Improving Symbolic Reachability Analysis by means of Activity Profiles

Gianpiero Cabodi, Paolo Camurati, and Stefano Quer

Abstract— **Symbolic techniques have undergone major improvements in the last few years. Nevertheless, applications are still limited by memory size and time constraints. As a consequence, extending their applicability to larger and real circuits is still a key issue.**

Within this framework, we introduce "activity profiles" as a novel technique to characterize Finite State Machines described by their transition relation. In our methodology a "learning phase" is used to collect activity measures. They are gathered, in an inexpensive way, for each Binary Decision Diagram node of the transition relation. They indicate the activity the node has been involved in, as an estimate of its correlation with space and/or time costs.

The above information can be used for several purposes. In particular, we present an application of activity profiles in the field of reachability analysis, to enhance memory and time performance of traversals. More specifically, we use transition relation subsetting in order to traverse the state transition graph in a non purely breadth-first, guided and multi-step fashion. Comparisons with other state-of-the-art approaches show that our sequence of partial traversals produces "best-ever" results for all the large benchmarks analyzed.

Keywords— **Binary decision diagrams (BDD's), symbolic techniques, formal verification, partitioning.**

I. Introduction

STATE-OF-THE-ART approaches for reachability analysis and formal verification of circuits, modeled as Finite State Machines (FSM's), exploit symbolic techniques based on Reduced Ordered Binary Decision Diagrams (ROBDD's or simply BDD's). They have been intensively used for their efficiency, and their application covers several fields of symbolic computation.

Given a sequential system modeled as a FSM, a *Symbolic Forward Traversal* is a breadth-first visit of the FSM state transition graph. Figure 1 shows a pseudo-code for this procedure.

```
  Traversal (T, S0)
1     Reached ← S0
2     From ← S0
3     New ← S0
4     while (New ≠ ∅)
5         To ← Image (T, From)
6         New ← To · ¬Reached
7         Reached ← Reached + New
8         From ← BestBDD (New, Reached)
9     return (Reached)
```

Fig. 1. Forward Traversal.

G. Cabodi and P. Camurati are with the Politecnico di Torino, Dip. di Automatica e Informatica, Corso Duca degli Abruzzi 24, I-10129 Turin, Italy.

S. Quer is with the Politecnico di Torino, Dip. di Automatica e Informatica, Corso Duca degli Abruzzi 24, I-10129 Turin, Italy (e-mail: quer@polito.it).

$\mathsf{T}(s,x,y)$ is the transition relation of the system[1], and S_0 is the set of initial states.

The traversal is accomplished by a series of image computations (function Image), from the initial states and going on until no new states are reached.

Reached states are accumulated in Reached. From is the input set for each image computations, and To is the resulting set. One can easily prove that From can be any set of states included in Reached, and including all the newly reached states from the previous iteration. So the BestBDD function (line 8), selects From in the above interval according to some optimality criteria (usually minimum BDD size).

Image computations are performed as a combined conjunction and existential quantification operation:

$$\begin{aligned} \mathsf{To}(y) &= \text{IMAGE}(\mathsf{T}(s,x,y), \mathsf{From}(s)) \\ &= \exists_{s,x}(\mathsf{T}(s,x,y) \cdot \mathsf{From}(s)) \end{aligned} \quad (1)$$

The above formula is the core operation of all symbolic reachability techniques. For medium-small circuits it is usually quite efficient. Nevertheless it reaches its limits on large practical examples. Several improvements have thus been proposed to deal with realistic circuit sizes in affordable time and memory. They are summarized in the following section.

A. Previous Works

Building the transition relation implies the following computation:

$$\mathsf{T}(s,x,y) = \prod_{i=1}^{n}(y_i \equiv \delta_i(s,x)) \quad (2)$$

where δ_i are next state functions, and n is the number of memory elements of the FSM.

In the "monolithic" approach all conjunctions are computed once and for all. Very often the existence of an input leading from a state to the next one is important, rather than its value, and primary inputs are abstracted from T, defining a new relation usually called the *non-deterministic transition relation.* As this naive approach too often incurs in the BDD explosion problem, the transition relation has been represented and used in *partitioned* or *clustered* forms. In the "partitioned" form, conjunctions remain implicit until an image is computed. "Clustering" generalizes this approach with partial computation of conjunctions, and a threshold based control on cluster BDD sizes. In both cases, efficiency of image computations relies on heuristic

[1] In the sequel we use s, x and y for present state variables, primary inputs, and next state variables, respectively.

partition sorting and *early quantification* of domain variables, whenever they are outside the true support of the remaining terms.

Dynamic variable reordering techniques have been introduced to adapt the variable order to the problem as the computations proceed. They can slow down an application by a large factor, but they often allow a successful completion of the application.

Over-approximate traversal methods are used to scale down the complexity of large problems, giving the tightest possible overestimation of the final reachable state set.

More recently, many researchers have followed the general trend of guiding symbolic traversals, by means of state space pruning or transition relation partitioning. On the one hand, these techniques are used for partial visits of the state graph, focusing on a given property or behavior. On the other hand, a sequence of properly guided traversals may be a more convenient way of visiting the whole state set. The result is an interleaved breadth/depth-first traversal, with better performance, especially in terms of memory. The real challenge, of course, is finding efficient problem decomposition and guided search strategies.

In [1], [2], [3] the authors propose different techniques to control the size of BDD's during the whole traversal process. The key idea is to use subsetting based on "high density". The density of a BDD is defined as the ratio between the number of states and the number of BDD nodes. Whenever a BDD has a size larger than a threshold, a "dense" subset is extracted and used for the next traversal steps. A full traversal is thus decomposed as a sequence of breadth-first partial[2] ones. Due to previously discarded reached states each local fixed point (of a partial traversal) may be a "dead-end". As a consequence, the convergence test, i.e., checking for a global fixed point, requires a "dead-end recovery" procedure, i.e., computing the image of the overall set of reached states.

Partitioning based on Shannon decomposition is the inspiring idea of [4], [5], [6], [7]. Partitions are obtained by selecting splitting variables or, in the more general case, "window" functions. In [4], [7] BDD's are automatically and dynamically partitioned, using Boole's decomposition theorem, whenever their size is larger than a threshold. Efficiency relies on good decomposition routines which have a low complexity and produce balanced and small size partitions. In [5] partitioning is a manual task and traversals follow the interleaved breadth/depth-first idea. The target is to optimize traversals following the "modes" or sub-behaviors of the circuit, to avoid intermediate memory blow-ups. In [6] the state space is partitioned statically using cubes as easily manageable "window" functions. Partial traversals are performed within sub-spaces, with different variable orderings allowed, and a communication cost to transfer BDD's among sub-spaces.

A technique related to [5] is developed in [8]. The authors propose "hints" to constrain the transition relation. Each hint is expressed as a constraint on primary inputs and/or state variables. The main effort is on accurate analysis of the circuits to find the right set of hints, and human experience is very important.

In [9] the authors introduce two main optimizations. First of all, they remove next state functions that functionally depend on other ones. This can be viewed as a dynamic application of memory element removal for sequential optimization. Secondly, they introduce a special purpose dynamic reordering scheme, exploiting the knowledge of present and next state variable sets.

