Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification

Yung-Te Lai and Sarma Sastry

Electrical Engineering - Systems Department
University of Southern California, Los Angeles, CA 90089-2562

Abstract

In this paper we present a new data structure called edge-valued binary decision diagrams (EV) as a representation of functions. An EV is an extension of ordered binary decision diagrams that allows for multi-level and hierarchical verification. We show that an EV is a compact and canonical representation for arbitrary integer functions. Hence the specification can be at a higher level than the implementation. Furthermore, the variable ordering strategy for an EV can be derived from a higher level functional specification instead of the gate level specification. Examples shown in this paper includes SN74L85[7], SN74181[7], a 64-bit comparator built from SN74L85 and a 6-bit adder built from SN74181.

1 Introduction

Ordered Binary Decision diagrams (OBDDs), proposed by Bryant [2], are a canonical and compact representation of boolean functions. Since their introduction, many improvements and extensions have been reported. Examples include sharing and attributed edge [8], complemented edges [6, 1], breadth-first manipulation [9] and heuristic variable ordering strategies [4, 6, 8]. Besides being able to show equivalence of boolean functions, OBDDs have also been used in multi-level logic synthesis [10], resynthesis for network optimization [5] and verification of finite state machines [3, 12].

MDDs [11] are an extension of OBDDs which allow non-terminal nodes to have more than two children and terminal nodes to assume integer values. Although the methods using MDDs [1, 11] work for arbitrary discrete functions, MDDs are too general to capture certain properties of functions in a specific domain, such as arithmetic functions.

In this paper, we present a new extension to OBDDs called edge-valued binary decision diagrams (EV) which allow for efficient manipulation of arithmetic functions. An EV provides some additional advantages over OBDDs. First, an EV can be derived from a higher level specification (e.g., 'z + y' as a specification of an adder) instead of requiring a vector of boolean expressions. Second, EVs allow for hierarchical verification (e.g., designs need not be flattened). EVs preserve both the canonical and compactness properties from OBDDs. An EV usually is much more compact than an OBDD when it is used to describe an arithmetic function. For example, an EV requires only 129 nodes to represent the specification of a 64-bit adder. An EV shares some of the disadvantages as an OBDD, namely, the number of nodes may grow exponentially and the size is very sensitive to the ordering of input variables. However, since we can specify a design at a higher level, we can order the input variables based on the higher level specification instead of the structure of a design.

In Section 2 we introduce EVs. In Section 3 we show how to use EVs to carry out verification and synthesis. In Section 4 we describe an extension of EVs, called structural EV (SEV) which allows the specification of conditional expressions (control signals) directly. Hierarchical verification using SEVs is described in Section 5. In Section 6 we describe an ordering strategy for SEVs based on the function of the circuit. Note: Proofs are omitted due to limitations on space.

2 Edge-Valued Binary Decision Diagrams

Definition 2.1 Given a set of binary variables \( V \) and a total ordering on these variables, an EV is a tuple \((c,f)\) where \( c \) is a constant integer and \( f \) is an element of the set \( S \) which is defined recursively as:

1. \( 0 \) is in \( S \).
2. \((x, l, r, v)\) is in \( S \) if the following conditions hold:

   (a) All the variables, if they exist, in \( l \) \( S \) and \( r \) \( S \) are greater than \( x \) \( V \) with respect to the given ordering.
   (b) It is not the case that \( l \equiv r \) and \( v = 0 \) where \( v \) is an integer.

Definition 2.2 An EV \((c,f)\) denotes the function \( c + f \) where \( f \) is the function denoted by \( f \in S \). \( 0 \) denotes the constant function \( 0 \) and \((x, l, r, v)\) denotes the arithmetic function \( x(v + l) + (1 - x)r \).

