INDIANA UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 398

Specification and Synthesis of Bounded Indirection

Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson

FEBRUARY 1994

Specification and Synthesis of Bounded Indirection [*]

Abstract

In this paper, we introduce *bounded indirection*, a restricted form of pointers, for system specification. Indirection provides a mechanism for compact descriptions of many complex control structures, such as interrupts, continuations, and dynamic connections between machines. We describe three kinds of indirection - state, value and net indirection for use in different aspects of system description. Transformations on indirection representations and methods for synthesizing bounded indirection within the framework of *behavior tables* are presented. The use of indirection in system specification and synthesis is illustrated using three large examples.

1 Introduction

Hardware description languages are widely used as a front-end for synthesis tools. HDLs have evolved from programming languages with many of their features and are extended with additional constructs for hardware specification. Some HDLs are primarily simulation languages with synthesizable subsets, whereas others are primarily used for describing synthesizable hardware. In this paper, we introduce the notion of *bounded indirection*, a restricted form of pointers as a synthesizable hardware description construct that can improve the expressiveness of HDLs. Indirection can model complex control structures as simple datapath descriptions, enabling a designer to write compact system descriptions. It can also be used to model selection mechanisms for connections between sequential components in a system.

[*]Research reported herein was supported, in part, by NSF: The National Science Foundation, under grants numbered MIP 89-21842 and MIP 92-08745.

Many hardware description languages have two dialects, one for behavioral descriptions, and another for structural descriptions. The behavioral dialects are used to specify the control flow of a system and the structural dialects are used to specify data flow. Most synthesis tools based on such languages use separate data structures - a control-flow graph and a data-flow graph as their internal representations to perform optimizations. These different internal representations restrict the ability of a designer to explore the design space, by eliminating the links between the behavioral and structural facets of the design.

Indirection blurs the distinction between behavior and structure by modeling behavior as structural descriptions. To model indirection effectively, we need a framework to represent both behavioral and structural facets of a design. Behavior tables [1] provide us with such a framework. In addition to behavior and structure, behavior tables can also model protocol and data abstraction aspects of a design in a unified framework.

1.1 Related Work

In recent years there has been substantial work in improving hardware description languages. Starting with register-transfer level descriptions, today a designer can choose from a variety of languages. As system synthesis continues to evolve, new features are being added to HDLs to model systems at higher levels of abstraction. VHDL [2] is emerging as a standard among HDLs, with a large number of language constructs, many of which are not synthesizable. With the addition of new features in the recent enhancement of VHDL [3], the simulation environment will be greatly improved. Unfortunately, the synthesizable subset of the language is still lagging far behind. *Access type* and *selected name* constructs in VHDL are similar to the bounded indirection construct discussed in this paper. But, these and other pointer constructs in VHDL can not be synthesized by most tools [4, 5]. The Verilog® [6] hardware description language lacks many of the constructs in VHDL, but has been quite popular due to its simplicity and availability of synthesis tools. Some aspects of behavior tables are similar to Cheng and Krishnakumar's extended finite state machine model [7], and Drusinsky and Harel's statecharts [8], but these finite state machine models do not support indirection.

Bounded indirection is a synthesizable structural construct, that can be added to any HDL to enhance the expressiveness of the synthesizable language. We describe a method for synthesizing bounded indirection. Transformations on bounded indirection constructs are also presented. The research reported here grew out of our design derivation methodology, which is based on first order functional algebra [9]. We present three substantial examples to illustrate the different uses of bounded indirection.

2 Behavior Tables

Behavior tables [1] are an extension of register transfer tables that can model control, datapath, protocol, and data abstraction facets of a system. A behavior table is a representation of a finite state machine that models system behavior. Each row in a behavior table represents a transition in the machine described by the table. The columns are divided into two sections, the decision section and the action section. The decision section represents the state and conditions that must hold for the transition to be executed. The action section represents the data flow through the functional units, ports, and the next state.

