Improving Symbolic Traversals by means of Activity Profiles

Gianpiero Cabodi   Paolo Camurati   Stefano Quer

Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

Abstract

Symbolic techniques have undergone major improvements in the last few years. Nevertheless they are still limited by the size of the involved BDDs, and extending their applicability to larger and real circuits is a key issue.

Within this framework, we introduce "activity profiles" as a novel technique to characterize transition relations. In our methodology a learning phase is used to collect activity measures, related to time and space cost, for each BDD node of the transition relation. We use inexpensive reachability analysis as learning technique, and we operate within inner steps of image computations involving the transition relation and state sets.

The above informations can be used for several purposes. In particular, we present an application of activity profiles in the field of reachability analysis itself. We propose transition relation subsetting and partial traversals of the state transition graph. We show that a sequence of partial traversals is able to complete a reachability analysis problem with smaller memory requirement and improved time performance.

1 Introduction

State-of-the-art approaches for reachability analysis and formal verification of circuits modeled as Finite State Machines (FSMs) exploit symbolic techniques based on Binary Decision Diagrams (BDDs).

Given the transition relation of a system, $\mathsf{TR}(s, x, y)$[1], and a set of states $\mathsf{F}(s)$, the set of states $\mathsf{T}(y)$ reachable in one step from the states in F is computed as

$$\begin{aligned} \mathsf{T}(y) &= \textsc{Image}(\mathsf{TR}(s,x,y), \mathsf{F}(s)) \\ &= \exists_{x,s}(\mathsf{TR}(s,x,y) \cdot \mathsf{F}(s)) \end{aligned} \quad (1)$$

This formula is the core computation of all symbolic reachability analysis problems. For medium-small circuit it is usually quite efficient. Nevertheless it reaches its limits on large practical examples. Several improvements have thus been proposed to the basic idea, in order to deal with realistic circuit sizes.

Transition relations have been represented and used in partitioned forms because of the BDD explosion when building them monolithically. Dynamic variable reordering techniques have been introduced to find good variable orders, with large improvements whenever intermediate results of computations may be individually optimized. Approximate traversals/verifications and abstractions of sub–components are other very popular approaches to scale down the complexity of large problems.

[1] In the sequel we use TR, s, x and y to indicate respectively a transition relation, present state variables, primary inputs, and next state variables.

Recently, many researchers have followed the trend of partial traversals, partitioning the problem and interleaving breadth and depth–first strategies, moving away from the traditional symbolic breadth–first analysis of the state transition graph. The advantages of such approaches are the following: (1) Focusing on a target property or behavior may result in dramatic space and time improvements; (2) even in the case of full traversals, the whole reachable state set is computed through a sequence of partitioned/partial traversals. Since traversals often produce the largest BDDs during intermediate steps, a sequence of simpler traversals is a good way of lowering intermediate peak memory requirements. Several techniques have been proposed to tailor partitioned/partial traversals. Among the other: BDD subsetting and decomposition based on high density [1, 2], over–approximate forward traversals to prune exact backward verification [3], partitioning based on Shannon decomposition [4, 5], manual insertion of *guards* in the hardware description [6].

In this work we propose a novel technique for characterizing the *activity* of recursive operators involving the transition relation. For each BDD node of a given transition relation we *dynamically* and *automatically* derive activity indicators (that we call *activity profiles*). These informations are collected during an initial *learning* phase. We use easy-to-perform reachability analysis as learning phase. The derived *profiles* might be used for several purposes, as an additional information to bias reordering, decompositions, and optimizations.

Among the possible applications, we exploit profiles to optimize reachability analysis in a mix breadth and depth–first analysis framework, using transition relation subsetting. The resulting technique reduces peak memory requirements and execution time of traversals.

This is an important issue today for reachability analysis and verification tools to allow them to reach a larger acceptance beyond the actual advocate community limits.

