A MODEL-BASED EXPERT SYSTEM FOR DIGITAL SYSTEM DESIGN

JUNG-GEN WU
National Taiwan University

YU HEN HU
University of Wisconsin, Madison

WILLIAM P.-C. HO

DAVID Y.Y. YUN
University of Hawaii, Manoa

The authors describe an expert system that generates digital system design from high-level specifications. The system is based on a three-phase model of digital system design. First, the high-level behavioral specifications are translated into a sequence of primitive behavioral operations. Next, these primitive operations are grouped to form intermediate-level behavioral functions. Finally, structural function modules are selected to implement these functions.

The advantages of this model-based reasoning technique is that more design solutions are possible and the design is easier to optimize.

Specifying a digital system often entails the use of both functional and behavioral descriptions. Functional blocks and their interconnections specify the structural components of the target system. As such, we can realize them using physical functional modules. Detailed input/output relationships, on the other hand, describe the behavioral aspects of the target system. Examples of these include the specifications of bus-interface and communication protocols, which contain no functional descriptions. Although a set of purely behavioral specifications allows design with complete technology independence, it also imposes more problems with its flexibility. These additional design complexities are a challenge to digital system designers. How do they find an efficient digital system design among all the potential candidates for a set of behavioral specifications?

The answer to that question lies in the definition of the design process. The primary task of digital system design is to transform high-level behavioral specifications into physical digital circuits. Naturally, then, as system complexity increases, design errors are more likely to occur and it becomes much more difficult to detect them. The solution we propose to this challenge of system complexity is an automated, intelligent, CAD tool that will help human designers to make crucial design decisions, prevent fatal errors, and maintain huge design-knowledge databases.

KNOWLEDGE-BASED DESIGN SYSTEMS

Several knowledge-based digital circuit design systems already exist. LSS,¹ Socrates,² and Dagon³ are automated design systems that map technology-independent combinational logic functions into a technology-dependent physical design. DAA⁴ mimics the human designers' usual approach in designing a digital system by encoding design knowledge into production rules. A preprocessor of DAA, called Bud,⁵ uses a bottom-up framework to perform a global analysis of the target system. Both the Vexed design system⁶ and a design paradigm proposed by Brewer and Gajski⁷ propagate constraints during the refinement of iterative specifications. Consequently, design constraints can be propagated both forward and backward during the reasoning process. This characteristic ensures that the system considers all interactions among subproblems.

The goal of Adam⁸ is to merge several design automation programs into a unified framework. The system uses planning techniques to build an abstract design plan that consists of possible sequences of design activities, which addresses the problem of how to use design tools most effectively.⁹ Synapse¹⁰ uses a set of algebraic primitive operations to

model the behavior of various levels of abstraction. The specification is expressed in algebraic form so that the system can use axioms and rewriting rules to simplify the input expression.

A MODEL-BASED EXPERT SYSTEM

Our system is called MBESDSD, short for model-based expert system for automated digital systems design. In a model-based expert system, the reasoning process is based on a model of the objects in the application and the specific operations that act on those objects.¹¹ In MBESDSD, the objects in the behavioral specification are various signals (electrical waveforms) acted on by five primitive operations: And, Or, Not, Store, and Transmit. These operations are sufficient to describe all possible behaviors of any digital circuit.

The five primitive operations are abstractions of the physical electrical waveforms. A *function* consists of several primitive operations. Instead of translating primitive operations directly into *functional modules*, we first group them into functions. Hence a function is also an intermediate representation between primitive operations and functional modules.

The purpose of the function is to act on signals to perform a more complicated task for a certain length of time. An example of a function is the operation to convert from parallel signals to serial signals. A function can also be an intermediate-level representation to bring purely behavioral descriptions (in terms of primitive operations) one step closer to their physical realizations. We find the function module in the final stage of synthesis. The function module is a physical entity that operates on its input signals and produces desired output signals to realize certain functions. If a network of interconnected function modules can realize the set of desired behaviors, then synthesis is complete.

With this model, the design process in MBESDSD proceeds through three distinct phases. In the first phase, called *behavioral compilation*, the system translates behavioral specifications into a sequence of primitive operations. It then analyzes the relations among these primitive operations to uncover their interactions and possible redundancies.

In the second phase, *function abstraction*, the system tries to find the optimal set of intermediate functions that will cover the set of primitive operations compiled from the behavioral specifications. MBESDSD uses a heuristic best-first search algorithm called A* to do this search.¹² It also uses planning and meta-planning techniques to help reduce the complexity of the search process.¹³

In the third phase, *structure implementation*, the system selects physical function modules to implement the functions. Here, the design task is to optimize certain design criteria. We have developed a function-module library in MBESDSD to maintain technology-dependent design knowledge. As technology evolves, MBESDSD can be easily updated with a more current module library. To match functions to physical function modules, the system again uses a heuristic search procedure.

COMPARING SYSTEMS

MBESDSD has several distinct features over existing knowledge-based systems for automatic digital system synthesis:

1. Most systems represent design knowledge by ad hoc rules. These rules often focus on local features and may thus be limited to local optimizations. MBESDSD, on the other hand, uses a formal heuristic search method to perform optimization and has a greater potential to achieve global optimization.

