# Biddy — a multi-platform academic BDD package

Robert Meolic
Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia
Email: meolic@uni-mb.si

**Abstract — Biddy is a BDD package under GPL, developed at the University of Maribor. It uses ROBDDs with complement edges, as described in the paper K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, 1990. Compared to other available BDD packages, Biddy's most distinguishing features are its specially designed C interface and an original implementation of automatic garbage collection. More generally, the Biddy project is not only concerned with the computer library, but also offers a demo application for the visualization of BDDs, called BDD Scout. The whole project is oriented towards a readable and comprehensible source code in C, which can be compiled unchanged on different platforms, including GNU/Linux and MS Windows.**

**Index Terms — Boolean algebra, binary decision diagram, symbolic manipulation of Boolean functions, formal methods, free software**

## I. Introduction

**Boolean algebra** is a mathematical structure applied within many engineering and scientific fields, especially those concerned with electronics, computers, and communications. The **Binary Decision Diagram** (BDD) is a data structure for representing Boolean functions. This representation has gained popularity because it is canonical, and thus tautology checking, satisfiability checking, and equivalence checking can be done in a constant time (after the BDD has been created). Moreover, it is a compact representation of many of those Boolean functions that arise during practical problems.

Binary decision diagrams are not just another theory. Many applications are heavily based on Boolean algebra and BDDs. Some successful examples are hardware design methods, e.g. logic synthesis [1], formal methods concerned with testing and verifying systems, e.g. symbolic model checking [2], and methods for knowledge representation and discovery e.g. the rough-set theory [3]. Recently, D. E. Knuth included an extensive section about BDDs in his famous monograph The Art of Computer Programming [4], where it states that "(BDDs) burst on the scene in 1986, long after old-timers like me thought that we had already seen all of the basic data structures that would ever prove to be of extraspecial importance" and that "(BDDs) have given me many more surprises than anything else so far". And last, but not least, a pioneering paper on BDD algorithms [5] is one of the most cited paper in the history of computer science!

The **BDD package** is computer software, more precisely a sort of mathematical library, which allows other programs to create and manipulate Boolean functions by using BDDs. Many different BDD packages are available, usually as a piece of free software. Among others, Wikipedia [6] lists ABCD [7], BuDDy [8], CAL [9], CMU BDD [10], CUDD [11], JDD [12], and Biddy [13], the package that this paper is about. Biddy is a minimalistic BDD package that includes only the necessary functions. It uses ROBDDs with complement edges, as described in [14]. Biddy can be distinguished mostly by its specially designed C interface and an original garbage collection that is not based on a classic reference count.

**Biddy** is based on a BDD package written at the University of Maribor in 1992 [15][16]. Hence, it can be categorized as one of the oldest BDD packages around (not to brag but even D. E. Knuth remembers he "didn't actually learn about binary decision diagrams until 1995 or so" [4]). In Maribor, the original BDD package was written in Pascal and ran on VAX 4000-600. The history of its further development can be summarized as follows. In 1994, A. Časar rewrote the package in C. In 1999 R. Meolic included it into EST, a tool for the formal verification of systems [17]. In 2006, the name Biddy first appeared. Separate library was formed in 2007. So far Biddy has been developed as an academic software (as defined in [18]), thus clean implementation has been preferred over efficiency optimization .

In 2003, whilst being a part of EST, Biddy was included within a survey of 13 BDD packages [19] and was one of two awarded mark A for code quality. No matter how subjective this classification is (the other A-graded package is the one maintained by the author of the survey :-) it reflects the main goal of Biddy to promote a readable and comprehensible source-code. The orientation towards educational purposes is also supported by the licence, which is GPL (published by FSF [20]), and the ability to be compiled and used on different platforms, including GNU/Linux and MS Windows.

Furthermore, this paper is organized as follows. Section II provides some basic terms and definitions about BDDs. In Section III, Biddy is briefly described from the user's point of view. Section IV gives details about the implementation. Section V introduces BDD Scout, a demo application for the visualization of BDDs which is being developed as part of the Biddy project. The conclusion summarizes the current state of the project.


doi:10.4304/jsw.7.6.1358-1366

## II. Binary Decision Diagrams

The term Binary Decision Diagram was coined by S. B. Akers in 1978 [21]. In his paper Akers predicted various applications for BDDs but did not describe any useful implementation. His work was extended in 1986 by R. E. Bryant who introduced Ordered BDDs and also showed computer algorithms for their manipulation [5]. In 1990, Reduced Ordered BDDs were invented together with efficient recursive algorithms [14][22], and afterwards the activities involving BDDs quickly became widespread [23][24][25][26][4].

A Binary Decision Diagram is a directed, acyclic graph with one root. Its leaves are called terminal nodes (also sink nodes) and are labelled either with '0' or '1'. All other nodes are called non-terminal nodes (also branch nodes) and are labelled by variables. Every non-terminal node has exactly two descendants called 'else' (the left) and 'then' (the right) edges, respectively. Boolean functions are associated with edges. A Boolean function represented by an edge is recursively calculated as

$$F = \bar{v} \cdot E + v \cdot T = ITE(v, T, E), \qquad (1)$$

where $v$ is a variable in the root (also called top variable), $E$ is a Boolean function represented by the root's 'else' edge, and $T$ is a Boolean function represented by the root's 'then' edge.