2 Overview of the Presented Approach

Given an FSM represented by its transition relation TR, we start by gathering TR usage statistics from partial exact or approximate traversals. Statistics are collected for each BDD node of

TR, and they are intended to record the involvement of that node during image computations. We observe recursive calls, operation cache hits, and amount of newly generated BDD nodes. We call activity profile of the TR the set of all the node statistics collected. The memory overhead associated to the node statistics (a few counters) is acceptable and often negligible, since TR nodes are usually a fraction of the BDD nodes generated in reachability, and we use them in low cost learning traversals. Anyway, every application using activity profiles should take into account this overhead.

Given an activity profile, we use it to perform TR subsetting, pruning nodes with a low activity, i.e., unused BDD nodes, or with a high activity, i.e., used too often or involved in expensive operations. The pruned TR is used to compute a subset of the whole reachable state set. Complete reachability is still possible by means of a sequence of partial traversals. This allows scaling down the space/time complexity of FSM traversals.

Similar in its inspiration to other recent works, this technique partially moves away from purely breadth–first traversal of the state transition graph. The method contains original contributions and similarities with known published techniques.

- Programs exploiting *learning* phases (either static or dynamic) are used in several fields of electronic CAD. In the field of reachability analysis, [6] exploits preliminary simulations to bias partial symbolic traversals. In our work, we introduce a new metric to characterize the activity of recursive BDD operators collecting statistics on TR usage within (inexpensive) symbolic traversals. TR activity profiles are collected in an *automatic* way, on a node by node basis, with low time and space overhead. The method can be viewed as a measure of the interaction between the TR and the reachable states through the IMAGE operator, and it could be generalized to other BDD operators as well.
- We follow the most recent trends in FSM reachability analysis: Not merely optimizing image computations, but simplifying the intermediate reachable state sets. Traversals are enhanced by working directly on their core function, the transition relation, as we introduce TR subsetting based on TR profiles computed in preliminary (*learning*) traversal steps. This is a novel "self-tuning" approach to scaling down the complexity of traversals by focusing them on the more "active" subset of TR. Moreover this is the way to isolate sub–behaviors of the FSM: We follow the most (or the least) active ones, given a heuristic evaluation function. Complete reachability is performed through a sequence of partial traversals, with a final one using the whole (non pruned) TR.
- The idea of a partial and focused traversal is presently adopted in many reachability based tasks. BDD subsetting was originally presented by Ravi and Somenzi [1]: They prune reachable state sets following a criterion of maximum density, i.e., the ratio between the number of states and the BDD size. Narayan *et. al.* [5] use window functions to partition (i.e., generate subsets of) a TR. Cabodi *et. al.* [4] use cubes and "idle" latches to generate TR partitions.

  We follow a different approach, since we do not look for high density BDDs, and we do not prune reachable states. Moreover, we do not use externally provided constraints to select proper working subspaces. Our solution is orthogonal and complementary to other approaches, aiming at a more *automated* process, with a learning initial phase biasing the subsequent process. We use a subset of the state transition graph, with benefits in terms of reachable state sets as a consequence. Moreover and more importantly, we exploit automatically generated activity informations directly coming from the traversal process. This is an additional and relevant input, making our pruning process inherently superior than any pruning scheme merely working on the BDD structure of the pruned BDD.

The remainder of the paper is organized as follows. The activity profile of a transition relations model is introduced in Section 3. Section 4 describes transition relation subsetting based on activity profiles, and a traversal approach exploiting this information. Section 5 shows some experimental results. Section 6 closes the paper with conclusions and some indications on possible future work.

3 The Activity Profile of a Transition Relation

We work in a traversal environment where IMAGE (see Equation 1) is the key operation. State–of–the–art approaches implement it as a sequence of relational products, i.e., a sequence of conjunction–quantification operations, applied to a conjunctively partitioned (clustered) transition relation.

