MVSIS

MVSIS Group
Minxi Gao, Yinghua Li, Jie-Hong Jiang, Yunjian Jiang, Subarnareka Sinha, Tiziano Villa, Robert Brayton
Electrical Engineering and Computer Sciences Dept.
University of California, Berkeley CA 94720
(mvsis-devel@ic.eecs.berkeley.edu)

August 21, 2001

Abstract

`MVSIS` is a program modeled after `SIS`, but the logic network it works on is such that all variables can be multi-valued each with its own range. We include all the technology-independent transformations of `SIS` for combinational logic synthesis as well as transformations specific to multi-valued nodes such as **`merge, pair_decode, encode, elim_part, print_part_value, print_range, reset_default`**. `MVSIS` has been made to have the look and feel of `SIS`. `MVSIS` can read and write `BLIF-MV` files with the **`read_blifmv`** and **`write_blifmv`** commands, or read `BLIF` files with the **`read_blif`** command.

1 Introduction

Multi-level multi-valued (MV) logic synthesis can have many applications including:

1. Logic synthesis for multi-valued hardware devices.
2. Initial manipulation of a hardware description before it is encoded into binary and processed by standard binary logic synthesis programs; MV is a natural way to describe procedures at a higher level.
3. A front end to a software compiler, since software lends itself naturally to the evaluation of multi-valued variables in a single cycle. Strong logic synthesis transformations can be applied to compilers aimed at embedded applications.
4. Asynchronous synthesis

We have developed and included techniques for combinational optimization of MV networks. Like `SIS` [1, 2], `MVSIS` is an interactive tool, and has been made to have the look and feel of `SIS`. When applied to purely binary networks, it behaves almost exactly like the technology independent part of `SIS`. In the sequel, the main components of `MVSIS` are described, the input specification, the MV-transformations and special commands, verification by simulation, followed by a few examples illustrating the use of `MVSIS` in the design process.

2 Design Specification

2.1 BLIF-MV Description Format

An MV circuit can be input to MVSIS as a netlist of MV-nodes (command: **read_blifmv**). We use a simple subset of BLIF-MV [3] used in VIS to specify the design. Such BLIF-MV files can be generated by the Verilog front-end to VIS (vl2mv) or be written out by VIS. Binary networks specified in BLIF (the format most commonly used in SIS) can also be read in (**read_blif**). After a design specification is read in, it is converted into an MV-network, a design representation used within MVSIS. An MV-network is a network of nodes; each node represents an MV-function with a single multi-valued output. The functions associated with each value (value-functions) of a node are stored in SOP form. There is one MV variable associated with the output of each node. An edge connects node i to node j if any of the value-functions at j depends explicitly on the variable associated with node i. The network has a set of primary inputs (all of which may be multi-valued) and a set of nodes, designated as the outputs of the network. An important distinction with other MV methods, is that in our representations, each variable can have a separate range of its own, including two values. All ranges are represented by the sets $\{0, 1, \ldots, n_i - 1\}$.

MVSIS 1.0 supports sequential MV-network with multi-valued latches. A latch, as any other variables in the network, can be multi-valued. In the BLIF-MV file, a latch can be specified using construct **.latch**, with the initial state specified using construct **.reset**. We only support BLIF-MV files with constant initial state values.

2.2 External Don't Cares

The original BLIF-MV file is extended to support external don't cares. Like in SIS, a don't care network is specified as a combinational network following the key word **.exdc**. The inputs of the EXDC can be a subset of the primary inputs plus the latch outputs; the outputs of the EXDC can be a subset of the primary outputs. The outputs of the EXDC are matched with those of the original network by names. They are binary variables, even though the original output may be multi-valued. Given a table for function f, its companion external don't care table defines a boolean function for f, whose minterms may produce any value for f (universal don't care minterms).

For multi-valued nodes, if a **.default** value is given, the node is completely specified. For a table without the **.default** value, if the sum-of-products specification does not cover all the minterms in the input space, it is incompletely specified. In this case, the unspecified minterms are assumed to be able to take any value at the output, namely don't cares. In some applications, like data mining, the unspecified minterms can consume a large space. Therefore, MVSIS 1.0 provides a way of extracting incomplete specification as external don't cares. In the currently implementation, if an incompletely specified table in the network is (i) a primary output, and (ii) all its fanins are primary inputs, then an EXDC network is extracted for that node.

In the future we plan to extract and add to the external don't care set also the minterms that are explicitly described as taking any value out of the range of an output function (universal minterms), since their semantics is the same as the one of unspecified minterms.

External don't cares are a special case of non-determinism, because they restrict it to the case when an input minterm assumes a unique output value or any output value (universe of the output variable). MVSIS 1.0 does not support partial don't cares, where an input minterm may assume any value out of a proper subset of the universe of the output variable.

3 Combinational Optimization

3.1 Node Simplification

The logic value-functions (one for each output value) at an MV-node are simplified with the `simplify` command which uses the two-level logic minimizer `ESPRESSO-MV`. The objective of a general two-level logic minimizer is to find a logic representation with a minimum number of implicants (cubes) and literals while preserving functionality. Satisfiability don't cares from the local fanins and subset support variables are used in the minimization unless the `-d` option is used. After simplication, the value-functions are replaced with simplified versions if the new functions have been improved according to the cost function in use.

For each node, one of the value-functions is selected as the default value. For example, for a binary output function, the onset is usually the primary value and the offset, the default value. The default value-function is never looked at unless a command requires it. For example, if the output of a binary function is used in the complemented form in a fanout, and the node is eliminated, then the complement must be computed to effect the elimination. The values of the nodes and statistics of the network are based only on the non-default value-functions. However, there is one command `reset_default` which looks at each node and chooses a default value for it based on the cost of the node. For example, if the cost function is the number of cubes, `reset_default` will cause value functions to be minimized, and the default value will be chosen to be the value whose function has the most cubes.

Currently, there are two cost functions which can be used. These can be selected using the `set` command. With `set cost 0`, the number of cubes in the SOP form is used. With `set cost 1`, the number of literals in the factored form is used. In the future, there will be more complex cost functions depending on the target of the application, e.g., the number of cubes, the number of nodes, the number of values, the number of fanins.