An Ordered Binary Decision Diagram (OBDD) is a BDD where variables occur along every path from the root to a leaf in strictly ascending order, with regard to fixed ordering. Algorithms appear to be much more efficient if they can assume the same variable order for the all involved OBDDs. Moreover, the size of an OBDD heavily depends on its variable order. An OBDD is a Reduced Ordered Binary Decision Diagram (ROBDD) if it contains neither isomorphic subgraphs nor nodes with isomorphic descendants. The most important property of ROBDD is the canonicity of the representation. The other one is node sharing (requires the same ordering of all ROBDDs), i.e. when more than one Boolean function is simultaneously represented, the merging of isomorphic subgraphs is applied between all of them. Hence, every node can belong to more functions.

An important extension of ROBDDs is the introduction of complemented edges. Every edge has an additional field (a single bit is sufficient), which is used to distinguish between the regular and complemented edges. A complemented edge complements the represented Boolean function. In this way, there is no need to keep terminal node '0' in the graph. It is replaced by the complement edge to terminal node '1'. The usual way to maintain the canonicity is to keep 'then' edges regular in every node. Examples of a ROBDD with and without complemented edges are given in Fig. 1.

## III. Manipulation Of Boolean Functions Using Biddy Library

Biddy is written in C. The source code is purified and improved in such a way that it could be compiled with a C or C++ compiler without errors and warnings. Precompiled packages include dynamically linked library (i.e. `*.so` on GNU/Linux, `*.dll` on MS Windows, `*.dylib` on Mac OS X), and the appropriate C header `biddy.h`[1]. Currently, there are no interfaces for other programming languages. The supplied C header is quite small because of nice abstractions. Its data declaration part is given in Fig. 2 (this is real code, only some comments have been removed).

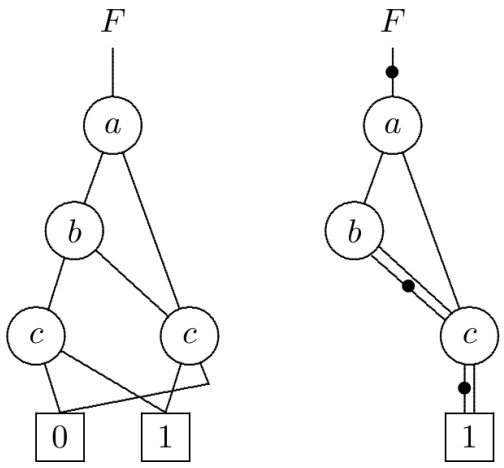


Figure 1. ROBDD without complemented edges and ROBDD with complemented edges for Boolean function

$$a \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + b \cdot \bar{c}$$

```
/* Constant definitions */
#define FALSE 0
#define TRUE !FALSE

/* Macro definitions */
#define Biddy_isEqv(f,g) \
  (((f).p == (g).p) && \
  ((f).mark == (g).mark))
#define Biddy_isTerminal(f) \
  ((f).p == biddy_termTrue.p)
#define Biddy_isNull(f) \
  ((f).p == biddy_termNull.p)

/* Type declarations */
typedef char Biddy_Boolean;
typedef char *Biddy_String;
typedef int Biddy_Variable;
typedef void (*Biddy_VoidFunction)();

/* Structure declarations */
typedef struct Biddy_Edge {
  void *p;
  Biddy_Boolean mark;
} Biddy_Edge;

/* Variable declarations */
EXTERN Biddy_Edge biddy_one;
EXTERN Biddy_Edge biddy_zero;
EXTERN Biddy_Edge biddy_null;
```

Figure 2. Data declaration part of `biddy.h`

[1] The header is included in development version, only.

Constant and macro definitions are self-explanatory and so are types `Biddy_Boolean` and `Biddy_String`. `Biddy_Variable` is a type used for refering to variables in BDD. The variables are stored in a table and referenced by indices. For the user, it is unimportant, how the variables are stored, they could be, for example, also stored in a binary tree and referenced by pointers (assuming that pointers are compatible with integers). BDD variable with the name '1' is created during the initialization of Biddy package (performed by function `Biddy_Init`), whilst all other BDD variables must be created explicitely with function `Biddy_FoaTerminal`. Type `Biddy_VoidFunction` is used to declare functions that extend Biddy's capability of memory management. These special functions are started each time Biddy tries to free redundant memory, and are intended to delete invalid entries in user caches.

The most important part of Biddy's external header is its declaration of structure `Biddy_Edge`. Each edge in a BDD corresponds to one Boolean function. `Biddy_Edge` consists of a pointer to a node and an optional mark. Void pointer is used o achieve separation of interface and implementation. Although this is not strict encapsulation (C does not have such a mechanisms as, for example C++), the user is encouraged to use the provided API functions, only, and not to travel through the BDD by direct usage of pointers nor rely on the node's internal structure. Among others, `Biddy_GetThen`, `Biddy_GetElse`, and `Biddy_GetVariable` are functions available in API.

The variables are defined by the keyword `EXTERN` which is in fact a macro defined before the data declaration part. On GNU/Linux and Mac OS X systems, this macro is simply expanded to keyword `extern` but on MS Windows it is expanded to a certain code which is required for the declaration of external symbols in DLL files (moreover, it is a different expansion whether you build a DLL or just use it). The edges `biddy_one` and `biddy_zero` represent Boolean functions 1 and 0, respectively, whilst `biddy_null` represents a non-valid (null) edge.

Biddy is capable of all the typical operations regarding Boolean functions.

