BLIF-MV: AN INTERCHANGE FORMAT FOR DESIGN VERIFICATION AND SYNTHESIS

by

R. K. Brayton, M. Chiodo, R. Hojati, T. Kam,
K. Kodandapani, R. P. Kurshan, S. Malik,
A. Sangiovanni-Vincentelli, E. M. Sentovich,
T. Shiple, K. J. Singh, H. Y. Wang

Memorandum No. UCB/ERL M91/97

1 November 1991

BLIF-MV: AN INTERCHANGE FORMAT FOR DESIGN VERIFICATION AND SYNTHESIS

by

R. K. Brayton, M. Chiodo, R. Hojati, T. Kam,
K. Kodandapani, R. P. Kurshan, S. Malik,
A. Sangiovanni-Vincentelli, E. M. Sentovich,
T. Shiple, K. J. Singh, H. Y. Wang

Memorandum No. UCB/ERL M91/97

1 November 1991

BLIF-MV: AN INTERCHANGE FORMAT FOR DESIGN VERIFICATION AND SYNTHESIS

by

R. K. Brayton, M. Chiodo, R. Hojati, T. Kam
K. Kodandapani, R. P. Kurshan, S. Malik,
A. Sangiovanni-Vincentelli, E. M. Sentovich,
T. Shiple, K. J. Singh, H. Y. Wang

Memorandum No. UCB/ERL M91/97

1 November 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BLIF-MV: AN INTERCHANGE FORMAT FOR DESIGN VERIFICATION AND SYNTHESIS

by

R. K. Brayton, M. Chiodo, R. Hojati, T. Kam
K. Kodandapani, R. P. Kurshan, S. Malik,
A. Sangiovanni-Vincentelli, E. M. Sentovich,
T. Shiple, K. J. Singh, H. Y. Wang

Memorandum No. UCB/ERL M91/97

1 November 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BLIF-MV: An Interchange Format for Design Verification and Synthesis

R.K. Brayton M. Chiodo R. Hojati T. Kam K. Kodandapani R.P. Kurshan S. Malik
A. Sangiovanni-Vincentelli E.M. Sentovich T. Shiple K.J. Singh H.Y. Wang

November 1, 1991

Abstract

This document briefly describes an interchange format for the specification of systems to be used as inputs to formal verification programs. The underlying system model is one of interacting, possibly non-deterministic state machines. The proposed format meets the requirements for an interchange format, namely simplicity and generality. Additionally, this format serves as an input format for digital synthesis systems and provides a suitable interface to the formal specification languages VHDL, Verilog and SDL. The proposed format is a natural extension of the BLIF format (Berkeley Logic Interchange Format), which is currently accepted as a standard for specifying digital systems at the logic level.

1 Introduction

At the *Workshop on Computer-Aided Verification* (Rutgers University, June 1990) a forum was established to discuss the creation of an interchange format for describing coordinating systems. This format should enable the exchange of design examples as well as comparisons of the performance of various verification systems on the same set of designs. Specifically, it was decided that the format selected should have the following attributes:

A. Simplicity: It must be unambiguous, simple to read and understand, and easy to manipulate. Writing a parser should be a relatively simple task.

B. Generality: It should be capable of describing a large range of hardware/software systems.

Among the suggestions which emerged from the forum was the KISS2 format [1] used as an interchange format by the logic synthesis community for describing finite state machines. The motivation behind this suggestion was that the systems being verified were state machines of some kind, so it made sense to use existing formats for describing them. In addition the same format can be (and is being) used for automatic synthesis. However, the following limitations of KISS2 were pointed out which could make it unsuitable for use as is:

1. It permits definition of only one state machine at a time. The systems of interest are networks comprised of interconnected state machines.
2. The state machine behaviors permitted are deterministic. Specification of non-deterministic behavior is often needed.
3. The behavior descriptions are two-level, i.e. the description is in the form of a PLA or two-level logic. While this may be suitable for representing control logic, it may lead to an inefficient or even infeasible representation for data path logic with arithmetic functions.
4. Various parallel composition schemes cannot be described.