In a model-based expert system, the reasoning process is based on a model of the objects in the application and the specific operations that act on those objects.

A circuit's behavioral specification may be composed of signal declarations, signal-purpose declarations, and behavioral events.

2. A rule-based system may have implicit knowledge redundantly encoded in several rules and a huge set of rules may be needed to accomplish a "good" design. MBESDSD explicitly expresses fundamental theorems of digital circuits and the behavior of logic functions and modules. As a result, redundancy and the inconsistency are greatly reduced.
3. Most existing systems use a top-down design strategy. In this type of design, functional blocks in the target system's description can restrict the structure of the final design. In MBESDSD, we use a bottom-up design strategy, abstracting essential functions from a set of purely behavioral specifications. Consequently, there is no structural information to restrict design options. A top-down design strategy is, however, more efficient in the design of large systems, while MBESDSD is suitable for medium circuit design.

KNOWLEDGE REPRESENTATION

A circuit's behavioral specification may be composed of three parts: signal declarations, signal-purpose declarations, and behavioral events. Signal declarations define the names, numbers of bits, and directions of all the input and output signals of the target circuit. Signal-purpose declarations characterize certain special properties of a signal. For example, the signal purpose "clock" has properties labeled "frequency" and "duty cycle." Behavioral events include the change of output states caused by the input. Figure 1 is an example, which shows the behavioral specification of an input sequence detector.

PRIMITIVE OPERATIONS

In our model, we assume that a digital circuit consists of three circuit types: combinational, sequential, and bus-interface. Consequently, we should be able to describe the behavior of any combination of these three types using five primitive operations mentioned earlier. The three primitive operations And, Or, and Not can describe any Boolean function and hence any combinational digital circuit. The fourth primitive operation, Store, characterizes the memory element of sequential circuits. Finally, the fifth primitive operation, Transmit, represents the transient behavior of a bus or interface signal whose values are clearly defined only for certain time intervals.

In MBESDSD, the description of an operation has the following regular expression format:

operation(input1, input2, ..., control, output, time)

Subsequent parameters of an operation are separated by commas. If a parameter is missing between its two corresponding commas, its value is either unknown or does not exist.

To simplify the design, we have made a few assumptions about the five primitive operations. First, each primitive operation will be a bit-wise and time-instant logic function, such that an operation can have at most one control signal, one output signal, and one timing parameter. Combinational operations will have no control signal. Store is triggered by the rising edge of the control signal, at which point input data is stored and sent to the output. Transmit is enabled when the control signal is 1 (high), at which point input data is transmitted to the output. And and Or may have more than one input, but the other operations can accept only one input.

Signal Input x, ck; /* x, ck, and z are */
 Output z: /* 1-bit signals */
 Function system clock ck; /* ck controls system timing */
 Event 0 If x = 1;
 Event 1 If (one clock after event 0) And (x = 0);
 Event 2 If (one clock after event 1) And (x = 1);

(b)

Figure 1. Timing diagram (a) and description (b) of an input sequence detector. If the signal *x* is changed in four consecutive clock cycles, the output signal turns to 1.

The timing parameter may have two formats: either a constant or a constant followed by a modifier r , which represents repeated operations. The operation $store(a, .b, l)$, for example, states that the information in signal a is stored to signal b at time l . On the other hand, $store(c, .d, lr)$ indicates that the operation is to be performed repeatedly for each clock cycle.

We also include two declaration statements, *value* and *input*, to link the value of signals at different time instants. A declaration statement has the following format:

declaration(signal, value, time)

A value declaration specifies that the particular signal will carry useful data after the specified time instant. An input declaration specifies the conditions of an input signal. The system does not have to generate the value of an input signal.

FUNCTIONS

As we stated earlier, a function is the collection of a set of primitive operations ordered in a specific sequence to perform some complicated, yet routine, tasks within a certain time. A function's characteristics are its input, output, and control signals as well as the timing parameters. We can control when the function is executed either by its control signals or by presetting execution at some specific time. In the prototype of MBESDSD, only a small subset of functions such as load, change states, count, parallel-to-serial conversion, and serial-to-parallel conversion are implemented.

A function is described by two sets of primitive operations: *covered* and *support*. Covered operations are primitive operations that characterize the behavior of the function. Support operations facilitate the proper execution of the function. Primitive operations that generate control signals and timing information are often used as support operations. Figure 2 is the behavioral description of the parallel-to-serial-conversion function. In this description, the flow of values is not specified explicitly. In fact, as long as correct values are delivered to the output at pre-specified times, their locations may be changed.

FUNCTION MODULES

As we stated earlier, function modules are physical entities that implement behavioral functions. The mappings between function modules and behavioral functions usually are not one to one. For example,

Figure 2. Representation of the Parallel-to-Serial Conversion function. The values of input signals i_1 through i_n at time t_1 are output sequentially from signal o starting at time t_2 . The support operations are the two control signals ld (load) and st (start) that synchronize the corresponding events.

Function modules are physical entities that implement behavioral functions. The mappings between these modules and behavioral functions usually are not one to one.

Technology-independent design knowledge consists of the behavioral descriptions of both functions and function modules, as well as the implementation methods of functions.