- **Tautology checking.** For the given Boolean function $G$ check if $G=1$. Macro `Biddy_isEqv` is suitable for performing this operation.
- **Equivalence checking.** For the given Boolean functions $G_1$ and $G_2$ check if $G_1=G_2$. Again, macro `Biddy_isEqv` is suitable for performing this operation.
- **Complement.** For the given Boolean function $G$ calculate $\overline{G}$. Function `Biddy_NOT` can be used to calculate complement.
- **Binary operations (AND, OR, etc.).** For the given Boolean functions $G_1$ and $G_2$ and the given binary operation $\times$, calculate $G_1 \times G_2$. Function `Biddy_ITE` is capable of calculating all binary operations on Boolean functions.
- **Restriction.** For the given Boolean function $G$, given variable $x$, and given constant $c \in \{0,1\}$ calculate $G|_{x=c}$. Function `Biddy_Restrict` calculates restriction.
- **Composition.** For the given Boolean functions $G$ and $H$ and given variable $x$, calculate $G|_{x=H}$. Function `Biddy_Compose` calculates composition.
- **Existential and universal quantification.** For the given Boolean function $G$ and given variable $x$, calculate $\exists x.G$ and $\forall x.G$. Functions `Biddy_E` and `Biddy_A` calculate the existential and universal quantifications, respectively.

As an example, a simple program using Biddy library is given in Fig. 3. It calculates the 13-th minterm of the Boolean function $F(a,b,c,d)$. On the Ubuntu (GNU/Linux) system, where Biddy library was installed from the available deb package, this program is compiled with the following command:

```
gcc -DUNIX -o mint13 mint13.c -lbiddy
```

On MS Windows (either 32 or 64 bit version) you have to use:

```
gcc -DWIN32 -DUSE_BIDDY -o mint13.exe
mint13.c -lbiddy
```

```
#include <biddy.h>

int createMinterm13() {
  Biddy_Edge a,b,c,d,TMP1,TMP2,F;

  a = Biddy_FoaTerminal("a");
  b = Biddy_FoaTerminal("b");
  c = Biddy_FoaTerminal("c");
  d = Biddy_FoaTerminal("d");
  TMP1 = Biddy_ITE(a,b,biddy_zero);
  TMP2 = Biddy_ITE(
    Biddy_NOT(c),d,biddy_zero);
  F = Biddy_ITE(TMP1,TMP2,biddy_zero);
  printf("F has %d nodes.\n",
    Biddy_NodeNumber(F));
}

int main() {
  Biddy_About();
  Biddy_Init();
  createMinterm13();
  Biddy_Exit();
  return 0;
}
```

Figure 3. A simple program using Biddy library

## IV. Implementation Details

Biddy sources consist of files `biddy.h`, `biddyInt.h`, `biddyMain.c`, `biddyStat.c`, and those used for compiling and packaging. File `biddyMain.c` includes definitions of constants, variables and functions, except those used for statistics; these have been separated and put into file `biddyStat.c`. Coding style, i.e. the naming of func-

tions, macros, and variables, the style of documentation, the organization and naming of source files etc. respects conventions from [27][2]:

- exported functions, macros, types, and structures have a prefix `Biddy_`, e.g. `Biddy_SimpleFunction`, they are all defined in file `biddy.h` (also called external header)
- internal functions, macros, types, and structures have a prefix `Biddy`, e.g. `BiddySimpleFunction`, they are all defined in file `biddyInt.h` (also called internal header)
- local functions, macros, types, and structures (visible to one file only) do not have a prefix, e.g. `SimpleFunction`,
- exported variables have a prefix `biddy_`, e.g. `biddy_simpleVariable`,
- internal variables have a prefix `biddy`, e.g. `biddySimpleVariable`,
- local variables (visible to one function or one file only) do not have a prefix, e.g. `simpleVariable`.

The main data structures are the Node Table, ITE Cache, EAX Cache, and some special lists for memory management. They are all declared in file `biddyInt.h`, as given in Fig. 4.

**Node Table** is a hash table with chaining. It stores all nodes and ensures a quick search for a node with given variable and references to descendants. It also prevents multiple instances of nodes with the same variable and references to descendants. Adding and searching for nodes are both done by function `Biddy_FoaNode` using the Find-Or-Add principle. The variable and descendants of a node are referenced by elements 'v', 'f', and 't', respectively. The chains of nodes are bi-directional (elements 'prev' and 'next') to allow for the deletion of directly addressed nodes. If a particular chain is empty, the hash table a contains null pointer. Otherwise, it contains a pointer to the first node in the chain. The first element of each chain has a special value for its 'prev' element (it is a pointer but not one to a regular node) to enable the correct relinking when it is being deleted. Moreover, the tricky node deleting procedure requires that 'prev' and 'next' are the first and second elements in the structure `BiddyNode`, respectively, and that the hash function used to spread the nodes across the table never returns zero.

