Preprint from Proceedings of GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Dresden, Germany, February 2006

Advanced Unbounded CTL Model Checking By Using AIGs, BDD Sweeping, And Quantifier Scheduling

Florian Pigorsch Christoph Scholl Stefan Disch

{pigorsch, scholl, disch}@informatik.uni-freiburg.de
Albert-Ludwigs-Universität Freiburg, Institut für Angewandte Wissenschaften,
D-79110 Freiburg im Breisgau, Germany

In this paper we present a complete method for verifying properties expressed in the temporal logic CTL. In contrast to the majority of verification methods presented in recent years, we support *unbounded* model checking based on symbolic representations of characteristic functions. Among others, our method is based on an advanced And-Inverter Graph (AIG) implementation, quantifier scheduling, and BDD sweeping. For several examples, our method outperforms BDD based symbolic model checking by orders of magnitude. However, our approach is also able to produce competitive results for cases where BDD are known to perform well.

1 Introduction

Given a sequential circuit and properties in some temporal logic like CTL or LTL, model checking is a method for verifying these properties [1, 2]. In the early nineties, by introducing *symbolic* model checking, Burch et al. substantially extended the class of systems which can be verified [3, 4]. In symbolic model checking binary decision diagrams (BDDs) [5] are used both for state set representation and for state traversal. Sets of states are represented by characteristic functions which in turn are represented by BDDs.

However, in the last few years SAT based techniques like Bounded Model Checking (BMC) [6, 7] have been attracting much interest, since industrial needs ask for methods avoiding the well known memory explosion problem which may occur during symbolic model checking of large circuits. BMC applied to certain properties (invariants or, more generally, LTL formulas) 'unfolds' the transition relation for k steps in order to find counterexamples. If no counterexample of length k is found, then k is increased and BMC is used again. The search for counterexamples can be stopped, if k is equal to the *diameter* of the system, i.e., the maximum length of all shortest paths between states in the system. In that case, BMC ends up with a proof of the property. Unfortunately, computing diameters of large systems turns out to be hard. The problem may be reduced to the validity check of a quantified Boolean formula (QBF) with alternating existential and universal quantifiers [6]. Since this check is usually hard for large systems, BMC is mostly used as an incomplete method for finding errors in practice.[1]

In this paper, we present a *complete* method for verifying properties expressed in the temporal logic CTL. Our method is based on a symbolic representation of sets of states. However, our symbolic representation relies on And-Inverter Graphs (AIGs) [9, 10] instead of BDDs. So far, And-Inverter Graphs have been successfully applied in combinational equivalence checking. Basically, they are Boolean circuits which consist of AND gates and

[1] Another possibility consists in increasing k up to the length of the longest simple path between two states [8]. Whereas it is easier to determine the length of the longest simple path than to determine the diameter of the system, the longest simple path may be exponentially longer than the diameter. If this is the case, unfolding the transition relation for such a large number of steps will be prohibitive.

inverters only. In contrast to BDDs, AIGs do not provide canonical representations of Boolean functions. Since we do not need canonical representations for representing sets of states, we are able to avoid memory blow-ups during the construction of (canonical) BDDs. On the other hand, checks for satisfiability or validity, which are needed during the model checking process, do not come for free as for BDDs, because there are different AIG representations for constants 0 and 1.

In order to obtain as much subexpression sharing as possible we make use of a special version of AIGs, the so-called functionally reduced AIGs (FRAIGs) which were introduced by Mishchenko et al. [11] in the context of logic synthesis, technology mapping and combinational equivalence checking. Like general AIGs, FRAIGs still form non-canonical representations of Boolean functions, but they have the additional property that they do not contain any pair of functionally equivalent nodes. This invariant is maintained during construction of FRAIGs by using a SAT solver. In addition, the construction of FRAIGs is assisted by functional simulation in order to avoid unnecessary SAT checks for pairs of nodes for which already simulation is able to prove non-equivalence.

The most difficult step during model checking using FRAIGs is the elimination of existential quantifiers. As in [12, 13, 14] existential quantifiers $\exists x f$ are eliminated by replacing them by $f|_{x=0} + f|_{x=1}$. Of course, in the worse case the elimination of one quantifier may double the size of the representation. Although it is not very likely that this worst case behavior can be avoided in random examples (since SAT checking is NP hard), we show in our experimental results that we succeed in limiting the increase in size by a clever choice of the order of quantifications ('quantifier scheduling') combined with other measures to limit the size of the representation. Our novel method for quantifier scheduling is based on estimations on the AIG sizes of the results after performing In Section 5 we motivate the importance of quantifier scheduling by giving an example and we describe the approach in more detail.