The format proposed in this document overcomes these limitations and has the desired properties stated earlier. It is an extension of the BLIF (Berkeley Logic Interchange Format) used by the logic synthesis community [1] to exchange circuit descriptions at the logic level (gate-level net lists), and incorporates the KISS2 format as a subcase. BLIF is used to describe digital circuits and thus signals/variables described there have binary values. However, our current needs involve general systems with variables, as in KISS2, which are not restricted to binary values. The extension of BLIF to handle multiple-valued (or symbolic) variables is simple. The proposed format subsumes the KISS2 description of single, deterministic machines with two-level behavior descriptions, and is powerful enough to support a variety of parallel composition paradigms for

interacting, non-deterministic machines with multi-level descriptions of behavior. The ability to specify multi-level behavior is essential for describing some types of circuits, such as arithmetic functions (whose two-level representations are exponential in the number of inputs). Specifically, the proposal has these additional attributes:

C. Non-determinism: The format supports non-determinism, which is important for representing abstraction, modelling delays, and defining system constraints such as fairness through the use of automata. Non-determinism is also used as the basis of support for various parallel composition paradigms (see below).

D. Synthesis: The format is compatible with a large number of existing hardware synthesis tools (MISII, SIS, NOVA, JEDI, etc).

E. Standards: In addition to supporting an interface with a variety of verification and simulation tools, the format supports an interface to substantial subsets of the formal specification languages VHDL, Verilog, SDL, and perhaps others, and can be easily generated from HDF (Hardware-Data Flow), the data structure used in BLIS (Behavior-to-Logic Interactive Synthesis) [4].

We propose using this extended format as the interchange format. We call it BLIF-MV since it is the multiple-valued extension of BLIF.

The issue of parallel composition requires further discussion. As there are a multitude of different notions of parallel composition, this could become a complicated issue if parallel composition operators were introduced directly into the format. Rather, it is proposed to have a very simple format which supports a large variety of parallel composition semantics at a meta-level. Specifically, it is proposed to have a format whose base semantics is purely synchronous, but which, through the use of non-determinism, is capable of modelling asynchrony, delay, interleaving semantics, true concurrency and so on, at the meta- or interpretive level. For example, asynchrony may be modelled, as in [3], through a non-deterministic choice of "pause" or "ready", where "pause" inhibits changing state, in each state machine at an asynchronous interface. Interleaving semantics can be modelled through non-determinism and a simple constraint preventing simultaneity of events. (In systems with implicit FIFO queues, these would become explicit, with the caveat that we limit ourselves to finite state systems.) True concurrency can be modelled likewise, utilizing non-determinism and "projections" to a global state. In summary, rather than to support explicitly a multitude of parallel composition operators, we propose a format which allows, through non-determinism, a variety of models of parallel composition to be represented.

There are some additional representation issues which thus far we have not addressed. While we have proposals for all of these, we feel they should be discussed in the wider forum. As their particular resolution does not conflict with the BLIF-MV format proposal, but rather augments that proposal, we separate these issues from the basic proposal.

1. Sequential constraints: It is often necessary to impose constraints upon sequences of events, such as automaton acceptance conditions and fairness. Apparently, all such constraints can be represented through automata. As the proposed format already supports automata transition structures, the issue may be narrowed to acceptance structures. Acceptance structures all are defined in terms of sets of states, sets of sets of states, sets of edges, and sets of pairs of these. Therefore, a fairly simple addition to the format could be provided to specify acceptance structures and outcome (accept or reject).