The strongest kind of node simplification that can be performed on a network, is implemented using the `fullsimp` (alias `fs`) command. To perform this function on a multi-level MV-network, an appropriate don't care set is first generated. Subsets of the satisfiability and observability don't care sets (SDC and ODC respectively) are used. The notion of compatible observability don't cares (CODC) used in `SIS` has been generalized to take MV-nodes into account [4]. Given these, MV-image computation techniques are used to map them to the local space of each node. An SDC of those nodes in the network whose support is a subset of the support of the node being simplified is also added to the local don't care set thus derived. This allows a form of Boolean substitution when `fs` is executed. Each node is then simplified by `ESPRESSO-MV` using this local don't care set.

During any of the above forms of simplification, if it is estimated that simplification will take too long, the node will be minimized with a simpler form of minimization or left unchanged. The complexity of an `ESPRESSO-MV` session is estimated by the number of cubes in the onset and don't care set, and the number of fanin variables. If this is too large, `ESPRESSO-MV` is not called. There is also a timeout for the `fullsimp` and `simplify` commands, controlled by the `-t` option. The specified time (in integer seconds) is shared among three time consuming computations; CODC computation, image computation, and `ESPRESSO-MV` minimization. If any one of these takes longer than the allocated time, the simplification for that node is terminated and only the local SDC is used for the node minimization. A default timeout value of 2 seconds is used.

3.2 Kernel and Cube Extraction

An important step in network optimization uses algebraic methods for extracting new nodes representing logic functions that are common factors of other nodes. Several techniques based on algebraic decomposition are part of `SIS`. Similarly, we have developed new algebraic techniques for MV-logic [5] which treat binary and multi-valued variables uniformly. They include methods for finding common sub-expressions, semi-algebraic division, decomposing a multi-valued network, and factoring a SOP form. The relevant commands and brief descriptions of their abilities are listed below.

1. The command `fx` looks at all the nodes in the network and tries to extract good common factors and create new nodes in the network, re-expressing other nodes in terms of these newly introduced nodes. It is one of the transforms used to break down large functions into smaller pieces. It has two options, `-q` and `-g`.

   The `-q` option generates candidate two-cube divisors by making each pair of cubes in a node value-function cube-free. These candidate two-cube divisors are made canonical and hashed into a table. A count is kept on the number of hits for each entry to obtain the value of a divisor. Complements are also kept if it is a two-cube expression. The divisor with the largest value is then extracted as a new node and substituted into all functions where applicable. For efficiency, the divisor table is only incrementally updated after each substitution.

   The `-g` option can generate additional divisors. This method extends `fx -q`. If no divisor is found by `fx -q`, `fx -g` first generates a set of candidate double-cube divisors, one for each function, by factoring each node in the network. It then divides these candidate divisors, used in the factorization, into all other nodes and computes their values. The divisor with the largest value is extracted. In general, this method can find divisors that `fx -q` cannot find, since factoring can sometimes find divisors that cannot be obtained by the other method. This method can take longer CPU times if the number of cubes in the nodes is large.

2. The command `decomp` does a complete factoring of each node, but instead of creating a factored form for each, decomposes the node according to its factorization. Thus more intermediate nodes are produced this way. Such intermediate nodes may not have been produced by `fx -q` or `fx -g`, so there is a possibility of finding better factors. After this, `resub` (see below) followed by `sweep` should be executed to eliminate duplicate factors. Then elimination can be done to clean up the network, possibly followed by `simplify` to look for Boolean substitutions.

3. Algebraic substitution of one node into another is performed in `MVSIS` using `resub [-d] [node-list]`. As an argument it takes a list of nodes that are to be algebraically substituted into all other nodes. If no list is given, all nodes are tried. All value-functions of the divisor are tried; the default value-function is also tried unless the `-d` option is given. We do not attempt to divide into the default function of other nodes, since their value-functions can be obtained by complementation. `resub` uses the new methods of "exact" semi-algebraic division, developed for multi-valued logic. There are two modes for this. If the divisor is a two-cube divisor, then a fast method based on matching is used; otherwise, a slower branch and bound method is used (called the satisfiability-matrix method). Although in theory all pairs of nodes must be looked at during `resub`, there are very effective filters to determine if no algebraic substitution is possible.

3.3 Network Manipulations

1. The command `collapse` collapses nodes in the network. If no arguments are given, it collapses the entire multi-level network so the SOP forms for each output are in terms of the primary inputs only. Thus the number of nodes in the network will be exactly the number of primary outputs. If a single node name is given as an argument, that node will be collapsed. If two names are given, one must be a fanin of the other, in which case the fanin node is collapsed into the fanout node. The collapsed node is removed from the network, if there is no other fanout.

2. The command `eliminate` eliminates all the nodes in the network whose value (as measured by the current setting for the cost function) does not exceed a specified threshold. The value of a node represents the total cost of the network with the node eliminated, minus the current cost of the network. If the value is not greater than the specified threshold, the node will be eliminated by collapsing the node into each of its fanouts. Of course, a primary input or a primary output will not be eliminated. The command iterates, since eliminating one node may affect the value of other nodes. The iteration continues until all remaining nodes have a value greater than the threshhold. However, if it is estimated that the elimination of a particular node will cause a blow-up in the number of cubes, the node will be kept (see the global parameter `el_limit` specified by the command `set`).

3. `merge` is a command unique to `MVSIS`. It takes a list of nodes and forces a merge of them into a single multi-valued node. In the worst case, if for example, there are k binary nodes in the list, it will create a single node with 2^k values. However, some new value-functions may be 0, in which case they are not created. In addition, if a pair of values always appears together in all the fanouts, then their functions will be merged into a single value-function. If no list is given, merge looks for a likely list of candidates and merges them if it can achieve a gain in the value for the network. This may result in several additional multi-valued nodes through multiple merges. `merge` can be given as argument a list of nodes to be merged. In this case, the nodes will be merged regardless of the gain, provided the merging does not introduce cycles. Cycles can happen if there is a path from some node A to some node B in the list, which passes through a node not in the list.