Other techniques for limiting the sizes of our representations of state sets are node selection heuristics and BDD sweeping. Whenever a new node is inserted into our FRAIG representation, we check whether there is already a node in the representation which is functionally equivalent to this new node (using SAT combined with simulation). If there is already a functionally equivalent node, we keep only one representation for the function and replace the representation of one node by the other (this is in contrast to [11] where various representations of the same function are kept for technology mapping purposes). In order to keep the overall size of the representation small we have to select carefully which representation is kept (see Section 4). BDD sweeping is known from combinational equivalence checking [9, 10] and builds BDDs for AIG nodes starting at the primary inputs until a certain node limit is reached. BDD sweeping is used there from time to time in order to identify equivalent nodes in the AIG. Since we are using SAT for maintaining the FRAIG invariant we do not need BDD sweeping with this objective. In contrast to the traditional use of BDD sweeping we make use of BDD sweeping in the cone of selected output functions of our FRAIG representation in order to compute smaller AIG representations. After one step of BDD sweeping we check whether our FRAIG representations decrease in size when parts of the FRAIG representation are replaced by subgraphs which are structurally equivalent to the BDDs computed during BDD sweeping.

The work by McMillan [15] is related to our approach in the sense that it also presents a method for unbounded model checking. In contrast to our approach it does not handle CTL properties, but invariants, and it does not use exact image computations, but overapproximations by so-called Craig interpolants. Whereas we are using sophisticated methods for compressing our state set representations, [15] only uses a simple method to identify functionally equivalent subformulas in overapproximated state set representations (i.e. building BDDs up to a small fixed size). Due to the overapproximated image computation the method of [15] needs to be applied iteratively on unfoldings of the transition relation for an increasing number of steps (as in Bounded Model Checking). Our method does not need several unfoldings, but it can be used in standard symbolic model checking just replacing BDD representations for state sets by AIG based representations.

Altogether, our approach includes the following novel contributions which are essential for the high quality of our experimental results (see Section 7):

- We developed methods for quantifier scheduling which are especially tailored towards our state set representations using FRAIGs. We can show that a proper scheduling of quantifications can lead from exponential representations to representations of linear size.

- The size of the FRAIG representations is limited by heuristics for node selection when functionally equivalent nodes are identified.

- We are using BDD sweeping as a method for non-local logic optimization of our FRAIG representations. BDD sweeping is controlled by heuristics based on the size of the AIG representations and on the success of previous runs of BDD sweeping.

We applied our representations of state sets and of transition functions to CTL model checking. We are using a standard CTL model checking algorithm based on symbolic representations of state sets. However, we make use of degrees of freedom in CTL model checking by preferring operations which are beneficial for our representation (see also Section 2).

Our experimental results prove the efficiency of our approach. We present several examples where BDDs are known to perform poorly and we demonstrate that our approach improves model checking for these examples by orders of magnitude. However, note that our approach is also able to produce competitive results for cases where BDD are known to perform well (which was not observed for approaches [12, 13], e.g.). We show in detail how our concepts such as quantifier scheduling, node selection heuristics and BDD sweeping as a non-local optimization step contribute to the success of our experiments.

The paper is structured as follows: We begin with a brief review of CTL model checking in Section 2. Then we describe both And-Inverter Graphs (AIGs) in general and the special version of AIGs we use as a data structure for model checking (Section 3). In Section 4 we describe our heuristics for node selection and in Section 5 we present our method for quantifier scheduling. AIG compression techniques by BDD sweeping are given in Section 6. After presenting experimental results in Section 7 we give some conclusions and future directions in Section 8.

2 Preliminaries

We use our FRAIG representation in the context of symbolic model checking [3, 4]. Symbolic model checking is applied to Kripke structures which may be derived from sequential circuits on the one hand and to a formula of a temporal logic (in our case CTL (Computation Tree Logic)) on the other hand.

An essential step in the recursive evaluation of CTL formulas is the preimage computation which computes for a set of states $Sat(\phi)$ the set of states $Sat(EX\phi)$ with at least one successor in $Sat(\phi)$:

$$\chi_{Sat(EX\phi)}(\vec{q},\vec{x}) := \exists\vec{q}'\exists\vec{x}'\Big(\chi_R(\vec{q},\vec{x},\vec{q}')\cdot\big(\chi_{Sat(\phi)}|_{\substack{\vec{q}\leftarrow\vec{q}'\\ \vec{x}\leftarrow\vec{x}'}}\big)(\vec{q}',\vec{x}')\Big) \tag{1}$$

(As usual χ_M means the characteristic function of set M, $\vec{x}$ represents the input variables, $\vec{q}$ the current state variables, and $\vec{q}'$ the next state variables. χ_R represents the transition relation of the Kripke structure.)