We aim at measuring the involvement and the impact of a transition relation in IMAGE operations on a BDD node by node basis. We use a set of activity counters associated with each TR node. We upgrade these counters within the relational products procedure using a *learning* methodology during "initial" IMAGE operations. In particular, given two functions f and g, and a variable v, we observe recursive calls of the relational products operators $\exists_v(f \cdot g)$. During this phase we consider a node as *active* if it produces a non-zero result. We also heuristically take into account the operation cache, which avoids sub–problem recomputation, and the cost of each relational product recursion in terms of newly generated BDD nodes.

More specifically, given the generic BDD node `t` of a transition relation, we define three counters recording the activity on the node:

- `t.act.rec`: Active recursions on `t`.
- `t.act.cacheHits`: Cache hits returning a result different from zero.
- `t.act.sizeCost`: Node increment within the BDD manager due to recursions on t.

The counter of cache hits is an activity indicator for the whole subtree rooted at `t`, that produced the cached results. We heuristically keep a single counter, ignoring here that several cache entries may refer to the same `t` node[†].

The `sizeCost` indicator is related to the amount of BDD nodes generated by recursions on `t`. Since exactly measuring the size of the results produced would require a higher computational overhead, we estimate size cost very efficiently as the overall increase in BDD nodes produced by a recursion, which is easily provided by global counters maintained by BDD packages. This

[†] Different $\exists_v(p \cdot t)$ may refer to the same `t` operand.

is not the size of the result (though often related to it), indeed, it is a good memory cost indicator of the operation performed.

Figure 1 shows our modified relational product operation, handling the previously defined counters. It describes the recursive procedure for the relational product (the RP function) $\exists_v (p \cdot t)$, with activity counters on the `t` factor.

```
RPPROFILE(v, p, t)
    if (terminal case)
        return (result)
    if (result of (RP, v, p, t) is cached)
        **if (result $\neq 0$)**
            **t.act.cacheHits++**
        return (result)
    let $\tau$ be the top variable of p and t
    $r_0 \leftarrow$ RPPROFILE(v, $p_{\overline{\tau}}$, $t_{\overline{\tau}}$)
    if ($\tau \in v$)
        if ($r_0 = 1$)
            $r \leftarrow 1$
            $r_1 \leftarrow 0$
        else
            $r_1 \leftarrow$ RPPROFILE(v, $p_\tau$, $t_\tau$)
            $r \leftarrow r_0 + r_1$
    else
        $r_1 \leftarrow$ RPPROFILE(v, $p_\tau$, $t_\tau$)
        $r \leftarrow$ reduced, unique BDD node for ($\tau$, $r_0$, $r_1$)
    cache the result of this operation
    **if (r $\neq 0$)**
        **t.act.rec++**
    **t.act.sizeCost += (total node increase)**
    return (r)
```

Figure 1: Relational Product with activity counters handling.

Relational product is quite similar to the AND Boolean operator, from which it differs whenever the top variable (τ) must be existentially quantified. This is done through an OR operation ($r_0 + r_1$). Our modifications are indicated by the bold type. The increments of active cache hits and recursion counters are conditioned by a non 0 result. Size cost is finally incremented in the last line before the return command. A negative increment is possible for size cost, whenever a recursion tree produces more released nodes then newly created ones.

Due to the nature of the node–by–node analysis, dynamic reordering is not allowed while gathering statistics: The drawback is not relevant for our application, since we presently collect profiles in learning phases, which are by far cheaper than the overall traversal process. Nonetheless, dynamic reordering could be allowed, with an abort–and–repeat scheme (like for most BDD recursive operators), with profiles created and used across sifting activations.

4 TR Subsetting using Activity Profiles

Given an activity profile, we may use it for TR subsetting, based on the following alternatives:

- Nodes with no or low activity may be pruned, i.e., replaced by the constant 0, the aim being to isolate the most important/active subset of a transition relation
- Nodes with an excessively high activity may be pruned, in order to remove those nodes of a transition relation that are too costly.

In the latter case we avoid removing the whole BDD rooted at the node, rather we clip it by pruning one of its cofactors, following an idea derived from the heavy branch subsetting technique [1]. Two choices are available: Pruning (replacing with 0) either the heavier or the lighter cofactor, where cofactor weight is given by its `sizeCost` indicator.