Latch can implement only part of the function Change States. RS Latch, on the other hand, can implement three functions: Set, Reset, and Store. Examples of function modules include: Decoder, Adder, Multiplexer, Register, Counter, and Shift Register. More complicated function modules include Microprocessors and CRT Controllers. A function module has input signals, control signals, memories, and output signals. Since a function module is a physical identity, we can use primitive operations to characterize all its input, output, and internal (memories) behavior. Hence, a function module has no undefined intermediate operations. Figure 3 depicts the representation of a function module called n -Bit Shift Register.

ORGANIZING DESIGN KNOWLEDGE

Figure 4 shows, technology-dependent design knowledge and technology-independent design knowledge are organized in different knowledge bases, or libraries. Technology-dependent design knowledge relates to physical packaging information such as the packaging types of function modules, the number of modules in each type, and the pin assignment of each type of package. Technology-independent design knowledge consists of the behavioral descriptions (using primitive operations) of both functions and function modules, as well as the implementation methods of functions.

The function library stores the behavior descriptions of all the functions. The library for function implementation methods stores design heuristics, which are similar to design rules for selecting function

Figure 3. Representation of the n -Bit Shift Register function module in terms of signals (a) and behavior (b). While ld is rising, the information in signal i_1 through i_n is stored to memory m_1 through m_n . While ck is rising, the information in i_0 is stored to m_1 , the information stored in each memory location except m_n is stored to another memory location. The internal signals m_1 through m_n are always connected to output signals.

modules in other existing expert design systems. A module library stores the behavior of each function module (technology-independent) and the available physical structure of that module (technology-dependent). Users need only load the appropriate module library before structure implementation. MBESDSD will synthesize an appropriate design using the specified physical structures.

Both the fundamental theorems of digital circuits design, such as Demorgan's theorem, and the output and timing properties of Store are technology-independent design knowledge and are encoded in the program as procedures.

ADVANTAGES OF PRIMITIVE OPERATIONS

By using these five primitive operations, we can get a fine granularity of knowledge representation. Hence, there is ample opportunity to further refine the set of given specifications using reasoning. Reasoning based on primitive operations has two main advantages. The first is that we can identify essential functions by merging redundant operations. For example, consider two signals *a* and *b*, where *a* changes state every clock cycle and *b* changes state every two clock cycles. That is,

store(a-, , a, 1r), and store(b-, , b, 2r)

Instead of implementing the behavior of signals *a* and *b* separately using perhaps two independent *counters*, MBESDSD has an essential function, called 2-Bit Count. This function realizes both primitive operations, which offers more opportunity to minimize the design.

The second advantage to basing reasoning on primitive operations is that rule-based systems with encapsulated design knowledge have limited flexibility for optimizing the design. Consider two examples of ad hoc rules:

1. If a function n-bit parallel-to-serial conversion is needed
Then use a module n-bit shift register.
2. If a function load n-bit data is needed
Then use a module n-bit shift register.

Neither of these rules explains why we cannot use the module Shift Register to implement the functions Parallel-to-Serial Conversion and Load Data. The rules contain some implicit knowledge about the functions and the module, but it is the behavior of storing input data and shifting data that makes the module implement the functions. The

Reasoning based on primitive operations allows us to refine the set of given specifications with a great deal of flexibility to optimize the design.

Figure 4. Knowledge organization in MBESDSD.

The design process in MBESDSD consists of behavioral compilation, function abstraction, and implementation of the structure.

knowledge of storing input data is implicitly and redundantly included in these two rules. The rule base needs to include all possible applications of Shift Register as rules to make the design knowledge base more complete.

In MBESDSD, we explicitly express the basic behavior of logic entities. In Figure 3, *store(l, ld, m_i)* operations perform the function Load Data. By applying consecutive *ck* control signals to *store(m_i, ck, m_i+1,)* operations, the *n*-bit data originally in *m₁ - m_n* is stored to *m_n* sequentially. The *transmit(m_n, 1, o_n,)* operation transmits the data on *m_n* to output *o*. This sequence of primitive operations performs the behavior of the Parallel-to-Serial Conversion function, as Figure 2 shows. Hence, MBESDSD needs only the behavior of functions and modules. It automatically deduces the rules about module application. This approach has the added benefit of alleviating redundancy in rule-based knowledge representation.

THE DESIGN PROCESS

The design process in MBESDSD consists of several phases, which include behavioral compilation, function abstraction, and implementation of the structure.

BEHAVIORAL COMPILATION

In this phase, the system translates the behavioral specification of the target system into a sequence of primitive operations. MBESDSD uses a one-pass compiler written in Pascal that runs on a VAX/780 computer.

Compilation starts with a one-to-one mapping from the given specification to primitive operations. Primitive operations are performed on signals to generate the conditions and state changes described in each event. For example, the signal purpose *clock* is translated into a sequence of Transmit operations with an input of 1 for certain time intervals and 0 for other intervals. After the translation, MBESDSD applies a post-optimization procedure using reasoning and inferencing to remove redundant primitive operations. The following description of an event illustrates compilation:

Event 3 If *i* = 1 Then (Complement *i*) -> *o* ;

The description means that while signal *i* is 1, the state of signal *o* will be the complement of *i*. This event is compiled into the following primitive operations:

value(*i*, 1, *t3*),
Not(*i*, , *pAD*, *t3*), and
transmit(*pAD*, , *o*, *t3*).