In the graphical representation of an EV \((c,f)\), \( f \) is represented by a rooted, directed, acyclic graph and \( c \) by a dangling incoming edge to the root node of \( f \). \( 0 \) is depicted by a square node labelled 0 and this
is the only terminal node. A non-terminal node is a quadruple \((x, l, r, v)\), where \(x\) is the node label, \(l\) and \(r\) are the two subgraphs (called left and right children) rooted at \(x\), and \(v\) is the label assigned to the left edge of \(x\). Figure 1 shows the EVs for the sum and carry function which implement a full adder. If \(x, y\), and \(z\) are boolean variables, the carry function is \((x \land y) \lor (y \land z) \lor (z \land x)\). If \(x, y\), and \(z\) are viewed as integers then the carry function is \(zy + yz + zz - 2zy^2\), whose EV is shown in Figure 1a. The function carry can be derived from the following equations:

\[
\begin{align*}
carry & = 0 + f_z \\
f_z & = x(0 + f_y) + (1 - x)f_y \\
f_y & = y(1 + 0) + (1 - y)f_x \\
f_x & = z(1 + 0) + (1 - z)0
\end{align*}
\]

An EV can represent both Boolean and arithmetic functions. Two examples of arithmetic functions represented by an EV are shown in Figure 2. From Figure 2, it is easy to see that it requires only \(n\) non-terminal nodes to represent an \(n\)-bit integer and \(2n\) non-terminal nodes to represent the addition of two \(n\)-bit integers.

In the above algorithm, terminal cases (line 3) occur when both \(f\) and \(g\) are 0, or when for example, \(op = \times\), \(c_1 = 1\), and \(f\) and \(g\) are isomorphic.

The following algorithm describes the function apply which takes \((c_f, f)\) and \((c_g, g)\) as arguments and returns \((c_h, h)\) such that \(c_h + h \equiv (c_f + f) \circ (c_g + g)\) where \(\circ\) can be any operator which is closed over the integers.

Algorithm A:
1. apply\((c_f, f), (c_g, g), op\)
2. {
   3. if (terminal case) return\((c_f, f) \circ (c_g, g))\);
   4. else if (computed\((c_f, f), (c_g, g), op\))
      return\(\text{comp-table}\((c_f, f), (c_g, g), op\))
   5. else {
      6. var = min\_var(root(f), root(g), ordering);
      7. \((c_1, h_1) = \text{apply}(L((c_f, f), var), \text{apply}(L((c_g, g), var), \circ))\);
      8. \((c_r, h_r) = \text{apply}(R((c_f, f), var), \text{apply}(R((c_g, g), var), \circ))\);
      9. if (equal((c_1, h_1), (c_r, h_r))) ans = (c_1, h_1);
      10. else if (exist(var, h_1, h_r, c_1 - c_r))
         ans = \((c_r, \text{node-table}(\text{var}, h_1, h_r, c_1 - c_r))\);
      11. else {
         12. insert(\text{node-table}, create\_node(\text{var}, h_1, h_r, c_1 - c_r));
         13. \((c_r, \text{node-table}(\text{var}, h_1, h_r, c_1 - c_r))\);
         14. }
      15. insert(\text{comp-table}, \((c_f, f), (c_g, g), \circ, \text{ans})
      16. return(\text{ans})
      17. }
      18. }

In the above algorithm, terminal cases (line 3) occur when both \(f\) and \(g\) are 0, or when for example, \(op = \times\), \(c_f = 1\), and \(f\) and \(g\) are isomorphic. The latter is a terminal case because \((1, 0) = 1 + 0 = 1\), \((c_f, g) = c_f + g\), and \(1 \times (c_f + g) = (c_f + g) = (c_f, g)\), thus the result can be returned immediately without traversing the graph. Two tables are used in Algorithm A: a comp-table to achieve computation efficiency (entries are stored in line 15 and used in line 4), and a node-table to ensure there are no duplication of nodes (line 10). If it is neither one of the terminal cases nor been computed before, then the operation requires a traversal of the graph based on the ordering of the root variables associated with \(f\) and \(g\). The function \(L((c_f, f), var)\) and \(R((c_f, f), var)\) both return \((c_f, f)\) if \(\text{root}(f) > var\), otherwise they return \((c_f + v_f, f_1)\) and \((c_f, f)\) respectively.

After the left and right children have been computed resulting in \((c_l, h_l)\) and \((c_r, h_r)\) (lines 7 and 8), the algorithm returns the pair \((c_h, h)\) determined in lines 9, 10 or 13. Line 9 is to guarantee that the case of \((z, k, k, 0)\) will not occur. In lines 10 and 13, the values \(c_h\) and \(v_h\) are set to \(c_r\) and \(c_1 - c_r\). This is to ensure that the value of right edge remains 0.
Example 2.1  An example of \( \text{apply}(0, f), (0, g), + \) is shown in Figure 3. Let the variable ordering be \( x_1 < x_2 \). Figure 3 (a) shows the initial arguments of apply; (b) is the recursive call of apply on line 7 whose result is (c). Similarly, another call to apply on line 8 and its results are shown in (d) and (e) respectively. The final result is shown in (f) which is generated from either line 10 or 13.

