An Efficient Multi-View Design Model for Real-Time Interactive Synthesis

Tedd Hadley, Allen C-H. Wu, and Daniel D. Gajski

Technical Report 92-35
April 20, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

hadley@ics.uci.edu

Mystic Tide Memory
Deformation
by Kawaguchi
(1962 - 2019)

Abstract

This report describes an efficient multi-view design model for real-time interactive synthesis of behavioral descriptions into layout data. We present a hybrid data structure which combines all of the design data needed throughout multiple levels of abstraction, including behavior, structure, and floorplan, into a single unified view. We also give a detailed time and space complexity analysis of the proposed design model, showing that it provides fast updating capabilities for incremental design changes but does not require an exorbitant amount of memory space. These features make this design model ideal for user-controlled synthesis systems that support incremental design and re-design tasks. Furthermore, the simplicity of the data structure allows easy implementation, maintenance, and extendibility.

Contents

List of Figures

1 Introduction

Typical high-level synthesis systems take as input a behavioral description and output a register-transfer level design implementing the given behavior. Several common intermediate tasks occur within the synthesis process to convert the abstract behavior to structure. These are transformations, scheduling, module selection, and allocation. A survey of high-level synthesis tasks is given in [McPC88] and [GDWL92].

In most high-level synthesis systems, each intermediate task is performed automatically, without any intervention by the user. In addition, each tool or procedure performing a task typically does not provide any indication as to what steps were taken or decisions performed to convert the input description to the output result. From the viewpoint of an experienced human designer, such "black-box" systems are unacceptable due to the inaccessible workings of the tools and algorithms involved. Hence, in recent literature there has been increasing interest in interactive synthesis systems and methodologies [CPTR89, WRJF92, BuEl89, Jenn91, Knap89].

An ideal system for interactive synthesis must allow the user to interrupt or interfere with the design process at any point. It must provide support for iterative, re-designing tasks such as rescheduling and rebinding. Furthermore, physical estimates and quality measures must be made readily and rapidly available in order to aid the user in the decision-making process. To support user tasks, an interactive system must provide graphical views into the design, visually describing the potentially complex relationships between behavior, structure, and floorplan, and allowing the user to immediately perceive the consequences of his or her decisions. To provide for these requirements, an interactive system needs an underlying data model which can combine all of the design data produced at multiple levels of abstraction into a single unified view. This data model must also provide a fast updating capability to support real-time incremental changes.

Previous work in unified design models has attempted to link the various design domains in order to achieve a single coherent data-structure. DDS [KnPa85] divides

design information into four subspaces having no implicit relationships with each other: behavior, structure, physical, and timing/control. These subspaces are then linked with explicit bindings. Another scheme is used by CORAL II [BlTK88], which maintains fine grained links between the behavioral specification and generated structure by storing link information externally for each design stage. Both of the above models provide a unified representation for design information in different design domains. However, the use of bindings which exist outside the design subspaces makes it difficult to implement the real-time response required for an interactive system.

In this report, we present an efficient design model for real-time interactive synthesis. Our design model uses a hybrid data structure to represent behavior, structure, and floorplan. This data structure is easily transformed into views (including behavior, datapath, control-unit, and floorplan) for interactive synthesis tasks. We also give a detailed space and time complexity analysis showing that this design model provides fast updating capabilities for incremental design changes. These features make the design model ideal for user-controlled synthesis systems that support incremental design and re-design tasks.

The rest of the report is organized as follows. Section 2 describes the design views needed for interactive synthesis. Section 3 presents the proposed design model. Section 4 gives a detailed time and space complexity analysis of the model. Finally, Section 5 summarizes our approach.

2 Design Views for Interactive Synthesis

There are several tasks in interactive synthesis that are unique from automatic synthesis. These tasks may be categorized as partial design, incremental design, and re-design.

Each task in interactive synthesis may require a combination of design views. This section describes the tasks in interactive synthesis and lists the design views needed for each task.

2.1 Interactive Tasks

Typically, interactive synthesis can be applied in one of two ways. In the first, the interactive-synthesis tool incorporates an initially complete or partial design (either generated automatically using synthesis tools or developed by the user), and then allows the user to improve it by re-scheduling, re-binding, module selection, structural partitioning, and floorplanning, guided by the information provided by the design views. In the second application, the user performs module selection, scheduling, binding, structural partitioning, and floorplanning incrementally according to the hints and quality measures provided by the views.

The basic interactive synthesis tasks are: (1) *module selection*, (2) *partial scheduling*, (3) *partial binding*, (4) *structural partitioning*, (5) *incremental scheduling and binding*, (6) *re-binding*, (7) *re-scheduling.* and (8) *floorplanning.*

Module selection is done to allocate an initial set of physical components for the target design. The set need not be complete; for example, it should not be necessary for the user to select every register needed in a design. Functionality, styles, and physical information of the modules available from the library should be provided to the user.