In [10] the authors propose a method to better use don't care sets. They present an improved approximate traversal method, which is able to over-approximate reachable states. The complement of the latter is an under-approximation of the input don't care set, which is used for optimized image computation in exact reachability. This is achieved by minimizing the set of support of the transition relation and improving the early quantification scheme.

Mixing random simulation with adaptive search and symbolic computation to explore a large number of controller interactions, is one of the key issues of [11]. "Guards" are used to determine which parts of the overall state space have been explored.

In Section I-B we introduce our approach, and in Section I-C we compare it with the above works.

B. Overview of the Presented Approach

The inspiring ideas of the work presented here are quite general: We propose to exploit a "learning" phase to "predict" the cost of future computations. Given a BDD based operation we want to optimize (i.e., we look for speed-up and reduced memory cost), we start by carrying out an initial *learning* phase. This phase focuses on the correlation between the operands (represented as BDD's), and the symbolic recursive operations performed on them. We define the *activity profile* of a BDD, involved as an operand in recursive operations, a set of statistics obtained as a result of this phase.

Once collected, we may use it for several purposes, as an additional information to bias decompositions, optimizations, and package tuning.

Among the possible applications, we exploit profiles to optimize reachability analysis in a mixed breadth-first and depth-first guided traversal framework, using transition relation subsetting. We made this choice because guided search techniques have been able to reduce the memory/time cost of traversals. Meanwhile, they still lack efficient ways of automatically biasing and guiding traversals. Selection of sub-spaces or sub-behaviors is often done manually, and whenever it is automatic, it mostly relies on the divide-and-conquer paradigm, with no particular skill in predicting the expected gain. Activity profiles open very promising trends in this domain, since they are based on a cause-effect relation between BDD's and efficiency of symbolic operations.

Given a FSM represented by its transition relation T, we start by gathering, *automatically* and *dynamically*, activity indicators for each BDD node of T, throughout a learning

[2] The term "partial" traversal will be used here-on to denote the visit of a subset of reachable states.

traversal. This traversal should be a good representative of the full symbolic task to be carried out, so that the activity indicators have a reliable prediction power. In order to obtain a low cost learning phase, our solution is an approximate traversal or a few exact traversal iterations. The purpose of this phase is to record the correlation of each node with the cost of the image computations performed. We observe, and measure, recursive calls and operation cache hits, which are related to the time cost of symbolic operations, and the amount of newly generated BDD nodes, which is more linked to memory usage.

Given the activity profile collected, we exploit it for a two-way guided traversal based on T subsetting. We discard nodes with low activity, i.e., used seldom or associated to minor behaviors of the FSM, or with high activity, i.e., used too often or involved in expensive operations. The resulting subset of the transition relation is used for a partial traversal, computing a subset of the overall reachable state set. This subset is the starting point for the following phase, a full traversal, using the entire transition relation.

A few more remarks are necessary.

The learning phase should be inexpensive or at least it should have a negligible cost with respect to the overall optimized process. The activity profile has a memory overhead of a few counters for each BDD node involved in the profile collection. In our application, this overhead is insignificant as the BDD nodes of the transition relation are usually a small fraction of the BDD nodes generated during the whole traversal. Nevertheless, any application using activity profiles should consider this overhead, and the trade-off between it and the performance improvements obtained.

Moreover, for prediction purposes, the learning phase should well represent the entire process. To this extent, our method is valuable as far as the activity recorded in the past is a good prediction for the future. Besides trading off the size of the learning process with the overhead introduced, this implies assuming that the overall traversal process is characterized by some repeated behaviors over different traversal iterations. We cannot provide a theoretical support for this assumption, but our experimental results confirm its validity.

Furthermore, the overall process should be automatic and rather self-tuning.

Due to their strict relation with BDD nodes, activity profiles are expected to be influenced by variable ordering, with better results whenever similar orderings are used in the profile learning phase and in the following guided traversal. As a conclusion, it should be noted that as far as reachability analysis is concerned, smarter and more complex guided search schemes are possible, but we believe that the experimental results obtained are sufficient to support the claims of the proposed method.

Some of the above contributions are full-length descriptions of results which are available in the conference literature [12]. More in particular, in this work we extend and improve the heuristics for transition relation subsetting. Thanks to this, and to a finer tuning of the overall methodology, we present a totally new set of experimental results, with reductions in the required memory, and up to one order of magnitude improvements in running time. Larger circuits have also been added, and to the best of our knowledge one circuit is traversed for the first time.

C. Comparison with Other Works

Similar in its inspiration to other recent works presented in Section I-A, our method presents original contributions and shares similarities with them.

- Programs exploiting *learning* phases are used in several fields of electronic CAD. In the field of reachability analysis, [11] exploits preliminary simulations to bias guided symbolic traversals. In our work, we introduce a new metric to characterize the activity of recursive BDD operators, by means of node related statistics. Activity profiles are collected on a node by node basis, with low time and space overhead. As far as we know, there are no similar approaches in literature.
- We follow the most recent trends in FSM reachability analysis: Not merely optimizing image computations, but producing cheaper intermediate reachable state sets, through non purely breadth-first guided traversal schemes. Our original contribution is exploiting information coming from traversal themselves as a source for improved reachability analysis. We work on the correlation between reachable states and the transition relation within symbolic traversal operations (namely the relational product operator), whereas previous works focus on reachable state sets [1], [2], [3], [4], [7], or the transition relation [5], [6], [8], without any consideration about their combined impact on performance. This allows us to isolate sub-behaviors of the FSM, and to follow the most (or the least) active ones, given a heuristic evaluation function.
- Concerning the heuristics to guide and focus traversals, state-of-the-art approaches rely on human decisions (e.g., selection of "idle" latches [5] or hints [8]), or automatic methods (e.g., working on the BDD density [1] or size [4]). We follow the second trend, and we seek for an *automated* and *self-tuning* process. Since we exploit information directly coming from the process itself (by means of learning), this is an additional and relevant input, which makes our solution inherently superior, yet orthogonal and complementary to other approaches in terms of self-tuning.

D. Road-map

Section II firstly describes the activity statistics we introduce, then shows how we collect them (Section II-B). After that, Section II-C shows how we use the above statistics to prune the transition relation of a FSM. This produces a subset of the transition relation, to be used for the subsequent reachability analysis phase. Section II-D discusses static and dynamic learning, and the relation between learning and variable reordering. Section III introduces the guided traversal, based on a two-step approach. We first use the subset of the transition relation, then the entire one. Finally, Section IV describes the experimental results obtained with this method. Section V closes the paper with some conclusions.

II. The Activity Profile

Our aim is characterizing the core symbolic operations of reachability analysis and formal verification. We specifically address traversals based on the transition relation, where the key operation is the relational product (combined conjunction and existential quantification).

