DD Package Generator

Suzana Stojković, Dragan Janković, Milena Stanković

Abstract: **Decision diagrams (DDs) are frequently used as efficient data structures for discrete functions representation and manipulation. One of approaches in DD packages development is a generic approach. Generic approach provides uniform development of DD packages working with different types of DDs. This paper presents the DDPG (DD Package Generator) program for the generation of C++ code of DD packages. Generated packages can manipulate with shared MDDs, and contain implementations of user-defined operations and spectral transforms.**

I. INTRODUCTION

Many problems in digital logic design, artificial intelligence, telecommunications, etc. are based on manipulations of discrete functions. Because of that, various methods for discrete functions representations were developed. Graph-based representation of Boolean functions, by Bryant 1986. [6], named Binary Decision Diagrams (BDDs), are very frequently used. Analogously, for the multi-valued discrete functions representation Multi-Valued Decision Diagrams (MDDs) are defined. For various applications, different types of DDs are defined. DD programming is a frequently decided problem in last years. In the papers [4] and [6] are defined the basic DD programming principles: to support dealing with shared DDs, to have unique node table, to support strong canonicity, to have unique compute table, to use efficient memory management. Existing DD packages (such as: CUDD, PUMA, BXD, VIS, SIS, [3],[4],[5],[6]) are efficient in work with any type of DDs or in work with some classes of discrete functions. Because of that, development of a DD package, which works with any type of DDs, is very useful. The paper [9] proposed a uniform approach in DD package development, named generic approach. On the base of that approach, DDPG (DD Package Generator) is build. DDPG is a program that generates C++ code of DD packages which work with a shared multi-valued DDs (MDDs). For code generating DDPG needs an input specification of DD package. Input DD package specification (DDP specifiacation) is a formal DD package description written in DDPSL (DD Package Specification Language) [10]. DDP specification describes operations and/or spectral transforms which are implemented in generated package.

Suzana Stojković, Dragan Janković, Milena Stanković
Faculty of Electronics, Beogradska 14, Niš, Yugoslavia
e-mail: **suza@elfak.ni.ac.yu**

II. DECISION DIAGRAMS AND SPECTRAL TRANSFORMS

Decision diagrams are acyclic directed graphs that contain nonterminal nodes, terminal nodes, and edges. Nonterminal nodes are labeled with a variables x_i in f and have q output edges. Output edges are labeled with a values of variable x_i. Terminal nodes contain the values of function f at the points defined by *n-tuples*, which label edges from the root node to the corresponding terminal nodes. In BDDs nonterminal nodes have two successors, while in MDDs nonterminal nodes have more then two successors. Fig. 1 shows BDD of binary function $f(x_1,x_2,x_3)$ defined by the truth-vector F=[0,0,0,0,0,1,1,1] and MDD of ternary function $g(x_1,x_2,x_3)$ defined by the truth-vector G=[0,0,0,0,2,1,2,0,0,1,1,1,2,2,2,2,0,0,1,1,1,2,2,2,2,0,0].

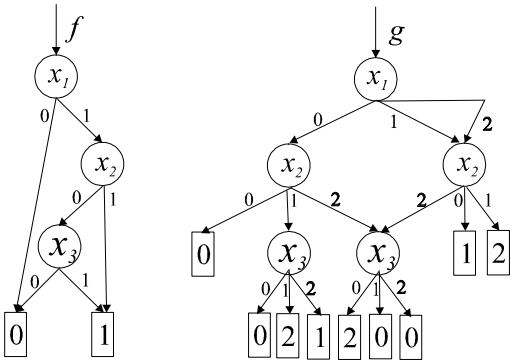

Fig. 1: BDD of function f (a) and MDD of function g (b).

Spectral transform (S), of q-valued n-variable discrete function f is defined as the product of transform matrix (T_n) and the truth-vector of the function f (F).

$$S = T_n \cdot F$$

where the transform matrix T_n is defined by:

$$T_n = T_{n-1} \otimes T_1$$

where T_1 is the basic (qxq) transform matrix, and $\otimes$ denotes the Kronecker product.

Calculation of spectral transform of functions represented by DDs is done by performing elementary transform (defined by basic transform matrix) in each nonterminal node of DD.

III. DD PACKAGE IMPLEMENTATION

Basic problems in DD package implementation are:

- to choose appropriate data structure for DD node representation;
- to select efficient algorithm for DD generation;
- to support basic principles for DD programming.

Data structure for BDD node representation, defined in paper [6], has been usually used in existing BDD packages (see Fig. 2).