Partial scheduling allows the user to specify time-steps at arbitrary locations in the design and assign operations to them.

Partial binding assumes a set of prior selected modules. Given a behavior, the user may wish to bind data-flow operators or variables to physical modules. While such a binding may imply a crude schedule, it should be possible for the user to do partial bindings given only a behavior and a set of allocated modules.

Structural partitioning allows the user to group highly connected modules into blocks.

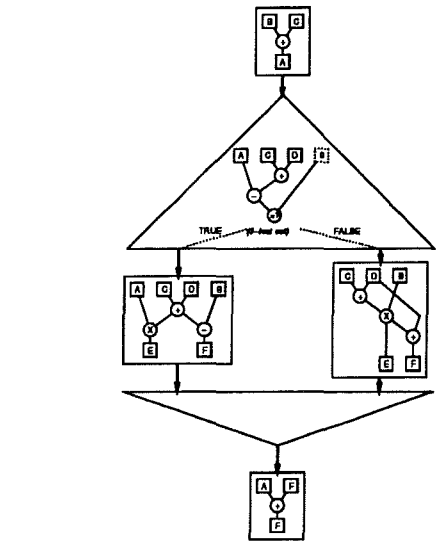
Incremental scheduling and binding is an iterative process where at each step the user adds a time-step to the behavior, assigns operations from the behavior to the time-step, and specifies modules to implement the data-flow operations.

Re-binding allows the user to explore alternate bindings by assigning operators or

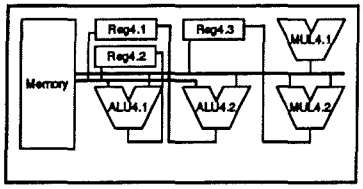
variables to different modules, or swapping two operator or variable assignments among two modules.

Re-scheduling allows the user to examine the results of moving operations to different time-steps.

Floorplanning tasks are interactive placement and routing, and possibly port repositioning and module rotation.



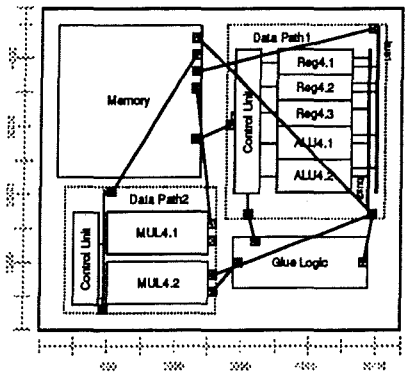
Control and Data-flow



Structure

State	Cond	Cond Value	Control Output	Next State
T1			ALU.CADD = 1	T2
T2			ALU.CADD = 1	T3
T3			ALU.CSUB = 1	T4
T4			COMP.CEQ = 1	T5
			R5.CLOAD = 1	
T5	R5.O[0]	'1'	ALU.CADD = 1	T6
		'0'	ALU.CADD = 1	T6
T6	R5.O[0]	'1'	ALU.CSUB = 1	T8
			MULT.CMULT = 1	
		'0'	MULT.CMULT = 1	T7
T7			ALU.CADD = 1	T8
T8			ALU.CADD = 1	

Time Steps



Floorplan

Figure 1: Views for interactive synthesis

2.2 Design Views

From the previous section it is clear that multiple views may be needed for each interactive synthesis task. The primary design views are (1) *control/data flow behavior*, (2) *time steps*, (3) *structure*, and (4) *physical*.

Figure 1 shows what the various views could look like from a user's perspective. Control and data-flow views capture the abstract behavior of the design. Time steps show the control values assigned in each state. The structural view shows the connectivity of the design and the current component set. The physical view shows the floorplan of the design in terms of blocks, data-paths, control units, modules, buses, and ports.

It is also necessary that all of the views required for a particular task coordinate closely together such that changes in one view will be reflected in others. Furthermore, all the design information available should be accessible from any one design hierarchy. For instance, Figure 2 shows an incremental scheduling and binding example. After

