Synthesis From Mixed Specifications

Vincent J. Mooney III
Computer Systems Lab
Stanford University
Stanford, CA 94305
mooney@aglaia.stanford.edu

Claudionor N. Coelho Jr.
Dep. de Ciência da Computação
ICEx/UFMG
Belo Horizonte, MG, Brazil
coelho@dcc.ufmg.br

Toshiyuki Sakamoto*
Giovanni De Micheli
Computer Systems Lab
Stanford University
Stanford, CA 94305
nanni@stanford.edu
* on leave from Toshiba

Abstract

We present a hardware synthesis system that accepts system-level specifications in both Verilog HDL and C. A synchronous semantics is assumed for both languages in order to guarantee a uniform underlying model. The rationale for mixed input specifications is to support hardware/software co-design by allowing the migration to hardware of system modules originally described in the C language.

We discuss assumptions and limitations of the input description style, a high-level synthesis system, and the application of such a system to some design examples.

1. Introduction

Hardware/software co-design of embedded systems [1] involves several tasks, some of which are supported by computer-aided design tools. Several approaches have been proposed for the *co-synthesis* of hardware/software systems. Their major differences lay in the style for hardware/software specifications and in the latitude that CAD tools have in refining the specifications into implementations that best leverage the features of hardware and software to reach the desired design objectives.

Co-synthesis approaches can be differentiated by considering the system-level specifications, which can be *homogeneous* (i.e., in a single specification language) or *heterogeneous* (i.e., involving multiple modeling paradigms). Hardware/software *partitioning* is a key problem in co-synthesis from homogeneous specifications. Partitioning determines the components of the system that will be implemented as hardware blocks and software routines. Some implementation objectives, such as performance, cost and programmability depend heavily on the partitioning choice.

Previous work on hardware/software partitioning addressed various facets of the problem. The COSYMA synthesis tool suite [2] partitions a system specification to accelerate software execution by using a dedicated hardware co-processor (to be synthesized). The original system model is described as a software program in an extension of the *C* programming language. The system model is compiled into a control/data-flow graph and a partitioning algorithm identifies the computational bottlenecks which are implemented as application-specific hardware. The VULCAN synthesis tool suite [3, 4, 5] uses instead a hardware model of the system (in the *HardwareC* language) and attempts to reduce the cost of a full-hardware implementation by transferring non-critical operations to routines executing on a standard processor or processor core. The system model is again compiled into a control/data-flow graph, which is partitioned yielding: (i) a set of software threads to be compiled and executed on the standard processor and (ii) the specification of the remaining hardware circuits, that can be synthesized as a netlist of logic gates. Hardware/software synchronization units for interfacing the processor to the application-specific logic are also automatically generated [3, 5, 6]. Both COSYMA and VULCAN partition the system specification with a fine granularity, i.e., partition blocks are sets of operations. Conversely, the CO-SAW tool suite [7] partitions systems with a coarse granularity, i.e., blocks correspond to processes. Whereas some optimality is lost in using a coarse granularity in partitioning, the resulting implementation is often closer to what designers expect, and interfacing hardware to software blocks is easier. Overall, cost/performance estimation of a partitioned imple-

mentation remains the key obstacle to surmount within partitioning, because it affects the partitioning choices and the final quality of the solution.

System designs that are modeled by heterogeneous specifications are often already partitioned into modules, each one described in the paradigm that best suits its nature. In this case, the hardware/software boundary is often predetermined, and co-synthesis does not involve a partitioning step. On the other hand, software/hardware *migration* (and vice versa) is applicable to heterogeneous specifications. Thus, software blocks specified in programming language can be implemented in hardware, using an appropriate hardware synthesis path. Similarly, software routines can be synthesized from hardware blocks.

Examples of co-design systems that use heterogeneous specifications are PTOLEMY [8], SIERA [9] and COWARE [10]. PTOLEMY is an environment and simulator for highly heterogeneous systems, SIERA is a prototyping environment and COWARE is a design system for embedded telecommunication applications. For example, COWARE supports inputs in C, VHDL, and DFL [11] (a data-flow language), with VHDL and DFL mapped to hardware while the C code is compiled onto a DSP processor [10].