4. `encode` is like the inverse of merge. It tries to find a good binary encoding for each multivalued variable in the network, including primary inputs and outputs. At the end, each signal is encoded as a binary signal, including primary inputs and outputs. Then a binary file can be written. As an option (`-i`), encoders and decoders can be put at the inputs and outputs which convert the network to its original multi-valued inputs and outputs. The internal signals are binary. This can be written as a `blif-mv` file and validated against the original file. The encoding heuristic starts from the outputs and in reverse topological order works back to the primary inputs. At each node, its outputs are encoded using the information on how its fanouts are used. The encode command is `encode` with option `- i` specifying that encoders and decoders should be put at the inputs and outputs.

5. `pair_decode` does bit pairing to create a new multi-valued node. It looks for a "best" pair of signals to pair together. Then it creates a new node with values equal to all the decodes of the pair. If both signals are binary, then a 4-valued node is created and algebraically substituted into nodes which algebraically depend on at least one of these decodes. Finally, any set of values of the new node, which always appear together in the fanouts, are merged into a single value of the new node. After this step, `simplify` should be executed to effect full substitution.

6. `elim_part` is a command like eliminate, except it works only on multi-valued output nodes and can eliminate some of the value-parts for the nodes. A specified threshold controls which parts are to be eliminated. The value given to each of the parts is heuristic and each part is ordered using this value. To see the accumulated value of the parts, the command `print_part_value` shows the ordering of the parts, and the accumulated values from least to greatest. For example, for a particular node `m` it might print out the following,

$$m: \ (110)\ 12\ 30\ 50\ 68\ 88\ 112\ 136\ 160\ 184\ (6\ 3\ 0\ 7\ 8\ 5\ 2\ 1\ 4\ )$$

   The name `m` is first followed by the value of the node as evaluated in `eliminate` (110), then a vector of the accumulated part values $12, 30, \ldots$, where 12 is the value of the 6^{th} part, 30 is 12 plus the value of the 3^{rd} part, etc. If the command `elim_part 30` is given, `MVSIS` will eliminate the first two values in the order (6 and 3) by merging their functions so that the range of the node is reduced by 1 and assigning them to the highest available value (here it is 7). A new default value is selected for the new function; the merged part may have a larger function so it would replace the previous default part. Fanout nodes will be modified accordingly to preserve the network functionality.

7. `sweep` successively eliminates single input intermediate nodes in the network and deletes nodes with no fanouts.

8. `undo` replaces the current network with the previous one. This is particularly useful if a forced merge results in a worse network. The combination `merge` and `undo` allows one to experiment with different mergings. `undo` treats `fs` special so that one is able to do two commands like `merge; fs` and `undo` will revert back to the circuit before the `merge`.

4 Sequential Optimization

Like in SIS, all combinational optimizations are performed on the combinational part of the sequential network. New command `extract_seq_dc` implements a reachability computation in the state space and computes the set of unreachable states, which are consequently treated as external don't cares (EXDC). In the case where there exists an EXDC network already, the unreachable EXDC is ORed with the existing one.

5 Other Commands

5.1 Iteration

Several commands have the option to apply the command a given number `n` times, or until a fixed point, i.e. no change occurs in the network. These commands are `eliminate, pair_decode, gx, fx` and `fullsimp`. The command structure is, for example `eliminate -i n <threshold>` where `n` is the iteration count. If no `-i` option is given, iteration to a fixed point is implied.

5.2 Printing, reading and writing

There is one write command, `write_blifmv`, and two read commands, `read_blifmv` and `read_blif`. The latter reads in ordinary `BLIF` files.

To view the results at any stage, there are several print commands which print to the console. `print` prints the SOP form of each value (including the default value if the option `-d` is given) of each node in the network. `print_factor` prints the factored form of each value-function (excluding the default) of each node in the network. Each of these can take, as argument, a list of names of nodes to be printed. In general, *, like `SIS`, stands for all a list of nodes in the network.

`print_range` prints out the size of the range for each variable; `print_value` the value of each node (according to the current cost function); `print_stats` the statistics of the network in terms of the network name, the number of primary outputs, the number of nodes, the number of cubes, and the number of literals in the cubes. `print_stats -f` also prints out the number of literals in the factored forms of the nodes (for a given function all values are taken into account except the default one). `print_io` prints the inputs and outputs of the network. If a list of nodes is given, it prints out the fanins and fanouts of each node in the list.

Sometimes, in order to view an output or factorization better, it is useful to change the names of the variables to short names using the command `chng_name`. It is a toggle between short names and the original names. Associated is a command `reset_names` which resets the naming of short names for the variables so that all variables appear in lexicographic order with no gaps in the naming. Thus inputs are named first, $\{a, b, c, \ldots\}$, then outputs, and finally intermediate nodes.

5.3 Setting global parameters

The `set` command sets various global parameters, which control the transformations. With no argument, `set` prints out the current values of the global parameters.

1. `cost` controls the type of cost function to be used in the evaluation of the value of a node. `set cost 0` uses the number of cubes as the cost function; `set cost 1` uses the number of literals in the factored form.

2. `time_limit`, controls the maximum amount of time (in seconds) that can be spent in factoring a single function. Since MV-algebraic factoring may take some time, this is useful in controlling the time spent in the factoring process, especially when the factoring is only being used to estimate the value of a node or to produce a readable output.

3. `el_limit` controls when a node will be eliminated. It may be that eliminating a node can result in a fanout becoming too large. For each fanout, we estimate the number of cubes that will result after elimination. If this estimate exceeds the maximum of `el_limit` and twice the largest cover of any node value, then the elimination is aborted.

4. `autoexec` can be given a command which will automatically execute after each command line is executed. A typical use is `set autoexec print_stats -f`, which will print out the statistics (including literals in the factored forms) of the network after each transformation.

5. `alias` is like `set`. It is used to create nick-names for various commands. For example `alias pfs print_stats -f` can be used to print out the stats of the network (including the number of literals in the factored forms) with the single command `pfs`. `alias` with no arguments will print out a list of all aliases defined so far. A typical set of aliases is incorporated in the file `mvsisrc`, which is executed when `MVSIS` is started.

5.4 Help and Scripts

1. help prints the set of commands available and with a single argument, a command, will print a detailed description of the command.

2. New command _print_version print out the new features included in the currently release compared with the previous one.

3. source reads and executes commands from a file (script file).

