MVSIS 2.0 Programmer's Manual

Donald Chai, Jie-Hong Jiang, Yunjian Jiang, Yinghua Li, Alan Mishchenko, Robert Brayton

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley CA 94720
mvsis-devel@ic.eecs.berkeley.edu

Abstract

MVSIS is a logic synthesis system, which enhances traditional binary logic synthesis with capabilities related to multi-value logic. This manual introduces the programming environment of MVSIS 2.0 and compares it with those of SIS 1.3 and VIS 1.4. The new data structures are described and the motivation behind them is explained. The goal is to help the reader with the general understanding of logic synthesis and the working knowledge of C programming get started writing his or her own application code, which manipulates binary or MV networks in MVSIS.

1 Introduction

MVSIS [5] is a logic synthesis system, which supports the data structures and procedures needed for technology-independent binary and multi-valued (MV) logic synthesis. The current implementation, MVSIS 2.0, features a completely new source code with only a few packages borrowed from SIS [7] and VIS [9]. However, as a programming environment, MVSIS is designed to have the feel of SIS and the look of VIS. The following points summarize the new features:

- Several modifications are made to the network/node data structures; in particular, the pin data structure is introduced.
- BDD/MDD functionality representation of the nodes pioneered in BDS [10] is supported along with the traditional SOP representation used in SIS.
- A new technique for storing the local functions (relations) of the nodes is used; as a result, all the node fanin variables are mapped into the same range of encoding BDD variables.
- New data structures are developed to represent and manipulate the general case of non-deterministic (ND) multi-valued (MV) relations describing the functionality of the nodes.
- Classical logic synthesis operations (*factor*, *eliminate*, *fast_extract* etc) are revised and re-implemented using the recent advances in processing of the binary data and the generality of multi-valued logic.

This manual assumes that the reader is familiar with the terminology of Boolean algebra, Boolean networks, logic synthesis, and decision diagrams. Familiarity with SIS or VIS programming environments would be helpful but it is not necessary.

In the sequel, whenever a reference is made to an MV object (network, node, SOP, relation etc), it is assumed that the same statement is also true for a binary object of the same type, unless it is explicitly stated that it does not work in the binary case.

The rest of the manual describes components of the MVSIS programming environment. Section 2 outlines the basic data structures. Section 3 presents the data structures to represent the node functionality in more detail. Sections 4-6 describe windowing, structural hashing, and the flexibility manager, respectively. The manual is concluded by Section 7 followed by a list of references.

2 Basic Data Structures

2.1 MVSIS Framework (*Mv_Frame_t*)

The MVSIS framework is a data structure containing general information about the current run of MVSIS. In particular, it stores the tables of command, aliases, and flags, the stack of backup networks, the pointers to the streams (output, error, and history), and other variables. Most of these variables are declared as global in SIS/VIS. MVSIS, on the contrary, does not have global variables, except a few statistical variables, such as those used by the CUDD package.

2.2 Network/Node (*Ntk_Network_t/Ntk_Node_t*)

The network/node data structures are similar to those of SIS/VIS. In this section, we describe the most important differences.

General Information. The network is composed of internal nodes, which have logic associated with them and two types of placeholder nodes: (1) combinational inputs (CIs), which include the primary inputs of the network and the latch outputs, and (2) combinational outputs (COs), which include the primary outputs and the latch inputs.

The latches are represented a sets of additional CI/CO pairs. If the input/output of a latch is also used as a primary output/input of the network, only one CO/CI is created and labeled accordingly.

Only single-output nodes are currently used. In this, MVSIS is similar to SIS and is different from VIS, which can read and represent multi-output nodes.

Pin Data Structure (*Ntk_Pin_t*). A *pin* stores information about one fanin or one fanout of a node. In each node, the fanin pins and the fanout pins form two double-linked linked lists. The MVSIS environment is geared towards logic synthesis when frequent changes to the network are performed. These changes make it necessary to add and remove nodes and node pins often. Linked lists are convenient for this purpose.

The concept of a *net*, as a wire connecting the output of a node with the inputs of the fanout nodes, is currently not explicitly represented in MVSIS. In the future releases, the net data structure (*Ntk_Net_t*) may be introduced, by associating it with the node and including in it the linked list of the fanout pins containing pointers to the fanouts.

Network Traversals. Most of the logic synthesis applications perform numerous traversals of the network structure. In SIS/VIS, such traversals are performed with the help of hash-tables and arrays. For example, when the nodes are collected in the DFS order in SIS, the hash-table is used to remember the visited nodes, while the array is used to collect the nodes in the DFS order.