ITE Cache and EAX Cache are fixed-size hash tables. **ITE Cache** stores the arguments and results from the performed ITE operations. All binary operations are implemented via the ITE operation, as shown in Fig. 5. This is efficient not only because one algorithm is sufficient for all operations but also because, in this way, all operations share the same cache. In order to avoid distinguishing the same calls (e.g. $\mathrm{ITE}(f,g,0)=\mathrm{ITE}(g,f,0)$), the arguments given to the ITE operation are transformed into a predefined normal form before being stored in the ITE Cache. This normalization is performed according to the rules given in [14]. The algorithm for operation ITE is shown in Fig. 6. **EAX Cache** stores the parameters and results of the performed existential and universal quantifications. One cache is efficiently used to store the results from both operations ($\exists a.f=\neg\forall a.\neg f$). Indeed, any cache is limited and if a certain result has been deleted (they are in fact overwritten), then the calculation has to be performed again.

```
/* NODE TABLE: a hash table with chaining */
typedef struct BiddyNode {
  struct BiddyNode *prev, *next, *list;
  Biddy_Variable v;
  Biddy_Edge f, t;
  int count;
} BiddyNode;
typedef struct {
  BiddyNode **table;
  BiddyNode **blocktable;
  int size;
  int generated, blocknumber, max, num,
      numf, foa, compare, add, garbage;
} BiddyNodeTable;

/* VARIABLE TABLE: dynam. allocated table */
typedef struct {
  Biddy_String name;
  int order;
  Biddy_Edge term;
  Biddy_Boolean value;
} BiddyVariable;
typedef struct {
  BiddyVariable *table;
  int size;
} BiddyVariableTable;

/* CACHE LIST: unidirectional list */
typedef struct BiddyCacheList {
  struct BiddyCacheList *next;
  Biddy_UserFunction gc;
} BiddyCacheList;

/* ITE CACHE: a fixed-size hash table */
typedef struct {
  BiddyNode *f, *g, *h;
  Biddy_Edge result;
  Biddy_Boolean hmark;
  Biddy_Boolean ok;
} BiddyIteCache;
typedef struct {
  BiddyIteCache *table;
  int size;
  int search, find, overwrite;
} BiddyIteCacheTable;

/* EAX CACHE: a fixed-size hash table */
typedef struct {
  BiddyNode *f;
  BiddyVariable v;
  Biddy_Edge result;
  Biddy_Boolean fmark;
  Biddy_Boolean ok;
} BiddyEAxCache;
typedef struct {
  BiddyEAxCache *table;
  int size;
  int search, find, overwrite;
} BiddyEAxCacheTable;
```

Figure 4. Structure declaration part of `biddyInt.h`

[2] These conventions originate in J. Ousterhout's Tcl/Tk Eng. Manual.

| Truth table | Symbol | Boolean base | ITE base |
|---|---|---|---|
| 0000 | 0 | 0 | 0 |
| 0001 | $f \cdot g$ | $f \cdot g$ | ITE( f , g,0) |
| 0010 | $\overline{f \rightarrow g}$ | $f \cdot \bar{g}$ | ITE(g,0, f ) |
| 0011 | f | f | f |
| 0100 | $\overline{f \leftarrow g}$ | $\bar{f} \cdot g$ | ITE( f ,0, g) |
| 0101 | g | g | g |
| 0110 | $f \oplus g$ | $f \cdot \bar{g} + \bar{f} \cdot g$ | ITE( f , $\bar{g}$, g) |
| 0111 | $f + g$ | $f + g$ | ITE( f ,1, g) |
| 1000 | $f \downarrow g$ | $\overline{f + g}$ | ITE( f ,0, $\bar{g}$) |
| 1001 | $f \equiv g$ | $f \cdot g + \bar{f} \cdot \bar{g}$ | ITE( f , g, $\bar{g}$) |
| 1010 | $\bar{g}$ | $\bar{g}$ | ITE(g,0,1) |
| 1011 | $f \leftarrow g$ | $f + \bar{g}$ | ITE(g, f ,1) |
| 1100 | $\bar{f} \rightarrow g$ | $\bar{f}$ | ITE( f ,0,1) |
| 1101 | $f \rightarrow g$ | $\bar{f} + g$ | ITE( f , g,1) |
| 1110 | $f \mid g$ | $\overline{f \cdot g}$ | ITE( f , $\bar{g}$,1) |
| 1111 | 1 | 1 | 1 |

Figure 5. Binary operations on Boolean functions

The efficiency of a BDD package depends heavily on the adequacy of the memory management, i.e. deleting nodes which were created during previous calculations but are no longer needed for further calculations. This operation is called garbage collection. For the deleted node, any record in any cache referencing this node (either as argument or result) is invalid, and may be reused. Biddy has the **Cache List** that keeps references to those functions intended for marking invalid entries in caches. ITE Cache and EAX Cache are registered on the Cache List during initialization, whilst any user-defined cache must be registered using function `Biddy_AddCache`.

The main problem with garbage collection is the detection of unnecessary nodes. Some help from the user is required because, during different calculations, a lot of temporary nodes are created and it is hard for the system to automatically guess which results will be used during further calculations. It could be assumed that all results will be needed in the future but this permissive strategy is, in fact, very poor for the majority of practical applications.

Biddy implements an original method called garbage collection with a formulae counter not used by other BDD packages. It works via an internal global variable `biddyCount` and an element `count` in every node. If a node's `count` is equal to zero then this is a fortified node, if it is equal to the value of `biddyCount` then this is a fresh node, and otherwise it is a bad node. Fortified nodes belong to those results from previous calculations which

```
Biddy_ITE(F,G,H) {
  normalization of arguments F,G,H
  if simple call return result
  if result in ITE Cache return result
  v = the smallest top variable of F,G,H
  T = Biddy_ITE(F,G,H)v=1
  E = Biddy_ITE(F,G,H}v=0
  result = Biddy_FoaNode(v,T,E)
  store F,G,H, and result into ITE Cache
  return result
}
```

Figure 6. Algorithm for ITE

are assumed (by the user) to be needed in the future. Fresh nodes have been created during the current calculation and must remain, at least until the end of this calculation. Bad nodes are safe to delete at any moment. Indeed, all descendants of a fortified node should be fortified and the descendants of a fresh node must not be bad. Biddy offers function `Biddy_isOK` to check whether a particular node is not bad.