6 Verification

MV-networks can be verified in MVSIS by simulation. The command validate verifies the combinational equivalence of two networks by simulating the networks on random vectors, and comparing the outputs. The number of random vectors can be provided by the user from the command line. Command gen_vec generates a specified number of random input vectors appropriate for the ranges of the primary inputs, and writes them into a file. This can be used by the command simulate, which simulates the network and produces the results at the primary outputs. If a formal MDD-based verification is desired, one can write out BLIF-MV files and envoke VIS using its command comb_verify.

In addition to the simulation based method, an MDD-based method is implemented in the new release. For combinational networks, it computes the global function for each output using MDD representation and compares the MDD structure; for sequential networks, it performs the same computation for each latch input as well. If a match can not be found among the latch variables of the two networks to be verified, no validation can be claimed by this method. Since MVSIS does not support non-deterministic networks, the new release includes a command called qcheck. It performs a quick check for non-determinism at the primary outputs by random simulation. If a network is non-deterministic and it is exercised by one of the random vectors, the command produces an error message; a completion without error message does not prove its determinism.

7 Examples Session

We illustrate MVSIS with some examples.

7.1 Example 1

The specification of the example matmul-c is given below in the BLIF-MV format (see BLIF-MV documentation in VIS).

```
#2 X 2 matrix mult over the ring Z_3
.model matmul
.inputs a11  a12  a21  a22
.inputs b11  b12  b21  b22
.outputs c11  c12  c21  c22
.mv a11,  a12,  a21,  a22  3
.mv b11,  b12,  b21,  b22  3
.mv c11,  c12,  c21,  c22  3
.table a11 a12 b11 b21 c11
0 0  - - 0
```

```
0 1  - - =b21
0 2  - 0 0
0 2  - 1 2
0 2  - 2 1
1 0  - - =b11
1 1  0 0 0
1 1  0 1 1
1 1  0 2 2
1 1  1 0 1
1 1  1 1 2
1 1  1 2 0
1 1  2 0 2
1 1  2 1 0
1 1  2 2 1
1 2  0 0 0
1 2  0 1 2
1 2  0 2 1
1 2  1 0 1
1 2  1 1 0
1 2  1 2 2
1 2  2 0 2
1 2  2 1 1
1 2  2 2 0
2 0  0 - 0
2 0  1 - 2
2 0  2 - 1
2 1  0 0 0
2 1  0 1 1
2 1  0 2 2
2 1  1 0 2
2 1  1 1 0
2 1  1 2 1
2 1  2 0 1
2 1  2 1 2
2 1  2 2 0
2 2  0 0 0
2 2  0 1 2
2 2  0 2 1
2 2  1 0 2
2 2  1 1 1
2 2  1 2 0
2 2  2 0 1
2 2  2 1 0
2 2  2 2 2
.table a11 a12 b12 b22 c12
0 0  - - 0
0 1  - - =b22
0 2  - 0 0
0 2  - 1 2
0 2  - 2 1
1 0  - - =b12
```

```
:
:
:
.table a21 a22 b11 b21 c21
0 0  - - 0
0 1  - - =b21
0 2  - 0 0
0 2  - 1 2
0 2  - 2 1
1 0  - - =b11
:
:
:
.table a21 a22 b12 b22 c22
0 0  - - 0
0 1  - - =b22
0 2  - 0 0
0 2  - 1 2
0 2  - 2 1
1 0  - - =b12
:
:
:
.end
```

The above example is stored in a file called `matmul-c`. We start `MVSIS` with the command `mvsis`. Using the `mvsisrc` file, the cost function is set to 1, i.e. cost is the number of literals in the factored forms and we change to the short names mode. The following aliases are used:

```
rl                read_blifmv
saf               set autoexec print_stats -f
fs  fullsimp
rsd               reset_default
pr                print_range
s                 simplify -t 2
pf                print_factor
m                 merge
pr                print_range
pio               print_io
el                eliminate
rsn               reset_name
pd               pair_decode

UC Berkeley, MVSIS 0.0.1 (compiled 13-Feb-01 at 5:54 PM)
changing to short-name mode
mvsis> rl matmul-c
mvsis> pr
{i}:    3
{j}:    3
{k}:    3
{l}:    3
a:      3
```

```
b:      3
c:      3
d:      3
e:      3
f:      3
g:      3
h:      3
mvsis> pio
primary inputs:   a b c d e f g h
primary outputs:  {i} {j} {k} {l}
mvsis> s
mvsis> saf
matmul:  4 nodes,  4 POs,  96 cubes(sop),  320 lits(sop),  160 lits(fact.)
mvsis> rsd
matmul:  4 nodes,  4 POs,  96 cubes(sop),  320 lits(sop),  160 lits(fact.)
mvsis> pd 1
m{0} = a{0}e{2} + e{0}
m{1} = a{0}e{1}
m{3} = a{1}e{2} + a{2}e{1}
n{0} = a{0}f{2} + f{0}
n{1} = a{0}f{1}
n{3} = a{1}f{2} + a{2}f{1}
o{0} = e{0}c{2} + c{0}
o{1} = e{0}c{1}
o{3} = e{1}c{2} + e{2}c{1}
p{0} = f{0}c{2} + c{0}
p{1} = f{0}c{1}
p{3} = f{1}c{2} + f{2}c{1}
q{0} = b{0}g{2} + g{0}
q{1} = b{0}g{1}
q{3} = b{1}g{2} + b{2}g{1}
r{0} = b{0}h{2} + h{0}
r{1} = b{0}h{1}
r{3} = b{1}h{2} + b{2}h{1}
s{0} = g{0}d{2} + d{0}
s{1} = g{0}d{1}
s{3} = g{1}d{2} + g{2}d{1}
t{0} = h{0}d{2} + d{0}
t{1} = h{0}d{1}
t{3} = h{1}d{2} + h{2}d{1}
matmul:  12 nodes,  4 POs,  64 cubes(sop),  184 lits(sop),  160 lits(fact.)
mvsis> s
matmul:  12 nodes,  4 POs,  56 cubes(sop),  96 lits(sop),  96 lits(fact.)
mvsis> rsn
matmul:  12 nodes,  4 POs,  56 cubes(sop),  96 lits(sop),  96 lits(fact.)
mvsis> pf
{i}{1} = p{2}t{2} + p{1}t{0} + p{0}t{1}
{i}{2} = p{2}t{0} + p{1}t{1} + p{0}t{2}
{j}{1} = m{2}q{2} + m{1}q{0} + m{0}q{1}
{j}{2} = m{2}q{0} + m{1}q{1} + m{0}q{2}
{k}{1} = n{2}r{2} + n{1}r{0} + n{0}r{1}
```

```
{k}{2} = n{2}r{0} + n{1}r{1} + n{0}r{2}
{l}{1} = o{2}s{2} + o{1}s{0} + o{0}s{1}
{l}{2} = o{2}s{0} + o{1}s{1} + o{0}s{2}
m{0} = a{0} + f{0}
m{2} = a{2}f{1} + a{1}f{2}
n{0} = c{0} + e{0}
n{2} = c{2}e{1} + c{1}e{2}
o{0} = c{0} + f{0}
o{2} = c{2}f{1} + c{1}f{2}
p{0} = b{0} + g{0}
p{2} = b{2}g{1} + b{1}g{2}
q{0} = b{0} + h{0}
q{2} = b{2}h{1} + b{1}h{2}
r{0} = d{0} + g{0}
r{2} = d{2}g{1} + d{1}g{2}
s{0} = d{0} + h{0}
s{2} = d{2}h{1} + d{1}h{2}
t{0} = a{0} + e{0}
t{2} = a{2}e{1} + a{1}e{2}
matmul:  12 nodes,  4 POs,  56 cubes(sop),  96 lits(sop),  96 lits(fact.)
mvsis> pr
{i}:    3
{j}:    3
{k}:    3
{l}:    3
a:      3
b:      3
c:      3
d:      3
e:      3
f:      3
g:      3
h:      3
m:      3
n:      3
o:      3
p:      3
q:      3
r:      3
s:      3
t:      3
```