2. Dense time: Discrete time can be introduced simply through counters. Recent interest in dense time (e.g. [2]) warrants support of this in the interchange format. Designation of minimum and maximum sojourn times in a state (as a function of state) appear to support all known needs in this regard. This can be represented using multi-valued variables understood to represent real time, without the need for any additional augmentation of the proposed format. These variables may be evaluated against minimum and maximum sojourn times using non-deterministic look-up tables. For simulation, these variables could be connected to real timers.

3. Stochastic behavior: A system of coordinating state machines defined through the proposed format may be interpreted as a stochastic process by designating a density distribution for the sojourn time in a state (in place of the bounds in 2. above), and probabilities on non-deterministic choice. Density distributions could be designated from a library (standard or user-supplied) through the use of embedded C-code in the machine specification, and such a designation would be easy to integrate into the proposed basic format. Defining the stochastic properties of a coordinating system in this fashion is completely general. Allowing the user to embed C-code in a machine specification provides a mechanism for specifying many constructs not explicitly supported by the format (however, such embedded code will be treated like a black box, and no analysis or verification will be performed on the code).

in1	in2	in3	out
a	green	in	on
b	red	out	off
c	blue	out	ready
c	blue	out	pause

Figure 1: Non-deterministic look-up table with multi-valued input variables in1, in2, in3, and multi-valued output variable out. Note the non-determinism represented by the last two rows.

4. Macro expansion: A macro facility could be important with respect both to succinctness and to tractability (in the case of analysis, which may operate on the macro representation rather than the expansion). This is especially important in the case of arrays of state machines (a set of coordinating state machines distinguished by index), arrays of variables, arithmetic expressions and relations. A multiplexor macro would be very useful for implementing general if-then-else type clauses with multi-valued variables. Such a facility could be integrated readily into the proposed format.

5. Input/Output encoding: The look-up table specification (see section 2) can be given with the inputs and outputs in symbolic form (as in KISS2) or in one-hot-encoded form using positional notation as used by ESPRESSO-MV. (Even in one-hot-encoded form, the codes can be interpreted as symbolic.) One-hot-encoded form is preferred because it is easier to parse and is identical to the ESPRESSO-MV format, which is the format already in use in the two-level, multi-valued logic minimizer ESPRESSO-MV.

6. Annotation: It may be desirable to annotate each machine with information about its behavior. For example, annotating a machine with information revealing its function (e.g. multiplier) would preserve some of the information passed down from a higher-level specification, and make it easier for a higher-level tool to recognize and utilize certain behavior. A simple extension of the format could be incorporated for the purpose of annotating machines.

2 Finite State Machine Specification

In this section, the specification of a single finite state machine will be described; the specification of a system of interacting finite state machines is described in the next section.

A description of a single finite state machine contains a list of input variables, a list of output variables, the transition structure, the output functions, and optionally a set of initial states and external don't care functions for the output and transition functions. For verification purposes, it may not be desirable to have each signal (e.g. each output or state variable) encoded (no encoding may be necessary to verify a design). For this purpose, multi-valued variables may be used for all signals in the machine (binary-valued variables being a special case). The transition and output behavior are described in terms of the inputs. This behavior is represented by a DAG, where each vertex is a multiple-input, single-output look-up table (see Figure 1), and each edge represents the dependency of one table on the output of another (in the figure, symbolic variable names are used rather than one-hot-encoded names). The output representing the state variable can be used in subsequent time periods by connecting it to the inputs of the look-up tables through a feedback latch (see Figure 2). Similarly, the latches can be used to connect logic blocks in a data path. A purely combinational block of logic can be described by using an interconnection of look-up tables with no latches.

By using a DAG of look-up tables (rather than a single look-up table), complex logical behavior (such as arithmetic functions) can be specified in this multi-level format that cannot be compactly described in a single look-up table. In addition, non-determinism in a machine can be described easily, using several lines in the look-up table to specify the several outputs resulting from a single input (Figure 1).

