二分決定木を用いた論理関数の質問処理

バレリー・ビャトキン 中野浩嗣 林達也
名古屋工業大学電気情報工学科

本論文では、複数の入力が同時に変化する場合に、論理関数をオンライン評価する手法について議論する。このオンライン評価法は、離散制御システムにおいてイベントに対する高速な処理に用いることができ、また、制御装置の応答時間を改善することができる。

Logic Evaluations as Processing of Queries using Binary Decision Diagrams

Valery Viatkin, Koji Nakano, Tatsuya Hayashi
Dept. of Electrical and Computer Engineering
Nagoya Institute of Technology
Showa-ku, Nagoya 466, JAPAN
{valery, nakano, hayashi}@elcom.nitech.ac.jp

The paper presents approach to evaluation of logic expressions in on-line execution of control algorithms for general case when arbitrary number of input variables is changing value. This approach essentially uses event-oriented nature of discrete control systems and allows to improve productivity of the computation and, correspondingly, response time of a controller. Given results are to be implemented for fast on-line logic processing in distributed control systems and other computer applications including real-time logic processing.

1 Introduction

Optimizing of logic computations is especially important in discrete control systems because they include a big deal of such a computations and interact with dynamic objects in real time. Hence, faster logic has computed, faster is response of the controller and wider its abilities. There are two general ways to improve an efficiency of such computations.

First one optimizes processing of logic using fast data presentations. Among most effective are binary decision diagrams (BDD) which guarantees that computation be completed and output generated in $O(n)$ time, where n is the number of variables included in computed Boolean expression. This is especially significant in the case when the model comprises the numerous functions whereas prior direct methods of formula interpretation computed the result in $O(\sum L(f_i))$ time, proportional to the sum of all the formula length [1]. Use of reduced and ordered BDDs (OBDD) proposed in [2] even more decreases memory requirements from exponential to polynomial for a big class of Boolean functions.

The other optimization approach regards to supervising interface within which the logic processing is operated. Usual cyclic interface of logic controllers is based upon the update of inputs and outputs between phases of model evaluation. Alternative routine is oriented on events, which activates the part of the model dependent on the given event. As event it is assumed the fact of a value change by certain input variable. Even a simple application of this principle joined with any particular technique of logic processing may considerably reduce required computations [3], [5].

The main contribution of this paper is development of algorithms that combine both these methods for general and more complicated case when event is tuple of inputs changed values simultaneously. Obviously, applying simple BDD traverse the result may be derived in $O(n)$ time for any length of the tuple. Our aim is to improve this results evolving ideas of event-driven BDD traverse from [4].

Two approaches to improve the computation efficiency are studied in the paper. The first one stated in Section 3 uses the property of discrete control systems which are idle during monitoring the same inputs in intervals between events. This time can be used to prepare some data to improve response time of the controller for upcoming events. Along this it is better to compute the function for as many input combinations as the time allows. These massive computation done with proposed data model and algorithms gives a favor of using some intermediate data left in the model by another passes. So total computation time is less than that of the sum of passes done independently.

Second part of the paper (sections 4,5)is devoted to optimization of immediate OBDD traverse to compute the function with certain data given after an event. Some modification of OBDD are proposed to derive the result in $O(k)$ time at best, where k is number of arguments, changed by event. All these modifications implement certain trade of memory to speed as well as all require some pre-evaluation - preparatory evaluations of some parameters used to compute result fast.

2 Evaluation and re-evaluation of Boolean function

Consider a Boolean function $f : \{0,1\}^n \to \{0,1\}$. For the initial input vector $A =< a_1, a_2, ...a_n >$, the value of f is $f(A)$. Let us consider an event, represented by $k-$tuple of indices $\sigma = (j_1, j_2, ..., j_k)$ $(1 \leq j_1 \leq j_2 \leq ... \leq j_k)$. This event means that j_i-th $(1 \leq i \leq k)$ argument a_{j_i} is changed to $\overline{a_{j_i}}$. We call k as a rank of event $\sigma : r(\sigma) = k$. We also term sub-array $< a_j, ...a_n >$ as $A_{j:n}$ and sub-event $(j_p, ..., j_k)$ as $\sigma_{p:k}$.

Let $\sigma(A)$ denote argument vector after the event and $\Sigma = ||\xi_j||_{,\overline{j=1,n}}$ is Boolean vector such that only those $\xi_{j_k} = 1$ which indices included in event: $j_k \in \sigma$. Then the value of input vector after σ could be derived as $\sigma(A) = A \oplus \Sigma$.