Let us refer to Equations (1) and (2), which may be combined and rewritten as:

$$\begin{array}{rcl} \mathsf{To}(y) & = & \textsc{Image}(\mathsf{T}(s,x,y),\mathsf{From}(s)) \\ & = & \exists_{s,x}(\prod_{i=1}^{n}(y_i \equiv \delta_i(s,x)) \cdot \mathsf{From}(s)) \end{array} \quad (3)$$

Following the standard approach outlined in Section I-A, i.e., clustering and early quantification, the transition relation is first clustered by properly sorting the terms and partially computing products. The result is $\mathsf{T}(s,x,y) = \prod_{i=1}^{m}(t_i(s,x,y))$, and image computation is expressed (with early quantification) as:

$$\begin{array}{rcl} \mathsf{To}(y) & = & \exists_{s,x}(\prod_{i=1}^{n} t_i(s,x,y) \cdot \mathsf{From}(s)) \\ & = & \exists_{s,x}(t_m(s,x,y) \cdot \ldots \cdot t_1(s,x,y) \cdot \mathsf{From}(s)) \\ & = & \exists_{s^m,x^m}(t_m(s,x,y) \cdot \ldots \cdot \\ & & \quad \exists_{s^1,x^1}(t_1(s,x,y) \cdot \mathsf{From}(s)) \ldots))) \end{array} \quad (4)$$

where the innermost products may be executed, with early quantification of variables not in the support of the outer terms.

The process results in a linear chain of relational products, usually producing peak BDD sizes within intermediate steps [1].

Our first target is measuring the correlation between the transition relation T and the set of states From during relational product operations (Equation (4)), on a BDD node by node basis. To achieve this result, we use a set of activity counters associated with each BDD node of T. We upgrade these counters within *learning* relational products.

The general idea is the following one. The image operator (Image) receives the two operands: The profiled BDD T (the transition relation) and the set From. It produces the result To and the activity profile of T, i.e., a set of counters associated with each node of T.

Our method works with both monolithic and partitioned/clustered transition relations. In the former case an image computation is a single relational product, in the latter one it is a sequence of relational products, each one involving a term of the transition relation (t_i), and an intermediate product (result of previous relational products). Other decomposition schemes (e.g., disjunctive forms) are also trivially supported.

A. Profile Description

In this section we describe the activity counters previously introduced.

Let us work on a relational product $\exists_{\mathtt{v}}(\mathtt{t} \cdot \mathtt{f})$, being `t` the profiled function, i.e., the one for which we collect activity measures. In case of a monolithic transition relation, `t` is the transition relation itself T, and `f` is the set From. In case of a partitioned/clustered transition relation, `t` is the generic partition/cluster $\mathtt{t_i}$, and `f` is an intermediate product previously obtained (see Equation (4)).

The activity profile is meant to record time and memory related measures. We thus observe recursive calls of the relational product operation, taking into account the operation cache, which avoids sub-problem re-computation, and the memory cost of each relational product recursion expressed in terms of newly generated BDD nodes.

More specifically, if `n` is the generic BDD node of `t`, we define three counters recording the activity within recursions on `n`:

- `n.rec`: Number of recursions on `n`, i.e., the number of times the node appears as an operand of a relational product recursive call.
- `n.cacheH`: Cache hits on `n`, i.e., the number of times `n` appears as an operand of a recursive call whose result is found in the operation cache table.
- `n.sizeC`: Node increment (size cost) within the BDD manager due to recursions on `n`, i.e., the difference between the total number of nodes after and before any recursion on `n`.

The recursive call counter, `rec`, is a direct indicator of the time spent working on node `n`, i.e., the higher this number, the more active is `n`. The cache hits counter, `cacheH`, is an indicator of activity stopping at `n` due to a cache hit. We heuristically keep a single counter, ignoring here that several cache entries may refer to the same node `n`. The `sizeC` indicator is related to the amount of BDD nodes generated by recursions on `n`. Since exactly measuring the size of the results produced would require a higher computational overhead, we prefer measuring the overall increase in BDD nodes produced by a recursion, which is easily provided by global counters maintained by BDD packages. This is not the size of the result (though often related to it), indeed, it is a good memory cost indicator of the operation performed. A negative increment is possible for size cost, whenever a recursion tree produces more released nodes than newly generated ones.

Example 1 shows a simple case of profile information associated with a BDD. Notice that the resulting memory overhead is proportional to the actual number of BDD nodes with activity profile, i.e., the transition relation.

Example 1: Figure 2(a) shows a generic BDD with three nodes x, y, and z. At this point, information on activity is not associated with BDD nodes. Figure 2(b) shows the same BDD after gathering the activity profile. Note that due to node sharing, statistic measures of a child node may be higher than the ones of a parent node (e.g., `y.rec` with respect to `x.rec`).

B. Collecting Activity Profiles

To collect the activity measures introduced in Section II-A, within recursive steps of the relational product operator, we concentrate on the general partitioned/clustered T, being the monolithic approach a sub-case.

Figure 3 shows the pseudo-code for the top level routine,

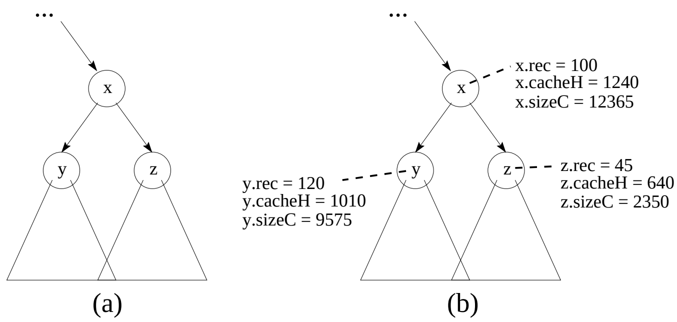

Fig. 2. BDD with Activity Profile.

i.e., an image computation (see Equation (4)) with learning activity. The intermediate product $\mathtt{P_0}$ is initially set to the set From. Then, the algorithm proceeds through an iteration, where function $\mathrm{RP}_{Profile}$, described in Figure 4, is called to compute the next intermediate product $\mathtt{P_i}$. To do that $\mathrm{RP}_{Profile}$ needs the set of quantifiable variables v, the new cluster $\mathtt{t_i}$, and the previous intermediate product $\mathtt{P_{i-1}}$. The overall product $\mathtt{P_m}$ is finally returned.

```
IMAGE_Profile(T(s, x, y), From (s))
    P0 ← From
    for i ← 1 to m
        v ← Early Quantifiable Variables of T for ti
        ti ← i-th Cluster of T
        Pi ← RP_Profile(v, ti, Pi−1)
    return (Pm)
```

Fig. 3. Image Computation with Learning.