When a new node is created, it becomes a fresh node. Moreover, all its descendants are being refreshed, i.e. they are changed into fresh nodes if they are currently bad. Refreshing is done by the recursive function `Biddy_Fresh`. When the calculation is finished, the user can fortify its result, i.e. change all its nodes into fortified ones. Fortifying is done by the recursive function `Biddy_Fortify`. At any time, the user can change all the fresh nodes into bad nodes by calling `Biddy_IncCounter`, which simply increments a variable `biddyCount`. This is usually used to separate different calculations, i.e. to mark all redundant nodes from the already finished calculations.

Function `Biddy_IncCounter` or any other function mentioned in the previous description does not actually start garbage collection, i.e. node deleting. This is started periodically by the system or explicitly by calling function `Biddy_Garbage`. Deleting nodes as often as possible is not the best strategy. A currently unnecessary node may become needed by the very next operation and it would have to be recreated. Hence, Biddy uses the following approach. An amount (a memory block) of BDD nodes is created during initialization. When all these nodes have been used, garbage collection starts to delete unnecessary ones. A new memory block full of BDD nodes is created if none of the nodes can be deleted. References to allocated memory block are stored in the **Memory Block Table** (element `blocktable` in structure `BiddyNodeTable`).

A special multi-purpose pointer `list` is used as part of every node in order to support efficient memory management. When a block of new BDD nodes is created, all the nodes are linked by this pointer into the **List of Free Nodes** (its beginning is referenced by pointer `biddyFreeNodes`). Nodes deleted by garbage collection are not deallocated from memory, they are just returned to this list. In this way, the number of time-consuming requests for allocating and deallocating memory is greatly reduced. When a new node is needed, it is simply taken

from the List of Free Nodes. For the nodes in use, the pointer `list` is reused to link the node to another list called **List of New Nodes** (its beginning and end are referenced by the pointers `biddyFirstNewNode` and `biddyLastNewNode`, respectively). List of New Nodes greatly reduces the time for garbage collection because only nodes from this list are checked and not the whole Node Table. It is even unnecessary to look over the whole list, as from a particular node forward to the end, the list contains only fresh nodes. This arbitrary node is referenced by the pointer `biddyFreshNodes` and this reference is transferred to the end of the list each time `BiddyCount` is incremented. During the garbage collection all fortified nodes are removed from the List of New Nodes and, hence, the next call will not be bothered with them. Indeed, they must be returned to the list if their status is changed from fortified to fresh. For all fortified nodes removed from the List of New Nodes, the pointer `list` is reused once again to connect them to the **List of Fortified Nodes**, which allows successive visiting of them. The algorithm for garbage collection with a formulae counter as used in Biddy, is shown in Fig. 7.

```
GarbageCollection () {
  if some bad nodes could exist {
    do all functions from Cache List
    forall nodes in List of New Nodes {
      if node is fortified
        move to List of Fortified Nodes
      if node is fresh
        do nothing
      if node is bad {
        delete it from Node Table
        move to List of Free Nodes
      }
    }
  }
}
```

Figure 8. Garbage collection in Biddy

If the status of the nodes is not controlled by the user, all nodes will remain fresh forever and the garbage collection simply does nothing. There is nothing wrong with this (for small examples this could even be the most efficient method), but you will be unable to manipulate large Boolean functions in such a way. A simple example of controlling garbage collection is given in Fig. 8. It shows the calculation of function F2, which requires calculation of a temporary function F1. After the construction of F2, function F1 is no longer needed, whilst function F2 itself is supposed to be a useful result. Thus, it is fortified and can be used during the calculation of F3. Function F1 must not be used during the calculation of F3 because garbage collection may delete some nodes of F1 before the calculation of F3 is finished.

Another, more complex example is shown in Fig. 9. Here, function G is being iteratively computed. The temporary functions F1 and F2 are created during each step. The final G may be huge, therefore we allow for the deleting of unnecessary nodes as the calculation goes along. We do not need intermediate results for G, only the final one, and therefore we do not use `Biddy_Fortify` during the calculation.

```
F1 = create a BDD
F2 = create a BDD using F1
Biddy_Fortify(F2);
Biddy_IncCounter();
F3 = create a BDD using F2
Biddy_Fortify(F3);
Biddy_IncCounter();
```

Figure 7. Simple example of controlling garbage collection

```
G = biddy_zero;
while (some condition) {
  Biddy_IncCounter();
  Biddy_Fresh(G);
  F1 = create a BDD using G
  F2 = create a BDD using G
  G = create a BDD using F1 and F2
}
Biddy_Fortify(G);
Biddy_IncCounter();
```

Figure 9. Another example of controlling garbage collection

## V. BDD SCOUT

**BDD Scout** is a tool for the visualization of BDDs (see the screenshot given in Fig. 10). It serves as a example application when demonstrating the capabilities of Biddy. There exist other free BDD visualization tools, for example BDDTCL [28], BDD Visualizer [29], and JADE [30]. BDD Scout has been developed completely independently of them and although the current version is really more a demo rather than a final product, it already includes comparable or even innovative functions.