The *value* declaration, which states signal *i* has value 1 at time *t3*, corresponds to the condition part (If *i* = 1) of the Event statement. The Not operation, which states that the value of *pAD* is the complement of *i*, is translated from the keyword Complement. Finally, the Transmit operation, which states that the value of *o* is the same as that of *pAD* at time *t3*, is compiled from the "->" operator. The compiler automatically generates symbols *t3* and *pAD* in Transmit.

After direct, one-to-one mapping from behavioral specifications to corresponding primitive operations, the system analyzes the primitive operations to uncover their interactions and to remove redundant operations. Types of these analyses, or inferences, include

- *Finding input signals whose condition may be used to control other functions.* For example, let a be an input signal rising at a specific time instant t as indicated in the declaration $\text{input}(a, \text{rising}, t)$. If another primitive operation, such as $\text{store}(b, c, t)$, is also to be performed at t , but the control signal is unknown, then a can be shared as the control signal of that Store operation.
- *Identifying special-purpose control signals.* For example, if a signal is active only at the beginning of the function and it precedes all other primitive operations in the compiled sequence of that function, then it is qualified as the *restart control* signal of a function.

In the second phase of design, the system maps the set of compiled primitive operations to a more abstract, intermediate function representation.

- *Deducing the timing relationship among primitive operations.* Consider, for example, the design of the sequence detector in Figure 1. Four events must occur sequentially: the conditions (values) of the input signal x must be 1, 0, 1, and 1 in sequence, before 1 is output to z . The system needs to know the condition (value) of x at every time instant for future reference. Hence, the compiler generates the *value* declarations to link these conditions, rather than generating the *input* declarations, which indicate independent input conditions. The result of compilation is

```
value(x, v1, t1),
value(x, v2, t1+1),
value(x, v3, t1+2),
value(x, v4, t1+3),
Not(v2, v5, )
And(v1, v5, v3, v4, z, t1+3)
```

The first four value declarations state that signal x has values $v1$, $v2$, $v3$, and $v4$ at time $t1$, $t1 + 1$, $t1 + 2$, and $t1 + 3$, respectively. The primitive operation Not complements the value of $v2$ to generate the value of $v5$. The last primitive operation ANDs the values of $v1$, $v5$, $v3$, and $v4$ to generate signal z . The Not and And operations come from the testing of values. The compiler generates the signal's value declarations.

FUNCTION ABSTRACTION

In the second phase of design, the system maps the set of compiled primitive operations to a more abstract, intermediate function representation. The translation from the primitive-operation representation to the function representation is a many-to-many mapping. That is, the system can map different groups of primitive operations to the same function, or it can map the same set of primitive operations to different functions. The design objective during this phase, is to find the optimal mapping that minimizes certain design-cost functions.

To find this mapping, MBESDSD uses a heuristic best-first search algorithm, called A*. The program is written in Pascal and runs on a VAX 11/780.¹⁴ In the algorithm, the task of finding a function to cover part of the set of compiled primitive operations is viewed as the expansion of nodes. Each node in the search tree contains four sets of elements: the set of selected functions, the set of covered operations, the set of uncovered operations, and the set of added operations. A*'s search strategy consists of the following steps:

1. Initialize the root node to contain the set of compiled primitive operations in the uncovered operations set. Initialize the remaining three sets with empty sets.

During function abstraction, the design objective is to select the smallest number of functions to cover every compiled primitive operation.

2. Repeat procedures A and B below until a solution (a node with minimum cost and an empty set of uncovered operations) is found.
 - A. Expand the current node. For each distinct primitive operation in the set of uncovered operations, identify functions that may cover this primitive operation. If the union of this identified function and the set of selected functions in the current node is not identical to the set of selected functions of some previously generated node, the system will generate a child node. For this purpose, copy the four sets (selected functions, added operations, covered operations, and uncovered operations) from the current node to the child node and add the newly selected function to the set of selected functions. Add those operations that this function covers to the set of added operations. Add the support operations to the set of uncovered operations. Delete operations that appear in both the set of uncovered operations and the set of added operations and move the deleted operations to the set of covered operations.
 - B. Select a new current node. First, compute the cost function h' of each new child node. Select the one that yields the minimum cost among all unexpanded nodes.

Figure 5 gives an example of node expansion. Initially, the uncovered set contains a primitive operation *store(sig1-, sig1,4r)*, in which sig1 changes states every four cycles, and some other primitive operations (*u*). After node expansion, the Load function, the 3-Bit Count function, and other functions may cover the Store operation. If the system selects Load, a support operation to generate a *load-control* signal every four clock cycles should be added to the uncovered set. If the system selects 3-Bit Count, it can use the system clock as a *count-control* signal so no support operation is needed. On the other hand, the two lower output bits in 3-Bit Count are two unused operations and should be put into the set of added operations.

During function abstraction, the design objective is to select the smallest number of functions to cover every compiled primitive operation. There is a constraint on the number of added (primitive) operations, however, which should be the fewest possible. To this end, MBESDSD uses the following heuristic cost function, called h' , to direct the search:

$$h' = \text{weight} * \text{number of uncovered operations} + \text{number of added operations} + \text{number of functions}$$

Increasing the weight on uncovered operations biases the strategy toward using more versatile functions, which usually cover more primitive operations. This strategy, in general, reduces search time complexity. The cost is that we increase the number of added primitive operations, and the new added operations are likely to be wasted.