We consider here the hardware synthesis component of a co-design environment, where a heterogeneous specification paradigm is used. Namely, system-level specifications in Verilog HDL and in C are supported as well as their synthesis into logic gates. We assume a *synchronous semantics* for both Verilog and C, so that the underlying model of computation is the same, and we subset both languages to support only synthesizable constructs.

Our synthesis system, called PARNASSUS synthesis system, targets control-dominated applications, and leverages previous research on modeling hardware using *control-flow expressions* (CFEs) [12, 13]. In PARNASSUS, CFEs represent an intermediate model applicable to both Verilog modules and C programs, that captures the global control-flow information in the system. PARNASSUS converts the input specifications into CFEs, and then synthesizes the corresponding datapath and control-unit.

PARNASSUS differs from other hardware synthesis systems in that it supports input specifications in a HDL and in a programming language. Since many hardware circuits are often modeled in C for fast evaluation or prototyping reasons, our system favors the migration of software models to hardware circuits.

The rest of the paper is organized as follows. Section 2 describes briefly the organization of the PARNASSUS system. Section 3 describes the assumptions and limitations of the input modeling style. Section 4 presents the control and data-path synthesis tools while Section 5 gives some experimental results and presents an example in detail.

2. Parnassus

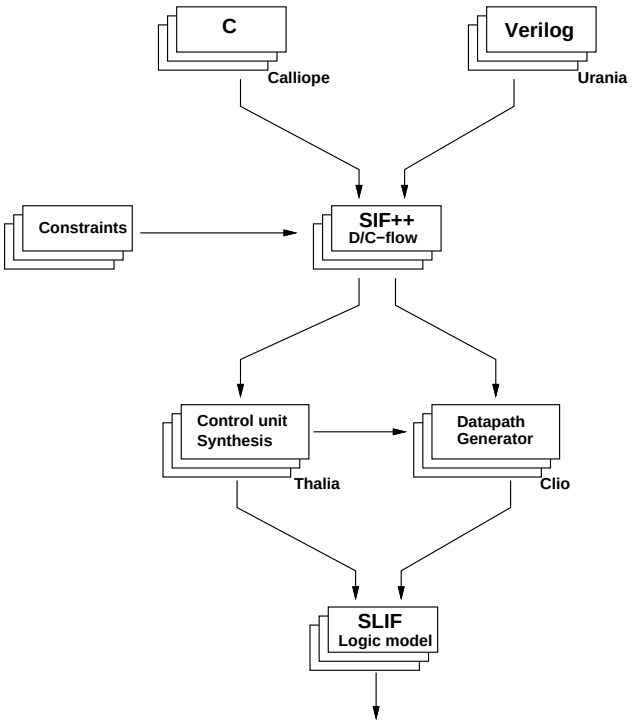

Figure 1. Block diagram of Parnassus Synthesis System

The PARNASSUS synthesis system consists of a set of tools, whose names recall those of the muses inhabiting the Parnassus mountain. The systems has two front-ends, called CALLIOPE and URANIA, which parse C programs and Verilog models respectively. These two tools extract information about the system being designed and store the data-flow and control-flow information in an intermediate format called *SIF++*.

PARNASSUS targets the design of concurrent systems, where synchronization and control-flow is of primary concern. (Some examples of the class of circuits that we consider are protocols for cache coherency or network communication.) For this reason, PARNASSUS synthesizes the control-unit separately from the datapath, by means of tools THALIA and CLIO respectively. The output of these tools is a logic-level description in the *SLIF* format, which can be mapped to a specific library by standard tools.

3. Modeling

PARNASSUS supports inputs in Verilog and C and thus digital systems can be modeled at different levels of detail, spanning the range from structural bit-oriented hardware descriptions all the way up to behavioral word-oriented software representations.

This approach matches design practice, where designers often describe their systems in a heterogeneous way, using description languages appropriate to the subsystem being implemented. Models in the C programming language are sometimes early functional models of hardware, whose automated synthesis reduces greatly the design time. Otherwise, C models may be software routines that designers decide to implement in hardware at a later stage. PARNASSUS supports the combined synthesis of Verilog and C models, even though none of its tools at the moment suggests the choice of a software or hardware implementation.