7.2 Example 2

The second example is in the file `aluack.mv`.

```
.model alu
.inputs a b carryin control
.outputs out carryout
.mv or 4
.mv and 4
```

```
.mv control 4
.mv out 4
.mv xor 4
.mv sum 4
.mv a 4
.mv sum1 4
.mv b 4
.table a b ->or
.default 3
0 0 0
0 1 1
0 2 2
0 3 3
1 0 1
1 1 1
2 0 2
2 2 2
.table a b ->and
.default 0
1 1 1
1 3 1
2 2 2
2 3 2
3 1 1
3 2 2
3 3 3
.table control or and xor sum ->out
.default 0
0 1 - - - 1
0 2 - - - 2
0 3 - - - 3
1 - 1 - - 1
1 - 2 - - 2
1 - 3 - - 3
2 - - 1 - 1
2 - - 2 - 2
2 - - 3 - 3
3 - - - 1 1
3 - - - 2 2
3 - - - 3 3
.table a b ->xor
.default 0
0 1 1
0 2 2
0 3 3
1 0 1
1 2 3
1 3 2
2 0 2
2 1 3
2 3 1
```

```
3 0 3
3 1 2
3 2 1
.table sum1 carryin ->sum
.default 0
1 0 1
0 1 1
2 0 2
1 1 2
3 0 3
2 1 3
.table a b carryin ->carryout
.default 0
1 3 - 1
2 3 - 1
3 3 - 1
2 2 - 1
3 2 - 1
3 1 - 1
3 2 - 1
3 3 - 1
3 0 1 1
2 1 1 1
1 2 1 1
0 3 1 1
.table a b ->sum1
.default 0
1 0 1
0 1 1
2 3 1
3 2 1
0 2 2
1 1 2
2 0 2
3 3 2
3 0 3
2 1 3
1 2 3
0 3 3
.end
```