Since an EV is an acyclic directed graph and there is no backtracking in the above algorithm, the worst case complexity is the product of the numbers of nodes of \( f \) and \( g \), and the size of the ranges of \( c_f + f \) and \( c_g + g \). In many practical applications, the number of nodes of an EV is small, but the size of the range of \( c_f + f \) is very large. For example, to represent an \( n \)-bit integer in \( (c_f, f) \), the number of nodes of \( f \) is \( n \) while \( c_f + f \) can be any one of \( 2^n \) different values. In the following subsections, we present several properties of EVs which can be used to reduce the computation complexity of apply in many situations.

2.1 Additive Property

An operator \( op \) is said to satisfy the additive property if

\[
(c_f + f) \ op \ (c_g + g) = (c_f \ op \ c_g) + (f \ op \ g)
\]

Two such examples are + and -. 

\[
(c_f + f) \pm (c_g + g) = (c_f \pm c_g) + (f \pm g)
\]

Because the values \( c_f \) and \( c_g \) can be separated from the functions \( f \) and \( g \), the entries of \( \text{comp-table} \) can be \((0, f), (0, g), \text{op}\) for this class of operators. After the computation of \((0, f), (0, g), \text{op}\) resulting \((c_f, h)\), we add \( c_f \ \text{op} \ c_g \) to \( c_h \) to have the complete result of \((c_f, f), (c_g, g), \text{op}\). Hence, every operation \((c_f, f), (c_g, g), \text{op}\) can share the computation result of \((0, f), (0, g), \text{op}\).

2.2 Subdomain Property

An operator \( op \) is said to satisfy the subdomain property if

\[
(c_f + f) \ op \ (c_g + g) = (c_f + f) \ op \ (c_g + g)
\]

where the domain of \( c_f \) is a subset of \( c_f \)’s. An example of this is when an expression is modulo a constant:

\[
(c_f + f) \ mod \ c = (c_f \ mod \ c + f) \ mod \ c
\]

In this case, \((c_f + kc) \ mod \ c \) can share the computation result of \((c_f, f)\) for any integer \( k \). Thus, we reduce the size of the range of \( c_f \) to the value of \( c \). When \( c \) is small, computation sharing is large; when \( c \) is large, then boundary property (next section) can be applied.

2.3 Boundary Property

The additive and subdomain properties often result in a significant savings in computation time. The boundary property also results in a reduction of computation time by increasing terminal cases (line 3 of procedure apply). As an example, consider the case of \( op = \text{mod} \) and \( c_f + g = c \), for some constant \( c \). If we know the maximum value of \( f \), then for all \( c_f \) such that \((c_f + f) < c \), we can have \((c_f + f) \ mod \ c = (c_f + f)\) without further computation.

Another example is from relational operators. Suppose the following result has been computed

\[
(c_f + f) > (c_g + g)
\]

Then for all \( c_f \) and \( c_g \) such that \((c_f + f) > (c_g + g)\) we have \((c_f + f) = (c_f + g) = 1\). To handle this class of behavior, we can use \((0, f), (0, g), >, c_f - c_g, 1\) as an entry of \( \text{comp-table} \). Alternatively, we can keep track of the maximum and minimum value of each non-terminal node and then include the following tests in the terminal conditions for operator >:

\[
(c_f + \min(f)) > (c_g + \max(g)) \Rightarrow 1
\]

\[
(c_f + \max(f)) < (c_g + \min(g)) \Rightarrow 0
\]

3 Verification and Synthesis Paradigm

In this section, we show how EVs can be used to perform functional verification and synthesis. First, we show as an example \( \text{sum}(x, y, z) \) and \( \text{carry}(x, y, z) \) implement the full adder \( x + y + z \). Given

\[
\text{imp}(x, y, z) = (\text{sum}(x, y, z), \text{carry}(x, y, z))
\]

\[
\text{spec}(x, y, z) = w_1 + 2w_2
\]

we want to show \( x + y + z = w_1 + 2w_2 = \text{sum}(x, y, z) + 2\text{carry}(x, y, z) \). The left hand side of the above expression implemented in EV is shown in Figure 4, and the EV for the right hand side is shown in Figure 5. By comparing the two rightmost EVs in Figure 5 and Figure 4, we verify that \( \text{sum}(x, y, z) \) and \( \text{carry}(x, y, z) \) implement \( x + y + z \).