As we stated earlier, the goal in function abstraction step is to find a minimum set of functions that covers all primitive operations. A subgoal is the selection of a function to cover part of the primitive operations. Since different functions may cover the same primitive operations, these subgoals interact. The simplest way of managing interactions is to determine the order in which subgoals should be achieved.¹⁶ This ordering ensures that the action in achieving one subgoal does not undo an already achieved subgoal or block an as yet unachieved one. The ordering strategy in MBESDSD is implemented implicitly in the cost function h' . The strategy is to select the function that covers the largest

number of uncovered primitive operations yet leaves the minimum number of unused (added) operations. To implement this strategy, MBESDSD uses planning and meta-planning techniques.

Planning is an artificial intelligence problem-solving approach that models the overall task (goal) as the conjunction of interacting subtasks (subgoals). It then attempts to solve the entire problem by solving individual subgoals, which are often easier to accomplish. Meta-planning is a planning method concerned with managing the interactions among subgoals. It provides guidelines to help determine the order of achieving each subgoal and to select the best solution among several alternatives. As such, meta-planning is concerned with the control of planning decisions—knowing when and how to make commitments to achieve a subgoal and when not to make them.¹⁵

Meta-planning helps improve search efficiency by pruning out unpromising search directions. Without meta-planning, some types of primitive operations may be covered by too many functions. For example, *store(a-, .a,8r)*, which says that signal *a* changes its state every eight clock cycles, can be covered by Load, Change States, and Count. Moreover, there are several possible Count functions. The operation may be the highest bit of 4-Bit Count with the system clock as its count-control signal. It may also be a 3-Bit Count function with a count-control signal that has twice the period of the system clock. It may also be a 2-Bit Count function with a count-control signal that has an even longer period. Without meta-planning, node expansion would generate all the child nodes that contain these functions. With meta-planning techniques, however, the system can choose the most promising of these functions—the one that contains only one unused operation between two operations. With meta-planning, for example, the 4-Bit Count function can cover the operations we just described only if there is an operation *store(b-, .b,2r)* (the second lowest bit). There is an unused operation (the third lowest bit) of the function between these two operations and another unused operation which is the lowest bit (the bit between the second lowest and nothing) of the Count function.

Another advantage of meta-planning is that it is relatively insensitive to the order in which the compiled primitive operations are presented.¹⁴ A search method guided by meta-planning usually gets results faster than one without it.

To illustrate the effectiveness of this program, we apply it to the input sequence detector in Figure 1. The functions found are

Figure 5. An example of node expansion.

*Meta-planning helps
improve search
efficiency by pruning
out unpromising search
directions.*

*In structural
implementation,
MBESDSD maps the set
of functions from the
second phase of the
design process to
physical function
modules.*

1. *Serial-to-Parallel Conversion.* This function covers the four value declarations. The values of input signal x are stored to outputs v_1 through v_4 sequentially.
2. *Not.* This function covers the Not primitive operation. It will invert the logic value of v_2 to generate signal v_5 .
3. *And.* This function covers the And primitive operation. It generates the output signal z by ANDing v_1 , v_3 , v_4 , and v_5 .

STRUCTURAL IMPLEMENTATION

In the third phase of design, MBESDSD maps the set of functions from the second phase to physical function modules. Like the mapping in the second phase, this mapping is modeled as a heuristic search process, but perhaps with more complicated constraints. One source of complication is packaging. As modern VLSI technology advances, more and more functional modules, such as logic gates and ALU subsystems, are put into a single package. In real-world design, when a function module is selected, the corresponding IC chip, which contains other unused function modules, is selected as well. We should try to use these spare parts to implement other functions and save hardware cost. The fewer the function modules in an IC package, the smaller the size and the lower the price. For a larger scale package, the unit cost for each module is lower. Where the optimization goal falls between these two extremes depends on the number of function modules required.

The heuristic cost function in this search is

$$\begin{aligned} \text{cost} = & w_1 * (\text{cost of physical entities}) + \\ & w_2 * (\text{expected cost of unimplemented functions}) - \\ & w_3 * (\text{expected cost of unimplemented functions that are} \\ & \quad \text{expected to be implemented by spare physical entities}) - \\ & w_4 * (\text{number of spare physical entities}) + \\ & w_5 * (\text{number of unimplemented functions}) \end{aligned}$$

The expected cost of an unimplemented function is the average cost of all possible implementation methods. The system calculates the cost of a physical entity as the sum of metrics multiplied by their respective weighting factors. Metrics are price, size, speed, and power consumption of the physical entity. The weighting factors are adjustable for different requirements on the final design. The third term in the cost function is for expecting to use the spare parts to implement the still-unimplemented functions. This prevents us from using smaller packages (which have lower unit package cost) only to find that a larger package is better. The fourth term is used to force MBESDSD to use as few function modules as possible. For example, if an And function is implemented by an AND gate in a 7408 TTL IC package, three spare modules are left. But, if an And function is implemented by three NAND gates in a 7400 package, only one spare module is left. The 7408 and 7400 have same size, price, and other properties except functionality. Thus, if there is no need for a function to be implemented, both approaches have the same physical cost. The one that has more spare parts appears to be a better design, since we can use the spare modules for future circuit modification if the specification changes, and we have not added any cost. The fifth term in the heuristic cost function lets more functions be implemented.

