DDDMP: Decision Diagram DuMP package
Release 2.0

Gianpiero Cabodi Stefano Quer

Politecnico di Torino
Dip. di Automatica e Informatica
Corso Duca degli Abruzzi 24
I–10129 Turin, ITALY
E-mail: {cabodi,quer}@polito.it

1 Introduction

The DDDMP package defines formats and rules to store DD on file. More in particular it contains a set of functions to dump (store and load) DDs and DD forests on file in different formats.

In the present implementation, BDDs (ROBDDs) and ADD (Algebraic Decision Diagram) of the CUDD package (version 2.3.0 or higher) are supported. These structures can be represented on files either in text, binary, or CNF (DIMACS) formats.

The main rules used are following rules:

- A file contains a single BDD/ADD or a forest of BDDs/ADD, i.e., a vector of Boolean functions.
- Integer indexes are used instead of pointers to reference nodes. BDD/ADD nodes are numbered with contiguous numbers, from 1 to NNodes (total number of nodes on a file). 0 is not used to allow negative indexes for complemented edges.
- A file contains a header, including several informations about variables and roots of BDD functions, then the list of nodes. The header is always represented in text format (also for binary files). BDDs, ADDs, and CNF files share a similar format header.
- BDD/ADD nodes are listed following their numbering, which is produced by a post-order traversal, in such a way that a node is always listed after its Then/Else children.

In the sequel we describe more in detail the different formats and procedures available. First of all, we describe BDDs and ADDs formats and procedure. Secondly, we concentrate on CNF files, i.e., how to translate a BDD or a forest of BDDs into a CNF formula and vice-versa.

2 BDD and ADD Support

In this section we describe format and procedure regarding BDDs and ADDs. We specifically refer to BDDs in the description as ADD may be seen as an extension and will be described later. First of all, we concentrate on the format used to store these structure, then we describe the procedure available to store and load them.

2.1 Format

BDD dump files are composed of two sections: The header and the list of nodes. The header has a common (text) format, while the list of nodes is either in text or binary format. In text format nodes are represented with redundant informations, where the main goal is readability, while the purpose of binary format is minimizing the overall storage size for BDD nodes. The header format is kept common to text and binary formats for sake of simplicity: No particular optimization is presently done on binary file headers, whose size is by far dominated by node lists in the case of large BDDs (several thousands of DD nodes).

2.1.1 Header

The header has the same format both for textual and binary dump. For sake of generality and because of dynamic variable ordering both variable IDs and permutations[1] are included. Names are optionally listed for input variables and for the stored functions. New auxiliary IDs are also allowed. Only the variables in the true support of the stored BDDs are listed. All information on variables (IDs, permutations, names, auxiliary IDs) sorted by IDs, and they are restricted to the true support of the dumped BDD, while IDs and permutations are referred to the writing BDD manager. Names can thus be sorted by variable ordering by permuting them according to the permutations stored in the file.

As an example, the header (in text mode) of the next state functions of circuit s27 follows:

```
.ver DDDMP-2.0
.mode A
.varinfo 3
.dd s27-delta
.nnodes 16
.nvars 10
.nsuppvars 7
.varnames G0 G1 G2 G3 G5 G6 G7
.orderedvarnames G0 G1 G2 G3 G5 G6 G7
.ids 0 1 2 3 4 5 6
.permids 0 1 2 3 5 7 9
.auxids 1 2 3 4 5 6 7
.nroots 3
.rootids 6 -13 -16
.rootnames G10 G11 G13
```

The lines contain the following informations:

[1] The permutation of the i-th variable ID is the relative position of the variable in the ordering.