It is well known that the same formula can also be computed based on transition *functions* δ_i of the sequential circuit instead of the transition *relation* R:

$$\chi_{Sat(EX\phi)}(\vec{q},\vec{x}) := \exists\vec{x}'\Big(\chi_{Sat(\phi)}|_{\substack{q_1\leftarrow\delta_1(\vec{q},\vec{x})\\ \ldots\\ q_m\leftarrow\delta_m(\vec{q},\vec{x})\\ \vec{x}\leftarrow\vec{x}'}}(\vec{q},\vec{x},\vec{x}')\Big) \tag{2}$$

In our implementation of the model checking procedure we always prefer Equation (2) over Equation (1), since the substitution operation is easy in the AIG context and can be performed in parallel for several substitutions. Although we use sophisticated methods to

prevent memory blow-ups due to quantification, in principle quantification needs special attention, since quantifying a single variable has the risk of doubling the size of the representation. If not needed, we do not take this risk and we avoid the additional effort of preventing the representation from increasing.

3 And-Inverter Graphs

Recently, And-Inverter Graphs (AIGs) [9, 10] enjoy a widespread application in combinational equivalence checking and Bounded Model Checking (BMC). They are simply a special kind of directed acyclic graphs representing boolean functions. There are three types of nodes: *and nodes* with two outgoing edges, modeling the Boolean conjunction of the functions represented by the two edges, *variable nodes* with no outgoing edges but labelled with a variable name, representing boolean variables, and a special terminal node with no outgoing edges, forming the constant 0 function. The edges of an AIG may contain negation marks that denote complementation.

Constructing AIGs using one level structural hashing [10] assures that we do not have two different nodes with the same pair of predecessors.

3.1 Functionally Reduced And-Inverter Graphs

AIG representations of Boolean functions are not canonical – for each Boolean function there exist many structurally different AIGs. Actually an AIG may contain functionally redundant nodes, i.e., nodes which are roots of structurally different subgraphs representing the same functions.

Redundant nodes lead to two problems: On the one hand, the graph structure is inefficient. Redundant nodes could be merged to reduce the graph size. On the other, checking the equivalence of two nodes needs additional effort.

To address these problems Mishchenko et al. [11] introduced the notion of functionally reduced AIGs (FRAIGs). The main idea is to check for equivalent nodes using SAT-based equivalence checking techniques while constructing an AIG and to merge them immediately. This approach establishes the *functional reduction property*: each node in an FRAIG represents a unique Boolean function (up to complementation).

3.2 An AIG Package for Model Checking

Since we use our AIG package for state set representations in CTL model checking, we have different requirements compared to usual packages for combinational equivalence checking. In this section we have a brief look at the key features of our package.

Apart from standard Boolean operations which are translated into AND operations and / or complementations we have to support substitution and existential quantification. Substitution of variables by functions is basically reduced to replacements of inputs of an AIG by subgraphs representing these functions and it can be easily performed for several variables in parallel. Existential quantification $\exists x f$ is reduced to $f|_{x=0} + f|_{x=1}$ (with optimizations described in the following sections).

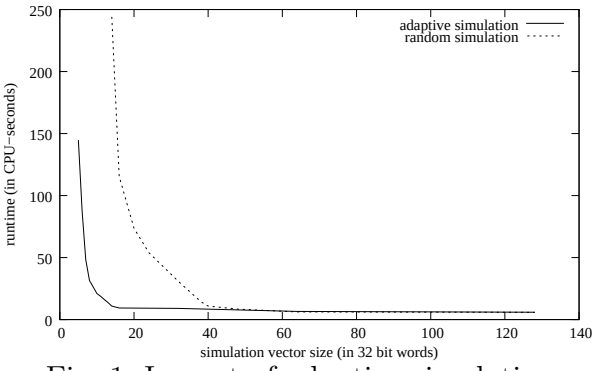

Fig. 1: Impact of adaptive simulation

Whereas in combinational equivalence checking only insertion of nodes has to be supported, we need efficient methods for the deletion of nodes. Nodes have to be deleted when certain state set representations are not needed any longer during the model checking procedure and when functionally equivalent nodes are merged into one representation. One-level structural hashing is applied as a fast technique for detecting isomorphic AIG nodes. The data structure we use for this

purpose includes measures that support the fast deletion of all occurrences of a node, while permitting lookups in constant time.