Figure 4 shows our modified relational product operator, $\mathrm{RP}_{Profile}$ handling the previously defined counters. It describes the recursive procedure for the relational product (the RP function) $\exists_{\mathtt{v}}(\mathtt{t} \cdot \mathtt{f})$, with activity counters on the t factor.

```
    RP_Profile(v, t, f)
1       if (terminal case)
2           return (result of terminal case evaluation)
3       let n be the top node of t
4       if (result of RP_Profile(v, t, f) is cached)
5           n.cacheH ← n.cacheH + 1
6           return (result)
7       let τ be the top variable of t and f
8       r0 ← RP_Profile(v, t_τ̄, f_τ̄)
9       if (τ ∈ v)
10          if (r0 = 1)
11              r ← 1
12              r1 ← 0
13          else
14              r1 ← RP_Profile(v, t_τ, f_τ)
15              r ← r0 + r1
16      else
17          r1 ← RP_Profile(v, t_τ, f_τ)
18          r ← reduced, unique BDD node for (τ, r0, r1)
18      cache the result of this operation
20      n.rec ← n.rec + 1
21      n.sizeC ← n.sizeC + (total node increment)
22      return (r)
```

Fig. 4. Relational Product with Activity Counters Handling.

Relational product is quite similar to the Boolean AND operator. The main difference appears whenever the top variable of t and f (τ, line 7) must be existentially quantified (lines from 10 to 15). In this case the returned result is generally obtained through an OR operation (r_0+r_1, line 15). Our modifications to the standard procedure are indicated in lines (5) and (20-21), showing increments in the number of cache hits, recursions, and size cost.

C. T *Subsetting using Activity Profiles*

Given the activity profile of a transition relation T, we use it to generate a subset of T. In order to obtain a BDD with expected lower activity, we may follow two main alternatives:

- Discarding nodes with no or low activity, the aim being to isolate the most important/active subset of a transition relation. We mostly adopt this choice with time related activity, i.e., recursion and cache hits counters, the intuition being to follow behaviors which were alive for most of the learning process,
- Pruning nodes with an excessively high activity, in order to remove those nodes that are (expected to be) too expensive. We use this heuristic with the memory related size activity.

In the first case, the pruned nodes are replaced by the constant 0. In the latter case we avoid removing the whole BDD rooted at the node, since we experimentally found that this results in too raw subsettings in our traversal scheme. So we just clip one of the cofactors, following an idea derived from the Heavy-Branch subsetting technique [1]. Two choices are available: Pruning, i.e., replacing with 0, the heavier cofactor, or the lighter one (where cofactor weights are given by the activity indicators). These choices are better detailed in the following example.

Example 2: Figure 5(a) reports the BDD of Figure 2 with its activity indicators. Let us suppose to prune BDD nodes with no or low activity, in particular nodes with recursion activity lower than 50. In this case node z has a recursion value below this threshold and it can be clipped away and substituted with the 0 leaf. This results in the BDD of Figure 5(b). Let us suppose now to prune high

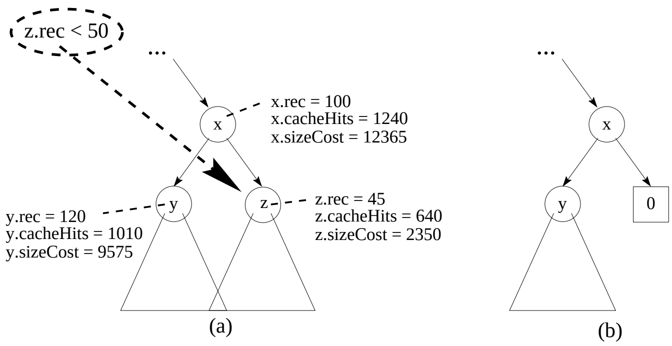

Fig. 5. Pruning Nodes with Low Activity from a BDD.

size activity with a threshold of 10000. The x node has sizeC over the threshold as indicated in Figure 6(a). If we clip the lighter cofactor (`z.sizeC = 2350`), we obtain Figure 6(b), whereas if we clip the heavier cofactor (`y.sizeC`

`= 9575`), we obtain Figure 6(c).

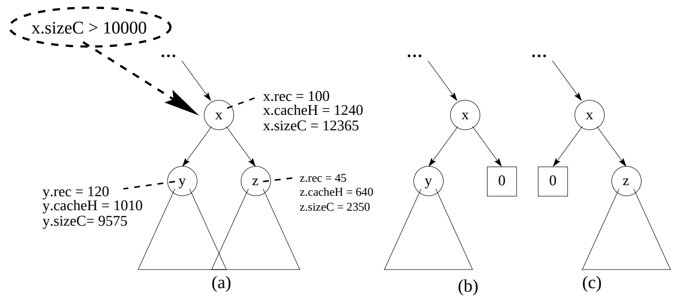

Fig. 6. Pruning Nodes with High Activity from a BDD.

The process results in a subset of the original `T`, with a simpler BDD, since it is derived from the original one, by replacing some of its nodes with the 0 constant. This is not always true for BDD's with complemented edges. In fact, a function and its complement, originally represented by the same BDD, may have different profiles and pruning, resulting in an overall larger BDD size. Overall, in the experimental result section (see Table III), we will show how pruning rarely influences the global size of the transition relation. In any case, we expect from the pruned BDD a lower activity for coming computations and this is the gain we are looking for.

The recursive pruning procedure is shown in Figure 7.

The BDD `t`, i.e., the transition relation or one of its clusters, is recursively visited depth-first. In case of a partitioned/clustered transition relation, the Prune function is iteratively called for all partition/cluster BDD roots. The `h` (heuristic), `th` (threshold), and `cH` (cache hits) parameters are used to guide the method, i.e., deciding how to prune subtrees from `t`. In our implementation `h` and `th` are selected by the user, while `cH` is used to collect the number of cache hits up in the recursion tree, to be incremented with the cache hits recorded in the present node `n` (line (28)) of `t` before recurring. Since a cache entry involving `n` stores the result of computations on the whole BDD rooted at `n`, we propagate cache hits down the recursion tree, and we heuristically assign the whole cached activity to all descendants of `n`.

The heuristic is selected on line (8). With the `Time` choice we aim at pruning BDD nodes with no or low computing time related activity, indicated by low recursions and cache hits. We use, for this purpose, three weighted terms (line 10): The number of recursions traversing node `n` (`n.rec`), the number of cache hits directly involving `n` as an operand (`n.cacheH`), and the number of cache hits above `n` in the recursion tree (`cH`). In particular, we use two terms for the cache activity, i.e., the one at the node `n` (`n.cacheH`) and the one above it (`cH`). w_1, w_2, and w_3 are set to 1 in the present implementation.