- Dddmp version information.
- File mode (A for ASCII text, B for binary mode).
- Var-extra-info (0: variable ID, 1: permID, 2: aux ID, 3: variable name, 4 no extra info).
- Name of dd (optional).
- Total number of nodes in the file.
- Number of variables of the writing DD manager.
- Number of variables in the true support of the stored DDs.
- Variable names (optional) for all the variables in the BDD/ADD support.
- Variable names for all the variables in the DD manager during the storing phase. Notice that this information was not stored by previous versions of the same tool. Full backward compatibility is guaranteed by the present implementation of the tool.
- Variable IDs.
- Variable permuted IDs.
- Variable auxiliary IDs (optional).
- Number of BDD roots.
- Indexes of BDD roots (complemented edges allowed).
- Names of BDD roots (optional).

Notice that a field

```
.add
```

is present after the dddmp version for files containing ADDs.

2.1.2 Text Format

In text mode nodes are listed on a text line basis. Each a node is represented as

```
<Node-index> [<Var-extra-info>] <Var-internal-index>
             <Then-index> <Else-index>
```

where all indexes are integer numbers.

This format is redundant (due to the node ordering, <Node-index> is and incremental integer) but we keep it for readability.

<Var-extra-info> (optional redundant field) is either an integer (ID, PermID, or auxID) or a string (variable name). <Var-internal-index> is an internal variable index: Variables in the true support of the stored BDDs are numbered with ascending integers starting from 0, and following the variable ordering. <Then-index> and <Else-index> are signed indexes of children nodes.

In the following, we report the list of nodes of the s27 next state functions (see previous header example):

```
.nodes
1 T 1 0 0
2 G7 6 1 -1
3 G5 4 1 2
4 G3 3 3 1
5 G1 1 1 4
6 G0 0 5 -1
7 G6 5 1 -1
8 G5 4 1 -7
9 G6 5 1 -2
10 G5 4 1 -9
11 G3 3 10 8
12 G1 1 8 11
13 G0 0 5 12
14 G2 2 1 -1
15 G2 2 1 -2
16 G1 1 14 15
.end
```

The list is enclosed between the `.nodes` and `.end` lines. First node is the one constant, each node contains the optional variable name.

For ADDs more than one constant is stored in the file. Each constant has the same format we have just analyzed for the BDD but the represented value is stored as a float number.

2.1.3 Binary Format

The binary format is not allowed for ADDs. As a consequence we concentrate only on BDDs in this section. In binary mode nodes are represented as a sequence of bytes, encoding tuples

```
<Node-code>
[<Var-internal-info>]
[<Then-info>]
[<Else-info>]
```

in an optimized way. Only the first byte (code) is mandatory, while integer indexes are represented in absolute or relative mode, where relative means offset with respect to a Then/Else node info. The best between absolute and relative representation is chosen and relative 1 is directly coded in <Node-code> without any extra info. Suppose Var(NodeId), Then(NodeId) and Else(NodeId) represent infos about a given node. <Node-code> is a byte which contains the following bit fields (MSB to LSB)

- Unused : 1 bit
- Variable: 2 bits, one of the following codes
  - DDDMP_ABSOLUTE_ID: Var(NodeId) is represented in absolute form as <Var-internal-info> = Var(NodeId) follows (absolute info)

- DDDMP_RELATIVE_ID: Var(NodeId) is represented in relative form as <Var-internal-info¿ = Min(Var(Then(NodeId)),Var(Else(NodeId)))-Var(NodeId)
- DDDMP_RELATIVE_1: the field <Var-internal-info> does not follow, because Var(NodeId) = Min(Var(Then(NodeId)),Var(Else(NodeId)))-1
- DDDMP_TERMINAL: Node is a terminal, no var info required

• T : 2 bits, with codes similar to V

- DDDMP_ABSOLUTE_ID: <Then-info> is represented in absolute form as <Then-info> = Then(NodeId)
- DDDMP_RELATIVE_ID: Then(NodeId) is represented in relative form as <Then-info> = Nodeid-Then(NodeId)
- DDDMP_RELATIVE_1: no <Then-info> follows, because Then(NodeId) = NodeId-1
- DDDMP_TERMINAL: Then Node is a terminal, no info required (for ROBDDs)

• Ecompl : 1 bit, if 1 means that the else edge is complemented

• E : 2 bits, with codes and meanings as for the Then edge

DD node codes are written as one byte. <Var-internal-index>, <Then-index>, <Else-index> (if required) are represented as unsigned integer values on a sufficient set of bytes (MSByte first).