The following additional aliases are used:

```
enm  encode -i
u undo
m  merge
vl  validate -n 1000

UC Berkeley, MVSIS 0.0.1 (compiled 13-Feb-01 at 5:54 PM)
mvsis> rl aluack.mv
alu:  7 nodes,  2 POs,  68 cubes(sop),  140 lits(sop),  128 lits(fact.)
mvsis> fs
```

```
alu:  7 nodes,  2 POs,  48 cubes(sop),  98 lits(sop),  96 lits(fact.)
mvsis> rsd
alu:  7 nodes,  2 POs,  49 cubes(sop),  98 lits(sop),  97 lits(fact.)
mvsis> u
alu:  7 nodes,  2 POs,  48 cubes(sop),  98 lits(sop),  96 lits(fact.)
mvsis> m
alu:  6 nodes,  2 POs,  41 cubes(sop),  80 lits(sop),  80 lits(fact.)
mvsis> fs
alu:  6 nodes,  2 POs,  41 cubes(sop),  80 lits(sop),  80 lits(fact.)
mvsis> vl aluack.mv
Networks are combinationally equivalent according to simulation.
alu:  6 nodes,  2 POs,  41 cubes(sop),  80 lits(sop),  80 lits(fact.)
mvsis> pr
{e}:    4
{f}:    2
a:      4
b:      4
c:      2
d:      4
h:      4
j:      4
k:      4
l:      9
alu:  6 nodes,  2 POs,  41 cubes(sop),  80 lits(sop),  80 lits(fact.)
mvsis> pf
{e}{1} = d{3}j{1} + d{2}l{2,6} + d{1}h{1} + d{0}l{1,2}
{e}{2} = d{3}j{2} + d{2}l{4,7} + d{1}h{2} + d{0}l{3,4}
{e}{3} = l{5,6,7,8}(d{2}l{8} + d{0}) + d{3}j{3} + d{1}h{3}
{f}{1} = c{1}j{0,3}k{0,1,3} + h{1,2,3}l{0,3,4,5,6,7,8}
h{1} = b{1}l{0,1,3,5} + l{7}
h{2} = b{2}l{0,1,3,5} + l{6}
h{3} = a{3}b{3}l{0,1,3,5}
j{1} = c{1}k{0} + c{0}k{1}
j{2} = c{1}k{1} + c{0}k{2}
j{3} = c{1}k{2} + c{0}k{3}
k{1} = l{2,6}
k{2} = h{3}l{5} + h{0,1}l{1,3,4}
k{3} = l{8}
l{0} = a{0}b{0}
l{1} = a{1}b{1}
l{2} = a{1}b{0} + a{0}b{1}
l{3} = a{2}b{2}
l{4} = a{2}b{0} + a{0}b{2}
l{5} = a{3}b{3}
l{6} = a{3}b{2} + a{2}b{3}
l{7} = a{3}b{1} + a{1}b{3}
alu:  6 nodes,  2 POs,  41 cubes(sop),  80 lits(sop),  80 lits(fact.)
mvsis> enm
alu:  22 nodes,  2 POs,  87 cubes(sop),  338 lits(sop),  199 lits(fact.)
mvsis> fs
alu:  14 nodes,  2 POs,  36 cubes(sop),  81 lits(sop),  77 lits(fact.)
```

```
mvsis> gx
alu:  16 nodes,  2 POs,  37 cubes(sop),  79 lits(sop),  77 lits(fact.)
mvsis> fs
alu:  16 nodes,  2 POs,  37 cubes(sop),  78 lits(sop),  75 lits(fact.)
mvsis> el 0
alu:  14 nodes,  2 POs,  35 cubes(sop),  79 lits(sop),  74 lits(fact.)
mvsis> pr
{e}:    4
{f}:    2
a:      4
b:      4
c:      2
d:      4
d0:     2
e0:     2
h0:     2
j0:     2
k0:     2
l0:     2
m0:     2
n0:     2
o0:     2
p0:     2
q0:     2
w0:     2
alu:  14 nodes,  2 POs,  35 cubes(sop),  79 lits(sop),  74 lits(fact.)
mvsis> pf
{e}{0} = d0{0}e0{0}
{e}{1} = d0{1}e0{1}
{e}{2} = d0{0}e0{1}
{f}{0} = l0{0}m0{0}(c{0} + h0{1}) + q0{1}
d0{1} = d{3}h0{1} + d{0,2}j0{1} + d{0,1}l0{1}
e0{1} = d{3}(k0{1}w0{0} + k0{0}w0{1}) + m0{1}(d{1,2}p0{0} + d{0,1}p0{1})
        + d{0,2}k0{1}l0{0}m0{0}
h0{1} = c{0}j0{1} + w0{1}
j0{1} = n0{1}o0{0}
k0{1} = j0{1}p0{0} + l0{1}p0{1} + m0{0}o0{1}
l0{1} = a{1,3}j0{0}
m0{1} = n0{0}o0{1} + p0{0}q0{0}
n0{1} = (a{0,2,3}b{0,1,2} + a{0,1,2}b{0,2,3})(a{1,2,3}b{0,1,3} + a{0,1,3}b{1,2,3})
o0{1} = q0{0}(a{1,3}b{1,3}n0{1} + a{0,2}b{0,2}) + a{1,2}n0{0}
p0{1} = a{1,2}b{1,2} + a{0,3}b{0,3} + b{0,2}o0{1}
q0{1} = a{0,1}b{0,1}
w0{1} = c{1}j0{0}
mvsis> vl aluack.mv
Networks are combinationally equivalent according to simulation.
```

7.3 Example 3

The third example has both latches and external don't cares

```
.model latch
.inputs a b
.outputs f
.mv f 3
.mv x 3
.table x a b -> f
.default 2
1 - 0  1
0 - -  0
.reset x
0
.latch f x
.exdc
.inputs a b
.outputs f
.table a b -> f
.default 0
1 1 1
.end
```

Given the semantics of `default` and of `.exdc`, the previous table is equivalent to the following one:

```
.model latch
.inputs a b
.outputs f
.mv f 3
.mv x 3
.table x a b -> f
.default 2
1 - 0  1
0 0 -  0
0 - 0  0
1 0 1  2
2 0 -  2
2 - 0  2
0 1 1  (0,1,2)
1 1 1  (0,1,2)
2 1 1  (0,1,2)
.reset x
0
.latch f x
.end
```

7.4 Example 4

The following example shows an FSM.

```
.model lion9
.inputs i0 i1
.outputs ns o0
.mv i0 2
.mv i1 2
```

```
.mv o0 2
.mv ps, ns 9 st0 st1 st2 st3 st4 st5 st6 st7 st8
.latch ns ps
.reset ps
st0
.table i0 i1 ps -> ns o0
1 0 st0 st1 0
0 0 st0 st0 0
0 0 st1 st0 0
1 0 st1 st1 0
1 1 st1 st2 0
1 0 st2 st1 0
1 1 st2 st2 0
0 1 st2 st3 0
1 1 st3 st2 1
0 1 st3 st3 1
0 0 st3 st4 1
0 1 st4 st3 1
0 0 st4 st4 1
1 0 st4 st5 1
0 0 st5 st4 1
1 0 st5 st5 1
1 1 st5 st6 1
1 0 st6 st5 1
1 1 st6 st6 1
0 1 st6 st7 1
1 1 st7 st6 1
0 1 st7 st7 1
0 0 st7 st8 1
0 1 st8 st7 1
0 0 st8 st8 1
.end
```

The following external don't care network is produced:

```
.exdc
.inputs i0 i1 ps
.outputs ns o0
.mv ps 9
.table i0 i1 ps ->o0
.default 0
1 0  (3,{7-8})  1
0 1  ({0-1},5)  1
1 - 8 1
0 0  (2,6)  1
- 1 0 1
1 1 4 1
.table i0 i1 ps ->ns
.default 0
1 0  (3,{7-8})  1
0 1  ({0-1},5)  1
1 - 8 1
```

```
0 0  (2,6)  1
- 1 0 1
1 1 4 1
```