Fig. 3 shows data structure for MDD node representation suggested in paper [7].

```
struct node
{
     node *high,*low;
     int index;
     int value;
     int id;
     int ref_counter;
     boolean mark;
}
```

Fig. 2: Data structure for BDD node representation.

```
typedef struct node *DDedge;
typedef struct node
{
     int ref;
     char value,flag;
     DDedge next, previous;
     DDedge edge[0];
} node;
```

Fig. 3: Data structure for MDD node representation.

There are different ways for discrete function representation (truth-vector, cubes, …). Because of that different algorithms for DD building were developed. All they contain series of logical operrations on DDs. One of the basic principles for DD programming is using the compute table where the results of previous calculation are stored. Due to, all logic operations are improved by using one universal operator. In the binary logic, it is *if-than-else* (ITE) operator, but in multi-valued logic it is the CASE operator.

ITE operator [4] is a Boolean function of 3 variables (F,G,H) defined as: **If *F* then *G* else *H***, and formal:

$$ite(F,G,H) = F \cdot G + \overline{F} \cdot H$$

In paper [9] is shown the relationship between definition table of an Boolean operator and realization of that operator by ITE. Definition table of an arbitrary Boolean operation is shown in Tab. 1.

Tab. 1: Definition table of Boolean operator OP.

OP	0	1
0	$v_{0,0}$	$V_{0,1}$
1	$v_{1,0}$	$V_{1,1}$

If two switching functions *a* i *b* are represented by BDDs, computation of *aOPb* can be realized by ITE operator as follows:

$$OP(a,b) = ite(a, ite(b, v_{1,1}, v_{1,0}), ite(b, v_{0,1}, v_{0,0})).$$

AND operator can be realized by using ITE operator as follows:

$$AND(a,b) = ite(a, ite(b,1,0), ite(b,0,0))$$
$$= ite(a, ite(b,1,0), 0) = ite(a,b,0)$$

CASE operator in *q*-valued logic is (*q*+1)-variable function defined as follows:

$$CASE(F, G^0, G^1, \cdots, G^{q-1}) = G^i \quad for \quad F = i \ .$$

CASE operator can be used for realization of any *q*-valued discrete function. Paper [9] shows the realationship between definition table of a *q*-valued function and the corresponding CASE operator. If *q*-valued operator (*qOP)* is defined by Tab. 2, then the operator *qOP* can be realized by CASE operator as:

$$qOP(a,b) = CASE(a, CASE(b, v_{q-1,q-1}, v_{q-1,q-2}, \cdots, v_{q-1,0}),$$
$$CASE(b, v_{q-2,q-1}, v_{q-2,q-2}, \cdots, v_{q-2,0}), \ldots,$$
$$CASE(b, v_{0,q-1}, v_{0,q-2}, \cdots, v_{0,0}))$$

Tab. 2: Definition table of q-valued operator q*OP.*

QOP	0	1	…	*q*-1
0	$v_{0,0}$	$v_{0,1}$	…	$v_{0,q-1}$
1	$v_{1,0}$	$v_{1,1}$	…	$v_{1,q-1}$
⋮	⋮	⋮	…	⋮
q-1	$v_{q-1,0}$	$v_{q-1,1}$	…	$v_{q-1,q-1}$

Spectral transform computation on DDs can be realized as a set of operations + and * (in corresponding algebraic structure). Because of that, ITE (CASE) operator is sufficient for spectral transform computation on BDD (MDD). In the paper [9] a generic approach in spectral transform implementiation is shown, too. If a spectral transform is defined by the basic transform matrix T, transformation in nonterminal node v can be performed as:

$$trans\_node(v) =$$
$$= node(ADD(MUL(t_{0,0}, v_o), ADD(MULL(t_{0,1} v_1), MUL(t_{0,2}, v_2))),$$
$$ADD(MUL(t_{1,0}, v_o), ADD(MULL(t_{1,1} v_1), MUL(t_{1,2}, v_2))),$$
$$ADD(MUL(t_{2,0}, v_o), ADD(MULL(t_{2,1} v_1), MUL(t_{2,2}, v_2))))$$

Fr om that, it appears that code for any operation and any spectral transform can be generated automatically if the corresponding definition tables are known.

IV. DD PACKAGE CODE GENERATION

DDPG (Decision Diagram Package Generator) generates C++ code of DD package on the base of input DD package specification. For input DD package specification formal language DDPSL (DD Package Specification Language) [10] was developed. DD package specification contains definition of *q*-value of logic, definitions of operations and spectral transformations which will be implemented in generated package. In the input specification, operations are defined only by definition tables, but transform definitions contain definitions of + and * operators, basic transform matrices, and C++ code blocks which will be inserted in a generated code. Contents of inserted code blocks will be described later (section 5). Generation process is devided into two steps. In the first step input specification is parsed. If there are any syntax or semantic errors in input specification, the generation process is stopped and error messages is shown. If there are not any errors, the generation process continues and in the second step the C++ code will be generated.