3.1. Modeling assumptions and limitations

Despite the different natures of Verilog and C, we require descriptions in either language to have the same underlying model of computation. This allows us to construct synthesis tools from the same intermediate form derived from both languages. We assume a *synchronous* operation. For the sake of simplicity, we consider a single clock. Data-flow is modeled by assignment in both languages, and abstracted as *operations* and *dependencies*. Operations, or groups of operations, take an integer number of cycles (possibly unbounded) to execute. Branching decisions are taken instantaneously, and iterations take also an integer number of cycles (possibly zero or unbounded).

The control-flow of the system being modeled is abstracted as a set of expressions, called CFEs [12], representing the serial/parallel flow of computation, branching, iteration, synchronization and exceptions. Such expressions have a deterministic finite-state machine (FSM) semantics, and can be compiled into specification FSMs representing the possible control-flow implementations.

In order to be able to reduce Verilog and C to this common data-flow and control model, some restrictions must be placed on the constructs that can be used, which will be detailed in the next sections. We also want to keep some correspondence between C and Verilog models. Thus, we use the notion of a C *process*, which is a procedure equivalent to a Verilog *module*. A C *function* is made equivalent to a Verilog function, by allowing one return value and prohibiting pass by reference in the arguments to the function call. Finally, a *procedure call* in C is equivalent to a Verilog *task*. This correspondence is displayed in Figure 2.

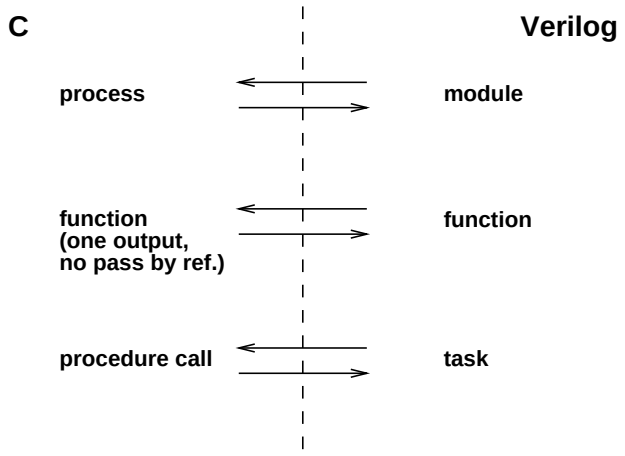

Figure 2. Correspondence between C and Verilog models

3.1.1 Verilog assumptions

We assume the Verilog code to be specified using one of three flavors: behavioral style, combinational style or structural style. These styles reflect current methodologies used by designers.

The behavioral style allows us to model single clock synchronous designs. In this style, we ignore the exact timing of the operations in Verilog, since this timing will depend on the particular component library chosen in datapath and logic synthesis.

Verilog `reg` variables in behavioral style hold their values once set, and so behave like latched outputs. All operations have their results placed in registers, including assignments to output ports. Thus, responses to external events cannot occur until the next cycle.

We implement a restricted form of the Verilog disable construct, where a process can only disable processes in its hierarchy.

The combinational style allows us to specify blocks of combinational logic. It assumes that all inputs to the logic appear in the sensitivity list of the logic blocks. Such blocks are presumed to finish their computation by the next clock cycle. We do not allow the same variable to appear in both the left and the right hand sides of a combinational logic; e.g., statement $b = b + a$; is disallowed. (Note that this is allowed in the behavioral style).

The structural style is used to specify the block diagram of the system to be synthesized. It assumes the

usual Verilog declarative semantics for structures.

3.1.2 C assumptions

We implement a subset of C as input to PARNASSUS. All control-flow constructs are supported (`continue`, `if-else`, `while`, `for`, `break`, and `switch-case`). Recursive functions and `goto` are not supported. We also do not support pointer manipulation or dynamic memory allocation at present (note that the system described in [15] does support some limited dynamic memory allocation).

3.2. Constraints and non-determinism

PARNASSUS supports the specification of constraints on hardware resources, on synchronization and on operation timing. Timing constraints can be used to serialize operations that have no data-flow dependencies and to model output events with the desired waveforms.