The specification of a single finite state machine is achieved with the **.model** construct, which introduces a new state machine by name:

```
.model model_name [model_attributes]
```

Optionally, attributes of the model can be specified, such as cell area, pin-to-pin delays, or a tag such as "black box" that indicates to synthesis and verification programs that the interior of the model is not to be altered.

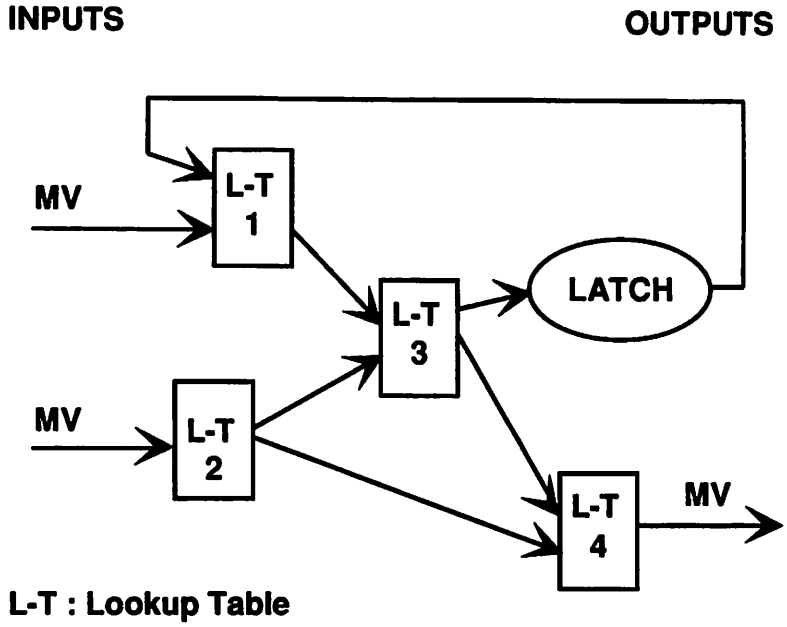

Figure 2: Network Representation of a Single Finite State Machine

The state machine input and output multi-valued variable name lists are specified using **.inputs** and **.outputs** constructs as follows:

```
.inputs in1 in2 ...
.outputs out1 out2 ...
```

Each multi-valued variable is given a set of values that it can assume with the **.mv** statement:

```
.mv symbolic_name n_values [value_0 value_1 ... value_n-1]
```

The **symbolic_name** is the name of the variable and **n_values** is the number of values that it can take. This is followed optionally by a list of the values. If the list is omitted, nonnegative integers are assumed. For example, **.mv x 3** indicates that the values **x** can take on are **0, 1, 2.**

For two variables to be connected (e.g. the output of one table to the input of another) they must have the same number of possible values. It is assumed that the first value of one variable is equivalent to the first value of the other, the second listed values are equivalent, and so on.

Each look-up table is specified by the **.names** construct as in BLIF, but taking multi-valued inputs and producing a multi-valued output. In the following example, **in1, in2,** ... are names of input signals to the look-up table (although not necessarily primary inputs to the state machine), and **out** is the name of the multi-valued output signal produced by this particular look-up table. The subsequent lines define the transition structure: **in1_value** is one of the values in the domain of **in1, out_value** is one of the values in the domain of **out,** and so on (see Figure 1).

```
.names in1 in2 ... out
in1_value_1 in2_value_1 ... out_value_1
in1_value_2 in2_value_2 ... out_value_2
.
.
.
```

Signals which define the next state are fed into a **.latch** construct (which implements delay). These signals can be fed back to the DAG and used as input in the next time period. The syntax for **.latch** is as follows:

```
.latch latch_input latch_output
```

where **latch_input** is a (multi-valued) variable which is either an input to the machine or the output of a look-up table, and **latch_output** is a (multi-valued) variable which is either an input of a look-up table or an output of the machine.