V. GENERATED CODE FEATURES

Generated C++ code is based on DDP class library. DDP class library contains all basic classes for a DD manipulation. The most important classes in DDP library are:

- `DDNode` – represents the MDD node,
- `DDEngine` – represents shared MDD,
- `DDNodeMaker` – abstract class which creates `DDNode`.

Class `DDNode` realizes both terminal and nonterminal nodes in MDD and contains the following attributes:

- unique node identifier;
- variable index associated with nonterminal node;
- value of terminal node;
- number of input edges in the node;
- dynamic vector of pointers to the successor nodes;
- pointer to the next node at the same level;
- mark field which indicates if the node is processed or not (it is used in different DD manipulations).

In the class `DDNode` abstract functions for the logical operations AND, OR and NOT (which are dependent on q-value of logic) and arithmetic operations ADD and MUL (which are dependent on algebraic structure) are declared. These functions will be implemented in generated class derived from the class `DDNode`.

The most important attributes in class `DDEngine` are:

- variables number;
- *q*-value of logic;
- output functions number;
- unique DD node table implemented as an hash table;
- unique computeTable – table of CASE operator computations whose entry contains input nodes pointers and result node pointer;
- array of the root nodes pointers.

In the class DDEngine, all basic functions for MDD manipulation are defined. Frequently used functions are:

- `bool createMDDfromCubes()` – creates MDD of function defined by cubes, returns true if creation is finished correctly;
- `bool createMDDfromTV()` – creates MDD of function defined by truth-vector, returns true if creation is finished correctly;
- `void levelExchange(unsinged k,unsigned l)` – exchanges levels k and l in the MDD;
- `void printMDD()` – prints MDD;
- `void printMDD(char k)` – prints MDD for k-th output function;
- `unsigned getValue(unsigned *m,char k)` – returns the value of k-th output function for minterm m;
- `unsigned getValue(unsigned long i,char k)` – returns value of k-th output function for index i;
- `void printTV(char k)` – prints a truth-vector of k-th output function;
- `unsigned size()` – returns nonterminal nodes number.

Class `DDNodeMaker` contains abstract function for `DDNode` creation. For each class, derived from `DDNode`, DDPG generates the corresponding node-maker class derived from `DDNodeMaker`. `DDEngine` class receives pointer to the node-maker class by constructor.

Class diagram of DD package generated by DDPG is shown on Fig. 4. In the diagram, DDP library classes are colored by blue and generated classes are colored by yellow. DDPG generates:

- `DDNode<q>` - class derived from `DDNode` class where functions for logical operations (AND, OR and NOT) and functions for each operations, defined in operation section of input DDP specification, are implemented in;

as well as for each transform, defined in transformation section of DDP specification, the following classes:

- `<trName><q>DDNode` – class in which operators + and *, for corresponding transform, are implemented;
- `<trName><q>DDEngine` – class in which spectral transform function is implemented.

In transform description in DDP specification, some code blocks describing global definitions and extra class members of classes `<trName><q>DDNode` and `<trName><q>DDEngine` can be specified. Global definitions, labeled as `DDNodeInclude` and `DDEngineInclude`, DDPG inserts at the beginning of files `<trName><q>DDNode.h` and `<trName><q>DDEngine.h`, but extra class members, labeled as `DDNodeClass` and `DDEngineClass`, inserts in definitions of the corresponding classes. At the end of each generated .CPP file DDPG generates comment:

```
// place your definitions here
```

If DDP specification will be changed, code part written after that comment will be saved into regenerated files.

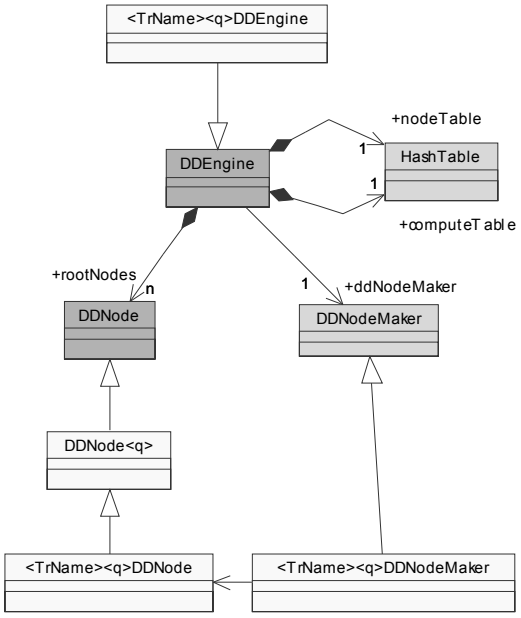

Fig. 4: Class diagram of generated DDPackage.