A machine is defined as $M = \langle T, s, n, P, R, C, S, \hat{I}, V \rangle$, where T is a non-empty set of transitions, s is the present state, n is the next state, P is the set of ports, R is a set of internal registers and combinational signals, C is the set of internal predicates, S is a set of states ($\#$ denotes any state in the machine), $\hat{I}$ is a set of reference sets that includes functions over register names and data input names, and V is the domain of values including the don't care value $\#$. The set of ports ($P = CI \cup CO \cup DI \cup DO$) is a union of the sets of control inputs, control outputs, data inputs, and data outputs. We adopt a convention that references to a register/port/signal r is written as $\hat{\mathrm{r}}$,and the prefixes @, \$ denote input and output dereference respectively.

A transition is defined as a function of the form $t = \ \mathit{if}\ [\mathit{satisfies}(t_d, \vec{d})]\ \mathit{then}\ t_a$, where the assignment function $t_d : C \cup CI \mapsto V$ and $t_d : \{s\} \mapsto S$. An assignment function t_d is satisfied with respect to the current conditions $\vec{d}$ if $(\forall c_i \in \{s\} \cup C \cup CI \, . \, t_d(c_i) = \vec{d_i} \ \vee \ t_d(c_i) = \#)$ is *true*. The action section of each transition is defined by the assignment function, $t_a : R \cup DO \cup CO \mapsto V \cup S \cup \hat{I}$ and $t_a : \{n\} \mapsto (V \cup S) - \{\#\}$. Each transition in the machine denotes a row in the behavior table. This model can be used to denote transitions from a set of states to a particular state or from a state to one of several states based on register contents. Transitions are indicated as $s_1 \xrightarrow{t} s_2$, where $t(s) = s_1$ and $t(n) = s_2$.

Our model enables us to define transitions to states based on register or port values. Interrupts and continuations can be modeled as transitions in behavior tables. Indirection allows us to control data flow based on register contents without "controller" intervention.

3 State Indirection

Behavior tables allow for state identifiers to be stored in registers. These can then be loaded as the *next state*, and used as a branching address. References to a register name in the next state column in a behavior table, instead of a specific state identifier, is called state indirection. The register in the next state column must be loaded with a state identifier before the transition involving state indirection is executed. State indirection is a useful mechanism to model interrupts, continuations, and procedure calls.

Decision Section			Action Section		
State	Interrupt	...	Next state	Future state	...
#	T	...	int-1	State	...
⋮	F	⋮	⋮	⋮	⋮
int-n	F	...	Future state	#	...

Table 1: Interrupt modeling

Interrupts can be modeled by saving the next state identifier in a register if there is an interrupt in any state. The register can then be used to return to the appropriate state after servicing the interrupt. Table 1 shows a typical behavior table with interrupts. The first row of the behavior table represents a jump to the "interrupt service routine" with the initial state int-1. The last row of the behavior table is a return from interrupt where the interrupted state is loaded back from the future state. Figure 1 shows the state diagram

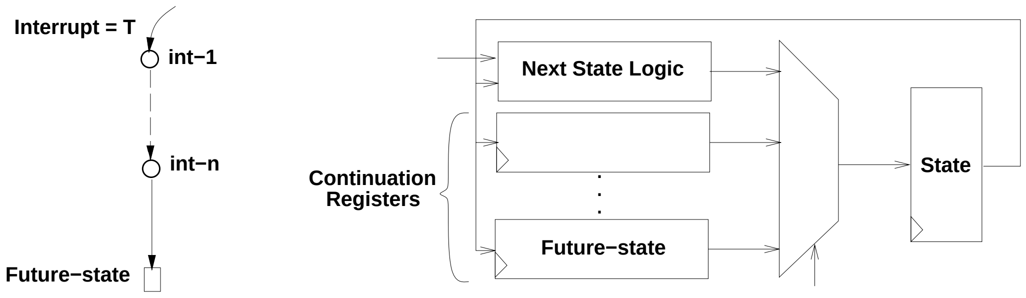

Figure 1: Interrupt service state diagram and schematic

for the interrupt service segment and the schematic from the behavior table description in Table 1. The box labeled Future-state denotes that the target state of the transition is based on the value of that register.