The *re-evaluation* problem is to compute value of $f(\sigma(A))$. An event can be considered as query to be processed by the model having the function f definition and current argument vector A.

2.1 Logic computations using binary decision diagrams

A BDD (Binary Decision Diagram) is a labeled directed acyclic graph $G =< V, E, ind, l >$ which represents a Boolean function f :

1. V is a set of nodes: exactly two nodes in V are termed *terminal nodes*. One of them is $0-node$, and the other is $1-node$. $value : V \to \{0,1\}$ is a marking assigning a Boolean value to a terminal node. If v is 0−node then $value(v) = 0$ or $value(v) = 1$ correspondingly. All other nodes are termed *variable nodes*. One of variable nodes is *root*.

2. E is a set of edges (i.e. ordered pairs of nodes) defined as follows: Each terminal node has one in-coming edge and no outgoing edges. Each variable node v has exactly two out-going edges $(v, hi(v))$ and $(v, lo(v))$. The root node has no in-coming edges and the other variable nodes have one or more incoming edges.

3. $ind : V \to [1, n+1]$ gives a number label to each node and $l : E \to [0,1]$ gives a Boolean marking to edges. Each variable node u is labeled by an integer in the range $[1, n]$. This means, that a variable $x_{ind(u)}$ is assigned to u. Terminal nodes have $ind(u) = n+1$. For each variable node u one of the two out-going edges is labeled by 0 and the other is labeled by 1. Node v, such that $l(u,v) = 0$ is called $lo(u)$ and such that $l(u,v) = 1$ is $hi(u)$.

The size of BDD is defined by the number of nodes in V denoted as $|V|$.

An *OBDD* (Ordered BDD) is a BDD such that for all directed edges $(v,u) \in E$ the inequality $ind(v) < ind(u)$ holds. Every node of OBDD is a root of some subgraph, which also is OBDD [2]. Given OBDD $G =< V, E, ind, l >$ such a sub-OBDD rooted in node $v(\in V)$ is termed as G_v. Correspondingly it defines Boolean function f_v as:

1. If v is either 0−node or 1−node, then $f_v = value(v)$ respectively.

2. If v is a variable node with two outgoing edges (v, v_0) and (v, v_1), then $f_v = \overline{x_{ind(v)}} f_{lo(v)} \vee x_{ind(v)} f_{hi(v)}$.

Functions $f_{lo(v)}$ and $f_{hi(v)}$ are termed *residual*. As it was shown in [2] OBDD may be reduced to be free of isomorphic subgraphs. Such reduced OBDD is canonical form of Boolean function definition. Thus it might be used as branch program to derive function result in $O(n)$ time with memory consumption that is often rather polynomial than exponential.

2.2 Interpretation of BDD and events

To describe formal way of OBDD interpretat ion let function $Step(v, x_{ind(v)})$ return a successor of v:
$Step(v, x_{ind(v)}) = hi(v)$ if $x_{ind(v)} = 1$ or $Step(v, x_{ind(v)}) = lo(v)$ if $x_{ind(v)} = 0$.

Given an input vector A the value of function f_w, defined by BDD rooted in w:
$f_w(x_{ind(w)}, ..., x_n) = val(w)$ if w is terminal or $f_w(x_{ind(w)}, ..., x_n) = f_{Step(w, x_{ind(w)})}(x_{ind(Step(w, x_{ind(w)}))}, ..., x_n)$.

Nodes $v_1 = Step(v, x_1)$, $v_2 = Step(v_1, x_{ind(v_1)})$, ..., $v_t = Step(v_{t-1}, x_{ind(v_{t-1})})$, where $ind(v_t) = n+1$, form a list $I_v(A) =< v, v_1, v_2, ..., v_t >$: $|I_v(A)| \leq n+1$ since $ind(v_j) < ind(v_{j+1}) < n+1$. This list is termed as *interpretation path* of OBDD G_v with current values A. Deriving the function result causes recursive computation of $Step(v_i, x_{ind(v_i)})$ in the nodes of $I_v(A)$. Interpretation path started from the root of G is termed just $I(A)$.

