A graphical hardware design language

Paul J. Drongowski
Jwahar R. Bammi
Ranganathan Ramaswamy
Sundar Iyengar
Tsu-Hua Wang

Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

Gdl is a graphical hardware design language that separates design decisions into three interrelated, but distinct domains: behavioral, structural and physical. Specific language features are provided to represent a design in each of these domains. This report describes the process model for Gdl. Functional behavior is separated into distinct activities called "processes" (autonomous control centers.) The computations performed by a process are specified in its behavior graph. Processes may communicate with each other through ports where the channel between two ports may be an abstract logical link or may be a physical bus. Provisions are made for synchronization. The paper concludes with an evaluation of Gdl and suggestions for future research.

1 Introduction.

Gdl is a graphical design language which addresses design issues at the architectural (specification, abstract machine) and organizational (description, implementation) levels. Gdl supports a separation of design concerns by providing specific and disjoint language features for behavioral, structural and physical (B/S/P) design decisions. These features are called **Gdl/b, Gdl/s** and **Gdl/p** and correspond to the behavioral, structural and physical domains, respectively. Within a particular domain, language features are provided to capture concerns at both the architecture and organization levels of abstraction. Thus, there are really six categories of language constructs as shown in Table 1.

[0]This work is supported by the Semiconductor Research Corporation (contract 86-08-092), the National Science Foundation (grant DMC85-08362) and the Microelectronics and Computer Technology Corporation. This paper will be submitted for publication; Prepublication courtesies are requested.

This document is an overview and evaluation of Gdl. We will concentrate primarily on the process and behavioral model for Gdl as the structural and physical representations are relatively conventional. More detailed information on Gdl is available in [2,5,21,36]. Following a comparison with other graph-based systems, the paper concludes with a summary of our experiences with the language and possibilities for future research.

2 The design process.

In order to put this discussion in perspective, a model of the design process is briefly described below (Figure 1.) The model is discussed in detail in [9,8].

The process of designing digital systems is a series of transformations of two types. One is a transformation of the design through several levels of representation. The other is the process of adding detail to a design within a given level by specifying how parts of the design are built from lower level building blocks. The levels of representation are *architecture*, *organization*, *logic* (both unilateral and switching), *electrical* and *layout* or geometry. Within a given level there are three views of a design, namely *behavioral*, *structural* and *physical* views. These views roughly correspond to the behavioral, structural and physical design decisions that must be made by an engineer during the design process. The process of refinement within a level ends when the design is expressed in terms of the primitives of that level of representation. These primitives are called *components* in our terminology.

We use the term "architecture" to refer to that level of abstraction which deals with abstract or "paper" machines. At this level, the engineer is most interested in specifying the general characteristics of a system which will drive trade-offs and choices during the remainder of the design process. The term "organization" refers to the actual implementation of a system with a fairly coarse degree of descriptive granularity. The kind of design supported at this level is typified by register transfer languages such as ISP' or AHPL. [19,32] Realizable design choices are made at the organization level.

3 Gdl.

Gdl seeks to support the representation of a design at both the architectural and organizational levels of abstraction. Agent (the design system based on Gdl) assists the specification of the architecture and its translation to a realizable organization. This section is a brief overview of Gdl at both levels.

3.1 Architectural Gdl.

The architectural level specification captures the abstract behavior of the system in Gdl. The behavior is specified in terms of the behavior of one or more *processes* that communicate with each other or with the external environment via ports on their boundaries employing well defined protocols. (Processes are discussed further in Section 4.) The structure that ties these processes together, is described in Gdl/s.

Gdl/s.

Structural entities at the architectural level are *processes*, *ports* and *channels*. Processes are computational activities that intercommunicate with each other through ports over channels. A port is associated with a particular process and is an interface through which digital (or perhaps analog) information will be transferred. Ports are typed in the programming language sense and electrical characteristics may also be specified if that information is known. Some information which will be stored within the system will eventually be "visible" at its structural boundary (interface.) Visible storage elements, called *variables*, are also declared in Gdl/s at the architectural level. Variables are typed and correspond to the programming model for the system. Variables must eventually be resolved into realizable storage circuits during organization level design.