To maintain the functional reduction property, we use a simulation guided, SAT based equivalence checking method known from the combinational equivalence checking domain as proposed in [11]. The idea is to avoid powerful methods for easy problems: If for a given pair of nodes simulation is already able to prove non-equivalence, more time consuming SAT checks are not needed. The simulation vectors are initially random, but they are updated using feedback from satisfied SAT instances. Figure 1 shows the impact of different simulation vector sizes and the use of learned simulation vectors in a typical model checking run: Depending on the size of the simulation vectors used, the dashed line shows the run times for the complete model checking run, when the learning of distinguishing simulation vectors from satisfied SAT instances is turned off. The solid line shows the same run times for the case that learning is turned on. At least for the smaller sizes of simulation vectors learning leads to considerable improvements of run times.

Additional features of our AIG package which are used during model checking are node selection, quantifier scheduling, and BDD sweeping. They will be described in the following three sections.

4 Node Selection in Case of Equivalence

When constructing a new node in an AIG, one will often encounter the situation that the current AIG already contains a functionally equivalent node which is the root of a structurally different subgraph. Since the functional reduction invariant must be maintained, only one of the two nodes can be kept and the other one needs to be removed from the AIG. Unlike the approach in [11] where the already existing AIG node is kept and all equivalent nodes are stored in a list of possible structural representations for later technology mapping, our AIG package tries to keep the memory consumption as low as possible and thus destroys redundant nodes. This strategy is vital for a successful employment of AIGs in the model checking domain.

The question is whether to preserve the old, existing node or the newly created one. We use two different heuristics to conquer the problem:

- h_{keep}. We always keep the old node and discard the new node. The drawback of this trivial method is the possible rejection of more efficient structural representations.
- h_{size}. We keep the node that structurally depends on less variables. If the nodes have equal support sizes, we consider the subgraphs (cones) rooted by the two nodes and select the node which has a smaller cone.

In our experiments (see Section 7) we will show that the naive node selection heuristics h_{keep} may result in high and even unmanageable node counts, while the more advanced one is able to reduce the AIGs to reasonable sizes.[2]

5 Quantifier Scheduling

During model checking we eliminate existential quantifiers $\exists x f$ by replacing them by $f|_{x=0} + f|_{x=1}$. In the worse case this elimination may double the size of the representation. Thus, after an existential quantification of a series of variables the size of the representation may potentially show an exponential blow-up. In this section we will present a heuristic method which aims at limiting this (potential) increase in size by a clever choice of the order of quantifications ('quantifier scheduling').

[2] In the case the used heuristics suggest to keep the old node, the only thing to do is to delete the new node. But if the new node is selected, we use a technique similar to implementation techniques known from BDD packages: All edges of the AIG pointing to the old node must be modified to reference the new node. Since the data structure used in our AIG package does not contain a directory of all edges, we need to use a more subtle replacement method: we actually transfer the data of the new node object into the old node object and then delete the new node. By doing this no edge has to be touched.

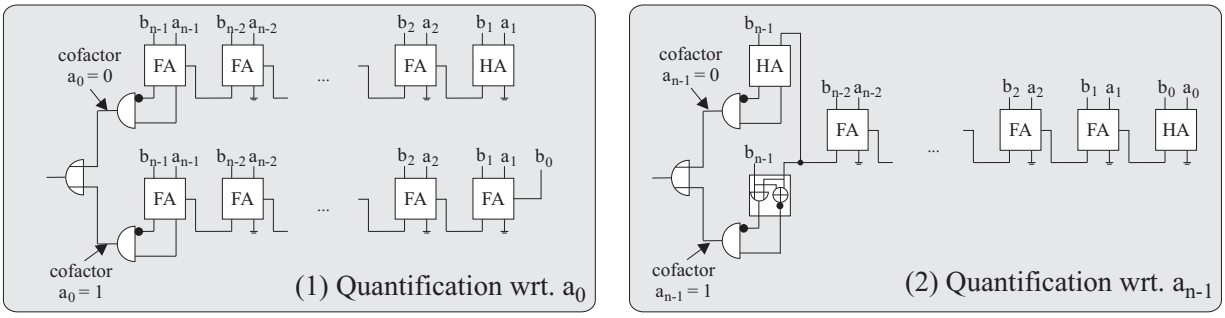

Fig. 2: $\overline{s_n} \cdot s_{n-1}$ after (1) quantification of a_0 and (2) quantification of a_{n-1}.