We can see that two rows in the behavior table, corresponding to two transitions in our machine model, can model the interrupt mechanism. Modeling the same mechanism in a classical finite state machine model or in a language without support for indirection would require $2n$ transitions for n states in the system.

3.1 Transformations

State indirection involves determining the next state of a machine based on the value of some register/signal. A bounded state indirection transition can be transformed into a set of transitions, each with a decision for a possible reference to a state identifier. Figure 2 shows the transformation on bounded state indirection.

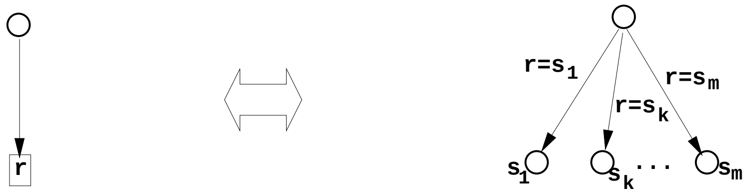

Figure 2: State indirection transformations

Definition 3.1: A transition with state indirection is equivalent to a set of transitions without state indirection if:

$$t_1 \overset{I_S}{\equiv} T_2 \overset{\text{def}}{=} (t_1(n) \notin S \ \wedge \ @(t_1(n)) \in I_S \ \Rightarrow \ \forall q \in I_S \,.\, \exists t_2 \in T_2 \,.\, t_2(n) = q) \\ \wedge \ (\forall t_2 \in T_2 \,.\, c \neq n \ \Rightarrow \ t_1(c) \equiv t_2(c))$$

The equivalent transition set must have a transition corrsponding to every possible next state. All other register/signal/decision values must be equivalent for the transitions.

3.2 Example: Interruptable DMA Controller

In this section, we use an interruptable direct memory access (DMA) controller as an example to illustrate the use of state indirection. The DMA controller is a process which coordinates access to the memory and performs block copying per request. The controller waits in an idle loop for a copy command. Upon a copy command, the machine enters a read-write loop, which copies a block of memory onto the destination address. The machine can be interrupted in the idle state or during a copy operation. On an interrupt the machine saves

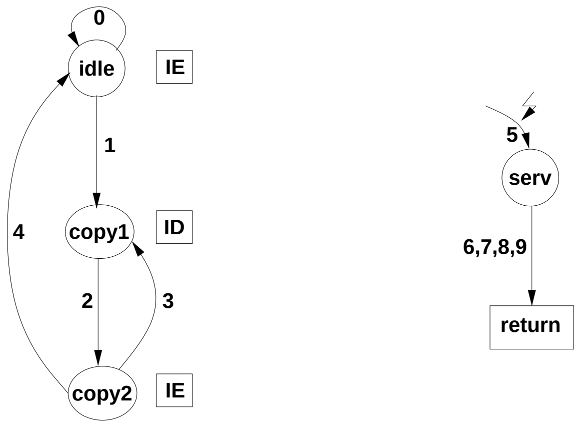

Figure 3: Interruptable DMA controller state diagram

the state in the return register and enters the interrupt service routine. The serv state performs the requested read or write and returns to the interrupted state. To prevent race conditions due to interruption, interrupts are disabled after a read in the read-write loop. Also if the request is within the region which is still to be copied onto, the operation is performed on the corresponding address within the region to be copied.

Figure 3 shows the state diagram for the interruptable DMA controller. Using behavior tables we can specify the controller in 4 states and 10 transitions, instead of 5 states and 15 transitions in a classical finite state machine model. The decision and action sections of the behavior table for this example are shown in Table 2. Compact interrupt representation allows significant reduction in the complexity of the specification. The states labeled $\boxed{\text{IE}}$ have interrupts enabled, and the states labeled $\boxed{\text{ID}}$ have interrupts disabled.