Gdl/b.

The communication behavior of the system is specified in an annotated graph schema which is founded in Petri Net theory. [29] In the communication behavior graph associated with a process, synchronization operations are modelled by shared places and the movement of control tokens. Data signals are written or read by assignment (transfer) statements associated with the graph transitions. The behavior graph specifies the nature and sequence of communication events. Assignment transitions also specify storage side-effects, that is, how the value of one or more variables may change as a result of firing the transition.

Gdl/p.

Because the architectural level involves the specification of abstract structure and behavior, the physical domain ordinarily would be empty. In Gdl/p, however, the overall characteristics of the implementation circuit technology and spatial process partition may still be specified. The partition shows the spatial grouping of processes indicating those processes which communicate locally (co-resident on a single chip) and those processes which must communicate across chip boundaries. Technological information includes, but is not limited to, maximum chip size, current and power constraints and expected inter-chip communication delay times. These physical specifications are needed during partitioning to assure the satisfaction of broad technology-dependent design constraints.

3.2 Organization level Gdl.

The organization level represents a "high level" physical structure. All the modules are physical hardware entities that must be realized in hardware and all the interconnections are physical data paths over which actual data transfers will occur. In short, the performance and cost trade-offs and binding decisions of abstract state elements, operators and transfers to physical hardware and data paths have been resolved and are specified by the designer. The design at this level completely specifies each function which has to be realized, all the data transfers which have to take place, and the control flow that has to affect these data transfers in order to achieve the desired behavior at a desired performance and cost level.

Each process in the architecture is mapped to a module. Modules may contain submodules, but they may not contain modules that are the implementation of another process. There are no nested processes in Gdl. Nested processes imply a hierarchical control and communication model along with the attendant performance disadvantages of hierarchical control.

The components of a module are physical state elements like register, register stack, memory, combinational elements, ports and an interconnecting structure of wires and busses. These components of the module may be organized in any convenient hierarchy of modules and components. The control graph associated with the module that implements the process describes the physical control flow in the organization that effects the data flow in the module to achieve the desired behavior. A suitable controller can be synthesized from the control graph yielding a complete system implementation.

Table 2 summarizes the relationship of architectural level components to organizational level components.

Gdl/s.

The Gdl/s organizational description is a formal block diagram called a "structure diagram" which instantiates one or more components or modules and describes their interconnection (Figure 3.) Variables and enabling conditions must be bound to real storage components and signals. Combinational components implement any primitive data operations that may appear in the Gdl/b.

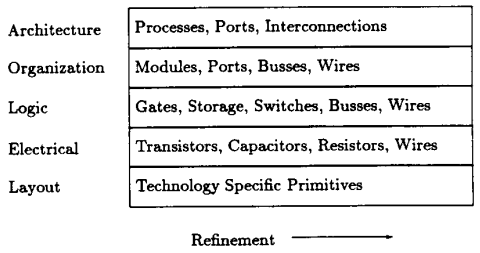

Figure 1: A Model of the Design Process.

	Behavior	Structure	Physical
Architecture	Control and communication graph	Process Port Visible state Channel	Spatial process partition
Organization	Control and communication graph	Process Port Module Component Connection	Technology Floor plan Pin map

Table 1: Separation of concerns in Gdl.

Architecture	Organization
Process	Module and Control Organization
Variables	State Elements
Operators	Modules/Combinational Elements
Ports	Ports/Control Signals
Interconnection	Wires/Busses
Control Flow	Control Organization

Table 2: Mapping of Architecture to Organization.

Gdl/b.

At the organizational level, the Gdl/b graph describes the scheduling of computational events within the Gdl/s datapath structure (Figure 2.) The graph must eventually be translated to a controller which implements the desired sequencing behavior. Thus, a list of control signal assertions must be constructed for each transition (event) in the Gdl/b graph. This information can be derived from the component library and the Gdl/s description.

Gdl/p.