BDDTCL was an early bird. It has not been updated for a long time and its capabilities are (according to the available screenshot) similar to BDD Scout. BDD Visualizer is a web-based application. It generates PDF documents. It is not as flexible as BDD Scout and the user can not adjust a generated graph nor interactively explore it. JADE is the most sophisticated software from these group. It is implemented in JAVA and produces nice graphics. It allows for the study of different variable ordering algorithms and allows for good navigation possibilities but, other that this, there is nothing spectacular. Being subjective, the output graphs of BDD Scout can sometimes be even more suitable for publications.

BDD Scout consists of two parts.

- The calculation part is written in C and consists of creating, importing, and exporting BDDs, calculating different statistics, and performing benchmarks.
- GUI and the drawing part are extensions of a separately developed Tcl/Tk application bddview, and allows to load and save a graph, adjust the graphical representation, and create PNG image and PDF file (by utilizing ghostscript).

The calculation part currently includes a parser for simple recursive BDD representation and a parser for the prefix form of Boolean functions used in IFIP/ISCAS benchmarks. For the time being, only one benchmark is implemented (the code is maintained in separate files `bddscoutIFIP.c` and `bddscoutIFIP.tcl`).

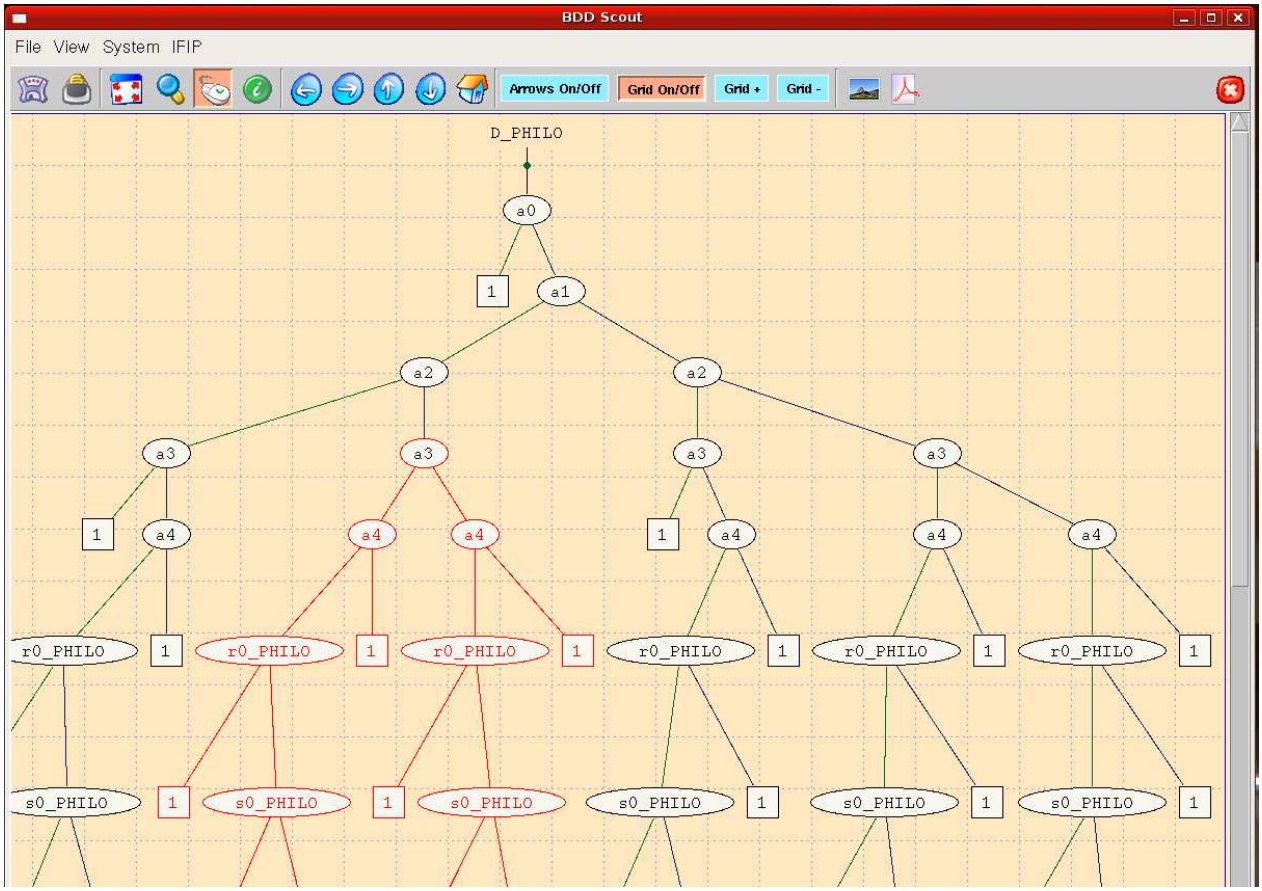


Figure 10. Screenshot of BDD Scout v1.0 (arrows off, grid on, part of the graph is being selected)

A recursive BDD representation recognized by BDD Scout has the following rules:

- The first word is the name of the BDD.
- The second word is the name of the top variable.
- Any variable name is followed by a description of two subgraphs given in parentheses.
- The name of the terminal node is '1'.
- Symbol * is used to denote complement edges.
- Symbol '0' is invalid and should not be used.
- Spacing and indentation are unimportant.