For events of 1-st rank the following lemma is valid:

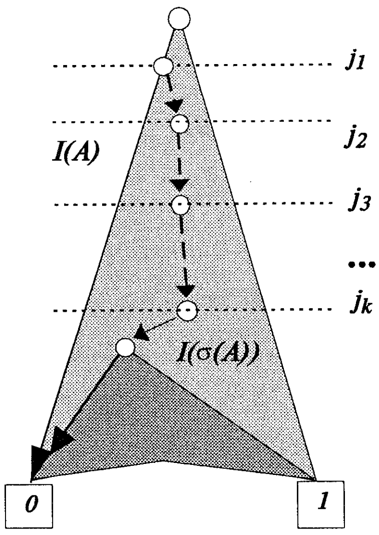

Figure 1: Interpretation changed by event

Lemma 2.1 *Given event $\sigma =< j_1 >$ and $I(A)$ the value of α_σ can be computed as:*

1. *if $\exists v \in I(A) : ind(v) = j_1$ then $\alpha_\sigma = f_{lo(v)}(A) \oplus f_{hi(v)}(A)$;*
2. *Otherwise $\alpha_\sigma = 0$;*

Proof: As far as $\alpha_\sigma = f(A) \oplus f(\sigma(A)) = [\overline{a_{j_1}} f_{lo(v)}(A) \vee a_{j_1} f_{hi(v)}(A)] \oplus [\overline{x_{j_1}} f_{lo(v)}(\sigma(A)) \vee x_{j_1} f_{hi(v)}(\sigma(A))]$. As far as $x_{j_1} = \overline{a_{j_1}}$ and $f_{lo(v)}(\sigma(A)) = f_{lo(v)}(A); f_{hi(v)}(\sigma(A)) = f_{hi(v)}(A)$ because both residual functions do not depend on x_j we have $\alpha_\sigma = [\overline{a_{j_1}} f_{lo(v)}(A) \vee a_{j_1} f_{hi(v)}] \oplus [a_{j_1} f_{lo(w)}(A) \vee \overline{a_{j_1}} f_{hi(w)}(A)] = f_{lo(w)}(A) \oplus f_{hi(w)}(A)$.

If $I(A)$ does not contain a vertex of index j_1, then $I(A) = I(\sigma(A))$ and $f(A) = f(\sigma(A))$, so that $\alpha_\sigma = f(A) \oplus f(\sigma(A)) = 0$. ■

Using the lemma 2.1 the following property of OBDD traverse for an arbitrary event can be proved:

Theorem 2.2 *Given event $\sigma =< j_1, j_2, ..., j_k >$ and $I(A) = \{u_1, u_2, ..., u_{t_A}\}$*

1. *if $\exists q = \min(p) : (j_p \in \sigma) \& (u \in I(A) : ind(u) = j_q)$ then $\alpha_\sigma = f(A) \oplus f_{Step(u,\overline{a_{j_q}})}(\sigma_{q+1:k}(A_{ind[Step(u,\overline{a_{j_q}})]:n}))$;*
2. *Otherwise $\alpha_\sigma = 0$;*

Interpretation $I(\sigma(A))$of OBDD with new input data combination after event σ differs from $I(A)$ in no more than k recursive applications of procedure *Step*, which are done with inverted operand $\overline{a_{j_i}}$ as this shown at the Fig.1

3 Boolean pre-evaluations using dependent traverses of a BDD

It is possible to improve an efficiency of a control system we increasing its loadness in an idle time intervals. We propose to use this time for computation the control logic function in advance for certain set of probable events. This action is termed as *pre-evaluations* or *pre-computations.*

Instead of the function's value it is possible also to pre-evaluate *sensitivity function α_σ* that is a Boolean function expressing ability of f to change the value as a result of event σ : $\alpha_\sigma = f(A) \oplus f(\sigma(A))$. Consequently $f(\sigma(A))$ can be expressed as $f(\sigma(A)) = f(A) \oplus \alpha_\sigma$. So that if the value of $\alpha_{\sigma,}$ had been computed for the given set of events it would be possible to reevaluate the function at constant time $O(1)$.

Every node of OBDD is root of some sub-OBDD [2]. So that it denotes some Boolean function. As *full evaluation* it is termed the evaluation of all this functions with assigning value to corresponding nodes. This value computed for particular node and stored as its parameter we denote as $mark(v)$. Then we prove that:

Lemma 3.1 *Full-evaluation of $mark(v)$ for all $v \in V$ is done in $O(|V|)$ time.*

Let us $mark_A(v)$ has three values - 0,1 and U (unknown). Assume that Initially $mark_A(v) =$ "U" for all non-terminal nodes and $mark_A(v) =$ "U". The following algorithm computes $mark$ in all nodes:

ALGORITHM 1 (FULL-BDD)

```
function Step(v:node; X: Boolean):node;
begin
    if terminal(v) then return(v)
    else if X=1 then return(v.hi)
        else return(v.lo)
end;
function Traverse(v: node): Boolean;
begin
    if mark(v)="U" then begin
        mark(v)=Traverse(Step(v,X[ind(v)]));
        return(mark(v))
    end
    else return(mark(v))
end;
procedure FullEvaluation;
begin
```

```
foreach node v in V do
    Traverse(v)
end.
```