Integers of any length are written as sequences of ”linked” bytes (MSByte first). For each byte 7 bits are used for data and one (MSBit) as link with a further byte (MSB = 1 means one more byte).

Low level read/write of bytes filters <CR>, <LF> and <ctrl-Z> through escape sequences.

2.2 Implementation

Store and load for single Boolean functions and arrays of Boolean functions are implemented. Moreover, the current presentation includes functions to retrieve variables names, auxiliary identifierss, and all the information contained in the header of the files. This information can be used as a pre-processing step for load operations. These functions allow to overcome few limitations of the previous implementations.

2.2.1 Storing Decision Diagrams

Dddmp_cuddBddStore and *Dddmp_cuddBddArrayStore* are the two store functions, used to store single BDD or a forest of BDDs, respectively. Internally, *Dddmp_cuddBddStore* builds a dummy 1 entry array of BDDs, and calls *dddmp_cuddBddArrayStore*.

Since conversion from DD pointers to integer is required, DD nodes are temporarily removed from the unique hash. This makes room in their *next* field to store node IDs. Nodes are re-linked after the store operation, possible in a modified order. Dumping is either in text or binary form. Both a file pointer (*fp*) and a file name (*fname*) are provided as inputs parameters to store routines. BDDs are stored to the already open file *fp*, if not NULL. Otherwise file whose name is *fname* is opened. This is intended to allow either DD storage within files containing other data, or to specific files.

2.2.2 Loading Decision Diagrams

Dddmp_cuddBddLoad and *Dddmp_cuddBddArrayLoad* are the load functions, which read a BDD dump file.

Following the store function, the main BDD load function, *Dddmp_cuddBddLoad*, is implemented by calling the main BDD-array loading function *Dddmp_cuddBddArrayLoad*. A dynamic vector of DD pointers is temporarily allocated to support conversion from DD indexes to pointers.

Several criteria are supported for variable match between file and DD manager, practically allowing variable permutations or compositions while loading DDs. Variable match between the DD manager and the BDD file is optionally based in *IDs*, *perids*, *varnames*, *varauxids*; also direct composition between *IDs* and *composeids* is supported. The *varmatchmode* parameter is used to select mathing mode. More in detail, two match modes use the information within the DD manager, the other ones use extra information, which support any variable remap or change in the ordering.

- varmatchnode=DDDMP_VAR_MATCHIDS allows loading a DD keeping variable IDs unchanged (regardless of the variable ordering of the reading manager.

  This is useful, for example, when swapping DDs to file and restoring them later from file, after possible variable reordering activations.

- varmatchnode=DDDMP_VAR_MATCHPERMIDS is used to allow variable match according to the position in the ordering (retrieved by array of permutations stored on file and within the reading DD manager). A possible application is retrieving BDDs stored after dynamic reordering, from a DD manager where all variable IDs map their position in the ordering, and the loaded BDD keeps the ordering as stored on file.

- varmatchnode=DDDMP_VAR_MATCHNAMES requires a not NULL varmatchmodes parameter; this is a vector of strings in one-to-one correspondence with variable IDs of the reading manager. Variables in the DD file read are matched with manager variables according to their name (a not NULL varnames parameter was required while storing the DD file). The most common usage of this feature is in combination with a variable ordering stored on a file and based on variables names. Names must be loaded in an array of strings and passed to the DD load procedure.

- varmatchnode=DDDMP_VAR_MATCHIDS has a meaning similar to DDDMP_VAR_MATCHNAMES but integer auxiliary IDs are used instead of strings. The additional not NULL varmathauxids parameter is needed.

- varmatchnode=DDDMP_VAR_COMPOSEIDS, uses the additional varcomposeids parameter as an array of variable IDs to be composed with IDs stored in file.