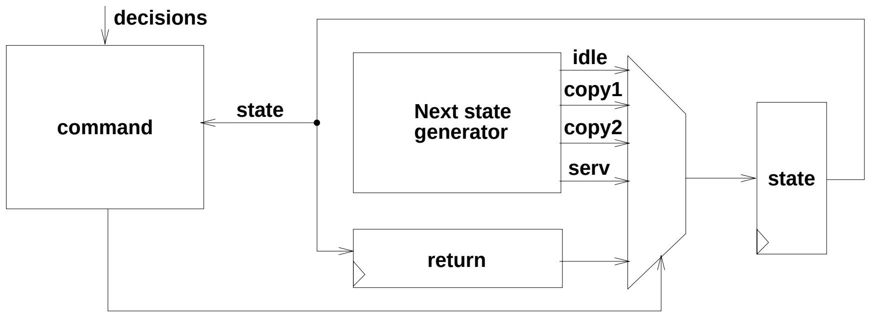

Figure 4: Interruptable DMA controller schematic

The schematic of the synthesized state machine for the DMA controller is shown in

Decision section

row	state	int-d	cpcmd?	rd?	(gt? n 0)	inrange
0	idle	false	false	#	#	#
1	idle	false	true	#	#	#
2	copy1	false	#	#	#	#
3	copy2	false	#	#	true	#
4	copy2	false	#	#	false	#
5	#	true	#	#	#	#
6	serv	false	#	true	#	true
7	serv	false	#	false	#	true
8	serv	false	#	true	#	false
9	serv	false	#	false	#	false

Action section

row	next	return	m	do	tmp	af	at	diff	n	inrange	int-d
0	idle	#	m	do	#	#	#	#	#	#	int
1	copy1	#	m	do	#	from	to	(- to from)	words	#	int
2	copy2	#	m	do	(rd m af)	(inc af)	at	diff	(dec n)	#	false
3	copy1	#	(wr m at tmp)	do	#	af	(inc at)	diff	n	#	int
4	idle	#	(wr m at tmp)	do	#	#	#	#	#	#	int
5	serv	state	m	#	#	af	at	diff	n	(in? a at n)	false
6	return	#	m	(rd m (- a diff))	#	af	at	diff	n	#	false
7	return	#	(wr m (- a diff) di)	#	#	af	at	diff	n	#	false
8	return	#	m	(rd m a)	#	af	at	diff	n	#	false
9	return	#	(wr m a di)	#	#	af	at	diff	n	#	false

Table 2: Behavior table for interruptable DMA controller

Figure 4. The state machine is synthesized out of the next and return columns in the action section of the behavior table (Table 2). Synthesizing the next column involves using a selector to select one of the different state identifiers and the return register based on the output from the command unit. The return register gets its input from only from the state register, so the selector for it is trivial and reduces to a wire. A binary representation of the state identifiers and truth values in the decision section of the behavior table (Table 2) are used to create a boolean decision table, which can be synthesized into the command unit by logic synthesis. The rest of the behavior table can be synthesized using sequential synthesis techniques [10].

4 Value Indirection

A simple construct in behavior tables can be used to specify that the value to be assigned to some register/signal is selected from a bounded set of signals based on the indirection value in a register/signal. The use of a value from a register/signal as a pointer to a specific signal from a bounded set of signals is called value indirection.

Table 3 shows the use of Ptr as an indirection pointer. A reference to R1 or R2 is stored

State	...	Next State	...	Ptr	R9
⋮	⋮	⋮	⋮	⋮	⋮
S1	...	S5	...	$\widehat{R1}$	...
S2	...	S5	...	$\widehat{R2}$	...
S5	...	S6	...	...	$@\mathsf{Ptr}_{I_1}$

Table 3: Value Indirection

in Ptr and used in the next transition to load R1 or R2 into R9. Figure 5 shows a fragment of the state diagram and the corresponding schematic for the behavior table shown in Table 3. One of the values from the reference set I_1 is selected by Ptr to be loaded into R9 through the control selector.