The process results in a subset of the original TR, with a simpler BDD, since it is derived from the original one, by replacing some of its nodes with the 0 constant[‡].

The recursive pruning procedure is shown in Figure 2. The BDD `t` (a transition relation) is recursively visited depth–first. Subtrees are pruned from `t`, depending on the chosen heuristic (`Recur` or `Size`).

```
PRUNE(t, th, heu, cacheHits)
    if (terminal case)
        return (t)
    cacheHits += t.act.cacheHits
    if ((PRUNE, t, th, heu, cacheHits) is cached)
        return (result)
    let $\tau$ be the top variable of t
    if (heu = Recur) and
            ((t.act.rec+cacheHits) $<$ th))
        return (0)
    if ((heu = Size) and
            (t.act.sizeCost $>$ th))
        if (RECURONTHENCOFACTOR (t))
            $r_0 \leftarrow 0$
            $r_1 \leftarrow$ PRUNE($t_\tau$, th, heu, cacheHits)
        else
            $r_0 \leftarrow$ PRUNE($t_{\overline{\tau}}$, th, heu, cacheHits)
            $r_1 \leftarrow 0$
    else
        $r_0 \leftarrow$ PRUNE($t_{\overline{\tau}}$, th, heu, cacheHits)
        $r_1 \leftarrow$ PRUNE($t_\tau$, th, heu, cacheHits)
    $r \leftarrow$ reduced, unique BDD node for ($\tau$, $r_0$, $r_1$)
    cache the result of this operation
    return (r)
```

Figure 2: BDD pruning based on activity profiles.

More specifically, with the `Recur` heuristic we aim at pruning BDD nodes with no or low activity. We use for this purpose the number of recursions traversing a node, augmented with the number of cache hits upper in the recursion tree. This is the reason why the cacheHits parameter is passed and incremented with the cache hits recorded in the present `t` node. In the second case (`Size`), we clip nodes that show a large size cost. The goal here is finding nodes of TR that produce BDD explosion (i.e., high increments in overall node amount) within image computations. Clipping of t is operated by recurring on one of the two cofactors, while forcing a 0 result for the other one. Cofactor selection is expressed by function RECURONTHENCOFACTOR (Figure 2), which takes a decision after comparing the size costs of the (then and else) cofactor root nodes. We allow either chosing the heavier or the lighter cofactor, since we experimentally found both choices valid, depending on the circuit.

[‡]This is not always the case for BDDs with complemented edges, where different profiles (and pruning) might be produced by a BDD and its complement

Going to the usage of pruned TRs, we propose a straightforward application of partial traversals. A full traversal is achieved through the following steps:

- *Learning* traversal & TR *pruning*: A limited number of fast exact or approximate traversal iterations is done to gather an activity profile. A subset TR_p of the transition relation is computed using function PRUNEand the activity profile.
- *Partial* traversal: A traversal which uses TR_p and produces a reachable state set R_p, which is a subset of the full one (R).
- *Full* traversal: A final traversal that computes R starting from R_p and uses (the original) TR.

The *learning* traversal is much cheaper than the other ones. 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, which starts from the result of the previous one, has a lower cost, compared with a standard traversal.

Of course, the scheme proposed is 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 TR subsetting.

5 Experimental Results

The presented technique is implemented within a traversal program built on top of the Colorado University Decision Diagram (CUDD) package [7]. Activity data are generated dynamically and associated to BDD nodes by means of hashing. This is due to the fact that BDD nodes in CUDD have no extra field available for user defined data. Our experiments ran on a 266 MHz Pentium II Workstation with a 256 Mbyte main memory, runnning RedHat Linux 5.2. Experiments were done with a 128 Mbyte memory limit. We present data on a few ISCAS'89 and ISCAS'89–addendum benchmarks, selected with different sizes, within the range of circuits manageable by state–of–the–art reachability techniques. Both the software and an updated set of experiments are available through the authors' home pages [8].

