CPU time for IWLS95 and GROUPMOD method

	IWLS95			GROUPMOD-method					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
ONE.mv.PPCliveness.c	19185	3	4456	19246	0	6	-50	1809	59
ONE.mv.contention.ct	17784	3	4456	19246	-7	6	-50	1809	59
PCIabnorm.mv.PCI.ctl	176276	14	28613	114041	35	11	21	17787	37
PCInorm.mv.PCI.ctl.o	81123	15	35124	69291	14	10	33	16749	52
TWO.mv.PPCliveness.c	263756	8	13118	272317	-3	11	-27	7393	43
TWO.mv.contention.ct	97622	7	11865	139081	-29	10	-30	7909	33
ethernet.define.213.	980429	6	14907	275055	71	11	-45	2989	79
gcd.mv.gcd.ctl.out	215222	2	7470	215222	0	3	-33	5707	23
minMax30.mv.minMax30	101042	3	7355	101042	0	4	-25	4521	38
multi_main.mv.multim	38694	5	14700	33423	13	10	-50	1842	87
p62_LS_LS_V01.mv.ccp	166074	23	49952	86196	48	23	0	29319	41
p62_LS_LS_V01.mv.p6l	452267	23	49952	141477	68	23	0	29319	41
p62_LS_L_V01.mv.ccp.	176540	23	49684	93166	47	23	0	27739	44
p62_LS_L_V01.mv.p6li	1617162	23	49684	192775	88	23	0	27739	44
p62_LS_S_V01.mv.ccp.	176540	23	49684	93166	47	23	0	27739	44
p62_LS_S_V01.mv.p6li	1614473	23	49684	192775	88	23	0	27739	44
p62_L_L_V01.mv.ccp.c	164244	23	48961	87708	46	23	0	28427	41
p62_L_L_V01.mv.p6liv	477543	23	48961	121978	74	23	0	28427	41
p62_L_S_V01.mv.ccp.c	168782	22	62479	72087	57	23	-4	25419	59
p62_L_S_V01.mv.p6liv	192410	22	62479	94688	50	23	-4	25419	59
p62_ND_LS_V01.mv.ccp	396642	24	63506	342875	13	24	0	26903	57
p62_ND_LS_V01.mv.p6l	5583160	24	63506	815232	85	24	0	26903	57
p62_ND_L_V01.mv.ccp.	356794	25	65964	483549	-26	24	4	33171	49
p62_S_S_V01.mv.ccp.c	147063	23	62209	64410	56	23	0	27880	55
p62_S_S_V01.mv.p6liv	153012	23	62209	84190	44	23	0	27880	55
p62_V_LS_V01.mv.ccp.	283494	24	58415	212132	25	24	0	31618	45
p62_V_LS_V01.mv.p6li	4483034	24	58415	1019022	77	24	0	31618	45
p62_V_S_V01.mv.ccp.c	213245	23	61795	118153	44	23	0	28363	54
p62_V_S_V01.mv.p6liv	964988	23	61795	269843	72	23	0	28363	54
packet.mv.packet.ctl	53790	3	9704	168874	-68	4	-25	8131	16
single_main.mv.singl	14936	2	6352	9360	37	6	-66	1831	71
single_main.mv.singl	14936	2	6352	9360	37	6	-66	1831	71
three_processor.mv.p	4621235	9	19750	6989715	-33	8	11	9756	50
three_processor_bin.	8779857	7	20387	5903397	32	7	0	6855	66
two_processor.mv.pro	903917	4	12311	179562	80	7	-42	1928	84
two_processor_bin.mv	252974	4	11610	65156	74	6	-33	2307	80
Total	34420245	538	1307864	19168810		568		641139	
Improvement				44%		-5%		50%	

Table 4.4: Comparison of OBDD-sizes for IWLS95 and GROUPMOD method

4.4 Conclusion

In the last two chapters we have developed new approaches for the partitioning of the transition relation of FSMs. These methods changed the paradigm in partitioning from ordering latches to grouping of the latches. We were able to improve the computation time in our experiments from 27% up to 74%, depending whether high-level information is available or not.

In the last chapter we introduced hierarchical image computation and a dynamic conjunction scheduling approach.

All the presented methods reduce the influence of the somewhat arbitrary partition size threshold that usually drastically influences the performance of the image computation. Thus, allowing a more stable image computation that requires less expertise on the user side.

Chapter 5

Reordering

The last two chapters were dealing with concepts for partitioning of the transition relation. Partitioning is a problem on the *application* layer of a formal verification package. Now, we consider a problem on the *representation* layer, i. e. optimization of the OBDD-representation size.

The only way to improve the size of the OBDD representation is to change its variable ordering. If the order of an already built OBDD has to be changed we speak of *dynamic reordering.* The most successful dynamic reordering strategy is the *Sifting* algorithm.

Sifting provides a good average performance and good results for general applications, but it is often problematic for really hard applications for various reasons:

- It is too time consuming.
- The results are insufficient.
- The quality of the results has a wide variation.

In symbolic model checking the most valuable resource is time. We want the reordering to be as fast as possible. Often, the reordering is not very successful in reducing the size of the OBDD as the represented functions become more complex and simply require more nodes for their representation. Then, reordering is a waste of time and we want to keep it as short as possible. On the other hand, there are situations in which we need to reorder the OBDD, e. g. if the functions became easier or a bad starting order has been chosen. In this case the reordering has to find a good order.

In the next sections we will describe two variable reordering strategies that both accelerate the reordering process, but without significant losses in quality. These strategies are called: *Sample Sifting for symbolic model checking* and *Block Restricted Sifting for symbolic model checking.*

The first one uses high-level information about the functions represented in the OBDD and is tailored for symbolic model checking. The second one is only depending on the symbolic model checking application. Therefore, it this heuristic is easier to integrate into an OBDD package, because no information has to be passed from the application layer to the OBDD layer.

5.1 Sample Sifting

In this section we present our first strategy to improve the dynamic variable reordering during symbolic model checking. This method uses *semantic* knowledge about the application and the represented functions, whose OBDDs have to be reordered.

The basis for our improved reordering technique is *Sample Sifting* introduced by Meinel and Slobodovà in [SM98], and independently by Jain, Adams and Fujita [JAF98]. This reordering method turned out to be a good basis for a reordering strategy for symbolic model checking. The advantages of Sample Sifting are:

- Sample Sifting is very fast.
- When depending on a specific application it produces very good results.
- Unsuccessful reordering attempts are finished quickly.
- In case of an unsuccessful sample reordering the whole reordering process can be aborted and the computation can be continued at the point it was stopped.

After describing the basic Sample Sifting strategy, we will explain our Sample Sifting strategy for symbolic model checking.

5.1.1 Basic Concept

Sampling is a common strategy in heuristic algorithm design: For a given problem instance $\mathbf{P}$, choose a smaller subinstance $\mathbf{p}$, solve the problem for $\mathbf{p}$ and apply the solution to $\mathbf{P}$. The benefit is that $\mathbf{p}$ can be solved with less resources than $\mathbf{P}$.

Applied to OBDD reordering we get the following basic sampling scheme:

1. Choose a sample.
2. Copy the sample to a different location.
3. Reorder the sample.
4. Apply the new order to the original OBDD.
5. If the result is insufficient, reorder back to starting order or repeat 1. to 4.

5.1.2 Sample Sifting Strategy for Symbolic Model Checking

The reordering problem for symbolic model checking turns out to be more challenging than for other OBDD applications like combinatorial verification. Therefore, we develop a more sophisticated strategy for Sampling during symbolic model checking. Because only a few reordering attempts occur in model checking, a single reordering attempt has to be conducted very carefully.

The Sample Sifting strategy for symbolic model checking consists of seven steps:

1. Analysis of the OBDD and its functions.
2. Determination of candidates for the sample.

3. Copying of the candidates into the sample.
4. Reordering of the sample.
5. Applying the new order.
6. Evolution of the new order.
7. Possible abort of the reordering.

We will now describe each step of our strategy in detail:

Step 1) Analysis

The OBDDs for the application are all kept in a single shared OBDD. We call this shared OBDD *manager* or *DD*, because other information like the variable order is also stored in it. Reordering always affects the complete manager. In the following we use the term *OBDD* for the OBDD representation of a single function in the manager and *root* for the pointer to the root node of an OBDD. The term *subOBDD* describes a subgraph of an OBDD that itself represents a valid subfunction of the function represented by the OBDD.

To determine the state of the operation we perform a detailed analysis of the OBDD, the current operation and the functions represented in the OBDD. This is done to guarantee a good sample and to estimate the chances for a successful reordering. The following values are computed or evaluated:

nvars: The number of support variables of the OBDDs in the manager.

ddsize: The number of live OBDD-nodes in the manager.

lastroots: The roots that have been involved in key operations during a certain period of the computation.

recentroots: A subset of *lastroots* that has been used in recent operations (determined by a constant e. g. 500)

lastrootssize: Number of shared OBDD-nodes of *lastroots.*

recentrootssize: Number of shared OBDD-nodes of *recentroots.*

temproots: The roots that have been found as nodes with external references (definition of a root), but do not appear in *lastroots.* These roots are temporary results of the current operation. They could become inner nodes of the function that is just computed or become obsolete at a later stage.

temprootssize: Number of shared OBDD-nodes of *temproots.*

currentop: The type and operands of the operation currently performed.

maxdd: The function with the largest OBDD-size in the manager.

maxddsize: Number of OBDD-nodes of *maxdd.*

The classification of the temporary roots (*temproots*) is crucial: Dynamic reordering is always triggered during an operation. Thus, there are always temporary roots that are part of the result of the current operation. These roots cannot be classified as important or not, because they neither belong to *last-* or *recentroots* nor do we know if they will persist over the end of the operation. On the other hand they might be part of a very costly operation, or even be the reason for the reordering. They can even represent a mayor fraction of the OBDD.