Modeling this mechanism in the classical finite state machine model or in a language without indirection would require n transitions or n-way case statement for n possible references in the reference set I_1, instead of a single transition in the behavior table. Further levels of indirection can be added to the system if needed for a more compact representation.

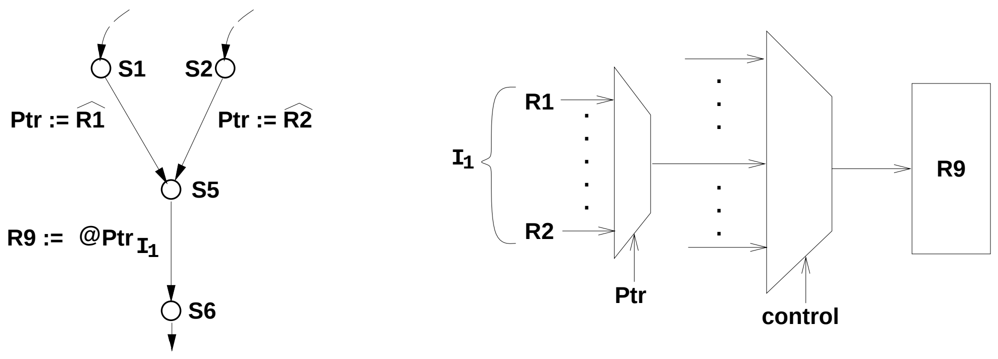

Figure 5: Value indirection: state diagram and schematic

4.1 Transformations

Value indirection involves determining the value of a register/signal in a machine based on the value of some register/signal. A bounded value indirection can be transformed into a set of transitions, each with a decision for a possible reference value. Figure 6 shows the transformation on bounded value indirection.

Definition 4.2: A transition with an indirect value reference and a set of transitions without the indirect reference are equivalent if:

$$t_1 \overset{I}{\equiv} T_2 \overset{\text{def}}{=} t_{a_1}(c) \in @I \Rightarrow \forall i \in \hat{I}\,.\,\exists t_2 \in T_2\,.\,t_{d_2}(t_{a_1}) = i$$
$$\wedge\ t_1(c) \notin @I \Rightarrow \forall t_2 \in T_2\,.\,t_1(c) \equiv t_2(c)$$

The equivalent transition set must have a transition with every possible dereference value as a decision, and all other values of the corresponding transitions must be equivalent.