MBESDSD implements the structural implementation phase in the C language, with a board-level digital circuit design as the target design. The modular implementation readily adapts to different VLSI design methods and criteria. Users simply change the cost functions, the library of function modules, or the library of function implementation methods.

MBESDSD uses two meta-planning strategies in this phase to cut computation time, which differ according to which functions are implemented first. One strategy implements functions with fewer implementation methods first. The second implements functions with higher implementation cost first. Spare modules in the selected packages may be useful for implementing functions that have more candidate implementation methods or less complex functions.

The input to the structural implementation phase consists of the abstracted functions of the input sequence detector. The weighting factors in the cost function w_1 through w_4 are set to 1, and w_5 is set to 2. More weights on the number of still-unimplemented functions are for expecting more functions to be implemented in each step so that the system can find results faster. MBESDSD uses the following function modules during this phase:

- a 74194 shift register package, which implements the Serial-to-Parallel Conversion function
- a 7404 inverter package with five unused inverters, which implements the Not function
- a 7420 4-input AND-gate package with one unused gate, which implements the And function.

Figure 6 shows the schematic diagram of a design generated during this phase.

DESIGN EXAMPLE

We found the design of a popular peripheral device, the ART asynchronous receiver/transmitter, useful in demonstrating the proposed design process using MBESDSD. The function of ART involves serial-to-parallel and parallel-to-serial conversion of 8-bit data. The following signals are involved in the operations:

- one 8-bit bidirectional signal: DB 0–7 (data bits 0 to 7)
- four 1-bit input signals: RD (read), WR (write), SI (serial data input), and CLK (system clock)
- three 1-bit output signals: SO (serial data output), TXRDY (transmitter ready), and RXRDY (receiver ready)

When the WR signal is rising, the system resets the TXRDY signal, indicating that the transmitter is busy with a serial output process. This process converts the 8-bit parallel input data to a sequence of eight 1-bit serial output data (SO). Similarly, when RD is low, the system resets the RXRDY signal, and converts the previous eight 1-bit serial data input (SI) (which was preceded by a 0 bit) into 1-byte parallel data output.

Figure 6. A design generated by the structural implementation phase of the design process.

We found the design of a popular peripheral device, the ART asynchronous receiver/transmitter, useful in design with MBESDSD.

```

Signal Input RD, WR, SI, CLK : 1,
DB : 8 ;
Output SO, TXRDY, RXRDY : 1,
DB : 8 ;
Function System Clock CLK ;
Event 0 If WR = Rising Then input DB,
  Reset TXRDY ;
  Next 0 -> SO ;
  Next bit 0 of input data -> SO ;
  Next bit 1 of input data -> SO ;
  Next bit 2 of input data -> SO ;
  Next bit 3 of input data -> SO ;
  Next bit 4 of input data -> SO ;
  Next bit 5 of input data -> SO ;
  Next bit 6 of input data -> SO ;
  Next bit 7 of input data -> SO ;
  Next 1 -> SO ;
  Next Set SO ;
  Next Set TXRDY ;
Event 1 IF SI = Falling;
  Next input SI ;
  Next input SI ;
  Next input SI ;
  Next input SI ;
  Next input SI ;
  Next input SI ;
  Next input SI ;
  Next Set RXRDY ;
Event 2 If ( after Event 1 ) And ( RD = 0 )
  Then ( bit 0 - 7 of input data of Event 1 ) -> DB,
  Reset RXRDY ;

```

Figure 7. Operations for the ART asynchronous receiver/transmitter encoded in a formal behavioral description language.

These operations are encoded in the formal behavior description shown in Figure 7. We are improving the syntax of the description language to include iteration structure. The description will be more compact when this is done. In the description, event 0 describes the behaviors during a serial output process, while events 1 and 2 describe the behaviors of a serial data-receiving process. Figures 8a and 8b show the equivalent timing diagrams of events 1 and 2, respectively.

As we described earlier, the first step in the design process with MBESDSD is to compile behavioral descriptions into a set of primitive operations. Figure 9 shows the compiled primitive operations for this example. Figure 9a's operations correspond to the behavior of the data-transmitting process described by event 0, while Figure 9b's and 9c's operations correspond to the behavior of the data-receiving process, which is described by events 1 and 2, respectively.

Next, in the function-abstraction phase, predefined functions are selected to cover the primitive operations. 11-Bit Parallel-to-Serial Conversion (F1) is selected to cover operations O2, O4, O5, O6, and O7. Other selected functions include

- 8-Bit Serial-to-Parallel Conversion (F2)
- 8-Bit Gate (F3)
- Reset (F4) and Set (F5) with output TXRDY
- Reset (F6) and Set (F7) with output RXRDY

During this process, MBESDSD automatically instantiates supporting primitive operations to provide the control signals required by the selected functions. Control-Signal Generation (F8) generates the control

Figure 8. Timing diagrams of ART, an asynchronous receiver/transmitter; transmitter **(a)** and receiver **(b)**.

signal of F5, a signal that is rising at time $t_0 + 11$ and is started by the WR signal. Another Control-Signal Generation function (F9) generates the control signal of F6, a signal that is rising at time $t_1 + 9$ and is started by the RD signal. To improve search efficiency, weight is added to the h' cost function given earlier to discourage the inclusion of unused operations. Consequently, functions that perform more operations than we actually need may be selected.