To take care of this effect, we compute the fractions of OBDD-nodes of the various *last-* , *recent-* and *temproots* of the total OBDD-size and choose nodes for sample according to this fraction. This supports to represent their importance and influence on the sample accordingly. The fraction for the *lastroots* is computed by:

$$lastrootfraction = \frac{lastrootsize \cdot samplesize}{ddsize^2}$$

The *recentrootfraction* and the *temprootfraction* are computed analogously.

Step 2) Determination of the candidates for the sample

The most fundamental parameter for Sample Sifting is the required OBDD-size for the sample (*samplesize*). This is the fraction of OBDD-nodes that should be copied into the sample. This parameter has the largest influence on the performance of the Sample Sifting. As smaller the sample is as faster it can be reordered, but a sample that is too small does not contain enough information of the original functions and leads to poor results if applied to the source manager. Sample sizes between 30% and 40% of the original OBDD-size seem appropriate for symbolic model checking . Candidates for the samples are first of all roots that have at least a certain number of nodes. We choose the number of support variables of the manager (*nvars*) as a lower limit. Smaller OBDDs cannot contain all variables and thus do not add useful information to the sample. OBDDs from *recentroots*, *lastroots* and *temproots* are chosen for the sample according to their fractions *lastrootfraction*, *recentrootfraction* and *temprootfraction*. When copying complete OBDDs to the sample, a very small number of large OBDDs can already suffice the size requirements for the complete sample, but reordering only a few functions of the manager does usually not lead to sufficient results. To allow a larger number of roots in the sample, only fractions of the candidate OBDDs are copied to the sample. The only exception is the largest OBDD (*maxdd*): This OBDD is copied completely into the sample, if it is involved in the current operation (*currentop*).

Step 3) Copying the candidates into the sample

There are various alternatives for the copying procedure, if only a fraction of a candidate OBDD should be copied into the sample:

Backtrack: This method was presented by Meinel and Slobodovà in [SM98]. It copies a sample by traversing the OBDD in recursive depth first style and copies a node to the sample, after its children have been copied to the sample. The copying is thus done in postorder fashion. When the size limit (*samplesize*) of the sample is exceeded, the copying process is canceled. Nodes, whose parents have not yet been copied become roots of the sample. The benefit of this method is that real subfunctions of the OBDDs are copied into the sample. The disadvantage is that for any real sample of an OBDD (i. e. <100%) at least the topmost variable is missing. Unfortunately, the top variables can be very important for the reordering. The reordering of the sample cannot determine a position for the missing variables. Thus, the missing variables have to be put externally into the computed order, e. g. to their original position. See Figure 5.2 for a sketch of the backtrack algorithm.

Visit: This method also copies nodes in depth first order, but nodes are copied to the sample as soon as they have been visited (preorder style). The result is a sample that is more likely to contain all variables of the OBDD. When the sample size limit is exceeded and the copying stops, it leaves the sample with nodes, whose successors have not been completely copied. To transform the sample into a valid OBDD representation those missing edges are connected to the 1-sink of the sample resulting in an overestimation of the original function. See Figure 5.3 for a sketch of the visit algorithm. The visit algorithm uses a computed table to avoid creation of redundant nodes that result from the overestimation.

Levelcopy: Both of the above methods copy the lower part of the OBDD completely, but only a small fraction of the upper part of the OBDD. To provide a more balanced sample of the OBDD we propose this method: The sample is copied in depth first style, but the sample size restriction is now applied on each level, i. e. only a given fraction of the nodes of each level is allowed to be copied into the sample. When a level reached its maximal node count, it will be skipped for the rest of the copying process. For the copying itself the visit method as well as the backtrack method can be used. When a certain percentage p ($0\% < p < 100\%$) of the nodes of each level is allowed in the sample the final size of the sample will roughly be the same percentage p of the original OBDD-size. The sample could actually be smaller due to elimination of redundant nodes that appear as nodes have to be skipped in the sample. See Figure 5.4 for a sketch of the levelcopy algorithm.

Figure 5.1 gives schematics for the results of the different copying methods. Note that the method levelcopy represents the profile of the OBDD manager best, while backtrack copies real subfunctions, instead of overestimations into the sample.

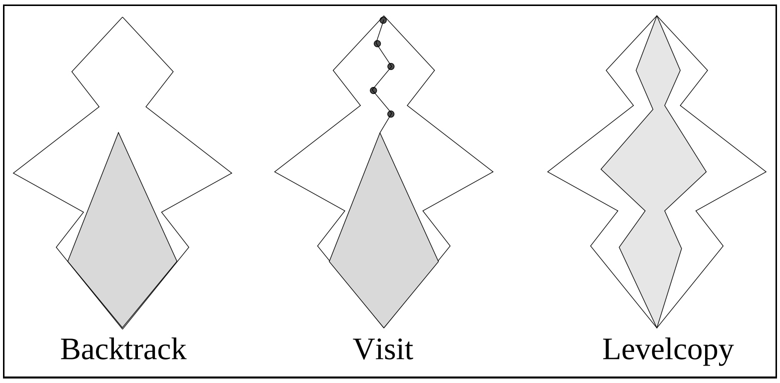


Figure 5.1: Schematic of different methods for copying a sample.

Step 4) Reordering of the sample

We use the standard Sifting to reorder the samples. The main parameter to control the quality and the performance of Sifting is the so called $MAXGROWTH$ factor, i. e. the allowed

```
CopyRecurBacktrack(dd,ndd,f,size,run_size){
    if(!size) return 0; /* size limit exceeded? */
    if(IsTerminal(f))    return(f==dd_one ? ndd_one : ndd_zero);
    result = lookup_table(f); /* already visited? */
    if(result) return result;
    f1=f→then; f0=f→else; /* have not been visited yet */

    ChooseRecursionOrderRandomly{
        result1=Ref(CopyRecurBacktrack(dd,ndd,f1,size,run_size));
        if(!result1) return 0; /* required size reached already */
        result0=Ref(CopyRecurBacktrack(dd,ndd,f0,size,run_size));
        if(!result0) return 0;
    }
    if(result0 == result1) result = result0; /* deletion rule */
    elsif(run_size ≥ size) return 0; /* required size reached already */
        else{
            result=CreateNode(ndd,f→index,result1,result0);
            add_table(f,result);
            run_size = run_size +1;
            Deref(result0); Deref(result1);
        }
    return result;
}
```

Figure 5.2: Algorithm for Sample Copying (Backtrack) in Pseudocode.

```
CopyRecurVisit(dd,ndd,f,size,run_size){
    if(IsTerminal(f))    return( f==dd_one ? ndd_one : ndd_zero);
    if ( run_size ≥ size) return ndd_one; /* size limit exceeded? overestimate */
    result = lookup_table(f); /* already visited? */
    if(result) return result;
    f1=f→then; f0=f→else; /* have not been visited yet */

    ChooseRecursionOrderRandomly{
    result1=CopyRecurVisit(dd,ndd,f1,size,run_size);
    result0=CopyRecurVisit(dd,ndd,f0,size,run_size);
    }
    if(result0 == result1) result = result0; /* deletion rule */
    else{ /* create result*/
        result=CreateNode(ndd,f→index,result1,result0);
        run_size = run_size + 1;
    }
    add_table(f,result);
    return result;
}
```

Figure 5.3: Algorithm for Sample Copying (Visit) in Pseudocode.

```
CopyRecurLevelcopy(dd,ndd,f,size,run_size){ /* visit */
    if(IsTerminal(f)) return( f==dd_one ? ndd_one : ndd_zero);
    result = lookup_table(f); /* already visited? */
    if(result) return result;

    level = Index→Level(dd,f→index); /* size limit exceeded for current level? */
    if(NodesAtLevel(ndd,level) > NodesAtLevel(dd,level) * sample_fraction){
        ChooseRandomly{
            f=f→then; /* skip level */
            f=f→else;
        }
        return CopyRecurVisit(dd,ndd,f,size,run_size);
    }
    f1=f→then; f0=f→else; /* have not been visited yet */
    ChooseRecursionOrderRandomly{
        result1=CopyRecurVisit(dd,ndd,f1,size,run_size);
        result0=CopyRecurVisit(dd,ndd,f0,size,run_size);
    }
    if(result0 == result1) result = result0; /* deletion rule */
    else{ /* create result*/
        result=CreateNode(ndd,f→index,result1,result0);
        run_size = run_size + 1;
    }
    add_table(f,result);
    return result;
}
```

Figure 5.4: Algorithm for Sample Copying (Level) in Pseudocode.

increase of the OBDD-size while sifting a single variable. The default value is 20% and we left it unchanged.

Using a more intensive sample reordering method might lead to very good orders that unfortunately cannot be reached by shuffling as intermediate results are too large. Instead, a less intense sample reordering, using a smaller MAXGROWTH factor might be more useful.

Step 5) Applying the new order

When the reduction of the sample is evaluated to be sufficient (e. g. the sample OBDD reduced to at least 95% of its original size) the newly computed order π' is applied to the original manager. Doing this and keeping the root pointers valid can only be done by a operation called *Shuffling*. Shuffling performs variable swap operations like sifting but towards a given variable order. Unfortunately, it suffers from the same problem as sifting: Temporary graph sizes can blow up, although the final result is small. To prevent us from memory overflow, we limit the increase of the OBDD-size during shuffling in the same way as for sifting, but this means that the manager probably cannot be shuffled to the new order π'. See Figure 5.5a for a schematic of an unreachable order. The example schematic shows the OBDD-size for shuffling from order π to order π'. The order π' cannot be reached, because the intermediate OBDD-size during shuffling exceeds the memory limitations.

During shuffling to the new order all intermediate OBDD-sizes are stored. If the final size is not the minimal size, the manager will be shuffled to the order with the minimal size (This process is not repeated to convergence).

Step 6) Evolution of the new order

Sample Sifting can find an order on a very remote location in the search space, unlike Sifting, which performs a local search. This order might be the best in its surrounding area, but often it is not. For this reason, we perform an additional very fast sifting that searches the local space for a better order π''. The maximal allowed increase of the OBDD-size is drastically reduced (5%) to keep this search very local. See Figure 5.5b for a schematic of order evolution. In this schematic the order π' decreases the OBDD-size in comparison to π but an evolutionary Sifting finds the even better order π''.

