MVSIS Code Generation Manual

Yunjian Jiang Robert Brayton
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{wjiang,brayton}@eecs.berkeley.edu

1 Introduction

This document describes the code generation feature in the MVSIS v1.1 release, including the design flow from high-level specifications and usage of the commands supported.

2 Specifications

When targeted for embedded system applications, the pure MV logic network is extended to have abstract data variables. These variables can be thought of as carrying an infinite range of values, which in the actual implementation can be mapped to any arbitrary type. Three additional types of nodes are supported for functions involving data variables: expressions, multiplexers and predicates. These are classified according to their input and output variables types, as shown in Table 1. In the examples in the table, *a*, *b*, *c* are multi-valued control variables, and *x*, *y* are data variables.

A multiplexer is defined as $f = f(y_c, y_0, \ldots, y_{n-1})$, where y_c is a MV-variable with n values, y_i, ($i \in [0, n-1]$) are data inputs. The output f is assigned to y_i if $y_c = i$. The data computation contained in predicate nodes and expression nodes are currently modeled as uninterpreted strings, but they must be arithmetic as definable by the semantics of the C language. As a result, these nodes cannot be reasoned about or simulated inside the MVSIS environment (only the control nodes can). The behavior of the entire network can be simulated only by generating C programs and then running the compiled program separately.

node types	operation	input	output	example
control	logical	MV	MV	$a\{0\}b\{1,2\}+a\{1\}$
data	arithmetic	data	data	`x + y`
multiplexer	assignment	MV/data	data	$c\{0\}$`x`$+c\{1\}$`y`
predicate	predicate	data	MV	`x > y`

Table 1: Node types in a control data network

MVSIS supports sequential MV-networks with multi-valued latches, i.e. storage devices that can hold any of a set of values, and latches for data variables.

Figure 1 shows an example of a control-data network with two latches, where bold wires indicate data variables. These networks can be derived from Esterel programs, by the following tool flow (for an Esterel program named simple.strl):

```
% esterel -causal simple.strl
% areaopt simple
% blifsc -ctbl simple.ctbl simple.opt.blif > simple.opt.sc
% scdc simple.opt.sc
% perl dc2mv.pl -p simple.opt.dc > simple.mv
```

Where `esterel, areaopt, blifsc, scdc` are tools released with the Esterel compiler. `esterel` is the Esterel compiler, which takes an Esterel program, performs static analysis, and generates a single extended finite state machine (EFSM) representation. Multiple modules, if specified in the Esterel program, will be synchronously composed into a single EFSM, which is the product machine of all modules. The result is an intermediate format that consists of a control portion in `BLIF` file and a data portion in a `ctbl` file . `areaopt` is logic optimization tool extended from SIS [1], which uses area minimization algorithms and latch removal [2]. Then `blifsc` combines the optimized control portion and the data table, and produces an intermediate format called sorted code (SC), which is in turn translated into declarative code (DC [3]) with `scdc`.

`dc2mv` is a Perl-based parser released with MVSIS, which converts the DC format into an extended BLIF-MV format. It works only on a subset of the DC formats generated by `scdc`. Specifically the following features in DC are not supported:

- Package tables, action tables, instance tables, node calls and action calls. It is assumed there is one package which defines a set of nodes. The nodes cannot be instances of other nodes defined elsewhere. It turns out that almost all DC files produced by Esterel satisfies this constraint.

- Structured types (array and record). With target application in control systems and not data processing, this is not a major limitation.

- Case and window statements for `equ` and `memo` construct.

- Dynamic variable dependency. In the data-flow like specification in the DC format, each definition can potentially be guarded by the dynamic value of a control variable. The current tool does not support this feature and assumes that all definitions are effective at run time, i.e. guarded by constant one. This is a limitation for some designs, e.g. the wristwatch example by G. Berry.

The DC format is a shared format among a number of synchronous languages, e.g. Lustre, Signal, Argos and StateChart. This makes MVSIS a common backend optimization and mapping tool for synchronous applications developed with all these languages.

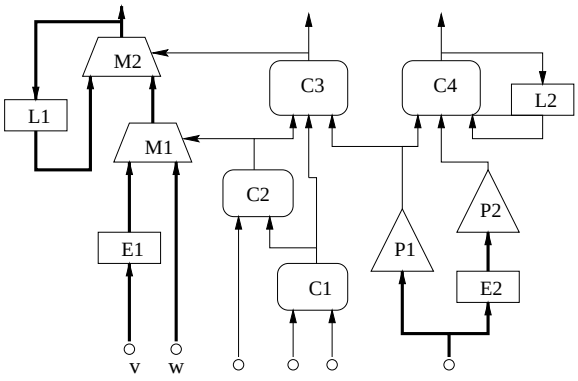

Figure 1: Control-data network

3 Software Synthesis

The software synthesis problem is, given such a multi-level control-data network, generate an efficient software implementation in C (or target dependent machine language), such that appropriate constraints are satisfied. These may include resource constraints, like RAM and ROM usages, and timing constraints. This is a mapping problem from a parallel execution model, originally targeted for hardware, to a sequential execution model, where all tasks share the same processing unit. The goal is to generate fast sequential simulation code with constraints on the code size. This may be achieved by reducing the time spent on each node, and reducing the total number of nodes that has to be evaluated. Therefore two sub-problems need to be solved: code generation for individual logic functions, and scheduling of node execution, detailed in the remainder of this section.

The code generation algorithm in MVSIS explores logic function evaluation using different logic representations, based on multi-valued decision diagrams (MDDs) and sum-of-products (SOPs). It also explores the node scheduling problem with static, quasi-static and dynamic scheduling.

Command `gen_c` takes an MV network and produces a C program. The generated program contains a main routine that generates random input vectors, and a network evaluation routine that simulates the network and returns the output vector. The C file is self-contained and is able to compile by itself across different platforms using different compilers.