Quant. Nr.	orig.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
UP	111	105	106	115	140	197	318	567	1072	2089	4130	8219	16404	32781	1
DOWN	111	105	99	93	87	81	75	69	63	57	51	45	39	33	1

Tab. 1: 14-bit-adder, function $\overline{s_{14}} \cdot s_{13}$: Number of AIG nodes after quantifications of single variables a_i, according to orders UP and $DOWN$

5.1 A motivating example

First of all, we give a motivating example which shows that the order of quantifications may be essential for avoiding memory blow-ups. Consider a simple carry ripple adder which computes the sum $(s_n, \ldots, s_0)$ for two operands $(a_{n-1}, \ldots, a_0)$ and $(b_{n-1}, \ldots, b_0)$. Now we want to compute the set of inputs $(b_{n-1}, \ldots, b_0)$ with the property that there is an input $(a_{n-1}, \ldots, a_0)$ with $2^{n-1} \leq \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i < 2^n$, i.e. with $s_n = 0$ and $s_{n-1} = 1$. The problem may be solved by computing a symbolic (BDD or AIG based) representation of $\overline{s_n} \cdot s_{n-1}$ based on the carry ripple circuit and by computing the existential quantification $\exists a_{n-1} \ldots \exists a_0 \overline{s_n} \cdot s_{n-1}$. The result of the quantification is a representation of a characteristic function for the set of inputs $(b_{n-1}, \ldots, b_0)$ fulfilling the given property. Since it is easy to see that this set of inputs is equal to $\mathbb{B}^n$, the final result has to be equal to 1.

Now we consider the two extreme cases for the order of quantification: The first order UP is $(a_0, \ldots, a_{n-1})$ (i.e. we start with the quantification of the least significant bit) and the second order $DOWN$ is $(a_{n-1}, \ldots a_0)$ starting with the quantification of the most significant bit. Figure 2.(1) shows the result of the quantification wrt. the first variable a_0 of order UP (for simplicity applied to the given carry ripple circuit, not to the corresponding AIG, which has roughly the same structure). The illustration shows that the propagation of constants 0 and 1 already stops at bit position 1 and no subcircuit sharing can be observed in the remaining circuit . Thus, the size of the circuit is almost doubled by quantification.

However, if we quantify variable a_{n-1} first, we obtain the situation shown in Figure 2.(2). In this case most parts of the circuit are shared between the positive and the negative cofactor. Since we use FRAIGs which identify functionally equivalent nodes, the corresponding FRAIG also shows this sharing. The duplication of the number of nodes as observed in the previous case can not be seen here.

The effect shown above continues during the following quantifications according to the orders UP and $DOWN$: As shown in Table 1 for the example of a 14-bit-adder, we observe an exponential blow-up of AIG nodes during quantification according to order UP, whereas the number of AIG nodes is monotonically decreasing for quantification order $DOWN$. Altogether our example shows that there may be an exponential gap wrt. AIG sizes between good and bad orders of quantification. So we have a strong need for heuristics computing good quantification orders.

5.2 A heuristic approach to quantifier scheduling

Here we present a method for quantifier scheduling which is especially tailored towards our state set representations using FRAIGs. Our greedy method is based on estimations on the sizes of the results after performing quantifier elimination. Before performing a quantification $\exists x f = f|_{x=0} + f|_{x=1}$ for some variable x and some function f represented

by a FRAIG, we compute an estimate on the FRAIG size of the final result:

- In a first step we consider the subgraph of the AIG representing f and for each $\epsilon \in \{0, 1\}$ we determine by two traversals of this subgraph the set R_ϵ of nodes which are not removed by propagation of constant ϵ.
- In a second step we compute an estimate for the node sharing between the representations of the positive and the negative cofactor: If a node n which occurs both in R_0 and R_1 is not connected to variable x by a path in the AIG graph, then it does not depend on x and thus the nodes corresponding to n in the representation of $f|_{x=0}$ and $f|_{x=1}$ are the same (due to the functional reduction property). So the set $R_{0,1}$ of all such nodes is our estimate for the set of nodes shared between $f|_{x=0}$ and $f|_{x=1}$.[3]
- Finally, our estimate for the size of $f|_{x=0} + f|_{x=1}$ is $1 + |R_0| + |R_1| - |R_{0,1}|$.

For reasons of efficiency our estimate does not consider node sharing between nodes for both cofactors on the one hand and nodes which are not in the subgraph representing f on the other hand. Additionally, possible restructuring of the AIG during node insertion (see Sections 4 and 6) is not taken into account. However, as shown by experimental results, our heuristic estimate seems to be reasonable for computing a good order for quantification: Whenever a number of variables $x_1, \ldots, x_n$ has to be quantified for a function f, we compute for each x_i our estimate of the size of $\exists x_i f$ and (greedily) start with quantification of the variable with smallest cost. Then the method is repeated to determine the next variable to be quantified and so on.