The following two features of MVSIS make hash-tables and arrays unnecessary in most of the network traversals performed by typical logic synthesis applications.

Traversal ID (*Ntk_Node_t: int TravId*). The traversal ID is a unique integer associated with each traversal of the network. Before a new traversal, the current traversal ID of the network is incremented. Once the node is visited, its traversal ID is set equal to the current traversal ID of the network. During the traversal when a node is visited, its traversal ID is compared with that of the network. If they are different, the node is visited for the first time. If they are the same, the node was visited earlier in the same traversal.

Specialized Linked List of Nodes. When there is a need to collect the nodes during a network traversal, the nodes are added to the specialized linked list, associated with the network. In this case, before performing the traversal, the specialized linked list of the network is cleared. Because each network owns only one such linked list, caution should be taken to make sure that the contents of the list are not modified by other procedures. To this end, it is recommended that the nodes collected into the list are used immediately (for example, by iterate through them) or saved into an array in the application code for future reference.

There are two iterators through the specialized linked list. One of them can only be used to access the nodes. Another one, called *safe iterator*, can be used to iterate and delete a node while iterating through it.

As an illustration of these ideas, refer to the code of procedure *Ntk_NetworkDfs*, found in *src/base/ntk/ntkDfs.c*.

Memory Management. In the current implementation, each network owns several memory managers, which are used to allocate/recycle pieces of memory of a fixed size, such as memory entries for individual nodes and pins, or pieces of memory of a flexible size, such as node names. These memory managers allocate memory using large chunks, which are then split into smaller pieces. The managers support constant-time allocate/free operations for the objects used inside MVSIS, but cannot return memory to the operating system as long as the network exists.

3 Functionality Data Structures

This section describes the main data structures, which are used to represent the node functionality.

3.1 Variable Maps (*Vm_VarMap_t/Vmx_VarMap_t*)

Two types of variable maps are used to represent the binary/MV input/output space of the node.

MV Variable Map (*Vm_VarMap_t*). The MV variable map represent the information about the number of values of each input/output variable of a node. The APIs to this data structure return such information as the number of values of individual variables or the total number of values of all input and outputs variables. To manipulate cubes represented in positional notation, there is another API, giving access to the first value of each variable in the contiguous array of values of all variables.

Encoded MV Variable Map (*Vmx_VarMap_t*)**.** An encoded variable map is defined as an extension of the MV variable map. It contains additional information about the encoding of MV variables using binary variables. This encoding is used to map the MDD representation of the nodes into the binary variables of the BDD package [8].

The extended variable map also stores the permutation of binary encoding variables (*pVmx->pOrder*) along with the

mapping of the MV variables into the binary variables (*pVmx->pBitsFirst*) and the number of binary variables needed to encode each MV variable (*pVmx->pBits*). Because of this permutation array, MV relations can be represented using BDDs, each of which has its own variable ordering stored in the extended map.

Both types of variable maps are cached in the hash tables internal to the variable map managers. As a result of caching, the variable maps of the nodes are shared rather than duplicated, which can noticeably reduce the memory consumption for large networks.

The variable map packages (*src/mv/vm* and *src/mv/mvx*) feature numerous APIs, to create new variable maps after the current one has been changed, for example, as a result of expanding the fanin space or dynamically reordering the variables in the underlying BDD.

3.2 Cubes and Covers (*Mvc_Cover_t/Mvc_Cube_t*)

The data structure to represent MV cubes and covers has been designed to provide convenience to the programmer, fast bitwise operations, and efficient memory management.

The differences compared to the well-known Espresso-MV [6] *pset_family* data structure are the following:

Combining Cubes into Covers. Linked lists rather than arrays are used to connect the cubes belonging to one cover. Linking cubes gives additional flexibility in those SOP operations, where cubes are manipulated individually. For example, some cubes may be removed from one cover and placed into another cover. This type of manipulation often takes place in factoring, co-factoring, tautology check, etc.

Bitwise Operations. The macros to perform bitwise operations are similar to those of Espresso. Additionally, these macros account for the special case, when the cubes in positional notation can be represented using one or two machine words. Cubes used in most of the logic synthesis applications fall into these two categories, because the input space of a node rarely includes more than 32 (or 64) values. On most computers, these macros are faster than the general type of macros used in Espresso.

Memory Management is used for all cubes and covers generated in the SOP-based computations for all networks.

3.3 MV SOPs (*Cvr_Cover_t*)

One way of representing the functionality of an MV node (a node with MV inputs and MV output) is to represent the values of the output variable (called *i-sets*) as functions of the input variables. In such a representation, each *i*-set is an MV-input binary-output function represented by its SOP. The array of SOPs, one for each *i*-set, constitutes the MV SOP representation of the node.