Multiple facets of a design implementation can be captured by means of *decision variables*, that can be used as switches among different implementation options. For example, modeling communication between two circuits may be achieved via a FIFO or via a memory. Both paths can be specified, and the implementation can be chosen during synthesis based on the value of the desired design objective.

4. Synthesis

4.1. THALIA Control Unit Generation

Synthesis of control units is divided into two phases. In the first phase, the control-flow input graph is translated into the algebraic representation of control-flow expressions [12]. This is performed by a traversal of the *SIF++* control-flow graph. The CFE representation is augmented with the design constraints. CFEs abstract also the behavior of the circuit environment, as a constraint on its implementation.

THALIA converts the CFE model to a finite-state machine representation, by manipulating the derivatives [12] of the CFEs, which model the global system states. THALIA next analyzes the finite-state system model, where the unreachable states violating synchronization constraints are eliminated. If the resulting machine is empty, then the system model is overconstrained. Otherwise, THALIA statically schedules operations in basic blocks such that the final implementation satisfies the desired constraints. Non-determinism in the original specification is resolved during the synthesis of control-units by selecting a deterministic implementation that optimizes the design goals.

THALIA finally outputs a controller finite-state machine in *SLIF* format and the constraints for the datapath generator in *KISS* format.

4.2. CLIO Datapath Generation

CLIO implements a straightforward datapath generator. The input to CLIO is the *SIF++* intermediate form and the *KISS* file generated by THALIA, containing the control information. CLIO takes also as input a library of functions, such as ALUs, multiplexers, shifters, etc., which are used as datapath components. Registers are inserted to ensure correct timing between data input and output as specified by the constraints contained in the *KISS* file generate by THALIA. The output of CLIO is a *SLIF* file. Since the control-unit generation and datapath generation are handled separately in PARNASSUS, both *SLIF* files they generate are needed for logic synthesis. For library binding, we have used CERES [16] with a standard library file, such as the LSI Logic 10k library.

5. Example and experimental results

In this section, we present an example of how a design can be successfully synthesized using the system described in the previous sections. We consider a control-dominated design containing concurrent models, communication, and complex interface constraints and show how it can be specified and synthesized in PARNASSUS.

For our example, we consider the transmission block of an ethernet co-processor, one which uses register variables to encode the different states of the protocol. The block diagram of the transmission block is shown in the context of the entire ethernet co-processor in Figure 3. The co-processor contains three units: an execution unit, a reception unit and a transmission unit. The transmission protocol executes concurrently and interacts through data transfers and synchronization. Thus, our example explores ethernet co-processor synchronization.

The transmission unit for DMA frame transmission is modeled by three concurrent processes, known respectively as *dma_xmit*, *xmit_frame* and *xmit_bit*. Process *dma_xmit* initiates a transmission by sending data to *xmit_frame* which includes the source and destination addresses of the DMA transfer. Then *xmit_frame* appends the appropriate header and tail information, such as the preamble, start frame delimiter, and parity

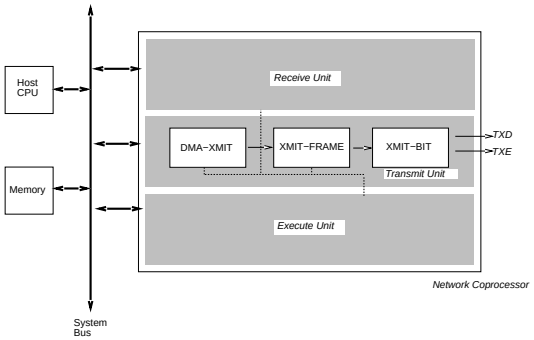

Figure 3. Ethernet co-processor transmission block diagram

bits. Upon receiving a byte from process *xmit_frame*, *xmit_bit* sends the corresponding bit stream over the line `TXD`. Thus, *xmit_bit* must receive each byte eight cycles apart, which constrains the rate at which the bytes are transmitted from *xmit_frame*. In our example, we specify *dma_xmit* in Verilog, *xmit_frame* in C and *xmit_bit* in Verilog.