The `Time` heuristic is the only case in which we prune nodes with low activity. All other heuristics we propose work with the `sizeC` activity and are based on pruning high activity. The goal here is finding nodes of `t` that are expected to produce BDD explosion (i.e., high increments in overall node amount) within image computations. Clipping

```
    Prune (t, h, th, cH)
1       if (terminal case)
2           return (t)
3       if (result of Prune (t, h, th, cH) is cached)
4           return (result)
5       let n be the top node and τ the top variable of t
6       pruneT ← 0
7       pruneE ← 0
8       case (h)
9           Time:
10              if ((w1·n.rec+w2· n.cacheH+w3· cH) < th))
11                  return (0)
12          AbsSizePruneLight:
13              if (n.sizeC > th)
14                  pruneT ← t_τ.sizeC < t_τ̄.sizeC
15                  pruneE ← ¬pruneT
16          AbsSizePruneHeavy:
17              if (n.sizeC > th)
18                  pruneT ← t_τ.sizeC > t_τ̄.sizeC
19                  pruneE ← ¬pruneT
20          NormSizePruneLight:
21              if (n.sizeC/n.rec > th)
22                  pruneT ← (t_τ.sizeC / t_τ.rec) < (t_τ̄.sizeC / t_τ̄.rec)
23                  pruneE ← ¬pruneT
24          NormSizePruneHeavy:
25              if (n.sizeC/n.rec > th)
26                  pruneT ← (t_τ.sizeC / t_τ.rec) > (t_τ̄.sizeC / t_τ̄.rec)
27                  pruneE ← ¬pruneT
28      cH ← cH + n.cacheH
29      if (pruneE)
30          r0 ← 0
31      else
32          r0 ← Prune (t_τ̄, h, th, cH)
33      if (pruneT)
34          r1 ← 0
35      else
36          r1 ← Prune (t_τ, h, th, cH)
37      r ← reduced, unique BDD node for (τ, r0, r1)
38      cache the result of this operation
39      return (r)
```

Fig. 7. BDD Pruning Based on Activity Profiles.

of `t` is performed by recurring on one of the two cofactors, while forcing a 0 result for the other one. Cofactor selection is obtained by comparing the size costs of the (then and else) cofactor root nodes. We allow either choosing the heavier or the lighter cofactor, since we experimentally found both choices valid, depending on the circuit.

More specifically, we use two different size related heuristics, both combined with the light or heavy branch pruning choice: In the `AbsSizePruneLight` and `AbsSizePruneHeavy` case we compare absolute size costs of the node `n` with the specified threshold. If the node cost overcomes the threshold we compare absolute size costs of the cofactor nodes, to decide which one to prune. With `NormSizePruneLight` and `NormSizePruneHeavy` we do the same, but size costs are normalized with respect to the recursion count. This means that we are now considering the average cost for a recursion on `n`, rather than the overall size cost.

As the experimental results will show, we attained better improvements with the size heuristics. In our opinion, this is due to the fact that size heuristics directly face the BDD blow-up problem. Size cost is, in fact, the most critical limit for reachability analysis, time cost being only a consequence.

D. Static and Dynamic Learning, and Variable Reordering

We can consider our methodology as the sequence of two distinct phases: During the first one we collect an activity profile, during the second one we use it. There are two possible ways of coordinating and interleaving the two phases:
- Static learning: Learning is an initial (off-line) operation resulting in an activity profile, which is used to properly start up a second phase with optimized computations.
- Dynamic learning: Learning is always on, integrated within optimized operations, and activity profiles are used whenever required.

There are no conceptual differences between the two approaches. The second one is a generalization of the first one, and it is more appealing to obtain a fully automated methodology. Nevertheless, dynamic variable reordering happens to be a discriminating factor to choose the method.

Practical issues would suggest to fully integrate learning with dynamic variable reordering. To this extent, dynamic variable reordering destroys the initial BDD structure and produces a new BDD. This has strong implications for the learning process, because of the statistics related with each node. Ad hoc theory or heuristics to transform an "old" profile into a "new" one after reordering should be developed, or a profile should be considered valid only between two dynamic reordering activations. In both cases the strict correlation between dynamic learning and dynamic reordering should be dealt with.

Conversely, static learning can disable dynamic reordering during the learning phase (much cheaper than the overall traversal), while enabling it in the subsequent (optimized) phase. For sake of simplicity this is the technique we prefer in the current implementation.

Furthermore, it is obvious that the prediction ability of a profile is related to the choice of a "good" learning sample. On the one hand the amount of information stored in the sample should be enough to provide a reliable prediction. This means finding a trade-off between size and complexity of the static learning phase (our current implementation) or the profile validity window in dynamic learning. On the other hand we rely on a certain uniformity of the traversal activity (future activity should repeat the past one). For instance the nearer the variable ordering during learning is to the one(s) used for subsequent computations, the better is the prediction. We verified this intuition experimentally, and we came up with a scheme in which a *pre-learning* phase is done to allow dynamic variable reordering converge to a good initial variable order.

III. Symbolic Traversal with Activity Profiles

The technique previously described is the core of a two-way guided traversal based on transition relation subsetting. The activity of a transition relation is measured in learning Image operations, and the resulting profile is used to guide reachability analysis.

A full traversal is achieved through the following steps:
- *Pre-Learning* traversal: A few iterations of an exact or approximate traversal are carried out to properly set up the following phases (e.g., to find a good variable ordering).
- *Learning* traversal & T *pruning*: A limited number of fast exact or approximate traversal iterations, where relational products are computed by the $\mathrm{RP}_{Profile}$ (Figure 4) function. Given the activity profiles collected, a subset T_p of the transition relation is computed using function Prune (Figure 7).
- *Partial* traversal: A guided traversal, where T_p is used to produce a reachable state set $\mathsf{Reached}_p$, a subset of the final one $\mathsf{Reached}$.
- *Full* traversal: A final traversal, computing $\mathsf{Reached}$ from $\mathsf{Reached}_p$, using the original T.

The *Pre-Learning* phase is used to adjust the proper settings for the *Learning* phase. In particular it is used to skip too-easy initial steps and to move the *Learning* process to a proper iteration of the breadth-first state space visit. Moreover is used to find a good variable order for the subsequent phases. This phase is optional as *Learning* may start immediately. The *Partial* traversal is intended to explore a subset of the State Transition Graph with lower peak memory requirements than a full traversal. The final *Full* traversal completes reachability analysis, starting from the result of the previous one.

Performance and memory/time costs are strongly related to the *Partial* and *Full* traversal phases, being the other ones mainly intended as preprocessing steps. Overall, we expect lower memory peaks, compared with a standard breadth-first traversal, with a certain balance of costs (in this two-way approach) between *Partial* and *Full* traversal.

Figure 8 and 9 summarize the above concepts in terms of pseudo-code.

Traversal (T, From, $\mathtt{maxI}$)
  $\mathtt{level} \leftarrow 0$
  $\mathsf{Reached} \leftarrow \mathsf{From}$
  $\mathsf{New} \leftarrow \mathsf{From}$
  while ($\mathsf{New} \neq \emptyset$ AND $\mathtt{level} < \mathtt{maxI}$)
    $\mathsf{To} \leftarrow$ Image (T, From)
    $\mathsf{New} \leftarrow \mathsf{To} \cdot \overline{\mathsf{Reached}}$
    $\mathsf{Reached} \leftarrow \mathsf{Reached} + \mathsf{New}$
    $\mathsf{From} \leftarrow$ BestBDD (New, $\mathsf{Reached}$)
    $\mathtt{level} \leftarrow \mathtt{level} + 1$
  return ($\mathsf{Reached}$)

Fig. 8. Modified Forward Traversal.