The Gdl/p description consists of one or more interrelated chip floor plans (Figure 4.) Each component or module in a floor plan must correspond directly to its counterpart in the Gdl/s description. Gdl/p also includes *pad* and *pin maps* that describe the interlevel connections from chip to package to board. Although we have concentrated on chip design, Gdl/p could be extended to address physical planning at board, chassis and rack levels within the standard packaging and interconnection hierarchy.

4 Processes.

Processes are the chief subject of Gdl/b at the architectural level. The spirit of a process is that it is an entity that reacts to stimuli from its environment and performs a well-advertised set of functions. A process represents one "autonomous control center." How values are communicated to or from a process is called its *communication behavior.* The *functional behavior* of a process describes how these values are computed.

The functional and communication behavior of a process is specified by the behavior graph of the process. There is only one behavior graph per process. The behavior graph of a process is a stylized form of Petri net, called a *Gdbl/b graph model.* The Gdl/b graph describes both the data flow and control flow that together exhibit the desired behavior of the process.

Transitions are annotated with assignments and expressions involving the abstract state elements (variables, both scalar and structured) and ports. [36] The abstract state elements (variables) are declared in the Gdl/s portion of the system description. In addition, a transition is annotated with an enabling (firing) condition which modifies the normal firing rule of Petri nets. A transition fires when all its input places are marked and the enabling conditions is true. Thus, enabling conditions can be used to resolve flow conflicts and to construct conditional branches and loops. An enabling condition must eventually be resolved to a signal within the Gdl/s structure (and routed to the controller.)

The places in the graph represent control states and are annotated with the assertion (again in terms of the values of the abstract state elements or ports) that must hold true in that particular control state. The directed arcs of the graph tie the places and transitions together in the normal fashion.

The firing of a transition represents one control step. For instance, if the abstract design is realized using a central clock, then each control step represents a state transition. Regardless of the style in which the control is realized, a state change occurs and the state assertion must hold at the end of the control step.

The idea of autonomous control center (process) arises from the observation that each process description represents a control flow that regulates the data flow between the abstract state elements of the process (in response to stimulus from the environment) to exhibit a behavior that is independent of the state of any other process in the design. How the control flow is implemented (synchronous or asynchronous, centralized or distributed, etc.) depends on the implementation of the process at the organizational level.

A subnet is a single entry, single exit (SESE) graph that permits the hierarchical decomposition and refinement of behavior. A subnet does not represent an independent control site; that is, a process does not contain subprocesses. Thus, every process in the system represents one

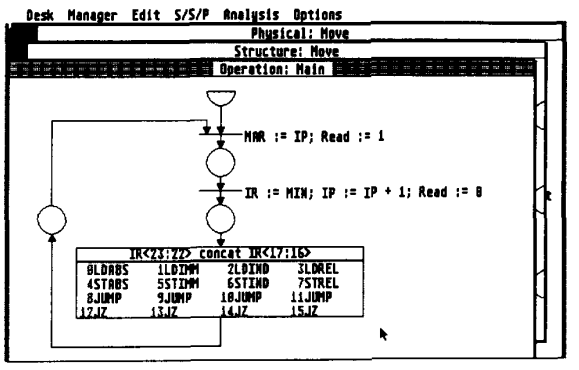

Figure 2: Gdl/b graph.

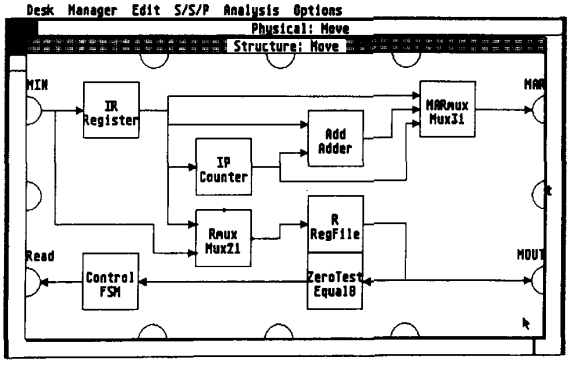

Figure 3: Gdl/s structural diagram.