Finally, in the structural implementation phase, function modules implement one or more functions selected in the previous phase. The following modules and IC packages contain these functions:

- an 8-bit tristate buffer (74244)
- four RS flip-flops (two 7474)
- 8-bit parallel-input shift registers (one 74166 for each)
- one 8-bit parallel-output shift register (74164)
- two 4-bit counters (74161)
- two 4-input NAND gates (7420)
- three inverters (7404)
- a 2-input AND gate (7408)

Figure 10a shows the finished circuit. One 4-bit counter, one 4-input NAND gate, and one or two inverters are implement one of the Control-Signal Generation functions (F8 or F9). MBESDSD first uses a 4-bit counter to implement the function. The support function for using the function module is a 4-Line-to-16-Line Decode function. Possible implementation methods of this function include a 4-line-to-16-line decoder and 4-input NAND gates with some Not functions as their support functions. In this case, a 4-input NAND gate with some Not functions is the best choice, since only one output of the decode function is needed and a NAND gate is much cheaper than a decoder. A predesigned nonrestartable control-signal generator (ctgen1) helps resolve the ambiguity in the condition of event 1. By using this module, we ensure that 0 data bits will not restart event 1.

During the search, the system uses a meta-planning heuristic to give higher priorities to functions with the fewest alternative implementation methods. The rationale is that functions with a larger variety of alternative implementation methods have a better chance to be realized as a byproduct—and are hence free of extra cost—by modules selected to implement other functions. We found that without such a heuristic, the time to complete a search would be prohibitively long for realistic design problems.

We ran this sample design on a VAX 8200 super-minicomputer. The behavioral compilation phase took one second in CPU time. The function-abstraction phase took 24.3 seconds with weights for uncovered operations set to 2. The structural implementation phase took 4.9 seconds.

Wu has examined the design generated by MBESDSD and made a minor improvement, which is shown in Figure 10b. Part of the components in the control-signal generator in Figure 10a have been removed. Since only one control signal is generated in this case, the design can be simplified. The use of a complicated module was due to insufficient design knowledge in the libraries we used during the structural-implementation phase. As more design knowledge is added to the library of function-implementation methods and the module library, the quality of the design should improve.

*During the search,
the system uses a
meta-planning heuristic
to give higher priorities
to functions with the
fewest alternative
implementation
methods.*

```

01. input(WR, rising, t0),
02. value(DBi, V0i, t0), for i = 0 to 7,
03. store(0, , TXRDY, t0),
04. transmit(0, , SO, t0 + 1),
05. transmit(V0i, , SO, t0 + 2 + i),
06. transmit(1, , SO, t0 + 10),
07. store(1, , SO, t0 + 11), and
08. store(1, , TXRDY, t0 + 11).

```

(a)

```

09. value(SI, falling, t1),
010. value(SI, V1i, t1 + 1 + i), for i = 0 to 7,
011. store(1, , RXRDY, t1 + 9).

```

(b)

```

012. input(RD, 0, t2), for t2 > t1 + 8,
013. transmit(V1i, , DBi, t2),
014. store(0, , RXRDY, t2)

```

(c)

Figure 9. Compiled primitive operations for the sample design using MBESDSD; behavior of the data-transmitting process described by event 0 (a), behavior of the data-receiving process described by event 1 (b), and behavior of the data-receiving process described by event 2 (c).

Running the sample design on a VAX 8200 supermini, behavioral compilation took one second in CPU time, while function abstraction took 24.3 seconds. Structural implementation took 4.9 seconds.

(a)

(b)

Figure 10. Circuit design of the ART asynchronous receiver/transmitter using MBESDSD's output (a) and as improved by a human designer (b).

Our digital synthesis system is perfect for designing small-to-medium digital circuits from purely behavioral specifications. Because MBESDSD uses extensive search operations in the two minimization steps—function abstraction and structure implementation—at least in its current form, it would be too cumbersome for the design of very large scale digital circuits. Such limitations are due primarily to MBESDSD's low level of design abstraction. Five primitive behavioral operations are all the system needs to optimize the design. Since these primitive operations describe the behavior of circuit building blocks at the register level, the complexities of digital system design are limited.

For the design of large-scale digital systems, we recommend adopting the top-down structural design approach to decompose the entire system into relatively independent subsystems and then applying MBESDSD to synthesize each subsystem. For this purpose, we plan to enhance the syntax of the input behavioral description language to allow the hierarchical description of large-system behavior. A more challenging task is to enhance the behavioral operation compiler so that we can decompose the system at the behavioral level.

A second future research direction is to incorporate information about how long it will take to get the optimal mapping between operations and functions, so that MBESDSD can also design real-time digital systems. We can do this by including delay information in the libraries of functions, and physical modules. Clearly, optimization will become more complicated, however.