Figure 8 is directly derived from the standard traversal routine described in Section I. In this case reachability analysis proceeds until the fixed point or a given limit for the number of iterations ($\mathtt{maxI}$) is reached. The algorithm models either an exact or an approximate traversal, the difference being the choice of a proper Image computation (exact or approximate [13]). Note that in general, although the over-approximation represents more states, its BDD is smaller than the one representing the $\mathsf{Reached}$ set, because several mutual dependencies between state variables disappear thanks to the approximation. Our approximate image computation is a straightforward one, obtained by separately computing the image of each transition relation cluster, and taking the Cartesian product of the results. This because approximate traversal is mainly used in the learning phase for learning purposes to compute a tight over-approximation of the exact result. Anyway, the implementation of the approximate analysis is not shown in the pseudo-code.

```
TRAVERSAL_Profile (T, S0, maxIPL, maxIL, h, th)
    Reached_PL ← TRAVERSAL (T, S0, maxIPL)
    Reached_L ← TRAVERSAL (T, Reached_PL, maxIL)
    T_Pruned ← PRUNE (T, h, th, 0)
    Reached_Partial ← TRAVERSAL (T_Pruned, Reached_PL, ∞)
    Reached_Full ← TRAVERSAL (T, Reached_Partial, ∞)
    return (Reached_Full)
```

Fig. 9. Forward Traversal.

Figure 9 describes a simplified version of our top-level routine. It directly calls the TRAVERSAL procedure of Figure 8 for each one of the previously described phases (Pre-Learning, Learning, Partial, Full). Each phase starts from the reached set collected by the previous phase, the only exception being the result of learning, which is not used (traversal resumes from the states reached by pre-learning).

The proposed scheme is obviously one among a large variety of choices for sequences of mixed breadth/depth-first traversals. We propose it as a simple way to show the potential improvements available through T subsetting. The main reason for the performance improvements is that the method avoids the intermediate blow-up of purely breadth-first traversals. Our method automatically prunes away some sub-behaviors (or parts of them) from T, and it fully explores the residual ones before restarting with the full transition relation. More flexible solutions are obviously possible, by allowing repeated *Learning-Partial* traversal sequences, but they are outside the scope of this paper.

As far as parameter choice and package tuning are concerned, the static learning scheme, with two-way guided traversal is not fully automated. Our solution introduces a few decision points under human responsibility: Approximate or exact image computation for pre-learning and learning, the maximum number of iterations, the pruning heuristic, and the threshold. This practice is very similar to other performance oriented parameter settings (e.g., cluster threshold, initial variable ordering, etc.) in the framework of reachability analysis. The user comes out with the right decisions after a few refinements and adjustments. To be more precise, since we mainly address hard-to-traverse circuits, we firstly determine a good initial variable ordering (by means of static and dynamic reordering in the Pre-Learning phase). Then we proceed trying to find the right tuning for the Learning phase. We try approximate image first, then the exact computation in case predictions do not yield good results in the subsequent phases. Going to the pruning heuristics, we found reasonable default values: We use the normalized size approach (either the NormSizePruneLight or the NormSizePruneHeavy) with an initial pruning threshold of about 80% of the maximum size cost obtained. Here our present implementation basically requires human decisions for a finer tuning, in order to balance the costs of the partial, and of the full traversal phases.

IV. Experimental Results

The technique presented has been implemented within a traversal program called Politecnico di Torino Reachability Analysis for Verification (PdTRAV), and built on top of the Colorado University Decision Diagram (CUDD) package [14]. Both the software used and the set of experiments obtained, and reported here, are available through the authors' home pages [15].

Activity data are generated and associated to BDD nodes by means of a hash table. This is due to the fact that BDD nodes in CUDD have no extra field available for user defined data. The method presented is implemented on a state-of-the-art traversal based on transition relation, i.e., using conjunctively partitioned transition relations, partition sorting, clustering, and early quantification. Other specific optimization techniques, which are deemed orthogonal to this approach, have been excluded from the experiments we present, to fully isolate the impact of activity profiles. In particular, the techniques presented in [4] would be applicable in both the partial and full traversal phases.

The results obtained are remarkable: All our experiments but one, a circuit traversed for the first time, ran with less than 50 Mbyte of main memory, and less than 2 hour of CPU time. To the best of our knowledge our results outperform all the data present in the literature on all the circuits analyzed.

We use a 400 MHz Pentium II Workstation with 128 Mbyte main memory, under the RedHat Linux 5.2 operating system.

We present data on full exact traversal of a few ISCAS'89 and ISCAS'89-addendum benchmarks, selected within the range of medium-large circuits for reachability analysis. We initially describe the activity profiles collected. We then move to the various phases of the reachability analysis process, which is characterized in terms of algorithmic choices, memory used, peak BDD size and CPU time. We finally compare our data with the best ones available in the literature.

Table I collects statistics on the circuits.

For each one, it shows the number of primary inputs (# I) and latches (# L), the sequential depth Depth, and the final number of reached states States.

Circuit s3384, as far as we know, is fully traversed here for the first time. As we use a multi-phase guided traversal the actual sequential depth of the circuit is overestimated.

We present three versions of circuit s5378, in which the subscript indicates the number of memory elements. Circuit $s5378_{179}$ is the original version, while $s5378_{164}$, and $s5378_{121}$ were generated through some optimization processes. In particular, circuit $s5378_{121}$ (usually called $s5378_{opt}$) has been obtained by iterative application of the redundancy removal procedure of [16], based on approximate traversal. The resulting circuit is sensibly smaller and easier to traverse than the original one. Circuits $s5378_{164}$ and $s5378_{179}$ are often found in the literature without dis-

TABLE I
CIRCUIT STATISTICS.

Circuit	# I	# L	Depth	States
s1269	18	37	9	$1.13 \cdot 10^{9}$
s1512	29	57	1023	$1.66 \cdot 10^{12}$
s3271	26	116	16	$1.31 \cdot 10^{31}$
s3330	40	132	7	$7.28 \cdot 10^{17}$
s3384	43	183	≤ 552	$2.31 \cdot 10^{45}$
s4863	43	183	4	$2.19 \cdot 10^{19}$
$s5378_{121}$	35	121	42	$2.58 \cdot 10^{17}$
$s5378_{164}$	35	164	44	$3.17 \cdot 10^{19}$
$s5378_{179}$	35	179	44	$3.17 \cdot 10^{19}$

tinction, since they have the same sequential depth and final number of states. As they have different intermediate behaviors within reachability analysis, we present both of them for sake of comparison and completeness.

Table II shows some statistics concerning the activity profiles of transition relations. Data are recorded through the learning traversals described in Tables IV and Table V.

We show average (Avg), standard deviation (StD), and maximum values (Max) of active recursions, cache hits and size cost. In order to allow comparisons, values are normalized with respect to the number of image computations (traversal iterations) performed. One could observe that these data roughly show dense distributions around the average values, with maximum values attained by a few nodes. This partially explains the power of the method as we are able to characterize and isolate “expensive” nodes of the transition relation. This is key for the memory oriented heuristics. In general a few nodes are pruned in order to achieve effective size clipping. Profiles are more uniform and homogeneous at the low activity values, so time oriented low activity pruning is expected to be a more regular process.