The set of all possible events of certain rank k we denote as Δ^k and $\aleph^k = \bigcup_{i=1}^{k} \Delta^k$. Total cardinality of this set is obviously $|\aleph^k| = \sum_{i=1}^{k} \frac{(n-1)^i}{i!} = O(n^{k+1})$. Further we consider problem of pre-evaluation of the function's value for all the arguments of $\aleph^k$ that is termed as $f(\aleph^k)$. Straightforward estimation of this computation complexity termed as $C(n,k)$ gives $C(n,k) \leq n\,|\aleph^k| = O(n^{k+2})$. Here it is assumed that evaluation of every data combination takes only n steps as if BDD traverse is accepted as way of function's computation.

Assume that the function is defined by BDD but computations done not independently that reduces complexity of computations. Consider subset $\aleph^k(i,j) \subset \aleph^k$ such that $\forall\sigma =< j_1, j_2, ..., j_p >\in \aleph(i,l) \Leftrightarrow j_i = l$; Then $\aleph^k = \bigcup_{l=1}^{n} \aleph^k(1,l)$. Given particular value of j_1 and $v \in I(A) : ind(v) = j_1$ pre-evaluation of the function f could be reduced to that of the $f_{Step(v,\overline{x_{j_1}})}$ for events $< j_2, ..., j_p >$ with rank decreased by 1. So that complexity of computations of $f(\aleph^k)$ is:

$$\begin{aligned} C\,(n,k) &= n + C(n-1,k-1)+ \\ &+C(n-2,k-1) + ... + C(k,k-1)+ \\ &+C(k-1,k-1) + ... + C(1,1). \end{aligned} \tag{1}$$

(There are n opportunities to select first event argument and transition from the correspondent node v to its idle successor by procedure $Step(v,\overline{x_{j_1}})$ takes one step, so that all the reduction adds n steps to the complexity expression plus recurrent references to computation complexities of functions denoted by BDDs rooted in $Step(v_{j_1}, \overline{x_{j_1}}), \overline{j_1{=}1,n})$.

Having this recurrent dependency reduced we eventually derive:

$$\begin{aligned} C(n,k) &= \sum_{t=1}^{k} \sum_{i=1}^{n-k} \alpha_{t-1}^{i}(n-t-i+1)+ \\ &+ \sum_{i=1}^{k} \alpha_k^i C(n-k-i+1,0)+ \\ &+ \sum_{i=1}^{k} C(k-i,k-i) \sum_{t=1}^{i} \alpha_t^{n-k}, \end{aligned} \tag{2}$$

where coefficient $\alpha_t^i = \sum_{p=1}^{i} \alpha_{t-1}^{p} = \alpha_{t-1}^{i} + \alpha_t^{i-1} = O(i^{t-1})$.

Thus, the first term of 2 can be estimated as $O(n^{k+1})$. As far as $C(p,p)$ is complexity of a BDD interpretation when all the arguments are changed:$n = k$ then it requires full traverse of the corresponding BDD that means $C(p,p) = p$. So that the third term might be transformed to $\sum_{i=1}^{k}(k-i)\sum_{t=1}^{i}\alpha_t^{n-k}$ and also be estimated as $O(n^{k+1})$. The second term contains $C(p,0)$ which terms complexity of BDD traverse with absence of any event that normally requires $O(p)$ steps of interpretation. Considering particular event processing this interpretation to be done after last event index j_k is passed. The total value of the second term does not exceed the complexity of full BDD evaluation. Thus: $C(n,k) = O(n^{k+1}) + O(|V|) \leq O(n^{k+2})$. Thus, if $|V| \leq O(n^{k+1})$ then dependent evaluation technique is chosen and result is $C(n,k) = O(n^{k+1})$ else independent traverse are more preferable and $C(n,k) = O(n^{k+2})$.

Recursive algorithm 2(MEP - Massive Event Pre-evaluation) computes values $f(\aleph^k)$ of the function defined by OBDD rooted in v. As v the root of OBDD is termed to which the algorithm is applied. Procedure *store(ev : event, value : Boolean)* used for storage of the results of pre-evaluation in proper place like k-dimensional table.