2.2.3 DD Load/Store and Variable Ordering

Loading of Decision Diagrams from file supports different variables ordering strategies, as already pointed out in the previous section. This allows or example storing different BDDs each with its own variable ordering, and to merge them within the same DD manager by means of proper load operations. We suggest using DDDMP_VAR_MATCHIDS whenever IDs keeps on representing the same entities while changing variable ordering. If this is not true, variable names (if available) or auxiliary

IDs are a good way to represent invariant attributed of variables across several runs with different orderings. DDDMP_VAR_COMPOSEIDS is an alternative solution, that practically corresponds to cascading DDDMP_VAR_MATCHIDS and variable composition with a given array of new variables.

3 CNF Support

3.1 Format

Given a BDD representing a function f, we develop three basic possible ways to store it as a CNF formula. In each method the set of clauses is written after an header part. Only the text format is allowed.

3.1.1 Header

The header part of each CNF file has basically the same format analyzed for the BDD/ADD files. For example the `.rootids` line indicates the beginning of each CNF formula represented by a single BDD. To be compatible with the DIMACS format each header line start with the character "c" to indicate a comment.

3.1.2 Text Format

The first method, which we call **Single-Node-Cut**, models each BDD nodes, but the ones with both the children equal to the constant node 1, as a multiplexer. Each multiplexer has two data inputs (i.e., the node children), a selection input (i.e., the node variable) and one output (i.e., the function value) whose value is assigned to an additional CNF variable. The final number of variables is equal to the number of original BDD variables plus the number of "internal" nodes of the BDD.

The second method, which we call **Maxterm-Cut**, create clauses starting from f corresponds to the off-set (i.e., all the paths-cubes from the root node to the terminal 0) of the function f. Within the BDD for f, such clauses are found by following all the paths from the root node of the BDD to the constant node 0. The final number of variables is equal to the number of original BDD variables.

The third method, which we call **Auxiliary-Variable-Cut**, is a trade-off between the first two strategies. Internal variables, i.e., cutting points, are added in order to decompose the BDD into multiple sub-trees each of which is stored following the second strategy. The trade-off is guided by the cutting point selection strategy, and we experiment with two methodologies. In the first method, a new CNF variable is inserted in correspondence to the shared nodes of the BDD, i.e., the nodes which have more than one incoming edge. This technique, albeit optimizing the number of literals stored, can produce clauses with a high number of literals[2]. To avoid this drawback, the second method, introduces all the previously indicated cutting points more the ones necessary to break the length of the path to a maximum (user) selected value.

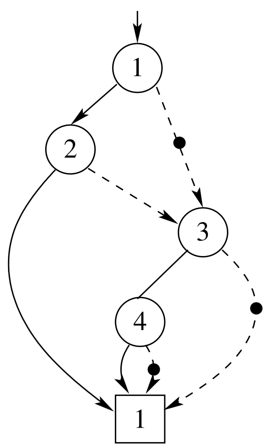
Actually, all the methods described above can be re-conducted to the basic idea of possibly breaking the BDD through the use of additional cutting variables and dumping the paths between the root of the BDD, the cutting variables and the terminal nodes. Such internal cutting variables are added always (for each node), never or sometimes respectively.

[2]This value is superiorly limited by the number of variables of the BDD, i.e., the longest path from the root to the terminal node.

While the *Single-Node-Cut* method minimizes the length of the clauses produced, but it also requires the higher number of CNF variables, the *Maxterm-Cut* technique minimizes the number of CNF variables required. This advantage is counter-balanced by the fact that in the worst case the number of clauses, as well as the total number of literals, produced is exponential in the BDD size (in terms of number of nodes). The application of this method is then limited to the cases in which the "off-set" of the represented function f has a small cardinality. The *Auxiliary-Variable-Cut* strategy is a trade-off between the first two methods and the ones which gives more compact results. As a final remark notice that the method is able to store both monolithic BDDs and conjunctive forms. In each case we generate CNF files using the standard DIMACS format.