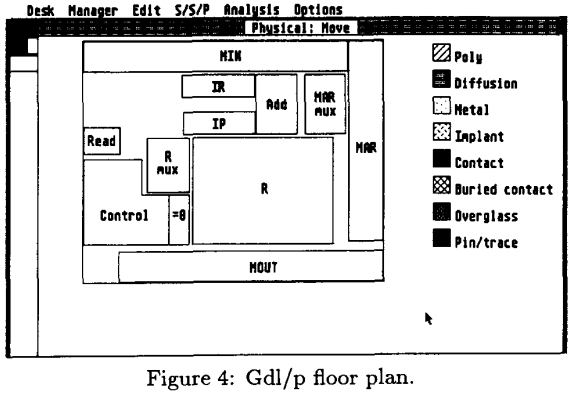

Figure 4: Gdl/p floor plan.

independent activity that communicates with its peers or environment. Processes do not form a hierarchy, but this does not preclude the situation in which the designer views the system as a single process that communicates with one or more subservient peers and so on. The designer may also alternatively view the aggregate behavior of communicating processes as a single process – these are just two equivalent views of the system as the designer goes through the process of refinement. The SESE style assists the flattening of the control graph.

Signal	State0	State1	State2	State3
Req	1	1	0	0
Ack	0	1	1	0

Table 3: Signalling states of a 4 cycle handshake.

Processes communicate with other processes in the system via ports using explicitly defined protocols. Communication behavior is addressed at a very early stage of the design (even at the architectural level) because of our conviction that communication considerations are just as important as functional considerations in the final quality of a design. [2]

The value of a port can represent two kinds of information, *data* or *control*, as specified in the Gdl/s description. Data and control ports may participate in the assignment annotation attached to a transition. An assignment to a control port, however, must be followed by an arc leading to a "synchronization place," which is a place where a token is exchanged with another process. The token will arrive at the other "half" of the synchronization place in the other process where it will enable a transition. The example in Figure 5 illustrates the interlocked exchange of information between two processes using the well-known four cycle handshake.

The number of synchronization places depends on the number of unique synchronizing states that are required by the protocol. Each shared place is annotated with the signal conditions (in terms of values of the ports involved) that have to hold true for that state. The transitions between these shared places are annotated with the changes in signalling conditions (changes in the values of the control ports involved) that have to occur for the next set of signalling conditions to hold true for the next shared place. In Figure 5, for example, four shared synchronization places are required, representing the signalling conditions shown in Table 3.

Once a process is specified, it may be repeatedly instantiated using the facilities of Gdl/s providing convenient support for regularity.

5 Tools.

Agent is the design environment based upon Gdl. Agent uses Gdl as the communication medium between its tools and the designer.

Two prototype tools have been constructed. The CGDP system (which executes on Apollo workstations) permits the creation and modification of Gdl control graphs and datapaths. [6,3] Presently, a floor planner is being added to the system. A rule-based consistency checker (implemented in Prolog) checks datapath covering and generally acts as the analytical portion of a syntax directed editor. CGDP and Gdl/s were also used as the front-end of a symbolic, constraint-based timing verifier. [20] The Agent/ST system (constructed on the Atari ST in Pascal under GEM)

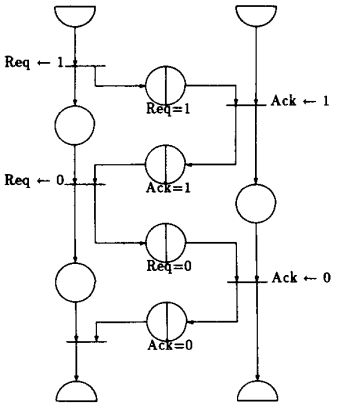

Figure 5: Four cycle handshake.