Algorithm 2

```
procedure doMEP(v: node; ev: event);
begin
    if terminal(v) then store(ev, v.value)
    else if ev[0]=k then begin
            v.mark=Traverse(v);
            store(ev, v.mark)
        end
        else begin {ev[0] < k}
            w=v;
            repeat
                w1=Step(w, not X[ind(w)]);
                doMEP(w1,ev+ind(w));
            until terminal(w);
            if ev[0] > 0 then store(ev,w.value);
        end
end;
```

Additional memory is required only for storage of the k-dimensional table that could be as large as actual memory size admits. Thus we had shown how using OBDD can considerably reduce massive preevaluations of Boolean functions. The last notion to this method is idea of progressive order of pre-evaluations. In every state there are could be specified set of most probable events, for which the function to be pre-evaluated at first. As

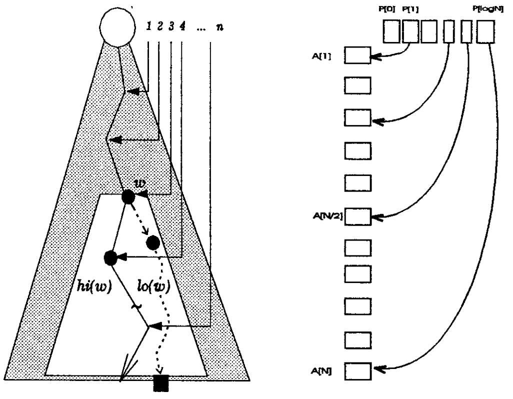

Figure 2: Pointers to all the vertices of PCI stored in each vertex and optimized pointer pattern

far as the probability of the event, in general, is decreasing with its rank, following pre-computations to be done with growing rank while time and memory allows.

4 Optimized BDD traverse

The drawback of above stated approach is that its effect shows itself in massive computations. Further we consider ways optimizing traverse of BDD in order to evaluate particular event.

4.1 Dynamical path pointing

Assume the function $p(v, m)$ returns pointer to node w with index m if such a node is contained in interpretation path starting from v or *nil* otherwise.

```
ALGORITHM 3
function p(v,m):*node;
begin
    w := v;
    while ind(w) < m do w:=Step(w,X[ind(w)]);
    if ind(w)=m then return(w)
    else return (nil)
end;
```

In this case handling of k-rank event could be done as $f = pass(root, 1)$, where function *pass* is defined recursevly as:

```
ALGORITHM 4
function pass (v,cnt):Boolean;
begin
    if (terminal(v)) return(v.value)
    else
        while ((cnt ≤ k) and
            ((ev[cnt] < index(v)) or (p(v, cnt)=nil)
                do inc(cnt);
        if (cnt > k) then return (*p(v,n+1).value)
        /* way to terminal from the last event */
        else
        return(pass(Step(v,not X[ind(v)]),cnt+1));
end;
```

If we keep all the values of the function p stored as array $P[1 : n]$ of pointers for every node then access to p takes only one step. Totally the procedure requires to pass $2k + 1$ vertex and to keep up to $O(n\,|V|)$ of additional pointers. However, recalculation time is spent for update the array of pointers in every node that has active path to those nodes, which have the outgoing path changed as a result of event, i.e. in nodes of $P(t + 1)$ with indices $j_1, j_2, ...j_k$. This leads to pass as many as all vertices of the BDD. Thus pre-evaluation time is estimated as $O(|V|)$.

This ratio could be changed to suit some requirements with applying of half-dividing technique. The optimization allows to store only $\log_2(i)$ of pointers in a node with index i (Fig.2). In this case each traverse between two consequent events takes as many as $\log_2(n)$ steps but response time increases to $k \cdot \log_2(n)$.

4.2 Static pointer lattice in BDD

An idea of static lattice we illustrate at first in the case of queries processing in one dimensional directed and ordered array. To reach cell with certain number in the linear structure with single pointer $p(A[i]) = A[i+1]$ to the neighbor takes as many as N steps. In fact, $A[j] = p(A[j-1]) = p(p(A[j-2])) = p(p(...p(A[1]))) = p_n(A[1])$ if we assign to the every cell access function $a(cell, index)$ returning pointer to the cell with certain index then in the linear structure $a(A[i], index) = a(A[i+1], index)$ $\forall i < index$.

Employing the technique of successive approximations with half dividing we split the array on two segments of equal size. First cell of the array has pointers to itself, to the middle and to the end of array ordered with index of referred cell. Each of two segments obtained as a result of half-dividing is treated with the same way. Procedure of half-dividing is applied until the length of segments is equal to 1. As a result every cell keeps direct pointer to all the cells which are the other edges of segments starting in it and whole the structure of references we obtain illustrated at Fig.3.