The initial state(s) of the machine are specified with the **.r** ("reset") construct:

```
.r latch1_output=latch1_value_1 latch2_output=latch2_value_1 ...
.r latch1_output=latch1_value_2 latch2_output=latch2_value_2 ...
.
.
.
```

where **latch1_output** is the name of the output of a particular latch, and **latch1_value** is a particular value in the range of the (multi-valued) latch output. Multiple reset states can be specified by including multiple **.r** constructs, at any level of the hierarchy.

Finally, the external don't care functions are specified in a similar way to the output and transition functions, and preceded with the **.exdc** construct:

```
.exdc
.names in1 in2 ... out
in1_value in2_value ... out_value
.names in1 in2 ... next_state
in1_value in2_value ... next_state_value
.
.
.
```

The following example illustrates the simple conversion from a machine described in KISS2 format to one described in BLIF-MV format. The state table in the KISS2 format is essentially duplicated to produce two tables describing the behavior of the state variable and the output variable, and a latch is added to model the delay in latching the state (this is fed back to the two tables for use in the subsequent computation). KISS2 description:

```
.i 1                  # number of inputs
.o 1                  # number of outputs
.s 3                  # number of states
stop st0 st2 off      # transition structure
go   st0 st1 on
stop st1 st1 off
go   st1 st0 on
stop st2 st2 off
go   st2 st0 on
.e
```

The corresponding BLIF-MV description is as follows:

```
.model example
.inputs vin                          # input variable name list
.outputs vout                        # output variable name list
.names vin prev_state next_state     # next state logic
stop st0 st2
go   st0 st1
stop st1 st1
go   st1 st0
stop st2 st2
go   st2 st0
.names vin prev_state vout           # output logic
```

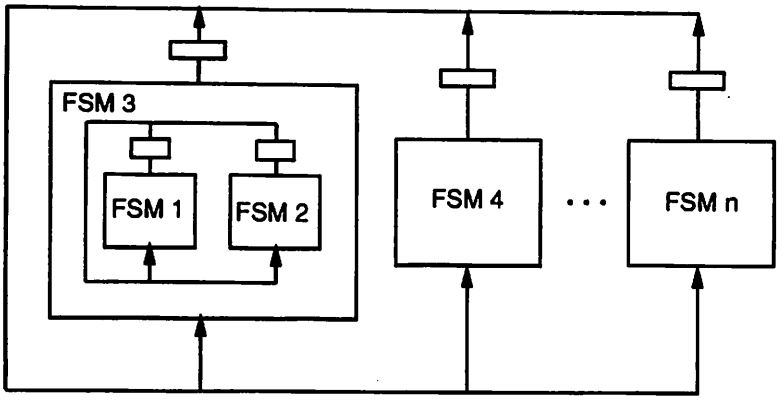

Figure 3: System of Communicating Finite State Machines

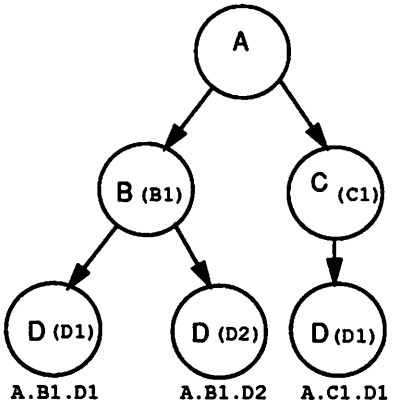

Figure 4: Unique Naming of Instances

```
stop st0 off
go   st0 on
stop st1 off
go   st1 on
stop st2 off
go   st2 on
.latch next_state prev_state      # latch for state variable
.end
```