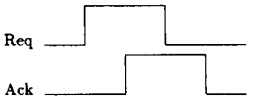

Figure 6: Four cycle signalling.

is also a multiwindow editor for Gdl. [4] Agent/ST can generate a Prolog compatible design description which can be analyzed for datapath covering, syntax and other forms of consistency checking. Code for timing analysis is currently under construction. We intend to meld the features of both system into a third (and hopefully final!) prototype of Agent on SUN workstations using NeWS.

6 Relationship to other work.

The graphical representation of system behavior and structure is not new. Early work in Petri Nets at MIT (Project MAC), the CWRU LOGOS system, UCLA's SARA and RTI ADAS are the most notable. One must also cite CAP/DSDL (Siemens/Dortmund) and CADOC (IMAG) although they will not be discussed here. [16,1,33]

CWRU LOGOS.

In the LOGOS system, a specification is separated into a control graph and a data graph. The data graph defines the storage cells, data operators and information flow paths between those objects. The control graph consists of control cells, operators and flow arcs. The control operators are attached to the data operators such that the control operator may initiate a data graph operation. The control operators (which include conditional, while loop, synchronizing merge, block entry and exit operations) provide interlocked, asynchronous control. [17,18,30,31]

LOGOS has its roots in the parallel program schemata of Karp and Miller. [22] Gdl/b remains closer to the Petri Net model, merely adding the notion of enabling condition to the basic firing rule to resolve conflicts and to construct conditionals, loops, cases, etc. The implementation of a LOGOS control graph in hardware requires a set of interlocked control elements. In the development of the Aids system (which is based on LOGOS), researchers at Heriot-Watt have examined the synchronous, microprogrammed implementation of LOGOS control graphs. [12,13] Gdl/b is amenable to either asynchronous or synchronous implementation as well. LOGOS has more "expressive" control operators while Gdl/b forces the designer to construct loops, etc. from transitions and a more fundamental firing rule.

The LOGOS data graph is a specification of the resources needed to perform the computation. Since LOGOS was intended to support the design of computing systems before an assignment of modules to hardware and software, the resources are not necessarily hardware components and binding to a physical representation may be postponed. Gdl/s identifies specific components or modules, thereby making an early binding to realizable design objects. This binding makes bottom-up speed, space and power analysis possible, but may limit the suitability of Gdl for software design. (This has not yet been explored.)

A comparison may also be drawn with data dependency driven techniques such as the C-MU *value trace*. [25,35] Using data dependencies, the binding to specific components and event schedules can be further postponed, permitting global technology-independent improvements ("optimizations.") This flexibility is suitable for fully automated behavioral synthesis. An interactive design aid (Agent, LOGOS, SARA, etc.) must give the designer the ability to manipulate event schedules and resources making explicit control and resource representations more appropriate to the task.

UCLA SARA.

The System ARchitects Apprentice (SARA) developed at UCLA under the direction of Gerald Estrin is a second generation specification and analysis tool. [10,11] SARA separates the specification into behavioral and structural parts. Structural design is conveyed in a SL1 description which is composed of modules, sockets and interconnections. The structural specification manages the design name space and is a carrier for subsystem behavioral specifications. The UCLA Graph Model of Behavior (GMB) is similar to LOGOS in that behavior is separated into control and data graphs. Tokens flow from node to node (thereby invoking data operations) and may be regulated by AND and OR input logic which provides fork and join operations. A GMB control graph is less cluttered than a Petri net, but can be directly transformed to an equivalent Petri net as shown by Peterson. [29]

The basic GMB has been extended with queueing primitives for performance oriented design. Through simulation, such measurements as average queue waiting time, average queue length, throughput, node utilization, etc. can be taken. Agent and Gdl seek to aid system partitioning without simulation. To perform layout sensitive delay analysis, physical planning information was included in Gdl. The role of Gdl/p is similar to that played by the Bottom-Up Designer (BUD) which supplies low level, technology dependent information to the Design Automation Assistant (DAA.) [26,27]

RTI ADAS.

The Architctural Design and Assessment System (ADAS) from Research Triangle Institute is also an extension of the Karp and Miller marked graph model and uses graphs to model the dataflow through a system. [14,15] Functionality is introduced into the graph using C language annotations. Graphs are timed and each transition takes some time to complete. A graph embodies a queueing model for the system and when it is simulated it provides information about node utilization and latency.