Then the mv network is encoded by `encode -i` (aliased by `enm`) and it is validated against the original one.

```
mvsis> enm
mvsis> wl
.model lion9
.inputs i0 i1
.outputs ns o0
.mv ns 9 st0 st1 st2 st3 st4 st5 st6 st7 st8
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->inf_ns_b2
.default 0
0 0 - 0 1 1 1
0 1 - 1 1 1 1
- 1 - 0 0 1 1
0 - 1 - - 1 1
1 - - - 0 1 1
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->o0
.default 1
0 0 0 - - 0 0
0 0 1 0 - 1 0
0 0 - 0 1 - 0
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->inf_ns_b1
.default 0
0 0 1 1 0 0 1
0 1 0 - - 0 1
0 1 1 0 1 - 1
0 1 1 - - 1 1
0 1 - 1 1 1 1
1 - - - 0 1 1
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->inf_ns_b0
.default 0
- 1 1 1 0 0 1
1 - - - 0 0 1
.table inf_ns_b0 inf_ns_b1 inf_ns_b2 inf_ns_b3 ->ns
.default st8
0 0 0 1 st0
0 0 0 0 st1
0 0 1 0 st2
0 0 1 1 st3
0 1 0 0 st4
0 1 0 1 st5
0 1 1 0 st6
0 1 1 1 st7
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->inf_ns_b3
.default 0
0 1 0 - 1 0 1
0 1 - 0 1 0 1
0 0 0 - 0 0 1
```

```
- - 1 - 0 1 1
- 1 - 0 0 1 1
1 - - - 0 1 1
.reset ps_b0
0
.latch inf_ns_b0 ps_b0
.reset ps_b1
0
.latch inf_ns_b1 ps_b1
.reset ps_b2
0
.latch inf_ns_b2 ps_b2
.reset ps_b3
1
.latch inf_ns_b3 ps_b3
.exdc
.inputs i0 i1 ps_b0 ps_b1 ps_b2 ps_b3
.outputs ns o0
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->o0
.default 0
0 - 1 0 0 0 1
- - 1 1 1 0 1
- 1 0 0 1 1 1
0 0 0 - 0 1 1
0 - 0 1 0 1 1
0 0 0 1 - 1 1
1 - - - 1 - 1
.table ps_b0 ps_b1 ps_b2 ps_b3 i0 i1 ->ns
.default 0
0 - 1 0 0 0 1
- - 1 1 1 0 1
- 1 0 0 1 1 1
0 0 0 - 0 1 1
0 - 0 1 0 1 1
0 0 0 1 - 1 1
1 - - - 1 - 1
.end
mvsis> validate lion9.mv
Networks are sequentially equivalent according to simulation.
```

The following example demonstrates `extract_seq_dc`. that computes sequential don't cares (unreachable states) and merges them with the exdc network. The example has multi-valued sequential don't cares (states $st5$ and $st6$ are unreachable from the initial state $st1$), to which additional don't cares are added after encoding because there is an unused code (101).

```
mvsis> rl lb2.mv
node ns is incompletely specified (exdc extracted)
node o0 is incompletely specified (exdc extracted)
mvsis> wl
.model lb2
.inputs i0 i1
.outputs ns o0
```

```
.mv ns 6 st1 st2 st3 st4 st5 st6
.mv ps 6 st1 st2 st3 st4 st5 st6
.table i0 i1 ps ->o1
0 0 st1 1
0 0 st2 0
0 0 st3 0
0 0 st6 1
0 1 st1 1
0 1 st3 1
0 1 st4 1
0 1 st5 1
1 0 st2 0
1 0 st4 0
1 0 st5 1
1 0 st6 1
1 1 st1 0
1 1 st3 0
1 1 st4 1
1 1 st5 0
1 1 st6 1
.table i0 i1 ps ->o0
0 0 st1 1
0 0 st2 1
0 0 st3 1
0 0 st6 1
0 1 st1 1
0 1 st3 0
0 1 st4 0
0 1 st5 1
1 0 st2 1
1 0 st4 1
1 0 st5 0
1 0 st6 0
1 1 st1 1
1 1 st3 1
1 1 st4 1
1 1 st5 1
1 1 st6 1
.table i0 i1 ps ->ns
0 0 st1 st1
0 0 st2 st2
0 0 st3 st2
0 0 st6 st1
0 1 st1 st1
0 1 st3 st3
0 1 st4 st3
0 1 st5 st1
1 0 st2 st4
1 0 st4 st4
1 0 st5 st5
1 0 st6 st5
```

```
1 1 st1 st3
1 1 st3 st3
1 1 st4 st4
1 1 st5 st3
1 1 st6 st6
.reset ps
st1
.latch ns ps
.exdc
.inputs i0 i1 ps
.outputs ns o0
.mv ps 6 st1 st2 st3 st4 st5 st6
.table i0 i1 ps ->o0
.default 0
0 0 (st4,st5) 1
1 0  (st1,st3)  1
- 1 st2 1
0 1 st6 1
.table i0 i1 ps ->ns
.default 0
0 0 (st4,st5) 1
1 0  (st1,st3)  1
- 1 st2 1
0 1 st6 1
.end
mvsis> extract_seq_dc
external don't care network has been created
mvsis> wl
.model lb
.inputs i0 i1
.outputs ns o0
.mv ns 6 st1 st2 st3 st4 st5 st6
.mv ps 6 st1 st2 st3 st4 st5 st6
.table i0 i1 ps ->01
0 0 st1 1
0 0 st2 0
0 0 st3 0
0 0 st6 1
0 1 st1 1
0 1 st3 1
0 1 st4 1
0 1 st5 1
1 0 st2 0
1 0 st4 0
1 0 st5 1
1 0 st6 1
1 1 st1 0
1 1 st3 0
1 1 st4 1
1 1 st5 0
1 1 st6 1
```

```
.table i0 i1 ps ->o0
0 0 st1 1
0 0 st2 1
0 0 st3 1
0 0 st6 1
0 1 st1 1
0 1 st3 0
0 1 st4 0
0 1 st5 1
1 0 st2 1
1 0 st4 1
1 0 st5 0
1 0 st6 0
1 1 st1 1
1 1 st3 1
1 1 st4 1
1 1 st5 1
1 1 st6 1
.table i0 i1 ps ->ns
0 0 st1 st1
0 0 st2 st2
0 0 st3 st2
0 0 st6 st1
0 1 st1 st1
0 1 st3 st3
0 1 st4 st3
0 1 st5 st1
1 0 st2 st4
1 0 st4 st4
1 0 st5 st5
1 0 st6 st5
1 1 st1 st3
1 1 st3 st3
1 1 st4 st4
1 1 st5 st3
1 1 st6 st6
.reset ps
st1
.latch ns ps
.exdc
.inputs i0 i1 ps
.outputs o0 ns
.mv ps 6 st1 st2 st3 st4 st5 st6
.table ps i0 i1 ->o0
.default 0
(st5,st6) - - 1
st2 - 1 1
 (st1,st3)  1 0 1
(st4,st5) 0 0 1
.table ps i0 i1 ->ns
.default 0
```

```
(st5,st6) - - 1
st2 - 1 1
 (st1,st3)  1 0 1
(st4,st5) 0 0 1
.end
mvsis> enm -v
node inf_ns is encoded as: value 0 - 001 value 1 - 010 value 2 - 000 value 3 - 011 value 4 - 100 value 5 -
mvsis> extract_seq_dc
external don't care network has been created
mvsis> wl
.model lb
.inputs i0 i1
.outputs ns o0
.mv ns 6 st1 st2 st3 st4 st5 st6
.table ps_b0 ps_b1 ps_b2 i0 i1 ->inf_ns_b2
.default 0
1 0 0 0 1 1
0 0 1 0 - 1
0 1 - 1 0 1
- 1 1 1 1 1
1 1 - 0 0 1
1 1 - 1 1 1
.table ps_b0 ps_b1 ps_b2 i0 i1 ->01
.default 1
- 0 0 1 1 0
0 - 0 0 0 0
0 1 - 1 0 0
0 0 - 1 1 0
.table ps_b0 ps_b1 ps_b2 i0 i1 ->o0
.default 1
0 1 1 0 1 0
1 - 0 1 0 0
0 0 0 0 1 0
1 1 - 1 0 0
.table ps_b0 ps_b1 ps_b2 i0 i1 ->inf_ns_b1
.default 0
0 - 0 0 0 1
0 1 1 1 - 1
1 1 - 1 1 1
0 1 - 1 0 1
.table ps_b0 ps_b1 ps_b2 i0 i1 ->inf_ns_b0
.default 0
1 - 0 1 0 1
1 1 - 1 - 1
.table inf_ns_b0 inf_ns_b1 inf_ns_b2 ->ns
.default st6
0 0 1 st1
0 1 0 st2
0 0 0 st3
0 1 1 st4
1 0 0 st5
```

```
.reset ps_b0
0
.latch inf_ns_b0 ps_b0
.reset ps_b1
0
.latch inf_ns_b1 ps_b1
.reset ps_b2
1
.latch inf_ns_b2 ps_b2
.exdc
.inputs i0 i1 ps_b0 ps_b1 ps_b2
.outputs inf_ns_b2 o0 inf_ns_b1 inf_ns_b0 ns
.table ps_b0 ps_b1 ps_b2 i0 i1 ->o0
.default 0
1 - - - - 1
- 1 0 - 1 1
0 0 - 1 0 1
- 1 1 0 0 1
.table ps_b0 ->inf_ns_b2
.default 0
1 1
.table ps_b0 ->inf_ns_b1
.default 0
1 1
.table ps_b0 ->inf_ns_b0
.default 0
1 1
.table ps_b0 ps_b1 ps_b2 i0 i1 ->ns
.default 0
1 - - - - 1
- 1 0 - 1 1
0 0 - 1 0 1
- 1 1 0 0 1
.end
```