Every node v is assigned with array of pointers $P_v[0 : l_v]$, where $P_v[0] = l_v \leq \log_2 N$ -length of the array, $P[1] =^* v$ - pointer to the cell itself, and $P[l_v]$ is pointer to the last cell of segment. $P[P[0]-$

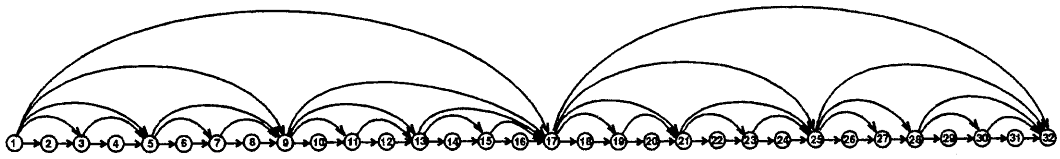

Figure 3: Accelerating access to array using pointers to half-divided sections

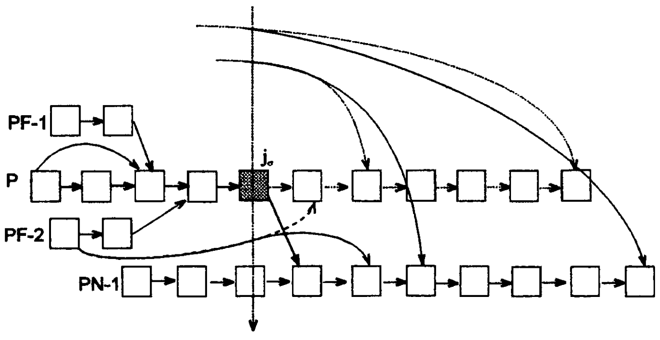

Figure 4: Modification of static pointers after event

1] refers to the middle of segment, $P[P[0]-2]$ to the middle of segment $[P[1], P[P[0]-1]]$ and so on. These pointers accelerate evaluation of the access function from N to $\log_2(N)$ steps at worst.

If we have a query defined as k - indices $j_1 \leq j_2 \leq ... \leq j_k$ its processing is consequent access to values of array cells with indices $j_1, ... j_k$. Its processing can be estimated as $O((k+1)\log_2[\frac{n}{k+1}])$. (The proof of this is almost obvious: Pass between two consequent cells $A[j_i], A[j_{i+1}]$ in the given structure can be estimated as $\log_2(j_{i+1} - j_i)$. Then the total duration of access is proportional to $\log_2 j_1 + \log_2(j_2 - j_1) + ... + \log_2(n - j_k) = \log_2[j_1(j_2 - j_1)...(j_n - j_k)]$. This can be upper bounded by $\log_2\left[\frac{j_1+(j_2-j_1)+...+(j_n-j_k)}{k+1}\right]^{k+1} = (k+1)\log_2(\frac{n}{k+1})$).

Traverse of OBDD PCI is like to above considered pass of linear array. The distinctive feature of PCI is that vertices of all consecutive indices not necessary included in the path. For that purpose let us consider rarefied array where indices are ordered but not consequent. Nevertheless, references are remained according the same idea as considered above. If any index is not presented in rarefied array, corresponding reference points to the first existed cell with larger index. Similarly for outgoing references - its start is assigned to next existed cell. Obviously, those references which started and pointed inside segment of consequent skipped indices are not being taken in account.

In the any cell of array except the first and the last, number of outgoing references coincides with the number of incoming ones. Consequently, for any index the total number of references in array, which points further it (outgoing and jumping over) is constant and equal to $\log_2 N$. That means in the rarefied array maximal number of outgoing references for every cell does not exceed $\log_2 N$ because those are formed as cell's own outgoing references and ones jumping over it from the skipped segment.

Theorem 4.1 *Number of nodes in processing a rarefied array is less or equal to that of consistent one.*

Thus consider procedure of the paths reordering that to be implemented after all required vertices had been passed. Consider any particular event with index j_σ. Let the path of current interpretation is P, and in node with index j_σ it switches to path PN as this shown at Fig.4. There are also some paths $PF1, PF2, ...$ which falls in P before j_σ and some others that falls in P after j_σ. It is required to redirect pointers from $P, PF1, PF2, ...$ that pass over j_σ from nodes of P to nodes of PN. To do that it is required to pass all the nodes of $OBDD$ and check whether they have such pointers. Then we find corresponding vertices of PN with the same or larger indices and change pointers to them. Thus, the complexity of this path modification also estimated as $O(\log_2 n\,|V|)$. In order to improve this estimation we outline subset of nodes $V' \in V$ termed as *critical nodes* - having more than one incoming edge. If these nodes store a number of pointers for all incoming edges then the pass of OBDD to modify pointers can be done in $O(\log_2 n\,|V'|)$. ■

5 Event decision diagrams

Event Decision Diagram (EDD) has a structure based on structure of BDDs. It is built for particular BDD in current state A of the input vector. EDD is a graph, having the same set of nodes V as parent BDD. EDD has the following structure:

1. Each variable node v in V has $n - ind(v) + 1$ outgoing edges uniquely labeled by $ind(v) + 1, ind(v) + 2, ..., n, n + 1$.