An ADAS system model has two graphs. The software graph is a description of the high level dataflow through software modules. The software graph in turn is mapped onto the hardware graph which represents the computational resources available to execute the system software. Through simulation the designer may determine if the set of resources is sufficient to execute software tasks with adequate performance.

Gdl uses its structural descriptions to define the resources available for computation. Since processes are associated with certain modules and multiple processes are permitted, Gdl can support a style of analysis similar to ADAS. A Gdl simulator is not currently available, however. We intend to perform static analysis on Gdl descriptions to support algorithmic partitioning in advance of detailed, relatively time-consuming simulation.

7 Experience and evaluation.

Experience with Gdl to date has been confined primarily to the organization level of design. We have applied the language to the description of the 6502 microprocessor (including external communication behavior) and several instructional processors (toys.) A student project is underway to take a Gdl description from the organization level all the way to geometry. Our comments are summarized below.

Early decisions.

Our primary complaint is that Gdl encourages a designer to make too many low level design decisions too soon. The visual expression of parallel and sequential execution flow is good since the engineer may address concurrency up front. However, Gdl only provides assignments (register transfers) on behavior graph transitions, thereby restricting the kinds and complexity of behavior that can be expressed. We would like to examine other kinds of annotations (queueing primitives) and specification styles such as functions or logic. [23,24] (The existing Gdl editor does not interpret transition fields and any additions should be easy to accomplish.)

Language problems.

The single entry - single exit restriction on subgraphs will sometimes force the introduction of "dead" transitions into the behavioral description. One case is a two way branch where one of the branches has an empty transition (no effect.) The SESE approach permits each subgraph to be analyzed in isolation for cleanness. ADAS permits multiple entries and exits and may therefore necessitate a flattening of the net before analysis.

Petri nets and related schemata are biased toward the description of asynchronous control systems. Descriptive problems arise when applying nets to synchronous systems. Synchronous control can be modelled by creating a clock subnet that periodically generates a token which is sent to all transitions in the main net. (Obviously this approach is not conflict free and the conventional firing rule must be modified.) The resulting graph will be very cluttered because all transitions must be connected to the central clock process. An alternative is to use timed transitions and to make all transition execution times equal. The third approach, which is used in Gdl, is to leave the interpretation of a graph with respect to timing discipline to the supporting design system. Thus, the timing analysis in Agent must handle both synchronous and asynchronous nets. The additional theory required led to a recent dissertation on reachability in conventional, subset fireable and timed nets. [37]

References

[1] Amblard, P. et al., CADOC: A functional specification and simulation tool, IEEE ICCAD '83, Santa Clara, September 1983.

[2] Bammi, J.R., and Drongowski, P.J., The Gdl/b process model, Computer Engineering & Science (CES), Case Western Reserve University (CWRU), Technical Report CES-86-11, October 1986.

[3] Drongowski, P.J., Ramaswamy, R., and Iyengar, S.R., An experimental assistant to support the construction and analysis of control graph datapath design, CES, CWRU, Technical Report CES-87-03, June 1987.

[4] Drongowski, P.J., An organization level story board for Agent – A VLSI designer's assistant, CES, CWRU, Technical Report CES-87-08, March 1987.

[5] Drongowski, P.J., The Gdl/p physical planning language, CES, CWRU, Technical Report CES-86-14, October 1986.

[6] Drongowski, P.J., A graphical, rule-based assistant for control graph - datapath design, Proc. ICCD'85, IEEE Computer Society, October 1985, pg. 208-211.

[7] Drongowski, P.J., **A Graphical Engineering Aid for VLSI Systems**, Computer Science Series, Harold S. Stone (Series Editor), UMI Research Press, Ann Arbor, Michigan, 1985.

[8] Drongowski, P.J., Representation in CAD: Model and semantics, Proc. of the 1985 ACM Computer Science Conference, March 1985, pg. 130-135.