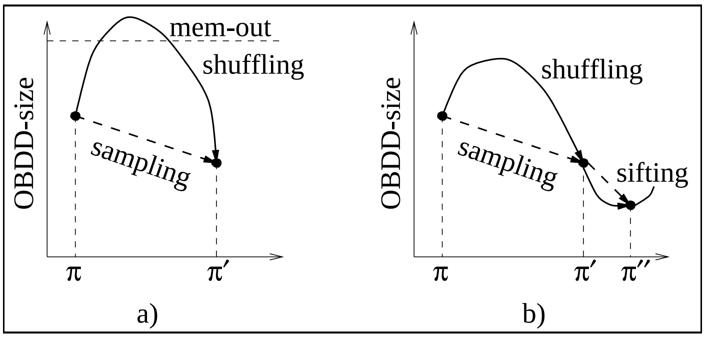


Figure 5.5: Schematic of an Unreachable Order a) and Order Evolution b).

Step 7) Possible abort of the reordering

Up to step 5 the source manager has not been reordered. Before actually reordering the manager, the cache has to be cleared and a garbage collection has to be performed. The result is a lower performance, because results of operations have been in the computed table before and need to be recomputed now. The garbage collection removes dead nodes, i. e. nodes with reference count 0. Allocating a node that was a dead node before the garbage collection is way more time consuming than reclaiming it from dead nodes. After the manager has been reordered, the temporary roots are no longer valid and the current operation (which can be very costly) has to be repeated. Sometimes, performing a time intensive shuffle operation results only in a negligible reduction in OBDD-size. Additionally, the current operation has to be repeated. Instead, it could be more useful to abort the complete reordering process and just increase the threshold for the next reordering. Then, the operation can be continued, where it was suspended.

Variations of the strategy

Step 4a) Evaluation of the sample order: After reordering the sample to the new order π' we can use the sample manager to evaluate the computed variable order π'. This could be done to either check for an abortion of the reordering or to get the real improvement of the OBDD-size. We can do this by copying other samples to the sample manager or even complete OBDDs. The problem for this operation lies in the transfer of OBDDs from one manager to another one with a different variable order. The standard method for transferring functions between different managers basically treats the OBDDs in the source manager as multiplexer circuits and rebuilds them in the new manager. Unfortunately, this method suffers from large (even exponential) intermediate OBDD-sizes. Another method that has been proposed by Meinel and Slobodovà [BMS95] called *Global Rebuilding* avoids this problem, but heavily uses cofactoring, which results in a slowdown of the operation. Also, it produces temporary nodes in the source manager. These nodes could become problematic, if the reordering is canceled.

Step 8) Repeating the sample process: To improve the quality of the result of the sample reordering, several sampling attempts can be performed during a reordering. This can either be done by choosing different functions, different fractions for the samples or by choosing a different method for sampling. Also, there is a choice whether to reorder the manager to the starting order π after the first reordering attempt or to continue with the new order π' and to base the next sampling attempt on this one.

Since, for model checking, samples of a certain size (30%-40%) are required to guarantee good results, we found several reordering attempts too time consuming. Instead, we focus on the quality of the samples to enable good results within one attempt.

Figure 5.6 summarizes the Sample Sifting for symbolic model checking strategy.

```
SampleSifting(mgr){
  data = AnalyzeManager(mgr);
  candidates = DetermineCandidates(mgr, data);
  newmgr = InitManager();
  CopyFunctions(newmgr,candidates,COPYMETHOD);
  Reorder(newmgr);
  if (size reduction of newmgr > MIN_SAMPLE_REDUCTION){
    Shuffle(mgr,order of newmgr);
    if ( sizereduction of mgr > MIN_REDUCTION){
      Reorder(mgr);
    }
  }
  else{
    abort Sampling;
  }
}
```

Figure 5.6: Algorithm for Sample Sifting in Pseudocode.

5.1.3 Experimental Results

In the following we will prove our Sample Sifting concept by benchmark experiments. We will first introduce the benchmarks we used. After giving some implementation details, we will discuss the results of computing the benchmarks using standard sifting. Then, we compare these results with the results using Sample Sifting.

Benchmarks

We use the publicly available SMV-traces of Yang [Yan98, YBO+98] for our experiments. SMV is the description language for the SMV-model checker [McM93]. SMV is comparable to Verilog or VHDL. Traces are recorded calls of OBDD operations done by the SMV model checker during the computation of a certain model. With the use of traces, one is not restricted to use the underlying OBDD-package of SMV. Instead one can use any OBDD package and/or own algorithms. Furthermore, traces have become the reference benchmark set for reordering during model checking. This allows a fair comparison of our heuristic to other approaches.

The SMV-models that underly the traces come from different sources and represent a range from communication protocols to industrial controllers:

- dme2-16: A distributed mutual exclusion protocol.
- dartes: A communication protocol of a complex Ada program.
- dpd75: Dining philosophers with dictionary.
- ftp3: The file transfer protocol.
- furnace17: A remote furnace program.
- key10: This protocol manages keyboard/screen interaction in a window manager.
- mmgt20: A distributed memory manager.
- over12: An automated highway system overtake protocol.
- futurebus: The futurebus cache coherence protocol.
- motors-stuck, valves-gates: Batch reactor system models.
- phone-async: An asynchronous model of a simple telephone system.
- phone-sync-CW: A synchronous model of a similar phone system with *call waiting*.
- tcas: A part of a preliminary version of the system requirements specification of TCAS II (Traffic Alert and Collision Avoidance System) for aircraft.

We did not use those benchmarks that did not trigger any reordering at all (short, abp11), and whose transition relation could not be build with our memory limitations (tomasulo).

Implementation

We implemented our Sample Sifting strategy in the CUDD Package [Som] (version 2.3.0). We applied Sampling during the construction of the transition relation and during model checking. An interface from the routines calling the OBDD operations to the sample routines is necessary to get a list of *recent roots*. For the combinatorial part of the computation, i. e. the construction of the transition relation, this interface is integrated in the main routine for ITE-operation that implements AND, OR, etc. For the model checking part, we have integrated an interface from the trace-driver to sampling routines. Thus, we get those OBDDs in the recent-root-list that are involved in the important model checking operations.

Experiments using Standard Sifting

All experiments were performed on Intel PentiumIII 500MHz Linux workstations with 500MByte datasize and CPU-time limited to 6 hours. For all computations we used the common technique of grouping present- and next-state variables, i. e. a pair of present-/next-state variables is always kept on adjacent levels. This setting accelerates reordering and results in better variable orders. The MAXGROWTH factor has been set to 20% (default value). The first reordering is triggered at 100,000 nodes. These settings are valid for all our experiments using SMV-traces.

	#Variables	CPU-time/s	Reorder-time/s	Reorder-time in %	#Reorderings	avg. Size Reduction	#Successful Reorderings	Peaknodes in 1000
dartes	198	242	214	88%	3	8%	1	583
dme2-16	586	1871	1190	63%	5	18%	3	4868
dpd75	600	2235	1363	61%	5	0%	0	3342
ftp3	100	598	335	56%	4	1%	0	3127
furnace17	184	1928	760	39%	5	21%	1	2339
futurebus	348	2329	666	28%	8	28%	8	2057
key10	140	422	326	77%	6	24%	3	1098
mmgt20	264	754	419	55%	4	2%	0	2904
motors-stuck	172	130	68	52%	4	36%	4	670
over12	174	1743	1499	86%	6	7%	2	4725
phone-async	86	1482	683	46%	5	8%	1	12570
phone-sync-CW	88	17006	14236	83%	10	14%	7	17203
tcas	292	3969	3783	95%	10	28%	10	6074
valves-gates	172	131	98	75%	5	35%	5	542
Total	3404	34840	25640	904	80	230	45	62102
Average	243	2488	1831	65%	5.7	16%	3	4435

Table 5.1: Overview of Computation and Reordering Effort for the Benchmarks using Standard Sifting

We begin our with benchmark experiments using standard Sifting for reordering to demonstrate the complexity of the benchmarks.

Table 5.1 shows the results and some additional statistics. The columns labeled #Variables and Peaknodes give an overview of the complexity of the designs. The complexity ranges from a medium (86) to a large (600) number of variables. A number of 600 variables is about, what is feasible with todays symbolic model checkers. The OBDD-size averages to 4.4 Mio. nodes, and thus is quite high. The average CPU time needed is 2488s and almost 2/3 of it are used for reordering (Reorder Time in %).

We have enlisted some more details for reordering: The column #Reorderings gives the number of reordering attempts for each design. The average number of 5.7 reordering attempts per design is quite low in comparison to e. g. symbolic simulation as used in combinatorial verification. Also, the average reduction per reordering attempt (avg. Size Reduction) and the number of successful reorderings (#Successful Reorderings), i. e. those reordering attempts that reduce the OBDD-size by more than 5%, is given.

We can distinguish three types of benchmarks:

1. Reordering fails: Reordering takes place, but it does not reduce the OBDD size. Here, an acceleration technique can exploit its full potential. Examples: dpd, ftp3.
2. Most reordering attempts fail, very few are successful: This is a more challenging task for an acceleration technique, because it has to deliver a good result at the right time, unsuccessful reordering attempts cannot be corrected. Example: furnace17.
3. Most reordering attempts are successful: In this case an acceleration technique has to prove that it produces overall good results in terms of reordering quality, because only a few reorderings occur. Examples: tcas, futurebus.

Experiments using Sample Sifting

On the basis of the above benchmarks we performed experiments with our Sample Sifting technique. We set the parameters for Sampling to the following values:

- Sample-size: 30%,
- Sample-reduction: 95%,
- DD-reduction: 95%,
- Sample-growth: 20%,
- Shuffle-growth: 20%,
- Copymethod: levelcopy,

Table 5.2 and Table 5.3 show the results of the benchmark experiments using our Sample Sifting strategy. Time is given in CPU-seconds and the size is given in peak-nodes in thousands (K).