8 Caveats

1. `MVSIS` only works correctly on deterministic networks, i.e. ones where any primary output as a function of the primary inputs, has at most one value per minterm. We do not check for non-determinism. If a network is non-deterministic, it can result in a new network that is not equivalent to the original. If a node is incompletely specified, unspecified minterms are assigned to the default value.

2. `MVSIS` can be applied to binary files by using the `read_blif` command. The results can be compared to those obtained by `SIS`. At this time we are still tuning the algorithms in `MVSIS` and comparing with `SIS` on binary files. `SIS` has a set of filters which are used to estimate when a result may blow up and if so it will not do the computation. We are experimenting with similar filters in `MVSIS`. Nevertheless, `MVSIS` is pretty competitive with `SIS` in terms of speed and results.

3. In general, some `MVSIS` commands do not have filters yet. Some examples are `reset_default` which computes the default value for each node and chooses another value-function to be the default if it is larger. This computation uses the complement operation and may blow up. We do have filters on `fullsimp`, `factor` and `eliminate`.
4. Another source of slow behavior can be `resub`, if there are many nodes in the network. As mentioned, formal verification can be done by writing a file and using `VIS`. For now, we have not built in a formal verification method, like `SIS` which can compute BDDs for two networks and compare them. Our verification is only by simulation through the `validate` command.
5. `MVSIS` is only being made available as an executable running under LINUX. This saves us a lot of development effort that would be expended in releasing source code which is compilable on various machines running under different environments. Perhaps in the future, source code can be released so that users can experiment with adding different algorithms, as `SIS` is used in academia and in industry. It is easy to port the code to `alphas` and `SUNs`. If there are users who desire `MVSIS` source code, we may work with individuals to make it available under special requests.
6. A `BLIF-MV` file can be generated using `vl2mv` which translates `verilog` to `BLIF-MV`. `vl2mv` is available as part of `VIS`.
7. The missing minterms of an incompletely specified PIPO node (i.e., a node whose outputs are primary outputs and whose inputs are primary inputs) are added to the external don't care set; whereas the missing minterms of an incompletely specified non-PIPO node are assigned to the current default value for that node.

Acknowledgements

We gratefully acknowledge the support of the SRC in funding this project under contract SRC-683.004. In addition, the logic synthesis class of Spring 1999 at Berkeley started `MVSIS` as a combined class project under the guidance of Subarnarekha Sinha, class TA. The class members were Yunjian Jiang, Niraj Shah, Scott Weber, Heloise Hse, Fernando De Bernardinis, David Chinnery, and Rupak Majumdar.

References

[1] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[2] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of the Intl. Conf. on Computer Design*, pp. 328–333, Oct. 1992.

[3] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang, "BLIF-MV: An Interchange Format for Design Verification and Synthesis," Tech. Rep. UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Nov. 1991.

[4] H. Savoj and R. K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks," in *Proc. of the Design Automation Conf.*, pp. 297–301, June 1990.

[5] M. Gao and R. K. Brayton, "Multi-valued multi-level network decomposition," in *Proc. of the Intl. Workshop on Logic Synthesis*, June 2001.