The above procedure can be reversed to become a procedure for functional synthesis. Again, we use the full adder as an example except now the goal \( \text{imp}(x, y, z) \) is not given. From the description of \( \text{spec}, \) we have

\[
\begin{align*}
\text{sum}(x, y, z) &= w_1 = \text{fa mod } 2 \\
\text{carry}(x, y, z) &= w_2 = (\text{fa} - (\text{fa mod } 2))/2
\end{align*}
\]
one of SPEC takes implementation, it takes less than 

Boolean expressions are really carrying out an addi-

2' 

eration Of 65 Of IMP our experimental implementation on SUN (129 

where (64) and (65) declare the number of bits. In 

Example 3.1 The design (IMP) is a 64-bit 3-level 

carry lookahead adder which has 129 inputs, 65 outputs, and 420 logic gates. The intended behavior 

(64) x y; 

unsigned c; 

return(x + y + c);

where (64) and (65) declare the number of bits. In 

unsigned65) add64(x, y, c) 

unsigned64) x y; 

unsigned c; 

return(x + y + c);

Figure 4: The EV derived from the specification of a full-adder

Figure 5: The EV derived from the implementation of a full-adder

where $f_a = x + y + z$. The following sequence of apply 

operations on EVs then can produce sum and carry automatically.

$\begin{align*}
(0, xy) &= \text{apply}(0, x), (0, y), +; \\
(0, fa) &= \text{apply}(0, x), (0, xy), +; \\
(0, sum) &= \text{apply}(0, fa), (2, 0), \text{mod}; \\
(0, temp) &= \text{apply}(0, fa), (0, sum), -; \\
(0, carry) &= \text{apply}(0, temp), (2, 0), /; 
\end{align*}$

4 Structure Edge-Valued Binary Decision Diagrams

As shown in previous section, EVs are useful for verifying functions performed by a data path. These functions include both arithmetic functions and relational operators. To include conditionals, for verifying control paths, we define a new data structure called SEV, as follows:

Definition 4.1 The set SEV is defined recursively as:

1. An EV is an SEV.
2. $(p \rightarrow t; e)$ is a (conditional) SEV if $p$ is an SEV with the range $\{0, 1\}$, and $t$ and $e$ are SEVs. The function denoted by $(p \rightarrow t; e)$ is that for any input assignment $a$, if $p(a) = 1$ then the function returns the value $t(a)$, otherwise it returns $e(a)$.
3. $[f_1, \ldots, f_m]$ is an SEV if $f_1, \ldots, f_m$ are SEVs. For any input assignment $a$, $[f_1, \ldots, f_m]$ returns the vector $(f_1(a), \ldots, f_m(a))$.

To develop the notion of a canonical SEV, we need to define reduction rules. To this end, we use two operators, ite and cofact which can transform the forms of SEVs without changing the functions they denote. ite takes a conditional SEV such as $(p \rightarrow t; e)$ as argument and returns an SEV $f$ which is similar to $t$ and/or $e$ in form. Furthermore, $(p \rightarrow t; e)$ and $f$ will denote the same function. Operator ite is essentially the one described in [1]. The other operator, cofact, does the reverse operation of ite. cofact takes an SEV $p$ in the range $\{0, 1\}$ and an arbitrary SEV $f$ as arguments and returns two SEVs $f_1$ and $f_0$ so that $(p \rightarrow f_1; f_0)$ denotes the same function as $f$. In the sequel, we'll use cofact1 and cofact0 to replace cofact so that cofactt1(p, f) = f1 and cofactt0(p, f) = f0. A description of the algorithm cofact1 can be found in [12]. With these two operators, we can easily change SEVs from a conditional form to a non-conditional form and vice versa.

Now, we define the following restricted form and relation on SEVs so that SEV can become canonical.