A third future research task is to automate the maintenance of the function and module libraries. We are using theorem proving to resolve the relationship between a function and a function module. When we add a new function module to the module library, we want to be able to add only its physical properties and its behavior (the primitive operations it performs). The resolution program applies the initial conditions of a function to the function module and tries to prove that the behavior of the function is true. If it is so proved, then the function module can implement the function and the system will add new information to the design library. We plan to have the system perform this process iteratively for every function in the library.

We believe the design of MBESDSD has set a new direction in digital-system synthesis by its use of problem-solving techniques based on artificial intelligence. The result is an approach that not only improves the quality and efficiency of the digital-system synthesis but offers a potential solution to other CAD problems as well.

*We plan to enhance the
syntax of the input
behavioral description
language to allow the
hierarchical description
of large-system behavior.*

ACKNOWLEDGMENTS

Y. Hu's work is supported by a grant from the National Science Foundation under contract MIP8896111.

REFERENCES

1. J. Darringer et al., "LSS: A System for Production Logic Synthesis," *IBM J. Research and Development*, Vol. 28, No. 5, Sept. 1984, pp. 537-545.
2. A. de Geus and W. Cohen, "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design & Test of Computers*, Vol. Aug. 1985, pp. 22-32.
3. K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proc. Design Automation Conf.*, June 1987, pp. 341-347.
4. T. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, 1985.

Jung-Gen Wu is an instructor in the Department of Information and Computer Education at National Taiwan Normal University, Taipei, Taiwan, where his research interests include automated synthesis of digital systems, computer architecture, and digital circuit design. Previously, he worked at the university's Tjingling Industrial Research Center in microprocessor applications and has been a visiting scholar in the Department of Computer Science and Engineering at Southern Methodist University, Dallas. He holds a BS, an MS and a PhD in electrical engineering from National Taiwan University and is a member of IEEE, ACM, and AAAI.

5. M. McFarland "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proc. Design Automation Conf.*, June, 1986, pp. 474-480
6. T. Mitchell, L. Steinberg, and J. Shulman, "A Knowledge-Based Approach to Design," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 5, Sept. 1985, pp. 502-510.
7. F. Brewer and D. Gajski, "An Expert-System Paradigm for Design," *Proc. Design Automation Conf.*, June 1986, pp. 62-68.
8. J. Granacki, D. Knapp, and A. Parker, "The ADAM Advanced Design Automation System: Overview, Planner, and Natural Language Interface," *Proc. Design Automation Conf.*, June 1985, pp. 727-730.
9. D. Knapp and A. Parker, "A Design Utility Manager: The ADAM Planning Engine," *Proc. Design Automation Conf.*, June 1986, pp. 48-54.
10. P. Subrahmanyam "Synapse: An Expert System for VLSI Design," *Computer*, Vol. 20, No. 7, July 1986, pp. 78-88.
11. P. Koton, "Empirical and Model-Based Reasoning in Expert Systems," *Proc. Int'l Joint Conf. Artificial Intelligence*, Aug. 1985, pp. 297-299.
12. E. Rich, *Artificial Intelligence*, McGraw-Hill, New York, 1983.
13. W. Ho and J. Wu, "Minimization in Digital Design as a Meta-Planning Problem," *SPIE Applications of Artificial Intelligence V*, Vol. 786, May 1987, pp. 12-19.
14. J. Wu et al., "Function Search from Behavioral Description of a Digital System," *Proc. Design Automation Conf.*, June 1987, pp. 574-579.
15. M. Stefik, "Planning and Meta-planning: MOLGEN, Part 2," *Artificial Intelligence*, Vol. 16, 1981, pp. 141-169.
16. Sacerdoti, E. D., "The Nonlinear Nature of Plans", *Proc. Int'l Joint Conf. Artificial Intelligence*, 1975, pp. 206-214.

No photo or biography is available for **William P.-C. Ho**.

Yu Hen Hu is an associate professor in the Department of Electrical and Computer Engineering at the University of Wisconsin, Madison, where his research interests include VLSI signal processing, spectrum estimation, fast algorithms and parallel computer architectures, and CAD tools for VLSI that use artificial intelligence. Previously, he was an assistant professor in the Department of Electrical Engineering at Southern Methodist University, Dallas. He holds a BSEE from National Taiwan University and a PhD in electrical engineering from the University of Southern California, Los Angeles. He was an associate editor of *IEEE Transactions on Acoustic, Speech, and Signal Processing* in system identification and fast algorithms. He is a member of IEEE, SIAM, and Eta Kappa Nu.

David Y.Y. Yun is professor of electrical and computer engineering at the University of Hawaii, as well as director of the Information Technology Division of the Pacific International Center for High Technology Research and director of the board for Hawaii's High Technology Development Corp. His research interests include system intelligence, information technology, design processes, and parallel and distributed computing. Previously he was professor and chairman of Southern Methodist University's Department of Computer Science and Engineering. He was also manager of the Symbolic Computation Group at IBM T.J. Watson Research Center. He holds a PhD in computer science and applied math from MIT.

Address questions and comments on this article to Y. Hu, Dept. of Electrical and Computer Eng., Univ. of Wisconsin, 1415 Johnson St., Madison, WI 53706; hu@engr.wisc.edu