Following options are supported for logic function evaluation (these assumes a static scheduling meaning all nodes are evaluated in a topological order):

- *SOP* Use sum-of-products representation of the logic functions. An input minterm is evaluated again each cube in the SOP table, by a single AND instruction [4]. There is extra run-time cost associated with aligning the bits of input variables in a one-hot encoded vector.

- *ZIP* Use a compressed version of the SOP, with priority don't cares. See the description for command `compress` below.
- *MDD* Use an MDD representation and generate branching program from the internal BDD structure. When an MV variable needs to be tested, the binary bits that are used for internal encoding is tested (similar to a binary search for the MV value).
- *ORDER* Same as the MDD approach, except that dynamic ordering is applied for each intermediate node.
- *HYBRID* Combines both SOP and ORDER approach, and uses SOP equations for small nodes (below three inputs) and ORDER approach for large nodes. This is designated as the default option.

In addition to static scheduling, the following two options are supported, which both uses the default hybrid approach for function evaluation. (refer to [5] for technical details)

- *Trigger* The triggerability of a node is computed statically, using the maximal observability don't care set (MODC) computed by command `fullsimp`. Nodes that are always observable at the primary outputs are statically scheduled; nodes that are possibly non-observable are scheduled at run time.
- *Event-driven* In this approach, all nodes are scheduled at run-time. A bit-vector is generated, with one bit associated with each node in the network indicating whether this node has been evaluated or not. For some applications, this gives superior performance.

Command `compress` is used for the generation of the *ZIP* code. It was discovered that given an ordering of all output MV values, the cubes of an i-set can be used as don't cares to minimized the i-sets lower in the order. This is called priority don't cares [4]. `compress` applies heuristics for the priority ordering of output values, for all control nodes in the network, and store the minimized i-sets in a separate structure. The ordering heuristics are described in [6].

An example run of the synthesis flow starting from BLIF-MV specification (assuming an event-driven scheduling with hybrid approach for function evaluation):

```
% mvsis> read_blifmv simple.mv
% mvsis> source mvsis.scriptd
% mvsis> fullsimp -d -s
% mvsis> compress
% mvsis> gen_c -m trigger simple.c
```

4 Comments

1. For examples that contain externally defined constants, variables and procedures, the `dc2mv` parser can produce the corresponding header file to be compiled with

the generated C program. The procedure themselves need to be defined in order to compile successfully.

2. Future work of the MDD-based approach is to explore MDD minimization using don't cares and free variable ordering. Results have been reported that such minimization could improve the code quality [7]. Specifically, we could use the CODC set computed by `fullsimp` to minimize the MDD representation. Future work would also explore generalized factoring diagrams as a new representation tuned for functional evaluation [5].

3. The data-path representation could be extended from the current black-box approach (uninterpreted strings) to Presburger arithmetic, as preliminary results show benefit of using this model [8].

Appendix: Extended BLIF-MV

The new BLIF-MV file format accepted by MVSIS v1.1 extends the one defined in MVSIS v1.0 manual with support for data-path, as defined below.

- **Abstract data** variables are specified using the `.n` construct:

```
.n local_time
```

- **Multiplexer** node assumes the first input is an MV variable, whose value range is at least the number of data input variables:

```
.mv time_zone 3
.n pacific_standard_time
.n beijing_time
.n london_time
.mux time_zone pacific_standard_time beijing_time lon-
don_time -> local_time
0 - - -  =pacific_standard_time
1 - - -  =beijing_time
2 - - -  =london_time
```

- **Expression** node takes an un-interpreted string (contained in double quote marks "") as input, which is assumed to conform to the semantics of the C language, and produces a data output.

```
.data pacific_standard_time -> pacific_daylight_time
"pacific_standard_time + 1"
```

- **Predicate** node is the same as an expression node, except that the output is a control variable (MV).

```
.data pacific_standard_time -> at_night
"(pacific_standard_time > 18) && (pacific_standard_time < 6)"
```

Acknowledgement

The authors would like to acknowledge Max Chiodo and Luciano Lavagno from Cadence Berkeley Labs for there insightful discussions and help in programming aspects. We gratefully acknowledge the support of the SRC in funding this project under contract SRC-683.004

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 Laboratory, Univ. of California, Berkeley, CA 94720, May 1992.

[2] E. M. Sentovich, H. Toma, and G. Berry, "Latch optimization in circuits generated from high-level descriptions," in *Proc. of the Intl. Conf. on Computer-Aided Design*, pp. 428–35, Nov. 1996.

[3] N. Halbwachs, *The declarative code DC, version 1.2a*. Vérimag, Grenoble, France, October 1995. unpublished report.

[4] Y. Jiang and R. K. Brayton, "Logic optimization and code generation for embedded control applications," in *Proc. of the Intl. Symposium on Hardware/Software Co-Design*, Apr. 2001.

[5] Y. Jiang and R. K. Brayton, "Software synthesis from synchronous specifications using logic simulation techniques," in *Proc. of the Design Automation Conf.*, June 2002.

[6] E. Dubrova, Y. Jiang, and R. K. Brayton, "Minimization of multiple-valued functions in post algebra," in *Proc. of the Intl. Workshop on Logic Synthesis*, Jun. 2001.

[7] C. Kim, L. Lavagno, and A. Sangiovanni-Vincentelli, "Free MDD-based software optimization techniques for embedded systems," in *Proc. of the Conf. on Design Automation & Test in Europe*, Mar. 2000.

[8] Y. Jiang and R. K. Brayton, "Don't care computation in minimizing extended finite state machines with presburger arithmetic," in *Proc. of the Intl. Workshop on Logic Synthesis*, Jun. 2002.