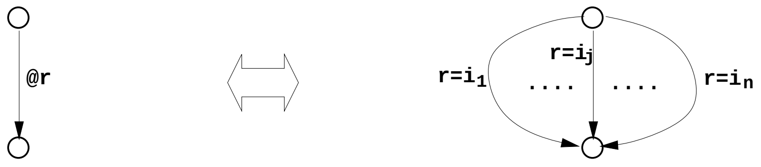

Figure 6: Value indirection transformations

4.2 Example: FM9001 microprocessor

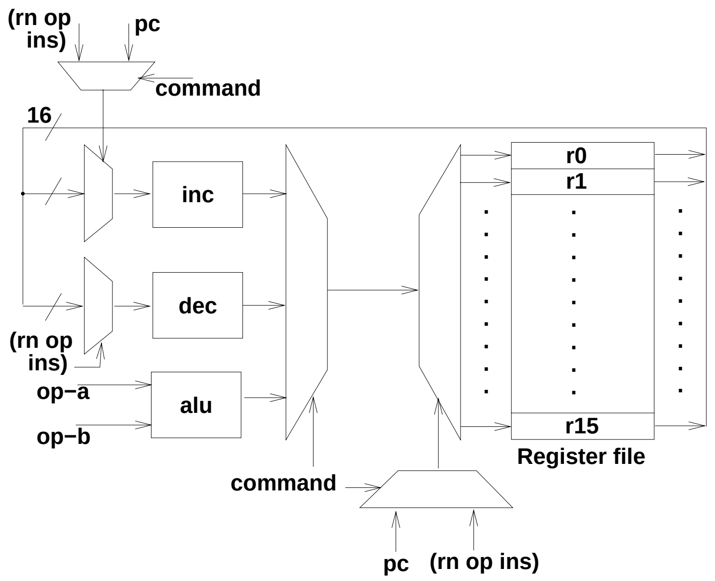

Figure 7: FM9001 register file datapath schematic

Let us consider a behavior table description of Hunt's FM9001 microprocessor [11] with instances of value indirection. The FM9001 is a 32-bit microprocessor and mechanically verified at the gate level using the Nqthm theorem prover. It has an internal register file with 16 general purpose registers any one of which can be used as the pc. Modeling all the different operations that can be performed on the different pairs of registers would require 16×3 transitions in a classical finite state machine model, but can be modeled by 3 transitions using indirection.

Decision section

row	state	(nlist-p oracle)	(a-imm-p ins)	(reg-dir-p (mode op ins))	(pre-dec-p (mode op ins))	(post-inc-p (mode op ins))	(store-resultp (store-cc ins) flags)
0	fm-intr	1	?	?	?	?	?
1	fm-intr	0	?	?	?	?	?
2	fm-fetch	?	?	?	?	?	?
3	fm-op	?	1	0	0	0	?
4	fm-op	?	0	1	0	0	?
5	fm-op	?	0	0	1	0	?
6	fm-op	?	0	0	0	1	?
7	fm-op	?	0	0	0	0	?
8	fm-alu-op	?	?	?	?	?	1
9	fm-alu-op	?	?	?	?	?	1
10	fm-alu-op	?	?	?	?	?	0

Action section

row	next-state	future	op	regs	mem	ins	op-a	op-b	b-addr
0	fm-intr	?	?	regs	mem	?	?	?	?
1	fm-fetch	?	?	regs	mem	?	?	?	?
2	fm-op	fm-op	a	($pc(inc @pc))	mem	(rd @pc mem)	?	?	?
3	fm-op	fm-alu-op	b	regs	mem	ins	?	(extend (a-imm ins) 32)	?
4	future	fm-alu-op	b	regs	mem	ins	op-b	@(rn op ins)	b-addr
5	future	fm-alu-op	b	($(rn op ins) (dec @(rn op ins)))	mem	ins	op-b	(rd (dec @(rn op ins)) mem)	(dec @(rn op ins))
6	future	fm-alu-op	b	($(rn op ins) (inc @(rn op ins)))	mem	ins	op-b	(rd (inc @(rn op ins)) mem)	@(rn op ins)
7	future	fm-alu-op	b	regs	mem	ins	op-b	(rd @(rn op ins) mem)	@(rn op ins))
8	fm-intr	?	?	($(rn op ins) (alu flags op-a op-b ins))	mem	ins	op-a	op-b	b-addr
9	fm-intr	?	?	regs	(wr b-addr mem (do-op flags op-a op-b ins))	ins	op-a	op-b	b-addr
10	fm-intr	?	?	regs	mem	ins	op-a	op-b	b-addr

Table 4: Behavior table for FM9001 microprocessor

Figure 7 shows the schematic for the register file datapath. This is synthesized from the regs column of the behavior table (Table 4). The register to be updated is selected by one of {$pc, $(rn op ins)} output indirections. inc, dec, alu form the functional units in the datapath. The input to inc is selected from the register file using the input indirection @(rn op ins), and the input to dec is selected using the input indirections @pc or @(rn op ins). Details about the derivation and synthesis of an implementation of the FM9001 can be found in [1, 12].

5 Net Indirection

Using the basic mechanisms developed in the previous sections, we will now define a powerful construct that can be used to selectively connect ports between behavior tables. Net indirection provides a mechanism for the connection of an input(output) port to one of several output(input) ports depending on the indirection value on another port. This enables us to specify dynamic connections between components in a system, such as buses and switches.