[9] Drongowski, P.J., System speed, space and power estimation using a higher level design notation, Proc. of the International Conference on Computer Design, IEEE Computer Society, October 1983, pg. 468-471.

[10] Estrin, Gerald, 1977 ACM/IEEE Symposium on Design Automation and Microprocessors, Palo Alto, 1977, pg. 54-59.

[11] Estrin, Gerald, SARA in the design room, 1985 ACM Computer Science Conference, New Orleans, March 1985, pg. 1-12.

[12] Foulk, P.W. and O'Callaghan, J., Formal description of computational structure in Aids, IEE Proceedings, Vol. 127, No. 2, March 1980, pg 55-63.

[13] Foulk, P.W., et al., Aids - An Integrated Design System for Digital Hardware, IEE Proceedings, Vol. 127, No. 2, March 1980, pg. 45-63.

[14] Frank, Geoffrey A., et al., An architecture design and assessment system for software/hardware codesign, 22nd ACM/IEEE Design Automation Conference, Las Vegas, June 1985, pg. 417-424.

[15] Frank, Geoffrey A., et al., An architecture design and assessment system, VLSI Design, August 1985.

[16] Frantz, D. and Rammig, F.J., The impact of an advanced CHDL on VLSI design, IEEE ICCD '83, November 1983, pg. 173-176.

[17] Glaser, E.L., Bradshaw, F.T. and Katske, S.W., LOGOS – An overview, IEEE CompCon '72, September 1972.

[18] Heath, F.G., and Rose, C.W., The case for integrated hardware/software design with CAD implications, IEEE CompCon '72, September 1972.

[19] Hill, F.J., and Navabi, Z., Extending Second Generation AHPL to Accommodate AHPL III, 4th IEEE and ACM International Symposium on Computer Hardware Description Languages, October 1978, pg. 47-53.

[20] Iyengar, S.R., Incremental timing verification in top-down, bottom-up designs using constraint propagation, Ph.D. dissertation, Case Western Reserve University, August 1987.

[21] Iyengar, S.R., Ramaswamy, R. and P.J. Drongowski, The Gdl/s structural design language, CES, CWRU, Technical Report CES-86-13, October 1986.

[22] Karp, R.M., and Miller, R.E., Parallel program schemata, Journal of Computers and System Science, Volume 3, 1969, pg. 147-195.

[23] Mateti, P. and Hunt, F.E., Precision descriptions of software designs: An example, IEEE CompSac '85, 1985, pg. 130-136.

[24] Mateti, P., CaseDL: A design specification language, Technical Report, Computer Engineering and Science, Case Western Reserve University, October 1987.

[25] McFarland, M.C., The value trace: A database for automated digital design, M.S. thesis, Carnegie-Mellon University, December 1978.

[26] McFarland, M.C., Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions, 23rd Design Automation Conference, June 1986, pg. 474-480.

[27] McFarland, M.C. and Kowalski, T.J., Assisting DAA: The use of global analysis in an expert system, 24th Design Automation Conference, June 1987, pg. 482-485.

[28] Peng, Z., Synthesis of VLSI systems with the CAMAD design aid, 23rd IEEE/ACM Design Automation Conference, June 1986, pg. 278-284.

[29] Peterson, J.L., **Petri Net Theory and the Modeling of Systems,** Prentice Hall, Englewood Cliffs, N.J., 1981.

[30] Rose, C.W., Bradshaw, F.T., and Katzke, S.W., The LOGOS representation system, IEEE CompCon '72, September 1972.

[31] Rose, C.W., LOGOS and the software engineer, Fall Joint Computer Conference, IFIPS, 1972, pg. 312-314.

[32] Rose, C.W., Rogers, L.R., and Straubs, R.V., "The N.mPc System Description Facility," 16th ACM/IEEE Design Automation Conference, June 1979.

[33] Saucier, G., et al., ASYL: A rule-based system for controller synthesis, IEEE Transactions on CAD, Vol. CAD-6, No. 6, November 1987, pg. 1088-1097.