Typically, a node has the default value, which the node outputs when all other values are not produced. (In the binary case, the on-set is typically represented explicitly, while the offset is assumed to be the default value.) In the current MVSIS implementation, the default value is marked by the NULL pointer in the array of *i*-sets in the MV SOP representation.

The special set of APIs of the MV SOP package (*src/mv/cvr*) allows us to call Espresso-MV for each *i*-set of the MV SOP. To this end, the current MVSIS cover/cube data structures are mapped into the corresponding Espresso structures, which are then given to Espresso. The result is transformed back into the MVSIS data structures. The runtime overhead for this transformation is in most cases negligible compared to the runtime of Espresso itself.

3.4 MV Relations (*Mvr_Relation_t*)

MV relations are the second major representation of the node functionality in the current implementation of MVSIS. Unlike the MVSOP representation, which represents the SOP of each *i*-set independently from that of other *i*-sets, an MV relation combines all *i*-sets into one data structure, by treating the input and output variables uniformly.

In the resulting representation, all the MV variables are ordered: first inputs, then outputs. Each MV variable is encoded using the smallest number of binary variables. The codes are derived in such a way as to evenly distribute the code minterms among the codes, which tends to reduce the BDD size. When all the variables are encoded, an MV relation is represented as a single BDD.

Sharing BDD Variable Spaces of Local MV Relations. The following convention regarding the use of BDD variables should be honored when programming with local MV relations. When the MV relation is *minimum-base* (when it does not include MV variables that do not contribute nodes to the MDD), it is always mapped into the topmost variables of the BDD manager.

This convention allows us to map all MV relations into the same, relatively small range of BDD variables. In this, MVSIS is different from VIS, which uses unique BDD variables to represent each MV variable. The advantage of the MVSIS representation is that substantially fewer variables are needed, especially for large networks. Using the same BDD variables for all relations increases the hit-rate in the computed table of the BDD package. The only disadvantage is that individual BDDs have to be remapped before they are composed. The overhead for remapping is almost always negligible, and therefore the advantages of the new representation by far outweigh the disadvantages.

BDD Variable Reordering. The individual BDD variables, which encode an MV relation, can be dynamically reordered to reduce the representation size. During reordering, the binary variables encoding the same MV variable are not kept contiguous; they can move independently up and down the order.

Dynamic reordering is performed using a specialized reordering engine REO (*src/bdd/reo*), compatible with the CUDD BDD manager. As a result, different relations can have different variable orders, yet they are stored in the same BDD manager, which is never reordered. The variable permutations specifying individual variable orders are represented in the extended variable map data structure, which is part of the MV relation data structure.

Other APIs. Numerous APIs to the MV relation package (*src/mv/mvr*) can perform such operations as checking whether the given relation is non-deterministic, well-defined etc, as well as computing cofactors, quantifying, and deriving new MV relations from the given one. Another class of APIs performs translation from the SOP into the MV relation representation and back. Fast generation of SOPs from BDD/MDD is based on [3].

BDD Package. The only currently supported BDD package is CUDD [8]. The motivation for this is that CUDD combines an overall good performance with efficient manipulation for Zero-suppressed Decision Diagrams, which are extensively used in MVSIS to convert BDD/MDD representation into the SOP representation.

4 Windowing (*Wn_Window_t*)

Windowing is a way of clustering nodes belonging to the same network. A non-trivial *window* includes a set of internal nodes of the network. In particular, it can be a single internal node or all internal nodes of the network.

Windows can be nested. In this case, it is possible to compute a container window around another window, called the *core*. The container window can be described by giving two parameters, which specify the number of levels of limited TFI and TFO, which should be included into it, starting from the nodes of the core.

The window with the given parameters is constructed by including into it the internal nodes of the current network, which fall into at least one of the following four categories: (1) the nodes of the window core; (2) the nodes in the TFI of the core, such that all the path from the core to them is less than the given parameter; (3) the nodes in the TFO of the core, such that all the path from the core to them is less than the given parameter; (4) all the nodes that are in the TFO of nodes (2) and in the TFI of nodes (3).

The windowing package (*src/graph/wn*) has a dedicated data structure to represent a window. This data structure includes the array of *window roots* (the window nodes that have at least one fanout outside of the window) and the array of *window leaves* (the nodes outside the window that fanin into at least one node in the window). The internal nodes of the window can be manipulated (collected, ordered, etc) using the APIs of the windowing package.