The most important result is that Sampling accelerates the reordering significantly. We achieved an overall improvement of 44% of the CPU-time. We were able to improve all of the 14 benchmarks. The improvements ranged up to 56%.

Interestingly, we could achieve even a small decrease in size, i. e. 1% overall. The Sampling improved the OBDD-size in 5 of the 14 cases. The improvement in size is not really a win, but we state this as time improvement *without* losses in OBDD-size.

	Sifting	Sampling	
	time/s	time/s	%
dartes	242	161	33
dme2-16	1871	1299	31
dpd75	2235	2059	8
ftp3	598	367	39
furnace17	1928	1415	27
futurebus	2329	1294	44
key10	422	224	47
mmgt20	754	495	34
motors-stuck	130	104	20
over12	1743	1013	42
phone-async	1482	1121	24
phone-sync-CW	17006	7469	56
tcas	3969	2452	38
valves-gates	131	126	3
Total	34841	19600	
Improvement		44%	

Table 5.2: Comparison of CPU-time in Seconds for Standard Sifting and Sampling

	Sifting	Sampling	
	size/K	size/K	%
dartes	583	614	- 5
dme2-16	4868	4198	14
dpd75	3342	3350	0
ftp3	3127	2963	5
furnace17	2339	3118	- 25
futurebus	2057	2013	2
key10	1098	1274	- 14
mmgt20	2904	3159	- 8
motors-stuck	670	717	- 7
over12	4725	4692	1
phone-async	12570	10525	16
phone-sync-CW	17203	17803	- 3
tcas	6074	6258	- 3
valves-gates	542	907	- 40
Total	62102	61591	
Improvement		1%	

Table 5.3: Comparison of Peak-nodes in Thousands (K) for Standard Sifting and Sampling

5.2 Block Restricted Sifting

In the last section we have shown how to apply Sample Sifting to accelerate symbolic model checking without losses in memory efficiency. But, Sampling has some disadvantages on the implementation side that limit an application:

- Routines on several layers of the BDD-package are involved.
- Interaction with the application layer is required.
- An additional OBDD-manager has to be maintained.
- Complex algorithms require a number of parameters to be set.

In this section we present an alternative strategy to accelerate symbolic model checking that is somewhat *orthogonal* to Sampling. The so called *Block Restricted Sifting* (BRS) restricts the search space of the sifting algorithm, resulting in a significant time improvement, but a size penalty has to be accepted. One benefit of BRS is that it is self-contained, i. e. only one routine within the reordering preprocessing is called. The number of parameters is very limited.

BRS was first presented by Meinel and Slobodovà [MS97], who applied it to combinatorial verification. It turned out that this strategy is in its basic form not well suited for symbolic model checking for the following reasons:

- In combinatorial verification and reachability analysis many reorderings occur. Bad results can be adjusted by other reorderings. Thus, the accelerating power can be used more excessively. In model checking a deviation of the results is more likely, because only a few reordering attempts occur. As a consequence the bad results would lead to a time penalty as larger OBDDs require more time for computation. The time improvement from reordering would be compensated.
- In [MS97] BRS and standard Sifting are interleaved to improve the quality of the results. During model checking too few reorderings occur to obtain significant acceleration when reordering methods are interleaved.

Our goal is to adjust Block Restricted Sifting for symbolic model checking in the way that it almost produces the same results as standard sifting, but in shorter time. To achieve this the BRS has to be application *dependent*. Nevertheless, we do not want the BRS to *interact* too much with the application to keep the implementation simple.

This section is structured as follows: We start with the basic BRS concept. Then, we present our *BRS for symbolic model checking* strategy. The section is concluded with benchmark experiments.

5.2.1 Basic Concept

One way to accelerate variable reordering is to restrict the search space, i. e. reduce the number of variable orders that are tested. A simple approach to restrict the search space as it is used e. g. in the CUDD-package is to allow a maximum growth of the OBDD-size during sifting a single variable (MAXGROWTH). This threshold is typically set to 20%. Another approach is the so called *lower bound sifting* [DG99], which computes the theoretical lower bound of the OBDD-size for the variable that is currently sifted. When this lower bound is no improvement of the already reached best result, the sifting process for this variable is stopped. But, the MAXGROWTH threshold often overrides these bounds.

The *block restricted sifting* approach follows the idea of dividing the search space into blocks, in which useful variable orders could be found. One observation is that the optimal position for a variable often lies relatively close to its original position, thus checking remote positions in the order is often just a waste of time, also it results in large intermediate OBDD-sizes. These large intermediate sizes often foreclose finding good remote positions for a variable.

The challenge is to find well suited blocks to divide the OBDD. An argument from the communication complexity theory is used to find good block borders: A good cut in a circuit (OBDDs are special multiplexer circuits) divides the circuit that only a low flow of information crosses the cut. A cut with a low flow of information in an OBDD is a level with a small number of nodes. To be more precise: It is a level with a small number of nodes in comparison to its neighboring levels. Another fact that supports the blocking concept is that when a variable is sifted only within a block the upper and lower cut lines, and thus the remaining OBDD, remains unchanged. This keeps the reordering effects local.

Basic Algorithm

The basic algorithm for block restricted sifting as it was described by Meinel and Slobodovà [MS97] consists of three major steps:

1. Compute the subfunction profile of the OBDD.
2. Determine block borders from the subfunction profile.
3. Reorder within the computed blocks.

The algorithm in more detail:

Compute subfunction profile: The information passed on a certain level of the OBDD does not only consist of the nodes of the level, but also of edges that cross this level. Nodes and edges together form the subfunction profile. The computation of the subfunction profile works as follows: The number of subfunctions on level i is represented as the ith entry of the array `SubFunc`. If the root node of a function is not on the toplevel of the OBDD, this function has to be added to all levels above the root node level. This has to be done for all root nodes, which have to be computed in advance. During the second phase all edges of all nodes are added in top-down manner to the subfunction profile. Edges that cross several levels have to be added to the respective entries of the array.

Determine block borders: The heuristic for finding block borders is controlled by three values:

- MINBLOCK: The minimal fraction of all variables that has to go in one block.
- MINUP and MINDOWN: The minimal relative increase (resp. decrease) in the subfunction profile to allow a new block border.

The heuristic evaluates the levels of the subfunction profile. Whenever the requirements for MINUP/MINDOWN are met and the current block is large enough (MINBLOCK), a new block border is created. If the next level fits even better, the block border is moved down.

Reorder within blocks: The reordering within the blocks is done by standard Sifting. The implementation in the CUDD-package is quite easy as the package provides handling of *variable groups*. Each block defines a variable group and reordering is allowed only within the groups.

5.2.2 BRS Strategy for Symbolic Model Checking

In the following we will describe the adaptations that are necessary to enable a successful application of the block restricted sifting strategy for symbolic model checking.

Computation of the subfunction profile

In a large OBDD the computation of roots can be a time consuming operation. In the operation *ApproximateSubfunctionProfile* (Figure 5.7) we omit computation of the roots. We only compute an approximation of the subfunction profile, but the introduced error is negligible and has almost no effect on the computation of the borders.

```
ApproximateSubfunctionprofile{
   foreach (varlevel){
      SubFunc[varlevel] = 0;
   }
   foreach (varlevel){
      foreach (node on varlevel ){
         for (both sons of node)
            if (NotMarked(son))
               for ( j = varlevel to varlevel of son)
                  SubFunc[j] = SubFunc[j] + 1;
         SetMark(son);
         }
      }
   }
   return SubFunc;
}
```

Figure 5.7: Algorithm for Approximation of the Subfunction Profile in Pseudocode

Block borders

In the procedure *ComputeBounds* we changed the best-fit strategy to a first-fit strategy, i. e. borders are no longer moved down in the order. This leads to more borders in the OBDD, especially in the lower part, where the subfunction profile naturally decreases and the best-fit strategy would move the border further downwards.

Additionally, we introduced a parameter MAXBLOCK that gives – similar to MINBLOCK – the maximum fraction of the total number of variables that is allowed in a single block. The effect is similar to the one of the first-fit strategy, but it also affects the top part of the OBDD. See Figure 5.8 for a sketch of the algorithm.

Reordering of the blocks

The reordering inside the blocks is done by standard Sifting. We used the standard threshold (MAXGROWTH) for the allowed growth of the OBDD-size during sifting of one variable (i. e. 20%). Using a smaller threshold would accelerate the reordering even more, but at the cost of poor quality of the found order. An alternative would be to use some of the time savings of the BRS to reorder the blocks more intensively, but this seems not appropriate for symbolic model checking.

```
ComputeBounds(s) {
   minblock=MINBLOCK*n;
   bl_start=$\Delta$=0;
   foreach (1 $\leq$ i < n-minblock) {
      bounds[i-1]=0;
      $\Delta$=s[i]-s[i-1];
      if($\Delta \geq$ MINDOWN*s[i-1]) {
         if ($\Delta$ > MINUP*s[i-1]) {
            if (i-bl_start > minblock) {
               bounds[i-1]=i-1; /* first-fit:new border*/
               bl_start=i-1;
            }
         }
      }
      else if(bl_start - i $\geq$ MAXBLOCK*n) {
         bounds[i-1]=i-1; /* new border*/
         bl_start=i-1;
      }
   }
}
```

Figure 5.8: Sketch of ComputeBounds in Pseudocode

Additionally, we introduced the concept of *weak boundaries*, i. e. we include a small number of variables (here: 4) above and below the current block in the reordering of the block. This overlap of the block borders drastically increases the quality of the results without affecting the runtime largely. The positive effect of the weak boundaries is that variables that have been placed on a wrong position in the variable order (e. g. by a static ordering heuristic for starting orders) can now cross the border to a different block. A few variables may move completely from the top to the bottom of the order. Figure 5.9 shows a schematic of an OBDD with weak boundaries.

During the reordering we assure that related present- and next-state variables always stay adjacent and remain in one single block.