There are implementations of all generated operations in DDNode<q> class. It follows that once defined operation (including OR and AND) can be used in different transforms as operator + or *. In this way, performances of object-

oriented programming are optimally used for efficient code generation.

Example. DDP specification for generation of DD package, wich deals with RMF and GF transforms in a 4-valued logic is shown in the Fig. 5. Class diagram of the corresponding generated DD package is shown at Fig. 6.

```
valueability = 4

// operation section
operator ADD_MOD4 = {
                        0, 1, 2, 3,
                        1, 2, 3, 0,
                        2, 3, 0, 1,
                        3, 0, 1, 2,
                    }
operator MUL_MOD4 = {
                        0, 0, 0, 0,
                        0, 1, 2, 3,
                        0, 2, 0, 2,
                        0, 3, 2, 1,
                    }

// transform section
transform RMF =  {
                    ADDoperator = ADD_MOD4
                    MULoperator = MUL_MOD4
                    {
                        1, 0, 0, 0,
                        1, 3, 0, 0,
                        1, 2, 1, 0,
                        1, 1, 3, 3,
                    }
                }
transform GF  =  {
                    ADDoperator = ADD_MOD4
                    MULoperator = MUL_MOD4
                    {
                        1, 0, 0, 0,
                        0, 1, 3, 2,
                        0, 1, 2, 3,
                        1, 1, 1, 1,
                    }
                }
```

Fig. 5: DDP specification.

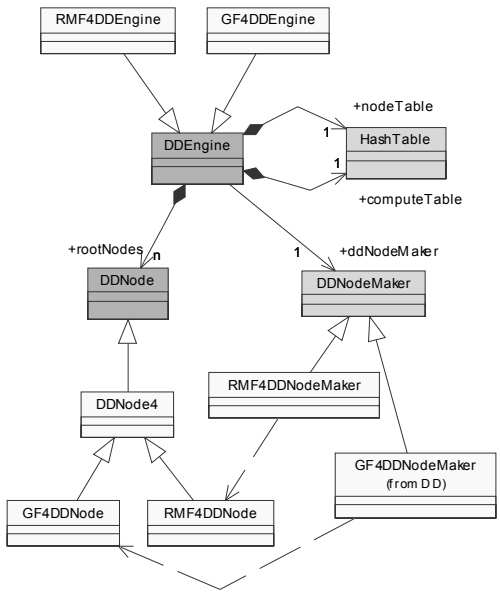

Fig. 6: Class diagram of package generated from specification shown in Fig.5.

VI. CONCLUSION

Generic approach in DD package implementation enables development of an automatic code generator for DD packages. Based on generic approach, tool DDPG is developed. For software generation by using DDPG, input specification is needed. Input specification is a formal specification, written on DDPSL, that contains definition tables of operations and transforms which will be implemented in generated package. In this way generated packages can manipulate with different types of DD-s. Base of generated C++ code is DDP library class which contains classes for basic manipulations over DDs. DDP library is developed by using the proposed basic principles of DD programming. This library is not finished. Current shortages of class library can be compensated by user-defined extra members in generated classes.

REFERENCES

[1] Sasao, T., Fujita, M., *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996.

[2] Stankovic, R. S.; Stankovic, M.; Jankovic, D., *Spectral Transforms in Switching Theory, Definitions and Calculations*, Nauka, Belgrade, 1999.

[3] Somenzi, F., *CUDD Rerease 1.1.1, 1996*

[4] Brace, K.S., Rudell, R. L., Bryant, R. E., "Efficient implementation of a BDD package", In *Design Automation Conference,* San Francisko, Juny 1991, 417-421.

[5] Hett, A., Drechsler, R., Becker, B., *The DD Package PUMA – An Online Documentation,* http://www.informatic.uni-freiburg.de/FREAK/puma/puma.htm, 1996.

[6] Bryant, R. E., "Graph-based algorithms for Boolean functions manipulation", *IEEE Trans. on Computers,* Vol. C-35, No. 8, August 1986, 677-691.

[7] Miller, D. M., Drechsler, R., "Implementing a multiple-valued decision diagram package", *Proc. 28th Int. Symp. on Multiple-Valued Logic*, Fukuoka, Japan, 1998, 52-57.

[8] Miller, D. M., Drechsler, R., "On the construction of Muliple-Valued Decision Diagrams", *Proc. 32d Int. Symp. on Multiple-Valued Logic*, Boston, USA, 2002, 245-253.

[9] Drechsler, R., Jankovic, D., Stankovic, R. S., "Generic implementation of DD Packeges in MVL" *Proc. EURIMICRO '99,* Milano, 1999. 352-358.

[10] Stojkovic, S., Jankovic, D., "DD Package Specification Language", ETRAN'2002, Banja Vrucica, Yugoslavia, june 2002, (in Serbian).