Definition 4.2 An SEV is reduced if

1. it is an EV, or
2. it is a conditional form of $(p \rightarrow t; e)$ where $p$ is not $(c, 0)$, cofact1(p, t) = t, cofact0(p, e) = e, and $t$ and $e$ are reduced, or
3. it is $[f_1, \ldots, f_m]$ and every $f_i$ is reduced.

Definition 4.3 Two SEVs $sev_f$ and $sev_g$ are in the same form if

1. both $sev_f$ and $sev_g$ are EVs, or
2. $sev_f = (p \rightarrow t_j; e_j)$, $sev_g = (p \rightarrow t_k; e_k)$, $t_j$ and $t_k$ are in the same form, and $e_j$ and $e_k$ are in the same form, or
3. $sev_f = [f_1, \ldots, f_m]$, $sev_g = [g_1, \ldots, g_n]$ and every pair of $f_i$ and $g_i$ is in the same form.

Lemma 4.1 Given two SEVs $sev_f$ and $sev_g$ which are reduced and in the same form, then $sev_f$ and $sev_g$ are isomorphic if and only if they denote the same function.
The following two algorithms can be used to convert an SEV derived from an implementation to the form derived from the specification. Note, both algorithms employ operators \textit{ite} and \textit{cofact} which can be implemented in polynomial time in the number of nodes of SEVs. 

**Algorithm B:** Converting \( f \) to a reduced form.

\[
\text{redu}(f) = \\
\begin{cases} 
\text{if} & (f \text{ is an EV}) \rightarrow \text{return}(f); \\
\text{else if} & (f \leftarrow (1, 0) \rightarrow t; e) \rightarrow \text{return}(\text{redu}(t)); \\
\text{else if} & (f \leftarrow (0, 0) \rightarrow t; e) \rightarrow \text{return}(\text{redu}(e)); \\
\text{else if} & (f \leftarrow (p \rightarrow t; e) \land p = (p' \rightarrow t'; e')) \rightarrow \text{return}(\text{redu}(\text{ite}(p) \rightarrow t'; e)); \\
\text{else if} & (f \leftarrow (p \rightarrow t; e)) \rightarrow \text{return}(\text{redu}(\text{cofact}(p, t))); \\
\text{else} & \text{return}(\text{redu}(f_1), ..., \text{redu}(f_m)) 
\end{cases}
\]

**Algorithm C:** Converting \( \text{sev}_j \) to the same form as \( \text{sev}_i \) assuming \( \text{sev}_j \) and \( \text{sev}_g \) have the same number of outputs.

\[
\text{convert}(\text{sev}_j, \text{sev}_g) = \\
\begin{cases} 
\text{if} & (\text{sev}_j \text{ is an EV}) \rightarrow \text{return}(\text{sev}_j); \\
\text{else if} & (\text{sev}_j = (p \rightarrow t; e)) \rightarrow \text{return}(\text{convert}(\text{cofact}(p, t)); \text{redu}(\text{cofact}(p, e))); \\
\text{else} & \text{return}(\text{convert}(f_1, ..., \text{convert}(f_m, g_m))) 
\end{cases}
\]

**Example 4.1** The implemented design (IMP) is the SN74L85 chip [7] which is a 4-bit comparator. This chip has 11 inputs, 3 outputs and 33 gates. The specification (SPEC) of the design may be described as:

\[
\text{unsigned}(3) \text{ comp}(4, x, y, gt, lt, eq) \\
\text{unsigned}(4) x, y; \\
\text{unsigned} gt, lt, eq; \\
\begin{cases} 
\text{if} & (x > y) \rightarrow \text{return}(1, 0, 0); \\
\text{else if} & (x < y) \rightarrow \text{return}(0, 1, 0); \\
\text{else} & \text{return}((gt, lt, eq)) 
\end{cases}
\]

It takes 0.05 seconds to generate the SEV of IMP which has 39 nodes and it takes 0.02 seconds to construct the conditional SEV of SPEC which has 25 nodes. The conversion from the SEV of IMP to the one of SPEC and then the comparison take 0.02 seconds.