Example 1 *Figure 1 shows an example of how our procedure works to store a small monolithic BDD. Figure 1(a) represents a BDD with* 4 *nodes. BDD variables are named after integer numbers ranging from* 1 *to* 4*, to have an easy-to-follow correspondence with the CNF variables. Figure 1(b), (c) and (d) show the corresponding CNF representations generated by our three methods. As in the standard format* p *indicates the total number of variables used (*4 *is the minimum value as the BDD itself has* 4 *variables), and* cnf *the total number of clauses.*

As a final remark notice that for this specific example the "Maxterm-Cut" approach is the one which gives the most compact CNF representation but also the clause with the largest number of literals (4*).*



(a)

```
p cnf 7 11
-5 3 0
-5 4 0
5 -3 -4 0
6 -2 0
6 -5 0
-6 2 5 0
7 1 5 0
-7 1 -5 0
7 -1 -6 0
-7 -1 6 0
7 0
```

(b)

```
p cnf 4 3
1 -3 -4 0
-1 2 3 0
-1 2 -3 4 0
```

(c)

```
p cnf 5 5
-5 1 0
5 -1 2 0
-3 -4 5 0
3 -5 0
-3 4 -5 0
```

(d)

Figure 1: (a) BDD; (b) "Single-Node-Cut" format; (c) "Maxterm-Cut" format; (d) "Auxiliary-Variable-Cut" Format.

3.2 Implementation

Store and Load for a single BDD or a forest of BDDs is currently implemented.

3.2.1 Storing Decision Diagrams as CNF Formulas

As far as the storing process is concerned three possible formats are available:

- DDDMP_CNF_MODE_NODE: store a BDD by introducing an auxiliary variable for each BDD node

- DDDMP_CNF_MODE_MAXTERM: store a BDD by following the maxterm of the represented function
- DDDMP_CNF_MODE_BEST: trade-of between the two previous solution, trying to optimize the number of literals stored.

See procedures Dddmp_cuddBddStoreCnf (to store a single BDD as a CNF formula) and Dddmp_cuddBddArrayStoreCnf (to store an array of BDDs as a CNF formula).

3.2.2 Loadinf CNF Formulas as BDDs

As far as the loading process is concerned three possible formats are available:

- DDDMP_CNF_MODE_NO_CONJ: Return the Clauses without Conjunction
- DDDMP_CNF_MODE_NO_QUANT: Return the sets of BDDs without Quantification
- DDDMP_CNF_MODE_CONJ_QUANT: Return the sets of BDDs AFTER Existential Quantification

See procedures Dddmp_cuddBddLoadCnf (to load a CNF formula as a single BDD) and Dddmp_cuddBddArrayLoadCnf (to load a CNF formula as an array of BDDs). See also Dddmp_cuddHeaderLoadCnf to load the header of a CNF file to gather information on the saved structure.

4 Test Program and Regression Tests

The *testddmp.c* file, provided with this distribution, exemplifies some of the above features. Moreover, in the *exp* experiments a few scripts, named *test¡n¿.script* are available for a sanity check of the tool and to take a look at some runs exemplification.

5 Documentation

For further documentation on the package see the on-line documentation automatically created from the source code files.

6 Acknowledgments

We are particular indebted with Fabio Somenzi, for discussions, advice, and for including the DDDMP package into the CUDD distribution. We also thank all the user of the package for their useful indication and comments on the it.

7 FTP Site

The package is singularly available from:

```
site: ftp.polito.it
user: anonymous
directory: /pub/research/dddmp
```

or directly from the author WEB pages:

```
WWW: http://www.polito.it/~{cabodi,quer}
```

8 Feedback

Send feedback to:


```
Gianpiero Cabodi & Stefano Quer
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi, 24
I-10129 Torino
Italy
E-mail: {cabodi,quer}@polito.it
WWW: http://www.polito.it/~{cabodi,quer}
```