As introduced in Section II-C the size of the transition relation is weakly modified by our pruning heuristic. This can be observed in Table III, which collects statistics on the pruning phase. For each circuit it reports the clustering threshold used to build the transition relation T, the number of clusters, # Cluster, and the size of T before and after pruning. Size is evaluated considering the sharing among clusters, i.e., shared BDD's are taken into account only once. The pruning heuristics used is indicated in column Heuristic.

As one can see from the table, pruning implies just a slight reduction of the overall size of the transition relation. Only in two cases, namely s1512 and $s5378_{121}$, the final size is reduced to about 50% of the initial one. This analysis indicates that blowing-up behaviors are often produced by a small part of the transition relation and great benefits can be obtained with small modifications of the overall size.

Table IV and Table V present an analysis of our optimized traversal. Here we show detailed statistics on our four-step sequence (*Pre-Learning*, *Learning*, *Partial* and *Full* traversal), and global results. For each circuit we only report the pruning heuristic which gives best global result. The choices done are reported in Table III.

Columns Pre-Learning, Learning, Partial Traversal and Full Traversal show detailed data for the four phases of the traversal.

Table IV shows the number of traversal iterations (Iter.) and memory usage (Mem). The last column, Global Data, indicates the size of the largest BDD (Peak) and the maximum memory used (Mem.) during all the phases. Exact image computation is always used except for the learning phase, where (a) and (e) denote approximate or exact image computation, respectively. Memory usages Mem. are reported in Mbytes ([Mb]).

Table V reports the CPU time, in seconds, for the different phases, and the total time (Total).

The Pre-Learning and Learning phases are almost always the cheapest ones. Partial traversal and the following full traversal are more balanced, with performance dominated by either of the two, depending on the subsetting.

Our experiments confirm that the size heuristics generally outperform the time one. We believe this is due to the fact that memory blow-up is the main obstacle for symbolic traversals of large circuits, so memory related statistics are a better measure for performance enhancement in this case. The results obtained with different size heuristics are often comparable. Normalized heuristics are generally more efficient than absolute ones. Concerning the decision on pruning the heavier or the lighter cofactor, our results show an overall balance. We thus avoid giving a-priory preference to either of the two choices.

Table VI compares overall performance of the profile based technique with the best data directly available in the literature. Only the best result for each circuit is presented from each paper. We also present data obtained with a popular publicly available tool, VIS [17] (version 1.3), compiled with the same BDD package (CUDD). In this case we run the experiments on our platform[3].

Data on [1] are not presented because improved by the ones presented in [2]. The differences between the results presented in this paper and the ones in [12] are due to better initial variable reordering (static, obtained by VIS in [12],

[3] A better performance using VIS may exist. For example, we know that circuit $s5378_{179}$ was traversed for the first time using VIS [18] on an Alpha machine with 1 Gbyte main memory, and using 440 Mbyte, and 286485 seconds. Anyway, this unpublished data is not reported in the table.

TABLE II

Activity Profiles Statistics.

Circuit	Recursions/Img			Cache Hits/Img			Size Cost/Img		
	Avg	StD	Max	Avg	StD	Max	Avg	StD	Max
s1269	0.3	0.4	6.6	0.1	0.4	6.1	0.0	0.5	15.9
s1512	5.0	9.6	100.8	1.3	5.1	95.8	0.3	1.7	31.3
s3271	0.3	0.4	3.4	0.1	0.3	2.9	0.1	0.6	16.8
s3330	98.1	302.9	4766.7	18.2	80.2	1556.7	20.1	242.3	10766.7
s3384	19.1	77.1	1563.3	4.0	20.1	554.7	17.8	87.0	2000.1
s4863	41.4	143.5	5900.0	3.9	29.8	1362.0	43.1	412.9	24200.0
$s5378_{121}$	0.2	0.1	1.6	0.0	0.1	1.1	0.1	0.2	2.5
$s5378_{164}$	56.3	426.0	9233.3	12.9	126.5	4300.0	38.5	1640.4	15900.0
$s5378_{179}$	4359.0	17100.0	211000.0	1086.0	5581.0	126000.0	-51.26	13655	228000.0

TABLE III

Transition Relation Statistics. ASPL, ASPH, NSPL, NSPH indicate, respectively, the AbsSizePruneLight, AbsSizePruneHeavy, NormSizePruneLight, and NormSizePruneHeavy heuristics. Time indicates the timing heuristic.

Circuit	Cluster Threshold	# Cluster	Initial Size	Heuristic	Pruned Size
s1269	5000	6	12122	NSPL	12093
s1512	5000	2	4283	NSPH	2032
s3271	500	17	6140	NSPH	5999
s3330	500	16	8039	NSPL	7638
s3384	100	57	4364	ASPH	3599
s4863	5000	37	86137	NSPH	83524
$s5378_{121}$	2000	13	19711	Time	6051
$s5378_{164}$	3000	19	29247	NSPL	28713
$s5378_{179}$	1000	25	11753	ASPL	11556

TABLE IV

Four-step Improved Traversal: Memory Performance.

Circuit	Pre-Learning		Learning		Partial Traversal		Full Traversal		Global Data	
	Iter.	Mem. [Mb]	Iter.	Mem. [Mb]	Iter.	Mem. [Mb]	Iter.	Mem. [Mb]	Peak	Mem. [Mb]
s1269	0	6.1	8 (a)	5.1	4	14.3	9	8.9	64027	14.3
s1512	20	5.2	20 (a)	5.1	1002	9.5	258	7.1	34330	9.5
s3271	5	8.0	10 (a)	5.1	6	15.3	10	43.9	569496	43.9
s3330	2	6.9	3 (e)	11.0	10	25.7	3	44.1	769287	44.1
s3384	3	8.4	3 (e)	7.6	548	94.0	3	16.3	2133339	94.0
s4863	2	13.2	2 (e)	12.2	3	13.4	2	13.5	48050	13.5
$s5378_{121}$	12	12.7	15 (a)	5.2	3	12.5	27	12.6	82295	12.6
$s5378_{164}$	3	13.3	3 (e)	12.3	62	23.3	39	42.8	716295	42.8
$s5378_{179}$	3	19.5	1 (e)	31.2	8	38.6	38	45.1	573646	45.1

TABLE V

Four-step Improved Traversal: Time Performance.

Circuit	CPU Time [sec]				
	Pre-Learning	Learning	Partial Traversal	Full Traversal	Total
s1269	0.0	0.6	8.4	1.9	10.9
s1512	0.4	0.9	114.2	7.2	122.7
s3271	1.1	0.2	54.1	411.9	467.3
s3330	0.4	2.6	127.6	120.6	251.2
s3384	38.2	0.7	94174.9	2007.8	96221.6
s4863	3.8	4.3	10.0	12.8	30.9
$s5378_{121}$	109.6	0.3	13.2	15.3	138.4
$s5378_{164}$	4.5	6.3	1199.3	12124.5	2375.1
$s5378_{179}$	7.1	31.2	1849.6	4697.6	6585.5

TABLE VI

Reachability Analysis Comparison: CPU Time in seconds. *ovf* means time or memory overflow, i.e., data on partial reachability analysis. – means data not reported.