To allow a fair comparison with state–of–the–art traversals, we compare our tool (without and with the optimizations presented here) with a popular publicly available tool, VIS [9] (version 1.3), compiled with the same BDD package (CUDD). The initial static variable ordering is the one generated by VIS (default settings) for the BDDs of the circuit. For all the experiments we experimentally determine proper cluster size thresholds (generally in the range from 100 to 5000 nodes). We enable dynamic reordering (group sifting with groups made up of corresponding present/next state variables) with the default settings of the CUDD package.

Table 1 collects statistics on the transition relations of the circuits.

It shows number of primary inputs (# I) and latches (# L) of the circuits. The sequential depth Depth of the circuit, and the final number of states States are also reported. For all circuits we first computed the BDDs of the next state functions, then we generated a clustered transition relation. Column # Cl indicates the number of clusters of the TR and column |TR| reports its BDD size in terms of BDD nodes.

We present two sets of experimental data. The first one describes the activity profiles collected. The second one compares standard reachability with the three–phase traversal using TR pruning (see Section 4).

Table 2 shows some statistics concerning the activity profiles of transition relations.

Data are recorded through the learning traversals described in the sequel. We present average, standard deviation and maximum values for active recursions, cache hits and size cost. These values are reported normalized to respect to the number of image computations performed in each case. Concerning the profiles, they roughly show dense distributions around the average values, with high maximum attained by a few nodes.

Optimized traversals, using the three–step sequence *Learning*, *Partial* and *Full* traversal, are described in Tables 3 and 4.

Memory usage Mem. is reported in Mbytes ([Mb]) and CPU time in seconds.

Table 3 compares overall performance of the profile based technique with our standard breadth–first implementation and VIS.

We report peak BDD size for intermediate products within image computations, memory, and execution time. Our standard traversal performance is very similar to VIS, with slight differences on cluster sorting, which explain different performance. Overall execution times are dominated by sifting (overall sifting time is reported between brackets), due to the choice done for our experiments. We believe this is the most common practice, although we are aware of better performance in some cases, working with optimal settings (cache size, optimum variable ordering, sifting disabled, etc).

We omit here circuits that we are not able to traverse with any technique. Our three–step traversal largely outperforms the standard one in terms of memory and of time in nearly all cases. It is also able to complete the traversal of s3271, that is aborted by standard traversals for memory limits. In the case of circuit $\mathsf{s5378}_{opt}$, we have a higher BDD node peak, but this was due to the first iteration of the full traversal, computing the image of the whole reachable state set resulting from the partial traversal. All other iteration had lower peaks.

A more detailed description of the partial traversal approach is shown in Table 4.

Iter. columns show the number of traversal iterations (image computations) of the different phases. Here we show more detailed statistics for all the three phases. The Learning phase is always the cheapest one. In the Iter. column, (a) and (e) indicate using approximate or exact reachability analysis, respectively. Partial traversal and the following full traversal are more balanced, with performance dominated by either of the two, depending on the TR subsetting done. It is also interesting to observe that the `Size` heuristic was better than the `Recur` one on all cases but one. We believe this is partly due to the fact that memory statistics are a better measure for performance of traversals, and partly to the looser efforts we made with the `Recur` heuristic.

6 Conclusions

We present a new approach to characterize the involvement and the impact of transition relations within the core operations of

Circuit	# I	# L	Depth	States	# Cl	$\lvert TR \rvert$
s1269	18	37	9	$1.13{\cdot}10^{9}$	19	14177
s1512	29	57	1023	$1.66{\cdot}10^{12}$	2	6230
s3271	26	116	16	$1.31{\cdot}10^{31}$	8	15554
s3330	40	132	7	$7.28{\cdot}10^{17}$	9	14537
s4863	43	183	4	$2.19{\cdot}10^{19}$	50	58601
s5378$_{opt}$	35	121	42	$2.58{\cdot}10^{17}$	5	9057

Table 1: Circuit and Transition Relation Statistics.