Process *xmit_frame* was specified as a program state machine as seen in Figure 4. Because we have to abort the transmission of a frame if `CCT` becomes true, we implemented the program state machine with a `while` loop which pools signal `CCT`, and a `case` statement on variable *state*, which determines the next state of the program state machine to be executed. Note that this state variable is not part of dataflow and it should be incorporated into the control-unit for *xmit_frame* [13]. CALLIOPE translates the C specification of *xmit_frame* to *SIF++*. Then THALIA extracts the control flow from the *SIF++* intermediate representation so as to include variable *state* in the synthesized controller. Processes *dma_xmit* and *xmit_bit* were both input to the PARNASSUS system using URANIA. All of the processes had CLIO generate the datapath.

Table 1 presents the results for the synthesis of *dma_xmit*, *xmit_frame*, and *xmit_bit*. In Table 1, the second column shows the number of lines of C the specification required, while the third column shows the number of lines of Verilog. Finally, the last column shows the number of gate equivalents the hardware required using the LSI Logic library.

We simulated the transmission unit using the logic

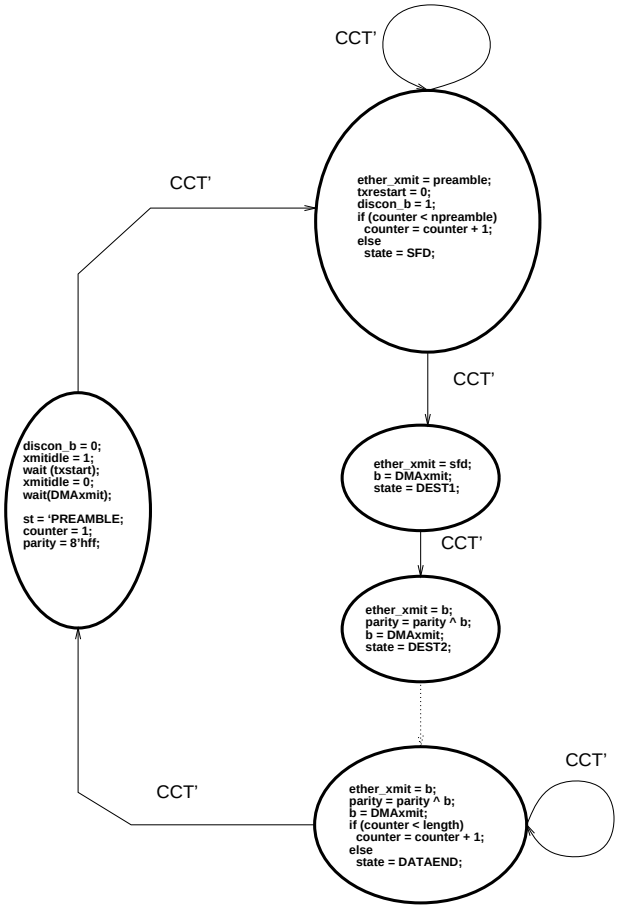

Figure 4. Program state machine for process *xmit_frame*

simulator MERCURY of the OLYMPUS system. The resulting waveform with a particular test pattern can be seen in Figure 5. Signal `xmit` contains the upper and lower bytes alternately of the source address for DMA transmission, while `Baddr` contains the destination address. Signal `xmit_byte` contains the data bytes to be transferred, which are placed serially on `TXD`. `TXE` is an enable signal.

6. Conclusion

The PARNASSUS system allows designers to perform system-level design from a multiple-paradigm input perspective. We have shown how one can synthesize synchronous control-dominated designs specified at a high level in C and Verilog. We have utilized the methodology of control-flow expressions to synthesize the control portion of control-dominated specifications. Some assumptions were necessary to guarantee a *syn-*

Process	# Lines C	# Lines Verilog	Area
dma-xmit		190	5552
xmit-frame	213		6218
xmit-bit		45	532