Consider the net connections where an input port a is connected to one of the output ports in the reference list I_1 depending on the indirect value in p_1. and the data output port b is connected to one of the input ports in the reference list I_2 depending on the value of p_2. The schematic corresponding to these net indirection primitives is shown in Figure 8. All the ports in the reference set associated with an indirect input(output) ports must be output(input) ports.

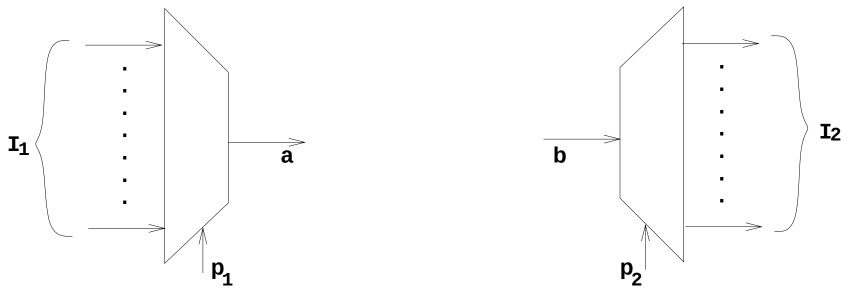

Figure 8: Net indirection primitives

5.1 Example: Bi-directional Bus

We now consider a general purpose bi-directional bus that provides port-to-port as well as broadcast channels for communication between ports in the system. Describing such a structure in a functional net-list specification would require creating and connecting modules for all the selectors. Using the net indirection primitives, we can specify a net-list at a higher level of abstraction, with much of the control mechanism built into the network, and it can be generated automatically.

Bus_out	B_1	B_2	...	M_{1_I}	M_{2_I}	...	IS_1	IS_2	...
@send_sel	@M_1P_{out}	@M_2P_{out}	...	(\$ IS_1 Bus_in)	(\$ IS_2 Bus_in)	...	@(1^{st} rec_sel)	@(2^{nd} rec_sel)	...

where:								
send_sel	$\in$	$\{\widehat{B_1}, \widehat{B_2}, \ldots\}$	rec_sel	$=$	$\langle r_1, r_2, \ldots \rangle$	r_j	$\in$	$\{\widehat{Z}, \widehat{M_jP_{in}}\}$
M_jP_{in}	$\in$	$\{\widehat{m_ji_1}, \widehat{m_ji_2}, \ldots\}$	M_kP_{out}	$\in$	$\{\widehat{m_ko_1}, \widehat{m_ko_2}, \ldots\}$	IS_j	$\in$	$\widehat{M_{j_I}} \cup \{Z\}$