Circuit	CPU Time [sec]								
	Other Approaches							Activity Profiles	
	[17]	[6]	[2]	[3]	[8]	[10]	[9]	[12]	This Paper
s1269	7609	906	290	248	47	–	–	68	11
s1512	818	1426	–	–	2373	–	–	215	123
s3271	*ovf*	*ovf*	–	>18000	–	13065	*ovf*	4983	467
s3330	8843	672	562	647	–	10134	*ovf*	960	251
s3384	*ovf*	–	–	–	–	–	–	–	96222
s4863	5310	*ovf*	–	1982	–	7120	243	528	30
$s5378_{121}$	483	–	575	–	–	–	–	248	138
$s5378_{164}$	*ovf*	–	–	–	–	*ovf*	–	–	2375
$s5378_{179}$	*ovf*	–	–	–	–	–	23400	–	6585

Reference	Machine
[17]	Pentium II, 400 MHz, 128 Mbyte
[6]	DEC Alpha, 250 MHz, 128 Mbyte
[2]	SUN Ultra-1, 1 Gbyte
[3]	DEC Alpha, 500 MHz, 512 Mbyte
	SUN Ultra 2, 250 MHz, 4 Gbyte
	Pentium Pro, 200 MHz, 512 Mbyte
[8]	Pentium II, 400 MHz, 1 Gbyte
[10]	SUN Ultra Sparc, 168 MHz, 128 Mbyte
[9]	SUN Ultra II, 296 MHz
	256 Mbyte for s4863, 512 Mbyte for $s5378_{179}$
[12]	Pentium II, 266 MHz, 128 Mbyte
This Paper	Pentium II, 400 MHz, 128 Mbyte

and already optimized here), extensions and improvements on the pruning heuristics (see Section II-C), and a finer tuning of the overall method. It is worthwhile noticing that also the CPUs used are different: A direct comparison on the CPUs used shows a speed-up ratio for this paper approximately equal to 7/10. Even taking into account this ratio, the improvements obtained are for some cases larger than one order of magnitude.

V. Conclusions

We present a new approach to characterize the involvement, and the impact, of the transition relation within the core operations of reachability analysis, i.e., the relational product operator. Activity profiles are introduced as node by node statistics gathered during a *learning* traversal phase.

Among the possible applications, e.g., decomposition, optimization, and package tuning, we describe guided partial traversals based on transition relation subsetting. Within this framework, we collect activity statistics, we prune the transition relation, we use the pruned result for an initial traversal phase, and we use the result of this phase to complete reachability analysis with the whole transition relation. Algorithms to collect activity statistics and pruning the transition relation are presented. Experimental results show performance gains, in terms of both memory and CPU time, over all the best data present in the literature.

References

[1] K. Ravi and F. Somenzi. High–Density Reachability Analysis. In *Proc. IEEE/ACM ICCAD'95*, pages 154–158, San Jose, California, November 1995.

[2] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and Decomposition of Binary Decision Diagram. In *Proc. EDA/SIGDA/ACM/IEEE DAC'98*, pages 445–450, San Francisco, California, June 1998.

[3] K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proc. IEEE ICCD'99*, pages 467–474, Austin, TX, October 1999.

[4] G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large Finite State Machine. In *Proc. IEEE/ACM ICCAD'96*, pages 354–360, San Jose, California, November 1996.

[5] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive Partitioning and Partial Iterative Squaring: an effective approach for symbolic traversal of large circuits. In *Proc. EDA/SIGDA/ACM/IEEE DAC'97*, pages 728–733, Anaheim, California, June 1997.

[6] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned–ROBDDs. In *Proc. IEEE/ACM ICCAD'97*, pages 388–393, San Jose, California, November 1997.

[7] G. Cabodi, P. Camurati, and S. Quer. Improving the Efficiency of BDD–based Operators by means of Partitioning. *IEEE Transactions on CAD*, 18(5):545–556, May 1999.

[8] K. Ravi and F. Somenzi. Hints to Accelerate Symbolic Traversal. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 250–264, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[9] H. Higuchi and Y. Matsunaga. Dynamic State Variable Removal and Reordering for Efficient State Traversal. Unpublished manuscript, November 1998.

[10] Y. Hong and P. A. Beerel. Symbolic Reachability Analysis of Large Finite State Machines Using Don't Cares. In *Proc. DATE'99*, pages 13–19, March 1999.

[11] M. Ganai and A. Aziz. Efficient Coverage Directed State Space Search. In *IWLS'98: IEEE International Workshop on Logic Synthesis*, Lake Tahoe, California, June 1998.

[12] G. Cabodi, P. Camurati, and S. Quer. Improving Symbolic Traversal by means of Activity Profiles. In *Proc. ACM/IEEE DAC'99*, pages 306–311, June 1999.

[13] H. Cho, G. D. Hatchel, E. Macii, B. Plessier, and F. Somenzi. Al-

gorithms for Approximate FSM Traversal Based on State Space Decomposition. *IEEE Transactions on CAD*, 15(12):1465–1478, December 1996.

[14] F. Somenzi. CUDD: CU Decision Diagram Package – Release 2.3.0. Technical report, Dept. of Electrical and Computer Engineering, University of Colorado, Boulder, Colorado, October 1998.

[15] http://www.polito.it/∼{cabodi,quer}.

[16] H. Cho, G. D. Hatchel, E. Macii, M. Poncino, K. Ravi, and F. Somenzi. Approximate Finite State Machine Traversal: Extensions and New Results. In *IWLS'95: IEEE International Workshop on Logic Synthesis*, Lake Tahoe, California, May 1995.

[17] R. K. Brayton et al. VIS. In *Proc. FMCAD'96, Lecture Notes in Computer Science 1166, Springer Verlag*, pages 248–256, Palo Alto, California, November 1996.

[18] F. Somenzi. Private Communication, December 1996.

Gianpiero Cabodi received the MS degree in EECS in 1984 from Politecnico di Torino and the Ph. D. degree in 1989. He is currently Associate Professor at Dipartimento di Automatica e Informatica of Politecnico di Torino. His research interests include hardware description languages, Binary Decision Diagrams, formal verification, logic and high-level synthesis, testing and ATPG.

Paolo Camurati received the MS degree in EECS in 1984 from Politecnico di Torino and the Ph. D. degree in 1989. He is currently Full Professor at Dipartimento di Automatica e Informatica of Politecnico di Torino. His research interests include hardware description languages, design for testability, testing and ATPG, formal verification.

Stefano Quer received the MS degree in EECS in 1991 from Politecnico di Torino and the Ph. D. degree in 1996. In 1994 he has been with the Department of Electronic Engineering and Computer Science of the University of California in Berkeley; in 1998 he has been with the *Advanced Technology Group*, at Synopsys Inc., Mountain View, California, and in 1999 with the *Alpha Development Group*, at Compaq, Shrewsbury, Massachussetts. He is currently Assistant Professor at Dipartimento di Automatica e Informatica of Politecnico di Torino, and consultant for Compaq Computer Corporation. His research interests include hardware description languages, logic synthesis, formal verification, simulation and testing of digital circuits and systems.