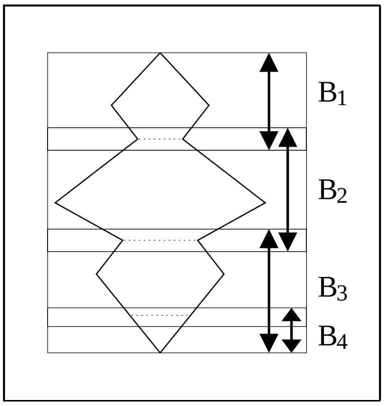


Figure 5.9: Schematic of an OBDD divided into Blocks.

5.2.3 Experimental Results

We used the same benchmarks as in Section 5.1 for our experiments. We compared our BRS strategy with standard Sifting. The parameters have been set to the following values:

- MINBLOCK = 15%,
- MAXBLOCK = 25%,
- MINDOWN = 0.02,
- MINUP = 0.01,
- OVERLAP = 4,
- Group present/next state variables.

The results are shown in Table 5.4 and Table 5.5. Time is given in CPU-seconds and the size is given in peak-nodes in thousands (K).

The BRS method wins in 13 of the 14 cases, resulting in an overall improvement of 36% with a maximum improvement of 61%. The losses in size summarize to marginal 3%. The BRS method was able to improve the size of four benchmarks. Concluding it can be said that BRS enables a significant improvement in computation time by only marginal losses in OBDD-size.

5.3 Conclusion

The most valuable resource in symbolic model checking is time. Unfortunately, it is often wasted by unsuccessful reordering attempts. We have developed two reordering strategies that are balanced in the way that they are very fast, but still compute sufficient results. The Sampling strategy and the BRS strategy improved the computation time in our experiments by 44% resp. 36% with no or only a small trade-off in OBDD-size.

	Sifting	BRS	
	time/s	time/s	%
dartes	242	97	60
dme2-16	1871	842	55
dpd75	2235	1360	39
ftp3	598	446	25
furnace17	1928	1374	29
futurebus	2329	3078	- 24
key10	422	163	61
mmgt20	754	534	29
motors-stuck	130	108	17
over12	1743	940	46
phone-async	1482	1393	6
phone-sync-CW	17006	9680	43
tcas	3969	2218	44
valves-gates	131	130	1
Total	34841	22365	
Improvement		36%	

Table 5.4: Comparison of CPU-time in Seconds for Standard Sifting and BRS

	Sifting	BRS	
	size/K	size/K	%
dartes	583	584	0
dme2-16	4868	4215	13
dpd75	3342	3325	1
ftp3	3127	2878	8
furnace17	2339	3056	- 23
futurebus	2057	2120	- 3
key10	1098	1256	- 13
mmgt20	2904	3026	- 4
motors-stuck	670	709	- 6
over12	4725	4778	- 1
phone-async	12570	10894	13
phone-sync-CW	17203	17475	- 2
tcas	6074	9841	- 38
valves-gates	542	627	- 14
Total	62102	64784	
Improvement		-3%	

Table 5.5: Comparison of Peak-nodes in Thousands (K) for Standard Sifting and BRS

Chapter 6

Conclusion

Key Results

In this work we have presented utilization of high-level methods on different levels of formal verification applications:

- We have demonstrated that a well suited assertion methodology is easy to integrate in the typical verification flow of an industrial tool.
- We have developed new approaches for partitioning of the transition relation of finite state machines. These approaches profit from high-level information, but are general enough to work efficiently without.
- Finally, we have shown how to handle the sometimes problematic use of dynamic variable reordering during symbolic model checking more efficiently. Often, dynamic variable reordering during model checking is useless, but sometimes it is important to guarantee completion of the computation. We developed strategies, that improve the run-time of the reordering without negatively affecting the OBDD-size.

We have shown that the performance of the very general OBDD technology can be drastically improved, if high-level methods are used in the right way, i. e. if information about the design, the structure of the design and the specific application is integrated into the algorithms and heuristics for formal verification. We have als shown that from experiences with high-level methods, often improved general methods can be deducted.

Possible Future Work

This section gives a brief outlook on possible directions for future work in this area.

First of all, we recommend the examination of *hybrid* methods that allow to choose from different partitioning methods. After creating the partitioned transition relation using different methods (e. g. RTLMOD/GROUPMOD or GROUPMOD/MLP [MHS00]) a decision about which approach to use, could be made by analyzing the resulting *cluster dependency matrices* (CDMs). For example the sparseness or the structure of the different CDMs could be compared.

Another topic could be an even tighter integration of dynamic variable reordering in the image computation process. The threshold that triggers the reordering (i. e. *OBDD-size_after_last_reordering*×2) is quite arbitrary. An advanced triggering heuristic for image computation should avoid a lot of unsuccessful reordering attempts. Also, the computation of starting orders for image computation has not yet been studied. Especially for the RTLMOD-method improvements should be possible.

Finally, as an application at the representation layer, we suggest interchanging and interleaving of Sampling and BRS. Reordering of samples using BRS would allow to reorder larger samples in shorter time.

Appendix A

Additional Tables

	RTL		Group	
	lcmp	time/s	time/s	%
`ONE.mv.PPCliveness.c`	40	9	9	0
`ONE.mv.contention.ct`	19	8	8	0
`PCIabnorm.mv.PCI.ctl`	304	167	222	-24
`PCInorm.mv.PCI.ctl.o`	206	39	57	-31
`TWO.mv.PPCliveness.c`	74	9364	2139	77
`TWO.mv.contention.ct`	37	125	106	15
`ethernet.define.213.`	301	14441	6088	57
`gcd.mv.gcd.ctl.out`	37	37	23	37
`minMax30.mv.minMax30`	8	12	7	41
`multi_main.mv.multim`	45	18	24	-25
`p62_LS_LS_V01.mv.ccp`	64	140	129	7
`p62_LS_LS_V01.mv.p6l`	99	444	130	70
`p62_LS_L_V01.mv.ccp.`	64	151	126	16
`p62_LS_L_V01.mv.p6li`	95	335	226	32
`p62_LS_S_V01.mv.ccp.`	64	153	126	17
`p62_LS_S_V01.mv.p6li`	95	318	235	26
`p62_L_L_V01.mv.ccp.c`	52	127	71	44
`p62_L_L_V01.mv.p6liv`	89	483	121	74
`p62_L_S_V01.mv.ccp.c`	75	122	69	43
`p62_L_S_V01.mv.p6liv`	118	278	116	58
`p62_ND_LS_V01.mv.ccp`	83	779	572	26
`p62_ND_LS_V01.mv.p6l`	128	13370	1798	86
`p62_ND_L_V01.mv.ccp.`	75	788	715	9
`p62_S_S_V01.mv.ccp.c`	43	61	66	-7
`p62_S_S_V01.mv.p6liv`	80	112	67	40
`p62_V_LS_V01.mv.ccp.`	108	368	354	3
`p62_V_LS_V01.mv.p6li`	153	16270	413	97
`p62_V_S_V01.mv.ccp.c`	82	169	146	13
`p62_V_S_V01.mv.p6liv`	127	595	279	53
`packet.mv.packet.ctl`	65325	7258	>6h	-66
`single_main.mv.singl`	107	8	10	-19
`single_main.mv.singl`	51	8	10	-19
`three_processor.mv.p`	244	8276	>6h	-61
`three_processor_bin.`	222	5708	>6h	-73
`two_processor.mv.pro`	264	75	84	-10
`two_processor_bin.mv`	141	33	80	-58
Total	69119	80649	79429	
Improvement			1%	

Table A.1: Comparison of CPU-time in Seconds for the RTL-method and the Group-Method

	RTL			Group					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
`ONE.mv.PPCliveness.c`	21229	4	2671	21362	0	5	-19	4971	-46
`ONE.mv.contention.ct`	21229	4	2671	21362	0	5	-19	4971	-46
`PCIabnorm.mv.PCI.ctl`	128310	10	24482	184722	-30	8	19	18676	23
`PCInorm.mv.PCI.ctl.o`	69291	10	20646	64961	6	11	-9	29081	-29
`TWO.mv.PPCliveness.c`	5363742	9	12828	2317020	56	10	-9	11730	8
`TWO.mv.contention.ct`	223656	10	12040	220979	1	10	0	15056	-20
`ethernet.define.213.`	6239111	17	1859	1614232	74	14	17	6471	-71
`gcd.mv.gcd.ctl.out`	215222	3	6466	215222	0	3	0	5366	17
`minMax30.mv.minMax30`	101042	2	6170	101042	0	2	0	3151	48
`multi_main.mv.multim`	33423	4	3092	33423	0	5	-19	4837	-36
`p62_LS_LS_V01.mv.ccp`	136280	20	56040	119996	11	23	-13	44697	20
`p62_LS_LS_V01.mv.p6l`	463209	20	56040	127833	72	23	-13	44697	20
`p62_LS_L_V01.mv.ccp.`	174275	18	53012	118284	32	23	-21	43117	18
`p62_LS_L_V01.mv.p6li`	293189	18	53012	197326	32	23	-21	43117	18
`p62_LS_S_V01.mv.ccp.`	174275	18	53012	118284	32	23	-21	43117	18
`p62_LS_S_V01.mv.p6li`	293189	18	53012	197326	32	23	-21	43117	18
`p62_L_L_V01.mv.ccp.c`	177614	18	54445	121246	31	23	-21	44027	19
`p62_L_L_V01.mv.p6liv`	507110	18	54445	121246	76	23	-21	44027	19
`p62_L_S_V01.mv.ccp.c`	144687	18	55820	107565	25	23	-21	45494	18
`p62_L_S_V01.mv.p6liv`	255775	18	55820	117042	54	23	-21	45494	18
`p62_ND_LS_V01.mv.ccp`	430404	20	54629	291233	32	24	-16	51339	6
`p62_ND_LS_V01.mv.p6l`	4091039	20	54629	821661	79	25	-19	49484	9
`p62_ND_L_V01.mv.ccp.`	436916	21	59711	306682	29	24	-12	52423	12
`p62_S_S_V01.mv.ccp.c`	133192	18	56727	97647	26	23	-21	44041	22
`p62_S_S_V01.mv.p6liv`	163585	18	56727	97647	40	23	-21	44041	22
`p62_V_LS_V01.mv.ccp.`	307382	19	58007	211863	31	24	-20	46795	19
`p62_V_LS_V01.mv.p6li`	5450872	19	58007	242396	95	24	-20	46795	19
`p62_V_S_V01.mv.ccp.c`	157448	19	53163	145859	7	24	-20	47214	11
`p62_V_S_V01.mv.p6liv`	345774	19	53163	204216	40	24	-20	47214	11
`packet.mv.packet.ctl`	62574	4	5191	51592	17	5	-19	10906	-52
`single_main.mv.singl`	9360	3	2227	9668	-3	3	0	2238	0
`single_main.mv.singl`	9360	3	2227	9668	-3	3	0	2238	0
`three_processor.mv.p`	4588114	4	5362	4722353	-2	9	-55	13522	-60
`three_processor_bin.`	3257556	4	5358	5086098	-35	9	-55	13902	-61
`two_processor.mv.pro`	86871	3	2838	75205	13	3	0	5027	-43
`two_processor_bin.mv`	63296	3	2831	131104	-51	3	0	6305	-55
Total	34629601	454	1168380	18645365		553		1028698	
Improvement				46%		-17%		11%	