2. Multiple edges are allowed, that is, the same two nodes may be connected by two or more edges.

EDD recursively denote the following Boolean function:

1. If v is either 0-node or 1-node then $f = value(v)$;

2. Let $y_{ind(v)+1} = x_{ind(v)+1}$, $y_{ind(v)+2} = \overline{y_{ind(v)+1}}x_{ind(v)+2}$, $\dots$, $y_{j+1} = \overline{[y_1 \vee y_2 \vee ... y_{j-1}]}x_{j+1}$, $\dots$, $y_{n+1} = \overline{[y_1 \vee y_2 \vee ... y_n]}$. If v is variable node with $n - ind(v)$ outgoing edges: $(v, v_{ind(v)+1}), ...(v, v_{n+1})$ such that $l(v, v_i) = i$ for all $i : (ind(v) + 1 \leq i \leq)$, then $f_v = \bigvee_{i=ind(v)}^{n+1} y_i f_{v_i}$

Note that $y_i = 1$ iff $x_{ind(v)+1} = x_{ind(v)+2} = \dots = x_{i-1} = 0$ and $x_i = 1$;

Given an input vector A, the value of $f_v(A_{ind(v):n})$ for $v(\in V)$ can be computed recursively as follows: if v is terminal node then $f_v = value(v)$, otherwise, a child u such that $y_{l(u,v)} = 1$ is selected and then f_u is computed recursively. Every such transition to recursive function is called as the step of computation. On the each step v is added to the path of interpretation $I'(A)$. The first step adds to $I'(A)$ the root of EDD, and the last also one of terminal nodes.

Lemma 5.1 *EDD computation is done in $O(|X|)$ steps, where as $|X|$ it is the number of $x_i = 1$ in vector X.*

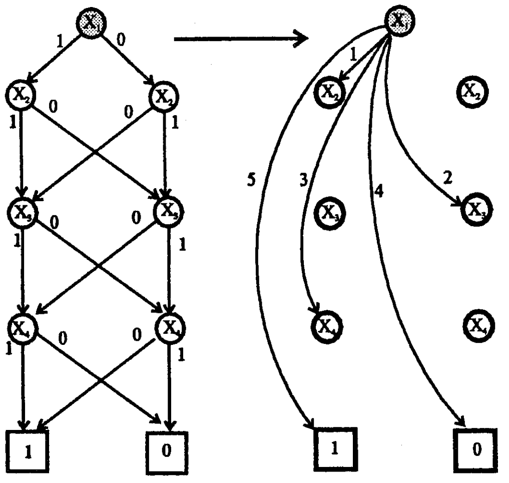

Figure 5: EDD edges for root node of OBDD

Proof: Consider $I'(A)$ generated during interpretation. For every node v_i $(1 < i < n + 1)$ it is follows from the definition of EDD that $x_{ind(v_i)} = 1$ if $ind(v_i) \leq n$. On opposite for any $x_i = 1$ there is node included in the interpretation path. Therefore number of $x_i = 1$ is exactly equal to $|I'(A)| - 2$. ■

The following algorithm implements the mapping $\psi : G' \to G$ generating unique EDD G' for given OBDD G:

ALGORITHM 5 *Foreach $v(\in V)$ we search $(n - ind(v))$ its children. To do that we pass OBDD with the following input data: $X^j = \{(x_i = 1)i = j, (x_i = 0)i \neq j\}_{\overline{j=ind(v),...n}, j \leq i \leq n}$. Each interpretation is done by k_j-times applied function $Step(v, x)$:$I(X^j) = (v_1^j, ..., v_{k_j}^j)$, where $k_j - 1 < j \leq k_j$; $v_i^j = Step(v_{i-1}^j, \theta_i^j), \theta_i^j = 1$ iff $i = j; \theta_i^j = 0$ otherwise. It means that interpretation of X^j terminates in first node with index greater or equal to j. This node last included in $I(X^j)$ accepted as v_j - child of v.*

Fragment of algorithm 5 implementation's result is shown at the Fig.5

Lemma 5.2 *EDD derived by the algorithm 5 denotes the same Boolean function as the parent BDD.*