3 System Specification

The previous section described how to create a single state machine. These machines, each of which is defined by a **.model** declaration, can be connected together to generate the system specification (see Figure 3). The hierarchical interconnection of these state machines is defined using the **.subckt** construct, allowing one to refer to **.models** within **.models**. The **.subckt** command is followed by a **model_name**, which refers to a specific state machine; this is followed by **i_name**, which is a unique name given to this particular instantiation within the model **model_name**. Each subcircuit can be uniquely named by prepending the instance name with a string of instance names that lead to that instance in the hierarchy. For example, in Figure 4, there are three occurrences of model D (the circles represent models, the names are model names, the names in parentheses are instance names). The two instances of D that are called by B are uniquely named D1 and D2. The three D instances are uniquely referred to as A.B.D1, A.B.D2, and A.C.D1. Finally a list of items of the form **io_pin=signal_name** is given, where **io_pin** is one of the input or output variables in the called model, and **signal_name** is any variable in the calling

model, and instance_attributes, **i_attributes**, such as "black box" (see Section 2) may be given. The syntax of the **.subckt** construct is as follows:

```
.subckt model_name i_name io_pin=signal_i . . . [i_attributes]
```

This mechanism will define the use of the model **model_name** and declare it's connectivity. Additionally, it allows an arbitrary hierarchy of models (correspondingly of state machines).

As an example, consider a system made up of two portions, `A` and `B` that communicate with each other. In turn `B` is made up of two parts `C` and `D` . The specification of such a system may appear as —

```
.model system  # The root of the hierarchy
.inputs
.outputs
.subckt A terminal=signal     # connectivity specification between
.subckt B terminal=signal     # the components and system terminals
.end      # of the system composed of A and B

.model A  # The description of model A
.inputs
.outputs
   .
   .                          # Definition of the model for A
   .
.end    # of model A

.model B
.inputs
.outputs
.subckt C terminal=signal  # connectivity specification
.subckt D terminal=signal  # between C, D and the terminals of B
.end  # of model B composed of C and D

.model C
.inputs
.outputs
   .
   .    # Definition of the model for C
   .
.end  # of model C

.model D
.inputs
.outputs
   .
   .    # Definition of the model for D
   .
.end  # of model D
```

Only the terminals declared as **.inputs** and **.outputs** can be used in expressing the connectivity between models (i.e. internal names in a model cannot be used to express connectivity). Thus the terminals (inputs and outputs) and the attributes of a model are the only abstractions that are seen by another state machine (model).

This format by itself does not enforce any particular delay semantics (one may specify purely combinational cycles); it is the designer's responsibility to ensure that no such conditions exist in the specification.

4 Concluding Comments

In conclusion we would like to highlight the strengths of BLIF-MV as an interchange format.

1. It is a simple description format. It is simple to read and understand, and utilizes the simplicity of a synchronous system for describing a wide range of coordination mechanisms.
2. It is a general interchange format. The same format can be used to describe either structure or behavior or a combination of both.
3. It supports non-determinism, which can be used for representing abstraction, modelling delays, defining system constraints, and supporting various parallel composition paradigms.
4. It is compatible with existing hardware synthesis tools. It encompasses the KISS2 description of single, deterministic state machines. When all the signals are binary-valued and deterministic, it reduces to the accepted BLIF description for a digital circuit.
5. It supports an interface to formal specification languages. In particular, it can serve as a gateway to other standard languages (e.g. VHDL and Verilog) because there is a simple map from a substantial subset of these higher-level languages to this format.
6. It supports an interface to hardware synthesis systems. The format subsumes KISS2, which is currently used as the input language for many state assignment and state minimization programs, and BLIF, which is used as the input language to many logic synthesis systems.

References

[1] Robert Lisanke (MCNC P.O. Box 12889 Research Triangle Park NC 27709). Logic Synthesis and Optimization Benchmarks: User's Guide, 1989.

[2] Rajeev Alur and David Dill. Automata for Modelling Real-Time Systems. In *ICALP*, 1990.

[3] R. P. Kurshan. Analysis of Discrete Event Coordination. In *LNCS*, pages 414–453, 1990.

[4] Gregory S. Whitcomb and A. Richard Newton. High-level Design Representation and Synthesis in BLIS. In *SRC TECHCON'90*, October 1990.