We would like to point out that in the case of our motivating example from above our heuristic method leads to quantification order $DOWN$ which produces a monotonically decreasing number of AIG nodes, whereas unfavorable orders like the order UP shown above lead to an exponential peak size in the number of AIG nodes before the final result 1 is computed.

6 BDD Sweeping

BDD sweeping [9, 10] is a well-known technique from combinational equivalence checking (CEC). It builds BDDs for AIG nodes starting at the primary inputs until a certain node limit is reached. Whereas in [9, 10] BDD sweeping is used in order to identify functionally equivalent nodes in the AIG, this is not needed in our case, since we always maintain the functional reduction property using SAT as described in Section 3. Here we use BDD sweeping as a means for non-local optimizations of our AIG representations. However, for reasons of efficiency both the number of BDD sweepings and the cost of a single BDD sweeping have to be limited.

From time to time, after certain operations of the AIG package, BDD sweeping is applied to the cone of the corresponding result. BDD sweeping builds a BDD for the cone of the given AIG node starting from the variable nodes and using AND and NOT operations. Variable reordering applied by the BDD package automatically tries to find an optimal variable order in terms of BDD node count. If BDD sweeping is able to compute the BDD for the given AIG node, then we check whether it makes sense to replace the cone of the AIG node by an AIG which is structurally equivalent to the BDD. Here we exploit the fact that any BDD node can be interpreted as a multiplexer, which can be transformed into an AIG with exactly three AIG nodes. Thus, if the size of the generated BDD is smaller than one third of the size of the given cone, we create an AIG from the BDD structure by recursively transforming the BDD nodes to their three-node AIG representation. When inserting this new AIG into the AIG package, node selection heuristics as described in Section 4 are used as usual with the additional effect that subgraphs of the new AIG may

[3] Situations where a node does not functionally depend on a variable x, but is (structurally) connected to x, may be possible in AIGs due to non-canonicity. However they are neglected for reasons of efficiency.

be replaced by smaller (functional equivalent) representations which are already present in the existing AIG graph.

In order to limit the *cost* of a single BDD sweeping we use a variable *BDD_limit*. Whenever the number of nodes in the BDD package is larger than *BDD_limit*, the BDD construction is aborted.

In order to limit the *number* of BDD sweepings, we decided to confine BDD sweeping to the results of cofactor operations which occur during existential quantification, since existential quantification of a variable is the only operation that has the risk of doubling the size of the representation. Moreover, BDD sweeping is not applied after *all* cofactor operations, but only after a small fraction of cofactor operations controlled by sophisticated heuristics based on the sizes of the results and the success of previous BDD sweepings. To avoid unnecessary BDD sweepings, BDD sweeping is only applied, if the AIG size of the current operation is larger than a some variable *AIG_limit*. *AIG_limit* is initialized to a certain constant (100 in our current implementation) and it evolves as follows:

- If a BDD is successfully built within the node limit *BDD_limit*, but it is not used in the AIG due to its size, *AIG_limit* is multiplied by a certain factor $f_1 > 1$ ($f_1 = 1.2$ in our current implementation).
- If the BDD construction is aborted, since *BDD_limit* is exceeded, *AIG_limit* is multiplied by some larger factor $f_2 > 1$ ($f_2 = 4$ in our current implementation).
- If a BDD is successfully built and a structural equivalent AIG is inserted into the AIG package, then *AIG_limit* is set to the size of the resulting AIG.
- Whenever BDD sweeping is not applied, since *AIG_limit* is too large, *AIG_limit* is decreased by multiplication with $1 - f_3 \cdot (\frac{1}{2})^{abort}$, where f_3 is a small constant ($0 < f_3 < 1$) and *abort* is the number of times the BDD construction is aborted due to an exceeded node limit (in our implementation we used $f_3 = 0.01$).

The presented heuristics ensure that unsuccessful BDD sweeping runs result in fewer BDD sweeping runs in the future.

The variable *BDD_limit* for aborting BDD constructions has to be high enough to allow for BDD variable reordering and is set to *AIG_limit* times 100 in our implementation.

Whenever BDD sweeping is aborted, it ends up with a number of BDDs for AIG nodes in the considered cone, since it computes BDDs beginning with the input variables of the cone. The procedure described above can be easily extended by making use of the BDDs computed so far. However, this feature is not yet realized in our prototype implementation.