Table A.2: Comparison of OBDD-sizes for the RTL-method and the Group-Method

	RTLMOD		GROUPMOD	
	lcmp	time/s	time/s	%
`ONE.mv.PPCliveness.c`	40	9	7	22
`ONE.mv.contention.ct`	19	8	6	25
`PCIabnorm.mv.PCI.ctl`	303	152	237	-35
`PCInorm.mv.PCI.ctl.o`	204	47	62	-24
`TWO.mv.PPCliveness.c`	74	3436	237	93
`TWO.mv.contention.ct`	37	150	85	43
`ethernet.define.213.`	300	818	572	30
`gcd.mv.gcd.ctl.out`	37	11	44	-75
`minMax30.mv.minMax30`	8	12	13	-7
`multi_main.mv.multim`	45	18	19	-5
`p62_LS_LS_V01.mv.ccp`	64	84	111	-24
`p62_LS_LS_V01.mv.p6l`	99	173	218	-20
`p62_LS_L_V01.mv.ccp.`	64	103	121	-14
`p62_LS_L_V01.mv.p6li`	99	252	244	3
`p62_LS_S_V01.mv.ccp.`	64	103	120	-14
`p62_LS_S_V01.mv.p6li`	99	260	299	-13
`p62_L_L_V01.mv.ccp.c`	52	78	102	-23
`p62_L_L_V01.mv.p6liv`	89	159	186	-14
`p62_L_S_V01.mv.ccp.c`	75	93	69	25
`p62_L_S_V01.mv.p6liv`	118	166	114	31
`p62_ND_LS_V01.mv.ccp`	83	559	527	5
`p62_ND_LS_V01.mv.p6l`	128	4544	2082	54
`p62_ND_L_V01.mv.ccp.`	75	577	794	-27
`p62_S_S_V01.mv.ccp.c`	42	62	41	33
`p62_S_S_V01.mv.p6liv`	79	67	67	0
`p62_V_LS_V01.mv.ccp.`	108	362	380	-4
`p62_V_LS_V01.mv.p6li`	153	6236	1772	71
`p62_V_S_V01.mv.ccp.c`	82	173	146	15
`p62_V_S_V01.mv.p6liv`	127	890	675	24
`packet.mv.packet.ctl`	65326	5068	10540	-51
`single_main.mv.singl`	106	8	7	12
`single_main.mv.singl`	19	7	7	0
`three_processor.mv.p`	244	2959	>6h	-86
`three_processor_bin.`	140	522	>6h	-97
`two_processor.mv.pro`	264	72	170	-57
`two_processor_bin.mv`	141	42	35	16
Total	69007	28280	63311	
Improvement			-55%	

Table A.3: Comparison of CPU-time in Seconds for the RTLMOD-method and the GROUPMOD-Method

	RTLMOD			GROUPMOD					
	Peakn	Parts	TRn	Peakn	%	Parts	%	TRn	%
ONE.mv.PPCliveness.c	21558	4	3220	19246	10	6	-33	1809	43
ONE.mv.contention.ct	21558	4	3220	19246	10	6	-33	1809	43
PCIabnorm.mv.PCI.ctl	116345	10	18061	114041	1	11	-9	17787	1
PCInorm.mv.PCI.ctl.o	69291	11	16993	69291	0	10	9	16749	1
TWO.mv.PPCliveness.c	2083768	10	13881	272317	86	11	-9	7393	46
TWO.mv.contention.ct	215686	10	10508	139081	35	10	0	7909	24
ethernet.define.213.	516625	17	2191	275055	46	11	35	2989	-26
gcd.mv.gcd.ctl.out	215222	2	6327	215222	0	3	-33	5707	9
minMax30.mv.minMax30	101042	2	6180	101042	0	4	-50	4521	26
multi_main.mv.multim	33423	6	1578	33423	0	10	-40	1842	-14
p62_LS_LS_V01.mv.ccp	124430	23	41246	86196	30	23	0	29319	28
p62_LS_LS_V01.mv.p6l	158021	23	41246	141477	10	23	0	29319	28
p62_LS_L_V01.mv.ccp.	132128	22	41091	93166	29	23	-4	27739	32
p62_LS_L_V01.mv.p6li	183674	22	41091	192775	-4	23	-4	27739	32
p62_LS_S_V01.mv.ccp.	132128	22	41091	93166	29	23	-4	27739	32
p62_LS_S_V01.mv.p6li	183674	22	41091	192775	-4	23	-4	27739	32
p62_L_L_V01.mv.ccp.c	117269	23	41165	87708	25	23	0	28427	30
p62_L_L_V01.mv.p6liv	189172	23	41165	121978	35	23	0	28427	30
p62_L_S_V01.mv.ccp.c	123551	23	42294	72087	41	23	0	25419	39
p62_L_S_V01.mv.p6liv	137714	23	42294	94688	31	23	0	25419	39
p62_ND_LS_V01.mv.ccp	299289	24	46550	342875	-12	24	0	26903	42
p62_ND_LS_V01.mv.p6l	1747044	24	46550	815232	53	24	0	26903	42
p62_ND_L_V01.mv.ccp.	380121	24	45076	483549	-21	24	0	33171	26
p62_S_S_V01.mv.ccp.c	97360	23	39901	64410	33	23	0	27880	30
p62_S_S_V01.mv.p6liv	97360	23	39901	84190	13	23	0	27880	30
p62_V_LS_V01.mv.ccp.	210147	23	45928	212132	0	24	-4	31618	31
p62_V_LS_V01.mv.p6li	2126524	23	45928	1019022	52	24	-4	31618	31
p62_V_S_V01.mv.ccp.c	142930	23	43513	118153	17	23	0	28363	34
p62_V_S_V01.mv.p6liv	442596	23	43513	269843	39	23	0	28363	34
packet.mv.packet.ctl	68473	4	4742	168874	-59	4	0	8131	-41
single_main.mv.singl	9360	4	884	9360	0	6	-33	1831	-51
single_main.mv.singl	9360	4	884	9360	0	6	-33	1831	-51
three_processor.mv.p	3062696	7	4838	6989715	-56	8	-12	9756	-50
three_processor_bin.	560970	7	5140	5903397	-90	7	0	6855	-25
two_processor.mv.pro	88215	5	1810	179562	-50	7	-28	1928	-6
two_processor_bin.mv	64924	5	2623	65156	0	6	-16	2307	12
Total	14283648	548	913714	19168810		568		641139	
Improvement				-25%		-3%		29%	

Table A.4: Comparison of OBDD-sizes for the RTLMOD-method and the GROUPMOD-Method

	IWLS95		RTL		Group		RTLMOD		GROUPMOD	
	lcmp	time/s	time/s	%	time/s	%	time/s	%	time/s	%
`ONE.mv.PPCliveness.c`	40	7	9	-22	9	-22	9	-22	7	0
`ONE.mv.contention.ct`	19	6	8	-25	8	-25	8	-25	6	0
`PCIabnorm.mv.PCI.ctl`	304	253	167	33	222	12	152	39	237	6
`PCInorm.mv.PCI.ctl.o`	206	56	39	30	57	-1	47	16	62	-9
`TWO.mv.PPCliveness.c`	74	303	9364	-96	2139	-85	3436	-91	237	21
`TWO.mv.contention.ct`	37	47	125	-62	106	-55	150	-68	85	-44
`ethernet.define.213.`	303	3321	14441	-77	6088	-45	818	75	572	82
`gcd.mv.gcd.ctl.out`	37	20	37	-45	23	-13	11	44	44	-54
`minMax30.mv.minMax30`	8	21	12	42	7	66	12	42	13	38
`multi_main.mv.multim`	45	34	18	47	24	29	18	47	19	44
`p62_LS_LS_V01.mv.ccp`	64	200	140	30	129	35	84	58	111	44
`p62_LS_LS_V01.mv.p6l`	99	831	444	46	130	84	173	79	218	73
`p62_LS_L_V01.mv.ccp.`	64	210	151	28	126	40	103	50	121	42
`p62_LS_L_V01.mv.p6li`	99	3511	335	90	226	93	252	92	244	93
`p62_LS_S_V01.mv.ccp.`	64	210	153	27	126	40	103	50	120	42
`p62_LS_S_V01.mv.p6li`	99	3601	318	91	235	93	260	92	299	91
`p62_L_L_V01.mv.ccp.c`	52	189	127	32	71	62	78	58	102	46
`p62_L_L_V01.mv.p6liv`	87	934	483	48	121	87	159	82	186	80
`p62_L_S_V01.mv.ccp.c`	75	121	122	0	69	42	93	23	69	42
`p62_L_S_V01.mv.p6liv`	118	231	278	-16	116	49	166	28	114	50
`p62_ND_LS_V01.mv.ccp`	83	830	779	6	572	31	559	32	527	36
`p62_ND_LS_V01.mv.p6l`	128	21039	13370	36	1798	91	4544	78	2082	90
`p62_ND_L_V01.mv.ccp.`	75	781	788	0	715	8	577	26	794	-1
`p62_S_S_V01.mv.ccp.c`	43	101	61	39	66	34	62	38	41	59
`p62_S_S_V01.mv.p6liv`	80	106	112	-5	67	36	67	36	67	36
`p62_V_LS_V01.mv.ccp.`	108	587	368	37	354	39	362	38	380	35
`p62_V_LS_V01.mv.p6li`	153	>6h	16270	24	413	98	6236	71	1772	91
`p62_V_S_V01.mv.ccp.c`	82	245	169	31	146	40	173	29	146	40
`p62_V_S_V01.mv.p6liv`	127	2168	595	72	279	87	890	58	675	68
`packet.mv.packet.ctl`	65326	5122	7258	-29	>6h	-76	5068	1	10540	-51
`single_main.mv.singl`	108	13	8	38	10	23	8	38	7	46
`single_main.mv.singl`	52	13	8	38	10	23	7	46	7	46
`three_processor.mv.p`	244	>6h	8276	61	>6h	0	2959	86	>6h	0
`three_processor_bin.`	122	>6h	5708	73	>6h	0	522	97	>6h	0
`two_processor.mv.pro`	264	676	75	88	84	87	72	89	170	74
`two_processor_bin.mv`	140	150	33	78	80	46	42	72	35	76
Total	69119	110740	80649		79429		28280		63311	
Improvement			27%		28%		74%		42%	