Proof: Consider certain input vector $A =< a_1, a_2, ..., a_n >$. Let number of $a_j \neq 0$ is equal to k and indices of non-zero elements are $< j_1, j_2, ..., j_k >$. Thus EDD interpretation path is

	Memory	*Reaction on event*	*Pre- Evaluation*
Conventional singb BDD-traverse	$O(\|V\|)$	n	-
Massive pre-computations using independent single traverses of OBDD	$O(\|V\|+n^{k+1})$	$O(1)$	$O(n^{k+2})$
Massive pre-computations using dependent traverses of OBDD	$O(\|V\|+n^{k+1})$	$O(1)$	$\min(O(n^{k+2}), \max(O(n^{k+1}),O(\|V\|))$
Dynamic-N pointers	$O(n\|V\|)$	k	$O(n\|V\|)$
Dynamic-log(N) pointers	$O(\log_2 n\|V\|)$	$O(k\log_2 n)$	$O(\log_2 n\|V\|)$
Static pointer lattice	$O(\log_2 n\|V\|)$	$O(k\log_2 \frac{n}{k})$	$O(\log_2 n\|V\|)$
EDD traverse	$O(n\|V\|)$	k	$O(n\|V\|)$

Figure 6: Parameters of developed methods compared to those of conventional traverse

consisted of $< v_1, v_2, ..., v_t >$. Obviously these nodes also included in the interpretation path of BDD. Consider EDD and BDD rooted in v_{j_k}. As far as last 1 in the A was with index j_1 all inputs with greater indices are equal to 0 and by the definition of EDD they have the same result ■

Let us introduce displaced function f_{X-A} : $f(A) = f_{X-A}(\mathbf{0})$[1]and $f(\sigma(A)) = f_{X-A}(\Sigma)$. Using that function the re-evaluation problem of f can be reduced to the problem of evaluation of $f_{X-A}(\Sigma)$.

Lemma 5.3 *For given BDD G denoting $f(A)$, a BDD G^0 denoting $f_{X-A}(\mathbf{0})$ has the same number of nodes $V' = V$ and its edges have the following property: $\forall v(\in V) : a_{ind(v)} = 0 \Rightarrow lo(v^0) = lo(v), hi(v^0) = hi(v)$; $\forall v(\in V) : a_{ind(v)} = 1 \Rightarrow lo(v^0) = hi(v), hi(v^0) = lo(v)$;*

Leaning upon this lemma the algorithm may be suggested to transform given OBDD denoting function f to OBDD denoting f_{X-A}. For all nodes u of the given OBDD such that $x_{ind(u)} = 1$ marking of edges $(v, lo(v))$ and $(v, hi(v))$ is trading places. That action can be done in $O(|V|)$ time.

Then EDD $E = \psi(G^0)$ constructed for G^0 by above stated algorithm 5, also denotes displaced function f_{X-A} that follows from lemma 5.2. The function's computation for event $\sigma \in \Delta^k$ may be complete in $t \sim O(k)$. The preprocessing time elapsed to built EDD is proportional to $O(n\,|V|)$.

6 Conclusion

Paper proposed methods to process tuple-events in logic control systems using OBDD. Estimated complexity of the methods collected in the table on Fig.6. For the control algorithm with about $n = 2000$ input signals and query of 10 entries conventional OBDD (row 1) traverse return result after has passed all n vertices, whereas developed methods returns output in: (2,3)- constant time, (4,7) about 10 steps, (5,6) about 80-110 steps. Estimation of other parameters is strongly dependent on size of the OBDD which for some functions might be polynomially dependent on n.

[1]We denote vector of dimension n filled with zeros as $\mathbf{0}$ and filled with ones as $\mathbf{1}$.

References

[1] P. Baracos, R. Hudson, "Advances in binary decision based programmable controllers", IEEE Transactions on Industrial Electronics., vol.35, pp.415-425, Aug.1988, 1988.

[2] R. E. Bryant, "Graph-based algorithms for Boolean Function Manipulation", IEEE Transactions on computers, vol. C-35, No.8, 1986

[3] John T.Welch, "The Clause Counter Map: An Event Chaining Algorithm for Online Programmable Logic", IEEE Transactions on Robotics and automation, February, 1995.

[4] V.Viatkin, N.Ishii, T.Hayashi, "Event oriented evaluations of binary decision diagrams", International workshop on discrete event systems, IEE, Edinburgh, 1996

[5] V.Viatkin, K.Nakano, T.Hayashi, "Evaluation of logic expressions based on event-oriented interpretation of marked functional diagrams", Society for instrumentation and control engineering of Japan, 35th annual international conference, Tottori, 1996