5 Structural Hashing(*Sh_Network_t/Sh_Node_t)*

An AND/INV graph is a non-canonical data structure used to represent Boolean functions. AND/INV graphs are similar to BDDs in that the functions are represented by the roots of a shared DAG with nodes being two-input AND-gates with optional inverters at the inputs and outputs. The leaves of the graph are the constant 1 node and the nodes representing the elementary variables.

The AND/INV graphs can be efficiently structurally hashed [2]. In fact, the structural hashing algorithm is built into the construction of the AND/INV graphs similar to the unique table lookup used in the construction of BDDs.

The current implementation of MVSIS allows for the generation and use of the AND/INV graphs that are functionally equivalent to the current network, or a window in the network. The uniformity and compactness of AND/INV graphs lead to an efficient implementation of such algorithms as symbolic simulation and circuit-based SAT [2].

The APIs of the structural hashing package (*src/graph/sh*) can construct and manipulate AND/INV graphs, similar to how BDDs are constructed and manipulated by calling the APIs of the BDD package.

6 Flexibility Manager *(Fm{sw}_Manager_t)*

The role of the flexibility manager is to compute the complete flexibility of a node or a group of nodes, taking into account their context in the network [4]. When the flexibility for a node is computed by a call to the flexibility manager, it should be used to optimize the node, before the next call to the flexibility manager.

The corresponding package (*src/opt/fm*) provides the APIs to start and free the flexibility manager, and to compute the flexibility for the nodes. The implementation supports two modifications of the flexibility manager, which differ in the scope of the surrounding areas of the network considered for deriving the flexibility.

Network Scope (*Fms_Manager_t*)**.** When this version of the flexibility manager is applied to the network, it first constructs the global BDDs for all the COs of the network and stores them away. This computation constitutes the preprocessing step. Next, the manager is called for individual nodes, in the order determined by the application. In each call, it uses the pre-computed global BDDs to derive the flexibility. In the end, the global BDDs are dereferenced and the manager is freed. Obviously, this manager can only be applied to the networks, for which constructing the global BDDs can be performed within the reasonable time and memory limits.

Window Scope (*Fmw_Manager_t*)**.** This version of the flexibility manager does not pre-compute the BDDs of the COs of the network, and therefore is independent of the

network size. When applied to a node, it first constructs a window around this node. Next, the global BDDs of the roots of the window are computed in terms of the leaves. These BDDs are used to derive the flexibility of the node. In the end of the call, both the global BDDs and the window are freed, while the resulting flexibility is returned to the user. In this flow, nothing is stored in the manager between the calls to the flexibility computation for individual nodes.

Both versions of the manager take advantage of the interleaved static variable BDD ordering. However, only the first version can currently perform the dynamic variable ordering.

7 Conclusions

In this paper, we briefly described the main data structures of the MVSIS environment.

Three other reference sources are available:

- MVSIS User's Manual [1] gives a high-level overview of MVSIS, describes the use of MVSIS command shell, and gives the summary of common logic synthesis commands.
- The source code [5] can, in many cases, be used as a reference material, due to the presence of the written notes preceding every procedure and comments included in the code.
- For compilation instructions, refer to *readme* file in the root directory of the project.

References

[1] D. Chai, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, R. Brayton. *MVSIS 2.0 Programmer's Manual*, UC Berkeley, May 2003.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, M. K. Ganai, Robust Boolean reasoning for equivalence checking and functional property verification, *Trans. CAD,* Vol. 21, No. 12, December 2002, pp. 1377-1394.

[3] S. Minato, "Fast generation of irredundant sum-of-products forms from binary decision diagrams. *Proc. SASIMI'92*, pp. 64-73.

[4] A. Mishchenko and R. K. Brayton, "Simplification of non-deterministic multi-valued networks", *Proc. ICCAD'02*, pp. 557-562.

[5] MVSIS Project Webpage: http://www-cad.eecs.berkeley.edu/Respep/Research/mvsis/

[6] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", *IEEE Trans. CAD*, Vol. 6(5), pp. 727-750, Sep. 1987.

[7] E. Sentovich, et al, "SIS: A system for sequential circuit synthesis", *Tech. Rep. UCB/ERI, M92/41*, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[8] F. Somenzi, *BDD package "CUDD v. 2.3.0."* http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[9] The VIS Group. *VIS: Verification Interacting with Synthesis*, 1995. http://www-cad.eecs.berkeley.edu/Respep/Research/vis

[10] C. Yang and M. Ciesielski. BDS 1.2: BDD-based logic synthesis system: http://www.ecs.umass.edu/ece/labs/vlsicad/bds/bds.html.