**Example 4.2** The implementation is the SN7481 chip which is a 4-bit ALU [7]. A partial specification is given below. Note: \text{un-comp}, \text{two} and \text{unsigned} perform type coercion. \text{un-comp} results in an unsigned integer, with the most significant bit being complemented, two means that the result is to be a two's complement integer.

\[
\text{SN7481}(M, S, A, B, \text{Cin}) \\
\text{unsigned} M, C, A, B; \\
\begin{cases} 
\text{if} & (M = 0) \\
\text{if} & (S = 0) \rightarrow \text{return}((\text{un-comp}(5)) A + (\text{Cin})); \\
\text{else} & \text{return}((\text{two}(5)) \rightarrow \text{Cin}); \\
\text{else} & \text{return}((\text{un-comp}(5)) A \rightarrow B \rightarrow \text{Cin}); \\
\text{else} & \text{return}((\text{unsigned}(4) \rightarrow \text{not}(A)); \\
\text{else} & \text{return}((\text{unsigned}(4) \rightarrow \text{not}(A \lor B)); 
\end{cases}
\]

Note that different output decodings including the difference of sizes are allowed for the same outputs. The conversion from the SEV of IMP to the one of SPEC and then the comparison take 0.02 seconds to complete.

**5 Hierarchical Verification**

In addition to providing the capability for multi-level verification, SEV is suitable for hierarchical verification, i.e., without having to completely flatten a component that has already been verified. Briefly, hierarchical verification is done by first deriving SEVs from the specification of each component. Now the inputs to each component are SEVs themselves. Therefore, the effect of interconnecting a set of components is realized by deriving an SEV given the component SEVs and the SEVs corresponding to the input arguments. The net result of this process is an SEV which is compared with the SEV derived from the specification of the entire system. Below we present two examples.

**Example 5.1** The design is a 64-bit comparator implemented through serial connection of 16 SN74L85s. A net list description of this design we use is:

\[
\begin{array}{c}
\text{out1} \quad \text{SN74L85} \; a0 \; a1 \; a2 \; a3 \; b0 \; b1 \; b2 \; b3 \; \text{gt} \; \text{lt} \; \text{eq} \\
\text{out2} \quad \text{SN74L85} \; a4 \; a5 \; a6 \; a7 \; b4 \; b5 \; b6 \; b7 \; \text{out1} \\
\vdots \\
\text{out16} \quad \text{SN74L85} \; a60 \; a61 \; \ldots \; b62 \; b63 \; \text{out15} \\
\text{Output} : \text{out16}
\end{array}
\]

where a net list has the form of: \textit{output_name module_name input_name_list}. The specification of this design is the same as the one in Example 4.1 except...
that the size declaration is changed from 4 to 64. The generations of implementation and specification SEVs take 0.26 and 0.39 seconds respectively, and the proof takes 3.35 seconds to finish.

Example 5.2 The design is a 64-bit ripple-carry adder implemented through serial connection of 16 SN74181s. The specification of this design is exactly the same as the one used in Example 3.1. The times to generate the SEVs for implementation and specification are 2.09 and 0.16 seconds respectively and the time to verify they are equivalent takes 0.98 second. Note that the generation of implementation SEV takes longer time while the verification takes less time than the case in Example 3.1. This is because 16 SEVs each with the sum of 4 bits instead of 64 SEVs each with the sum of 1 bit are generated in this example.

6 Ordering Strategy

In our implementation, variable ordering is based on the following rules:

1. Use the specification rather than implementation to determine the ordering.

2. Variables occurring in p have lower indices than the ones in t or e for a (p -> t; e) SEV.

3. Variables having larger coefficient have lower indices than the ones having smaller coefficients

The advantage of rule 1 is that it is much easier to determine a good ordering at the functional level. Rule 2 matches the experimental results reported in [2]. The main reason of rule 3 is to be able to utilize the boundary property described in Section 2.3.

7 Concluding Remarks

As shown in the examples, EVs can be used to perform equivalence checking between boolean expressions and arithmetic expressions with relational operators. SEVs allow for the verification of both data path and control path. Furthermore, the time required to carry out the equivalence checking between different levels is comparable to the time required for tautology checking in OBDDs. As with OBDDs, the size of SEVs may grow exponentially. From an implementation point of view, OBDDs require only 22 bytes [1] to represent a node, whereas EVs require a bit more space. Memory management problems can usually be avoided by using arithmetic expressions as the specification and hierarchical verification whenever it is possible.

Acknowledgment

This research was supported in part by a grant from the National Science Foundation, award No. MIP-9111206.

References