Table A.5: Comparison of CPU-time in Seconds for all Partitioning Methods

	IWLS95	RTL		Group		RTLMOD		GROUPMOD	
	Peakn	Peakn	%	Peakn	%	Peakn	%	Peakn	%
ONE.mv.PPCliveness.c	19185	21229	-9	21362	-10	21558	-11	19246	0
ONE.mv.contention.ct	17784	21229	-16	21362	-16	21558	-17	19246	-7
PCIabnorm.mv.PCI.ctl	176276	128310	27	184722	-4	116345	33	114041	35
PCInorm.mv.PCI.ctl.o	81123	69291	14	64961	19	69291	14	69291	14
TWO.mv.PPCliveness.c	263756	5363742	-95	2317020	-88	2083768	-87	272317	-3
TWO.mv.contention.ct	97622	223656	-56	220979	-55	215686	-54	139081	-29
ethernet.define.213.	980429	6239111	-84	1614232	-39	516625	47	275055	71
gcd.mv.gcd.ctl.out	215222	215222	0	215222	0	215222	0	215222	0
minMax30.mv.minMax30	101042	101042	0	101042	0	101042	0	101042	0
multi_main.mv.multim	38694	33423	13	33423	13	33423	13	33423	13
p62_LS_LS_V01.mv.ccp	166074	136280	17	119996	27	124430	25	86196	48
p62_LS_LS_V01.mv.p6l	452267	463209	-2	127833	71	158021	65	141477	68
p62_LS_L_V01.mv.ccp.	176540	174275	1	118284	32	132128	25	93166	47
p62_LS_L_V01.mv.p6li	1617162	293189	81	197326	87	183674	88	192775	88
p62_LS_S_V01.mv.ccp.	176540	174275	1	118284	32	132128	25	93166	47
p62_LS_S_V01.mv.p6li	1614473	293189	81	197326	87	183674	88	192775	88
p62_L_L_V01.mv.ccp.c	164244	177614	-7	121246	26	117269	28	87708	46
p62_L_L_V01.mv.p6liv	477543	507110	-5	121246	74	189172	60	121978	74
p62_L_S_V01.mv.ccp.c	168782	144687	14	107565	36	123551	26	72087	57
p62_L_S_V01.mv.p6liv	192410	255775	-24	117042	39	137714	28	94688	50
p62_ND_LS_V01.mv.ccp	396642	430404	-7	291233	26	299289	24	342875	13
p62_ND_LS_V01.mv.p6l	5583160	4091039	26	821661	85	1747044	68	815232	85
p62_ND_L_V01.mv.ccp.	356794	436916	-18	306682	14	380121	-6	483549	-26
p62_S_S_V01.mv.ccp.c	147063	133192	9	97647	33	97360	33	64410	56
p62_S_S_V01.mv.p6liv	153012	163585	-6	97647	36	97360	36	84190	44
p62_V_LS_V01.mv.ccp.	283494	307382	-7	211863	25	210147	25	212132	25
p62_V_LS_V01.mv.p6li	4483034	5450872	-17	242396	94	2126524	52	1019022	77
p62_V_S_V01.mv.ccp.c	213245	157448	26	145859	31	142930	32	118153	44
p62_V_S_V01.mv.p6liv	964988	345774	64	204216	78	442596	54	269843	72
packet.mv.packet.ctl	53790	62574	-14	51592	4	68473	-21	168874	-68
single_main.mv.singl	14936	9360	37	9668	35	9360	37	9360	37
single_main.mv.singl	14936	9360	37	9668	35	9360	37	9360	37
three_processor.mv.p	4621235	4588114	0	4722353	-2	3062696	33	6989715	-33
three_processor_bin.	8779857	3257556	62	5086098	42	560970	93	5903397	32
two_processor.mv.pro	903917	86871	90	75205	91	88215	90	179562	80
two_processor_bin.mv	252974	63296	74	131104	48	64924	74	65156	74
Total	34420245	34629601		18645365		14283648		19168810	
Improvement		0%		45%		58%		44%	

Table A.6: Comparison of Peak-nodes for all Partitioning Methods

	Sampling	BRS	
	time/s	time/s	%
dartes	161	96	+ 39
dme2-16	1299	841	+ 35
dpd75	2059	1360	+ 33
ftp3	367	446	- 17
furnace17	1415	1373	+ 2
futurebus	1293	3078	- 57
key10	224	163	+ 27
mmgt20	494	534	- 7
motors-stuck	104	108	- 3
over12	1013	940	+ 7
phone-async	1121	1393	- 19
phone-sync-CW	7469	9680	- 22
tcas	2451	2217	+ 9
valves-gates	126	130	- 2
Total	19600	22364	
Improvement		-14%	

Table A.7: Comparison of CPU-time in Seconds for Sampling and BRS

	Sampling	BRS	
	size/K	size/K	%
dartes	614	584	+ 4
dme2-16	4198	4215	- 0
dpd75	3350	3325	+ 0
ftp3	2963	2878	+ 2
furnace17	3118	3056	+ 1
futurebus	2013	2120	- 5
key10	1274	1256	+ 1
mmgt20	3159	3026	+ 4
motors-stuck	717	709	+ 1
over12	4692	4778	- 1
phone-async	10525	10894	- 3
phone-sync-CW	17803	17475	+ 1
tcas	6258	9841	- 36
valves-gates	907	627	+ 30
Total	61591	64784	
Improvement		-5%	

Table A.8: Comparison of Peak-nodes in Thousands (K) for Sampling and BRS

Bibliography

[Azi97] A. Aziz et. al., *Texas-97 benchmarks*, http://www-cad.EECS.Berkeley.EDU/Respep/Research/Vis/texas-97.

[BFG93+] R. I. Bahar and E. A. Frohm and C. M. Gaona and G. D. Hachtel and E. Macii and A. Pardo and F. Somenzi, *Algebraic decision diagrams and their applications*, In Proc. IEEE International Conference on Computer-Aided Design,1993

[BW96] B. Bollig and I. Wegener, *Improving the Variable Ordering of OBDDs is NP–complete*, IEEE Transaction on Computers, 45(9), 1996.

[BLW95] B. Bollig and M. Löbbing and I. Wegener, *Simulated Annealing to Improve Variable Orderings for OBDDs*, Int. Workshop on Logic Synthesis, 1995

[VIS] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy and T. Villa, *VIS: A System for Verification and Synthesis*, Proc. of Computer Aided Verification (CAV'96), 1996.

[Bry86] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 1986.

[Bry91] R.E. Bryant, *On the Complexity of VLSI implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication*, IEEE Transaction on Computers, 40, 1991.

[Bry92] R. E. Bryant, *Symbolic Boolean manipulation with ordered binary decision diagrams*, ACM Computing Surveys, 24(3), 1992.

[BCL91] J. R. Burch, E. M. Clarke and D. E. Long, *Symbolic Model Checking with partitioned transition relations*, Proc. of Int. Conf. on VLSI, 1991.

[BMS95] J. Bern, Ch. Meinel and A. Slobodová, *Global Rebuilding of OBDDs - Tunneling Memory Requirement Maxima* Proc. of Computer Aided Verification (CAV'95), LNCS 939, Springer-Verlag, 1995.

[Bur90] J. R. Burch, E. M. Clarke, D. L. Dill, L. J. Hwang and K. L. McMillan, *Symbolic model checking:* 10^{20} *states and beyond*, Proc. of Logic in Computer Science (LICS'90), 1990.

[CAL] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton and A. Sangiovanni-Vincentelli, *High Performance BDD Package Based on Exploiting Memory Hierarchy* In Proc. of ACM/IEEE Design Automation Conference, 1996.

[CB96] A. Crews and F. Brewer, *Controller Optimization for Protocol Intensive Applications*, in Proc. of the European Design Automation Conference 1996, Geneva, Switzerland, 1996.

[CBM89] O. Coudert, C. Berthet and J. C. Madre, *Verification of Synchronous Machines using Symbolic Execution*, Proc. of Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, Springer, 1989.

[CCJ+01] P. Chauhan, E. M. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith and D. Wang, *Non-linear Quantification Scheduling in Image Computation*, Proc. of International Conference on CAD (ICCAD'01), 2001.