Circuit	Recursions/Img			Cache Hits/Img			Size Cost/Img		
	Avg	StD	Max	Avg	StD	Max	Avg	StD	Max
s1269	0.55	0.89	11.44	0.22	0.69	16.89	0.06	1.92	119.89
s1512	4.96	9.60	101.25	1.25	5.15	95.95	0.25	1.70	31.30
s3271	28.22	91.75	1563.45	7.43	33.71	791.18	6.76	272.32	2314.32
s3330	12.00	17.40	133.02	0.44	1.34	16.32	14.66	145.43	7848.54
s4863	129.43	464.54	5657.54	14.50	171.54	14344.54	9.43	140.54	8772.43
s5378$_{opt}$	0.23	0.31	2.43	0.29	0.11	1.67	0.72	2.21	3.83

Table 2: Activity Profiles.

reachability analysis. Activity profiles are introduced as node by node statistics gathered during *learning* traversal steps.

Among the possible applications, e.g., reordering, decomposition and optimization, we describe partial traversals based on transition relation subsetting. Experimental results show performance gains in terms of memory requirement and CPU time.

Future works will investigate more sophisticated techniques to collect and use profiles. Relation with dynamic reordering has to be better investigated. Moreover, dynamic learning is an interesting path to pursue. Whereas a few circuit have a quite uniform profile, other have a very different behavior during reachability analysis. One way to cope with that is to apply the learning phase more than once during reachability analysis, e.g., between image computations or sifting activations. Moreover, we want to explore other activity indicators and pruning heuristics, as well as more sophisticated partial traversal, i.e., multi-phase traversal as opposite as two-phase. In parallel, one target is to investigate further applications, e.g., other BDD operators, variable ordering, package tuning.

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] G. Cabodi, P. Camurati, and S. Quer. Efficient State Space Pruning in Symbolic Backward Traversal. In *Proc. IEEE ICCD'94*, pages 230–235, Cambridge, Massachussetts, October 1994.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

Circuit	VIS			Original Method			Activity Profiles Method		
	Peak	Mem. [Mb]	Time [sec]	Peak	Mem. [Mb]	Time [sec]	Peak	Mem. [Mb]	Time [sec]
s1269	1287619	60.7	7609 (7151)	1654441	71.4	3892 (3468)	69615	11.5	68 (39)
s1512	56177	13.2	818 (166)	40058	19.5	1649 (83)	12218	9.6	215 (110)
s3271	$-$	$-$	$-$	$-$	$-$	$-$	610185	56.1	4983 (3521)
s3330	514839	47.5	8843 (7249)	1368358	94.2	16934 (9172)	30904	15.5	960 (720)
s4863	701506	49	5310 (4256)	805165	55	4810 (4341)	52335	181.5	528 (430)
s5378$_{opt}$	53951	13.6	483 (124)	62381	15.6	703 (529)	82295	13.4	248 (134)

Table 3: Reachability Analysis Comparison. $-$ means memory overflow.

Circuit	Heuristic	Learning			Partial Traversal			Full Traversal		
		Iter.	Mem. [Mb]	Time [sec]	Iter.	Mem. [Mb]	Time [sec]	Iter.	Mem. [Mb]	Time [sec]
s1269	Size(Heavy)	10 (a)	4.9	1	5	11.5	60	8	9.3	7
s1512	Size(Light)	20 (a)	5.1	1	1023	9.6	204	259	8.2	11
s3271	Size(Light)	5 (e)	18.2	12	7	20.1	164	12	56.0	4807
s3330	Size(Heavy)	1 (e)	5.2	0	12	13.2	140	6	15.5	820
s4863	Size(Light)	2 (e)	12.2	8	3	13.4	431	3	181.8	89
s5378$_{opt}$	Recursion	16 (a)	11.3	3	16	12.1	207	28	13.2	25

Table 4: Three-step Improved Traversal.