Table 1. Results for the synthesis of the transmission unit

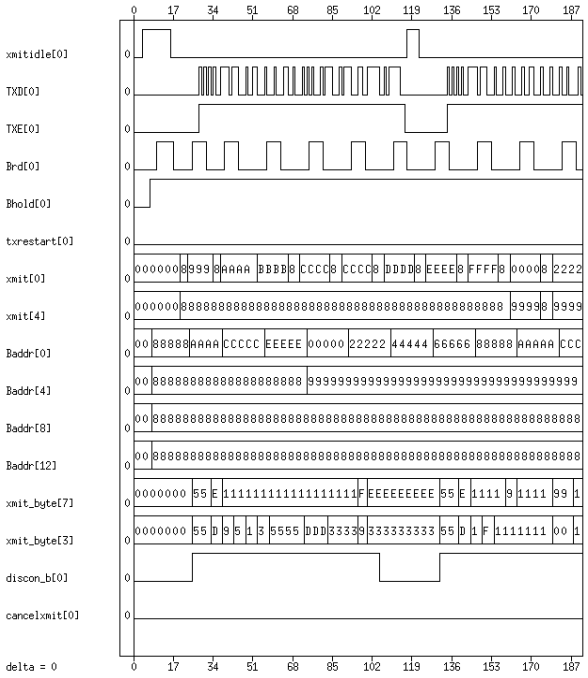

Figure 5. Waveform Display of Synthesized Transmission Unit

chronous semantics for both Verilog and C; such conditions preserve common underlying model for both. Thus, the system potentially allows designers to migrate their code from one language domain to another as they further refine their system implementation.

Acknowledgements

This research was sponsored by ARPA, under grant No. DABT63-95-C-0049.

References

[1] G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design,* Kluwer Academic Publishers, Norwell, MA, 1996.

[2] J. Henkel, Th. Benner, R. Ernst, W. Ye, N. Serafimov and G. Glawe, "COSYMA: A Software-Oriented Approach to Hardware/Software Co-design," *The Journal of Computer and Software Engineering,* Vol. 2, No. 3, pp. 293-314, 1994.

[3] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Boston, MA, 1995.

[4] R. Gupta, C. Coelho, and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," *Proceedings of the 29th Design Automation Conference,* pp. 225-230, June 1992.

[5] R. Gupta and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers,* pp. 29-41, September 1993.

[6] R. Gupta and G. De Micheli, "Program Implementation Schemes for Hardware-Software Systems," *IEEE Computer,* Vol. 27, No. 1, pp. 48-55, January 1994.

[7] Jay K. Adams and Donald E. Thomas, "Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems," *International Symposium on System Synthesis,* pp. 10-15, September 1995.

[8] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal on Computer Simulation,* Vol. 4, pp. 155-182, April, 1994.

[9] M. B. Srivastava and R. W. Broderson, "Rapid-Prototyping of Hardware and Software in a Unified Framework," *Proceedings of the IEEE International Conference on Computer-Aided Design,* pp. 152-155, 1991.

[10] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Co-design of DSP Systems," in G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design,* pp. 75-104, Kluwer Academic Publishers, Norwell, MA, 1996.

[11] P. Willekens, et. al., "Algorithm Specification in DSP Station using Data Flow Language," *DSP Applications,* pp. 8-16, January 1994.

[12] C. N. Coelho Jr. and G. De Micheli, "Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions," *IEEE Transactions on CAD/ICAS,* (to appear), and Technical Report CSL-TR-96-694, http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-96-694, Stanford, CA, April, 1996.

[13] C. N. Coelho Jr., *Analysis and Synthesis of Concurrent Digital Systems Using Control-Flow Expressions*, Ph.D. Thesis, Technical Report CSL-TR-96-690, http://elib.stanford.edu/Dienst/UI/2.0/Describe/stanford.cs%2fCSL-TR-96-690, Stanford, CA, March, 1996.

[14] Verilog-XL Reference Manual, Version 1.6, 1991.

[15] G. de Jong, B. Lin, C. Verdonck, S. Wuytack and F. Catthoor, "Background Memory Management for Dynamic Data Structure Intensive Processing Systems," *Proceedings of the IEEE International Conference on Computer-Aided Design,* pp. 515-520, November 1995.

[16] G. De Micheli, D. Ku, F. Mailhot, T. Truong, "The Olympus Synthesis System," *IEEE Design & Test of Computers,* pp. 37-53, October 1990.