[CCQ02] G. Cabodi, P. Camurati and S. Quer, *Dynamic Scheduling and Clustering in Symbolic Image Computation* Proc. of Design and Test in Europe (DATE,02), 2002.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla, *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, 1986.

[CM90] O. Coudert and J. C. Madre, *A unified framework for the formal verification of sequential circuits*, Proc. of IEEE Int. Conf. on Computer-Aided Design, 1990.

[CM95] O. Coudert and J. C. Madre, *The implicit set paradigm: A new approach to finite state system verification*, Formal Methods in System Design, 6(2), 1995.

[DBG95] R. Drechsler and B. Becker and N. Göckel *A genetic algorithm for variable ordering of OBDDs*, Int. Workshop on Logic Synthesis, 1995.

[DDG98] R. Drechsler, N. Drechsler and W. Günther, *Fast Exact Minimization of BDDs*, Proc. of Design Automation Conference, 1998.

[DG99] R. Drechsler and W. Günther, *Using lower bounds during dynamic BDD minimization*, Proc. of Design Automation Conference (DAC'99), 1999.

[FK98] L. Fix and G. Kamhi, *Adaptive Variable Reordering for Symbolic Model Checking*, Proc. IEEE International Conference on Computer-Aided Design, 1998.

[FS90] S.J. Friedman and K.J. Supowit, *Finding the Optimal Variable Ordering for Binary Decision Diagrams*, IEEE Transactions on Computers, 39(5), 1990.

[FOH93] H Fujii, G. Ootomo and C. Hori, *Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams*, Proc. of Int. Conf on Computer Aided Design, 1993.

[GJ78] M. R. Garey, and M. Johnson *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1978.

[GB94] D. Geist and I. Beer, *Efficient Model Checking by Automated Ordering of Transition Relation Partitions*, Proc. of Computer Aided Verification CAV'94, 1994.

[HB98] U. Holtmann, P. Blinzer, *Design of a SPDIF Receiver using Protocol Compiler*, in Proc. Design Automation Conference 1998, San Francisco, 1998.

[HKB96] R. Hojati, S. C. Krishnan, R. K. Brayton, *Early qantification and partitioned transition relations*, Proc. of. IEEE Int. Conf. on Computer-Aided Design, 1996.

[HJ96] R. D. M. Hunter and T. T. Johnson, *Introduction to VHDL*, Chapman & Hall, 1996.

[JAF98] J. Jain, W. Adams and M. Fujita, *Sampling schemes for computing OBDD variable orderings*, Proc. IEEE International Conference on Computer-Aided Design, 1998.

[Kuk96] Y. Kukimoto, *BLIF-MV*, Tech. Report , U.C. Berkeley, EECS Dept., May 31, 1996.

[Lon92] D. E. Long, *Efficient implementation of an OBDD package*, Carnegie-Mellon-University1992.

[MWB88] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment*, Proc. of the 25th Design Automation Conference, 1988.

[McM93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[MHS00] I. Moon, G. D. Hachtel and F. Somenzi, *Border-Block Triangular Form and Conjunction Schedule in Image Computation*, Proc. of Formal Methods in CAD (FMCAD'00), LNCS 1954, 2000.

[MJH+98] I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan and C. Pixley, *Approximate Reachability Don't cares for CTL Model Checking*, Proc. of Int. Conf. on CAD (ICCAD'98), 1998.

[MKR+00] I.Moon, J. Kukula, K. Ravi and F. Somenzi, *To Split or to Conjoin: The Question in Image Computation*, Proc. of Design Automation Conference (DAC'00), 2000.

[MSS99] Ch. Meinel, K. Schwettmann, and A. Slobodová, *Application Driven Variable Reordering and an Example Implementation in Reachability Analysis*, Proc. of ASP-DAC'99, Hongkong, 1999.

[MS97] Ch. Meinel and A. Slobodová, *Speeding up Variable Reordering of OBDDs*, in Proc. of the International Conference on Computer Design, 1997.

[MST97] Ch. Meinel, F. Somenzi, and T. Theobald *Linear sifting of decision diagrams.* Proc. 34th ACM/IEEE Design Automation Conference, 1997.

[MST97a] W. Meyer, A. Seawright, F. Tada, *Design and Synthesis of Array Structured Telecommunication Processing Applications*, in Proc. of the Design Automation Conference, 1997.

[MS00] Ch. Meinel and C. Stangier, *Speeding Up Image Computation by using RTL Information*, Proc. of Formal Methods in CAD (FMCAD'00), LNCS 1954, 2000.

[MS00a] Ch. Meinel and C. Stangier, *Speeding Up Symbolic Model Checking by Accelerating Dynamic Variable Reordering*, Proc. of IEEE 10th Great Lakes Symopsium on VLSI, 2000.

[MS01] Ch. Meinel and C. Stangier, *A New Partitioning Scheme for Improvement of Image Computation*, Proc. of ASP Design Automation Conference (ASP-DAC'01), 2001.

[MS01a] Ch. Meinel and C. Stangier, *Hierarchical Image Computation with Dynamic Conjunction Scheduling*, Proc. of IEEE Int. Conf. on Computer Design, 2001.

[MS01b] Ch. Meinel and C. Stangier, *Data Structures for Boolean Functions. BDDs - Foundations and Applications*, In: Computational Discrete Mathematics, Ed. H. Alt, LNCS 2122, Springer, 2001.

[MS02] Ch. Meinel and C. Stangier, *Modular Partitioning and Dynamic Conjunction Scheduling in Image Computation*, In Proc, ACM/IEEE Int. Workshop on Logic and Synthesis, 2002.

[MT98] Ch. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*, Springer-Verlag, 1998.

[Min93] S. Minato, *Zero-suppressed BDDs for set manipulation in combinatorial problems*, In Proc. 30th ACM/IEEE Design Automation Conference,1993.

[MIY90] S. Minato, N. Ishiura, and S. Yahima, *Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation*, Proc. of the 27th ACM/IEEE Design Automation Conference, 1990.

[Moo65] G. E. Moore, *Cromming more Components onto Integrated Circuits*, Electronic Magazine, vol. 38, no. 8, 1965.

[Ran95] R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley and B. Plessier, *Efficient BDD Algorithms for Synthesizing and Verifying Finite State Machines*, Proc. of Int. Workshop on Logic Synthesis (IWLS'95), 1995.

[Rud93] R. Rudell, *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*, Proc. of ACM/IEEE Int. Conf. on Computer-Aided Design , 1993.

[SW97] P. Savický and I. Wegener, *Efficient Algorithms for the Transformation between Different Types of Binary Decision Diagrams*, Acta Informatica, 34, 1997.

[SB94] A. Seawright and F. Brewer, *Clairvoyant: A Synthesis System For Production-Based Specification*, in IEEE Trans. on VLSI Systems, 1994.

[Sea96] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, *A System for Compiling and Debugging Structured Data Processing Controllers*, in Proc. of the European Design Automation Conference ,1996.

[SIS] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton and A. Sangiovanni-Vincentelli, *Sequential Circuit Design using Synthesis and Optimization*, in Proc. of Int. Confererence on Circuit Design, 1992.

[Sie98] D. Sieling. *On the Existence of Polynomial Time Approximation Schemes for OBDD Minimization*, Proc. of STACS'98, LNCS 1373, Springer Verlag, 1998.

[SW93] D. Sieling and I. Wegener, *Reduction of BDDs in Linear Time*, Information Processing Letters, 48(3), 1993.

[SM98] A. Slobodová and Ch. Meinel, *Sample Method for Minimization of OBDDs*, Proc. of Int. Workshop on Logic Synthesis, (Tahoe City,CA), June 1998.

[Som] F. Somenzi, *CUDD: CU Decision Diagram Package*,
ftp://vlsi.colorado.edu/pub/ .

[SH01] C. Stangier and U. Holtmann, *Applying Formal Verification with Protocol Compiler*, in Proc. of EUROMICRO Digital System Design (DSD'01), Poland, 2001

[THY93] S. Tani, K. Hamaguchi, and S. Yajima, *The Complexity of the Optimal Variable Ordering Problem of Shared Binary Decision Diagrams*, Proc. of ISAAC, LNCS 762, Springer, 1993.

[The98] T. Theobald, *Transformation Techniques for Decision Diagrams in Computer-Aided Design* Shaker, 1998.

[TM91] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991.

[Weg00] I. Wegener, *Branching programs and binary decision diagrams - theory and applications*, SIAM Monographs on Discrete Mathematics and Applications, 2000.

[Woe01] P Woelfel, *New Bounds on the OBDD-Size of Integer Multiplication via Universal Hashing*, Proc. of STACS 2001, LNCS 2010, 2001.

[Wol86] P. Wolper, *Expressing interesting properties of programs in propositional temporal logic*, in Proc. 13th ACM Symp. on Principles of Programming, 1986.

[Yan98] B. Yang, *SMV Models*, http://www.cs.cmu.edu/~bwolen/software/, 1998.

[YBO+98] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi, *A Performance Study of BDD-based Model Checking*, In Proc. Formal Methods in CAD (FMCAD'98), 1998.

Christian Stangier

Fujitsu Labs of America
595 Lawrence Expressway
Sunnyvale, CA 94086
USA

Geboren am 2. August 1969 in Solingen.

Akademischer Werdegang

Schule			
8/75	–	6/79	Grundschule: Maischützenschule, Bochum
8/79	–	6/88	Gymnasium: Gymnasium am Ostring, Bochum
		6/88	Abitur, Leistungskurse: Mathematik, Physik
Universität			
10/89	–	6/96	Universität Dortmund: Hauptfach: Informatik, Nebenfach: Physik
		6/96	Diplom Diplomarbeit: Speicherverwaltung bei der BDD-Synthese
9/96	–	12/99	Graduiertenkolleg "Mathematische Optimierung" an der Universität Trier
1/00	–	4/02	Promotionsstudent am Lehrstuhl für Theoretische Informatik, Universität Trier
	seit	10/02	Member of Research Staff, Fujitsu Labs of America, Sunnyvale, CA, USA