Table 5: Bus specification

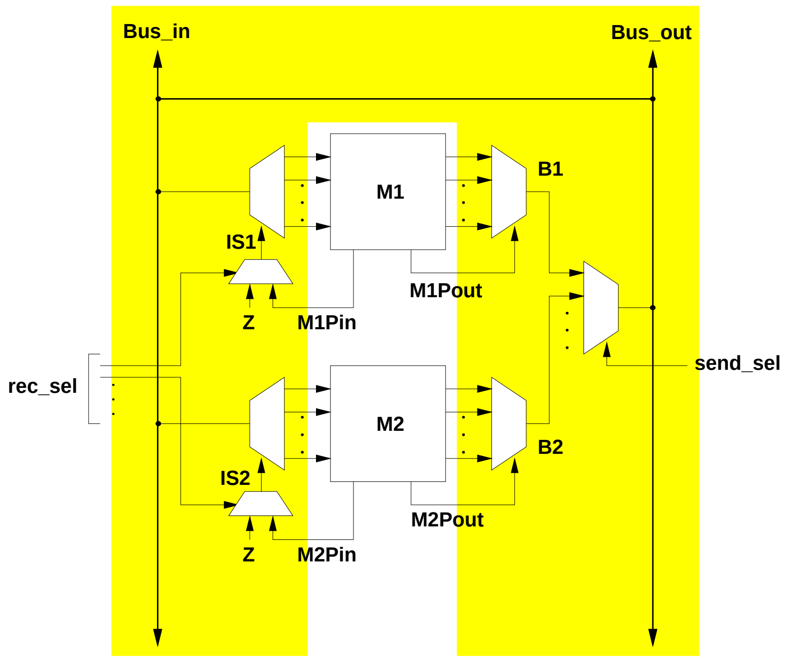

Figure 9: Bi-directional Bus schematic

Table 5 shows the net specification for the bus. The range of each of the indirection signals are also shown. **send_sel** ranges over references to bus output signals from each machine and selects one of them. Similarly, **rec_sel** is a vector of bits, one for each component on the bus, which ranges over references to one of the input ports of the machine or Z (high impedance). Both **send_sel** and **rec_sel** are input ports of the bus circuitry. Each machine connected to the bus has a set of input and output ports. The output ports M_iP_{out} and M_iP_{in} specify which output port and input port of machine M_i are active.

Synthesizing the net description shown in Table 5 results in the shaded area of the schematic shown in Figure 9. Each of the reference sets defined in the net specification corresponds to a selector in the schematic. M_1P_{out} selects one of the output ports of M_1 to be connected to B_1. IS_1 selects none or one of input ports of M_1 depending on the value of

r_1, which in turn selects a reference to the input ports or Z.

6 Conclusions and Future Work

In this paper, we presented indirection, a novel hardware specification construct. By restricting the range of each indirect signal to a bounded set of references, we were able to make it a synthesizable construct. Bounded indirection is similar to pointers in programming languages, but with the important difference that bounded indirect signals are defined over small pre-defined types of signals names, where as pointers are defined over a broad range of values in a type class. The synthesizability of this construct and a clear description of its basis in a finite state machine model make it particularly attractive for inclusion into hardware description languages. The three substantial examples presented here provide ample evidence about the usefulness of bounded indirection for "real" design problems.

We are currently exploring avenues for optimizations on the systems synthesized from behavior table specifications with bounded indirection. One such optimization is to allocate the minimum number of selectors for each indirect construct based on the same reference set by allocating the same selector for indirections in different rows of the behavior table; using a graph coloring algorithm.

The research reported here is work-in-progress. A design environment based on behavior tables which includes bounded indirection is being implemented using Motif widgets in C and Scheme.

References

[1] K. Rath, M. E. Tuna, and S. D. Johnson, "Behavior tables: A basis for system representation and transformational system synthesis," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, IEEE, Nov. 1993. to appear.

[2] S. 1076, *IEEE Standard VHDL Language Reference Manual.* IEEE, 1987.

[3] Berge, Fonfoua, Maginot, and Rouillard, *VHDL 92.* Kluwer, 1992. in press.

[4] R. Vemuri, P. Mamtora, P. Sinha, N. Kumar, J. Roy, and R. Vutukuru, "Experiences in functional validation of a high level synthesis system," in *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pp. 194–201, June 1993.

[5] W. Ecker and S. März, "System-level specification and design using VHDL: A case study," in *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1993.

[6] D. E. Thomas and P. Moorby, *The Verilog® Hardware Description Language.* Kluwer, 1991.

[7] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pp. 86–91, June 1993.

[8] D. Drusinsky and D. Harel, "Using statecharts for hardware description and synthesis," *Transactions on CAD*, vol. 8, pp. 798–807, July 1989.

[9] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations.* Cambridge: MIT Press, 1984. ACM Distinguished Dissertation 1984.

[10] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proceedings of International Conference on Computer Design*, pp. 328–333, IEEE, Oct. 1992.

[11] W. A. Hunt, "A formal HDL and its use in the FM9001 verification," in *Mechanized Reasoning in Hardware Design* (C. Hoare and M. Gordon, eds.), Prentice-Hall, 1992.

[12] B. Bose and S. D. Johnson, "DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation," in *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods*, Springer, 1993.