An example of recursive BDD representation is given in Fig. 11. The obtained graph is given in Fig. 14.

The prefix form of Boolean functions (as used in IFIP/ISCAS benchmarks) has the following rules:

- The file is optionally started with the set of variables given in the parentheses (to determine variable ordering).
- There can be many Boolean functions within the same file.
- Spacing and indentation are unimportant but the function's name and symbol = must be given on the same line.
- Supported operators (and also reserved words) are NOT, OR, AND, and EXOR, written either uppercase or lowercase.

The example of prefix form of Boolean functions is given in Fig. 12. The obtained graph is given in Fig. 14 (it is the same as in previous example).

```
Biddy
B (* i (d (1) (y (* 1) (1)))
       (d (y (* 1) (1)) (1)))
  (  i (d (1) (y (* 1) (1)))
       (d (y (* 1) (1)) (1)))
```

Figure 11. A recursive BDD representation supported by BDD Scout

```
(B i d y)
s1 = (or B (not y))
s2 = (or B i d)
s3 = (or B (not i) (not d))
s4 = (or (not B) i (not d) y)
s5 = (or (not B) (not i) d y)
Biddy = (and s1 s2 s3 s4 s5)
```

Figure 12. A prefix representation of Boolean functions supported by BDD Scout

Application **bddview**, which is used in the drawing part of BDD Scout is a single Tcl/Tk script. It is a graph viewer only and does not directly use Biddy or any other BDD package. However, it is not a general graph viewer; it is in many ways optimized to visualize ROBDDs with complement edges. Internally, bddview uses a special textual representation that contains exact coordinates of all nodes. In order to show graph for a particular BDD, BDD Scout in the first place produces an appropriate description in the bddview format. Program **dot** from the graphviz package [31] is utilized to determine position of each node.

The bddview format consists of the following constructs:

- label <n> <name> <x> <y>
- node <n> <name> <x> <y>
- terminal <n> 1 <x> <y>
- connect <n1> <n2> <type>

Here, <n> is the unique number (integer), <name> is a string, <x> and <y> are the coordinates (integer), and <type> is a label, which can be one of the following:

- 's' stands for single line,
- 'si' stands for inverted single line,
- 'r' stands for line to 'then' successor,
- 'l' stands for line to 'else' successor,
- 'li' stands for inverted line to 'else' successor,
- 'd' stands for double line.

The following remarks should be considered:

- (0,0) is the top left-hand corner,
- only one label is supported,
- single line and inverted single line should be used to connect a label and a node, only,
- line to the right successor cannot be inverted,
- when using double line, the line to the left successor is automatically inverted.

An example of bddview format is given in Fig. 13, and the obtained graph is (again) given in Fig. 14.

```
label 0 "Biddy" 100.0 10
node 1 "B" 100 60
node 2 "i" 100 125
node 3 "d" 50 175
terminal 4 1 50 240
node 5 "y" 100 225
terminal 6 1 100 290
node 7 "d" 150 175
terminal 8 1 150 240
connect 0 1 s
connect 1 2 d
connect 2 3 l
connect 2 7 r
connect 3 4 l
connect 3 5 r
connect 5 6 d
connect 7 5 l
connect 7 8 r
```

Figure 13. An example of bddview format

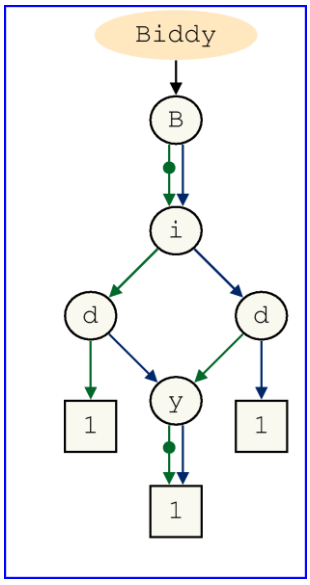


Figure 14. The graph described by textual representations given in Fig. 11, Fig. 12, and Fig. 13

## VI. CONCLUSION

Biddy is a BDD package suitable for educational purposes and also usable in prototyped research tools. It has already been used for quite some time in different projects. Hence, it is very unlikely that it contains major bugs. Nevertheless, it is still being actively developed and upgraded whilst many other free BDD packages are no longer supported by their authors. Building and installation procedures are not ideal, as yet, but precompiled binary packages are being tested for working on various systems. Debian and rpm packaging has been added, recently.

This paper has not compared the effectiveness of Biddy with other popular BDD packages. By considering the goals of the project so far, it was found that such a comparison would be unsuitable, as yet (which does not mean that Biddy is much slower than others). Biddy uses the classical depth-first approach. It also uses common data structures (node table, cache tables), which are implemented straightforwardly without tricky shortcuts. A simple implementation style has been used intentionally to improve the readability of the source code. Some optimizations are planned in the future. In addition to this work, the package will soon be extended with different algorithms for reordering.

The most original part of Biddy is its implementation of garbage collection. This method using a formulae counter is still being investigated and probably, even more advantages will be seen. Memory management is, of course, the main factor of any powerful BDD package and, hence, it will get a lot of attention during the ongoing research.