7 Experimental Results

We performed a number of experiments for evaluating our approach. The examples Coherence, DAIO, Lock, Picojava/ICU, and Barrelshifter are taken from the *VIS Verification Benchmark* set [16].[4] Moreover, we used a pipelined ALU ('PALU') similar to the one presented in [3]. The pipelined ALU includes 16 registers, a combinational adder, a combinational multiplier, and bitwise operations, all with a bit width of 16 bits. For the pipelined ALU we checked the CTL formula $\phi = AG\big(\text{"}\mathtt{R}_2 := \mathtt{R}_0 \oplus \mathtt{R}_1\text{"} \rightarrow \big((AX)^2\mathbf{R}_0 + (AX)^2\mathbf{R}_1 \equiv (AX)^3\mathbf{R}_2\big)\big)$ (similar to formula (1) from [3]).[5]

All experiments were performed on a 3 GHz Pentium4 workstation running Debian Linux. We used a timeout of 12 CPU hours.

[4] The number of registers in the barrelshifter was increased from 4 to 10.

[5] Given an *exor* operation in the instruction register the formula basically checks whether the contents of the destination register in three steps are the same as the *or* operation of the contents of the operand registers in two steps. This would be true for an *or* operation in the instruction register, but is obviously not true for the *exor* operation.

In a first experiment we evaluated the effect of our node selection heuristics from Section 4. Table 2 lists the peak node counts, numbers of node replacement steps, and run times for the two different proposed node selection heuristics: the naive method h_{keep} (always keeping the already existing node), and h_{size}. In this experiment we turned off BDD sweeping, because the naive method h_{keep} would never use the results of BDD sweeping. (Thus the results would be biased towards h_{size}, since it can exploit BDD sweeping whilst h_{keep} does not profit from it.)

The results clearly show that the node selection heuristics are of great importance for obtaining good results: The heuristics h_{size} lead to a considerable decrease in peak node counts. The most impressive examples are Picojava and DAIO where the peak number of AIG nodes for the naive method are by 124% and 244% higher than for h_{size}. Not only the node counts, but also the run times are greatly reduced by h_{size}, in the case of Picojava from 40.2 CPU seconds with h_{keep} to 8.4 CPU seconds with h_{size}, in the case of DAIO even from 5.7 CPU hours with h_{keep} to about 7 CPU minutes with h_{size}. Results for example 'Coherence' could not be given here, since – without BDD sweeping – the computation did not finish within our limit on CPU time, i.e., BDD sweeping is essential for success with this benchmark (see also experiments below, Table 3). Since the node selection heuristics h_{size} seems to be the best choice, we always use this method in the following.

Name	h_{keep} (naive method)		h_{size}		
	nodes	time	nodes	repl.	time
Lock	12196	6.5	10241	1531	5.7
Picojava	24021	40.2	10747	810	8.4
DAIO	65046	20571.1	21879	18907	421.6
Coherence	-	> 12h	-	-	> 12h
Barrel	6037	20.4	6037	0	20.4
PALU	6221	4.0	6221	288	2.4

Tab. 2: Impact of node selection heuristics

Name	original order		quantifier scheduling							BDD-	VIS
	nodes	time	nodes	time	BDD sweep	BDD success	BDD limit	SAT	SAT equiv	MC time	(v 2.0) time
Lock	631	0.33	629	0.38	$7.2 \cdot 10^{-3}$	0.74	0.00	0.020	0.99	0.04	< 0.1
Picojava	8130	9.6	2943	6.6	$1.5 \cdot 10^{-3}$	0.19	0.00	0.051	0.40	1.10	12.2
DAIO	925	0.79	925	0.81	$2.4 \cdot 10^{-3}$	0.89	0.00	0.160	0.92	0.39	1.5
Coherence	116303	943.4	40454	137.1	$2.2 \cdot 10^{-4}$	0.41	0.00	0.013	0.73	17.00	0.4
Barrel	6037	32.0	6037	32.3	$6.5 \cdot 10^{-5}$	0.00	1.00	0.021	0.37	> 12h	> 2GB
PALU	6221	26.2	6215	78.0	$7.7 \cdot 10^{-5}$	0.25	0.75	0.011	0.79	> 12h	> 2GB

Tab. 3: Experiments with BDD Simulation and Quantifier Scheduling

In the following experiments we consider our method with BDD sweeping turned on. In the second experiment shown in Table 3 we evaluated the effect of quantifier scheduling and in the third experiment we compared our results to BDD based CTL model checkers (to VIS in the last column and to our own implementation of a CTL model checker in the column before).