Biddy's educational role can be very interesting. It can be simply used to illustrate interesting programming paradigm. On the other hand, it can be used to explore the axioms and theorems of Boolean algebra (by the equivalence checking of Boolean formulae). Its more obvious usage is to help students understand the details of the BDD package. And last, but not least, it can be used as an engine for research applications as, for example, demo implementation of Quine-McCluskey minimization [32] or very real-world formal verification of systems, e.g. stuck-at faults detection [33]. Moreover, BDD Scout can be extended in order to show how BDD is constructed, step by step, how BDD representation changes if a different variable ordering is selected, to show the content of the cache, to demonstrate how the cache hits the speed of BDD computation, how garbage collection is triggered, etc.

## REFERENCES


[1] S. N. Yanushkevich, D. M. Miller, V. P. Shmerko, R. S. Stankovic. Decision diagram techniques for micro- and nanoelectronic design handbook. CRC Press, 2006.

[2] C. Baier J.-P. Katoen. Principles of Model Checking. The MIT Press, 2008.

[3] Q. Wei, T. Gu, "Symbolic Representation for Rough Set Attribute Reduction Using Ordered Binary Decision Diagrams", Journal of Software, Vol. 6, No. 6, 2011, pp. 977-984.

[4] D. E. Knuth. Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 2009.

[5] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, Vol. C-35, No. 8, 1986, pp. 677-691. Reprinted in M. Yoeli, Formal Verification of Hardware Design, IEEE Computer Society Press, 1990, pp. 253-267.

[6] Wikipedia: Binary decision diagram. On-line (21/10/2011). http://en.wikipedia.org/wiki/Binary_decision_diagram

[7] ABCD. On-line (21/10/2011). http://fmv.jku.at/abcd/

[8] BuDDy. On-line (21/10/2011). http://buddy.wiki.sourceforge.net/

[9] CAL. On-line (21/10/2011). http://embedded.eecs.berkeley.edu/Research/cal_bdd/

[10] CMU BDD. On-line (21/10/2011). http://www-2.cs.cmu.edu/~modelcheck/bdd.html

[11] CUDD. On-line (21/10/2011). http://vlsi.colorado.edu/~fabio/CUDD/

[12] JDD. On-line (21/10/2011). http://javaddlib.sourceforge.net/jdd/

[13] Biddy. On-line (21/10/2011). http://lms.uni-mb.si/biddy/

[14] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient Implementation of a BDD Package", In: 27. ACM/IEEE Design Automation Conference (DAC'90), 1990, pp. 40-45.

[15] A. Časar, R. Meolic, "Predstavitev logičnih funkcij z minimalnimi urejenimi binarnimi odločitvenimi grafi", Elektrotehniški vestnik, Vol. 59, No. 5, 1992, pp. 299-307. In Slovene.

[16] A. Časar, R. Meolic, "Representation of Boolean functions with ROBDDs", 1993. Presented at IEEE Region 8 Student Paper Contest, Paris-Evry 1993. Published in: IEEE Student paper contest: regional contest winners 1990-1997, IEEE, 2000.

[17] EST. On-line (21/10/2011). http://lms.uni-mb.si/EST/

[18] S. Paumier, "Why academic software should be Open Source", INFOtheca: Journal of informatics and librarianship, Vol. X, No. 1-2, 2009, pp. 51-54.

[19] G. Janssen, "A Consumer Report on BDD Packages", In: 16th Symposium on Integrated Circuits and Systems Design, 2003, pp. 217.

[20] Free Software Foundation, Inc. On-line (21/10/2011). http://www.fsf.org/

[21] S. B. Akers, "Binary decision diagrams", IEEE Transactions on Computers, Vol. C-27, No. 6, 1978, pp. 509-516.

[22] S. Minato, N. Ishiura, S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", In: 27th ACM/IEEE Design Automation Conference (DAC'90), 1990, pp. 52-57.

[23] R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification", In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD '95), 1995, pp. 236-243.

[24] R. Drechsler, B. Becker. Binary decision diagrams: theory and implementation. Springer, 1998.

[25] C. Meinel, T. Theobald. Algorithms and Data Structures in VLSI-Design: OBDD – Foundations and Applications. Springer-Verlag, 1998.

[26] R. Ebendt, G. Fey, R. Drechsler. Advanced BDD optimization. Springer, 2005.

[27] S. Edwards, G. Swamy, "The VIS Engineering Manual", 1996. On-line (21/10/2011). http://vlsi.colorado.edu/~vis/prgDoc.html

[28] BDDTCL. On-line (21/10/2011). http://www2.parc.com/csl/members/kpartrid/

[29] BDD Visualizer. On-line (21/10/2011). http://www.cs.uc.edu/~weaversa/BDD_Visualizer.html

[30] JADE: Implementation and Visualization of a BDD Package in JAVA. On-line (21/10/2011). http://www.informatik.uni-bremen.de/agra/eng/jade.php

[31] Graphviz. On-line (21/10/2011). http://www.graphviz.org/

[32] A. Duşa, "Enhancing Quine-McCluskey", 2007. COMPASSS Working Paper WP 2007-49. http://www.compasss.org/pages/resources/wpfull.html

[33] A. Časar, Z. Brezočnik, T. Kapus, "Exploiting symbolic model checking for sensing stuck-at faults in digital circuits", Inf. MIDEM, Vol. 32, No. 3, 2002, pp. 171-180.


**Robert Meolic** received his Ph.D from the University of Maribor, Slovenia in 2005. He is currently an Assistant Professor at the Faculty of Electrical Engineering and Computer Science at the same university. His main research interests include Boolean algebra, binary decison diagrams, temporal logic and model checking. Dr. Meolic is a member of IEEE and the Slovenian Electronic Communication Society SIKOM.