First of all, columns 2 and 3 show the peak node counts and run times for the unmodified quantification order ('original order') and columns 4 and 5 give the same results for our quantifier scheduling heuristics from Section 5. It can be observed that quantifier scheduling always improves the peak node counts; for Picojava, e.g., by a factor of 2.8, for Coherence by a factor of 2.9. With exception of PALU, run times are always better or at least in the same range, for our slowest example Coherence run times are even reduced from 15.7 CPU minutes to 2.3 CPU minutes.

Columns 6-10 give some more details for our experiments with BDD sweeping and quantifier scheduling turned on: Column 6 shows the number of applications of BDD sweeping divided by the total number of attempts to insert a node into the AIG. Column 7 shows the numbers of successful applications of BDD sweeping (i.e. the numbers of BDD sweepings where the results are used in the AIG package) divided by the total number of BDD sweepings. And finally column 8 shows the numbers of aborted BDD sweepings (due to exceeded node limits) divided by the total number of BDD sweepings. BDD sweeping is only applied from time to time in all cases and in cases where BDD sweeping is not very successful (especially for examples 'Barrel' and 'PALU') our heuristics from Section 6 work in the sense that unsuccessful BDD sweeping runs result in fewer BDD sweeping runs in the future. Column 9 shows the number of SAT checks divided by the total number of attempts to insert a node into the AIG, column 10 shows the fraction of

SAT checks which lead to the result that the compared nodes are functionally equivalent. Although we always maintain the functional reduction property of our FRAIGs, the results show that the assistance of SAT by simulation and structural hashing as described in Section 3.2 assures that SAT is applied only for 1-5% of all node insertions. Moreover, the high percentage of SAT checks proving functional equivalence of two nodes shows the effectiveness of simulation in avoiding unnecessary SAT checks for nodes which are not equivalent.

The last two columns give a comparison of our results to BDD based CTL model checking. For the last two examples, Barrel and PALU, the BDD based model checkers were not able to provide a result (for our own model checker we had to abort due to a CPU limit of 12 hours, VIS quickly ran out of memory (2GB)). However, these examples did not form a problem for the model checker presented in this paper and we could solve them within a few seconds.

In contrast, for the remaining benchmarks taken from the VIS Benchmark set, BDDs are known to perform well and these examples could be solved quickly by the BDD based model checkers. However, note that also for these examples we could observe that our approach succeeded in producing competitive results within a few seconds.

8 Conclusions and Future Work

We presented an approach to symbolic CTL model checking based on And-Inverter Graphs as a representation of characteristic functions. Several methods such as functional reduction using simulation assisted SAT checks, node selection heuristics, quantifier scheduling, and BDD sweeping contribute to the success of our approach. Although the experimental results for our preliminary implementation already appear to be impressive, we believe that there is still room for improvement. Among others, we will work on future improvements of the heuristics presented in Sections 4, 5, and 6. In addition, we will investigate whether methods for structural SAT solving [9] will be useful in our context and we will explore whether it sometimes makes sense to switch to lazy methods for AIG compression instead of our eager one.

References

[1] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[2] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[5] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Constuction and Analysis of Systems*, volume 1579 of *LNCS*. Springer Verlag, 1999.

[7] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conf.*, 1999.

[8] M. Sheeran, S. Singh, and G. Stalmarck. Checking Safety Properties Using Induction and a SAT-solver. In W.A. Hunt Jr. and S.D. Johnson, editors, *FMCAD*, volume 1954 of *LNCS*, pages 407–420. Springer, 2000.

[9] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Int'l Conf. on Comp. Design*, pages 459–464, 2000.

[10] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. M.K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Trans. on CAD*, 2002.

[11] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K. Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 03 2005.

[12] P.A. Abdullah, P. Bjesse, and N. Een. Symbolic reachability analysis based on sat-solvers. In *Tools and Algorithms for the Constuction and Analysis of Systems*, volume 1785 of *LNCS*. Springer-Verlag, 2000.

[13] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 124–138. Springer Verlag, 2000.

[14] G. Cabodi, M. Crivellari, S. Nocco, and S. Quer. Circuit based quantification: Back to state set manipulation with unbounded model checking. In *Design, Automation and Test in Europe*, 2005.

[15] K.L. McMillan. Interpolation and SAT-Based Model Checking. In W.A. Hunt Jr. and F. Somenzi, editors, *Int'l. Conf. on CAV*, LNCS. Springer, 2003.

[16] The VIS Group. VIS Verification Benchmarks. http://vlsi.colorado.edu/∼vis/.