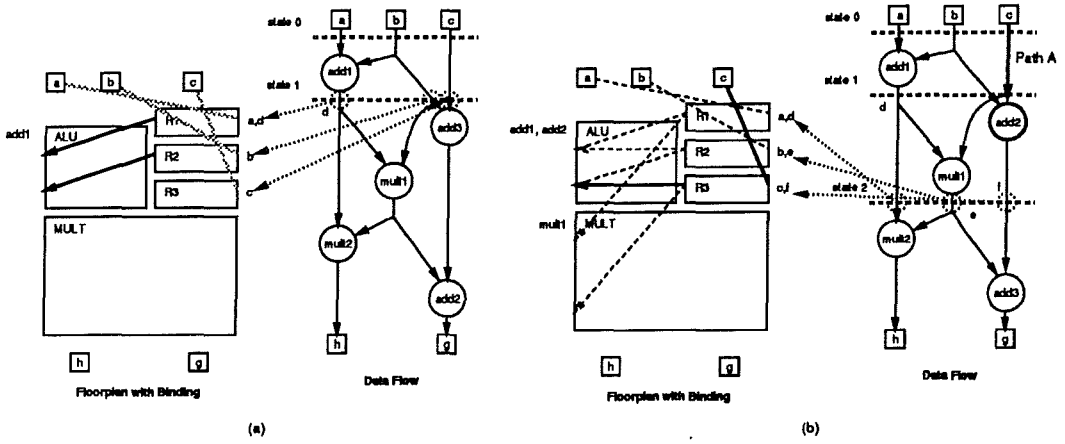


Figure 2: Incremental scheduling and binding example

assigning the three ports (A,B,and C) into separate registers, the user assigns a timestep for the add operation and binds it to the ALU. In the second step (Figure 2(a)), the user assigns a timestep for the add operation and binds it to the ALU. The result of the add operation must be bound to a register. The structural and floorplan views provide information about which variable is assigned to which register. By examining these

views, the user can determine that the port variable *a* is no longer used in the design. Since it is bound to register R1, that register is free to accept a new value. Moreover, if the user wishes to know the corresponding layout connection of a data-flow path (for instance *Path A* in Figure 2(b)), the floorplan view can provide that information, as illustrated in Figure 2(b).

3 Design Model

To support the interactive synthesis tasks described in the previous section and to provide a highly coordinated view environment, an efficient multi-view design model is needed. This section describes our design model. We first describe the data structure, and then show how the design behavior, structure, and floorplan views are represented.

3.1 Data Structure

In our design model, we use a hybrid data structure to represent behavior, structure, and floorplan (Figure 3). For simplicity, the chip level will be described as a single node representing the goal design and forming the root of the tree (although the data structure and subsequent analyses can be trivially extended to include a fourth level of hierarchy: the MCM level). All design hierarchies are represented using a single hierarchical graph data structure. The relationships between different design hierarchies are initiated by a grouping process. For instance, a structural node will contain a set of behavioral nodes and edges, while a floorplan-module node will contain a set of structural nodes, as indicated in Figure 3. In order to combine the nodes in different hierarchies, each node in one hierarchy may be a parent or child of nodes in another hierarchy. This means that a node may have multiple parents, one for each possible hierarchy.

The data structure is composed of only two entities: the supernode and the superedge (Figure 4). A supernode has one or more input and output ports. Each port may connect to a superedge. Each superedge connects two ports belonging to two supernodes,

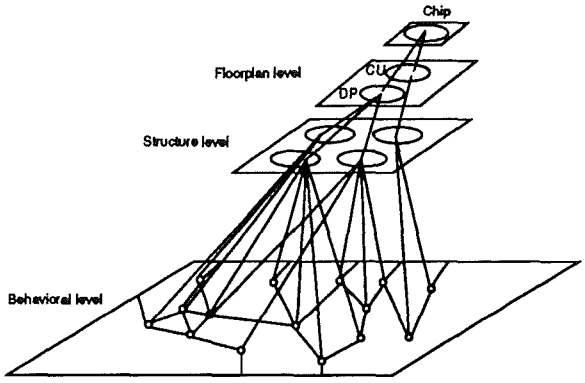


Figure 3: The hybrid data structure.

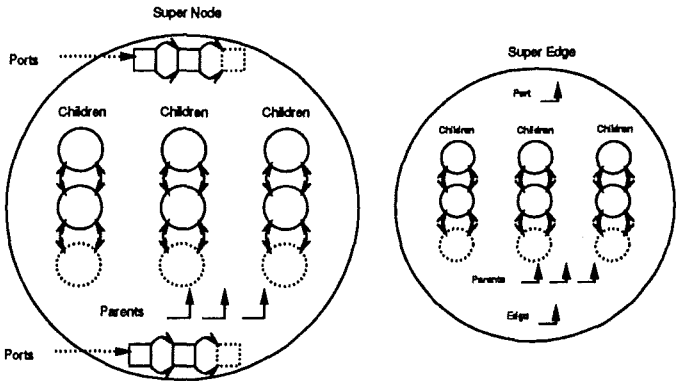


Figure 4: The two data structure entities.

respectively, in the same hierarchy. A supernode may have zero or more sets of children. Each set contains one or more supernode or superedge children belonging to another hierarchy. Each supernode or superedge has a unique set of attributes that establish the context in which it is used.

Having only two major data types in the data structure considerably simplifies the implementation. Only one set of routines needs to be written and debugged to support all possible operations on the connectivity of the data structure. In addition, the simplicity of the data structure allows new design views to be added with a minimum of effort. Typical high-level synthesis system implementations use unique data structures for each design view. This requires a separate implementation for each data structure, which increases the amount of time spent debugging applications and decreases the overall reliability of the system.

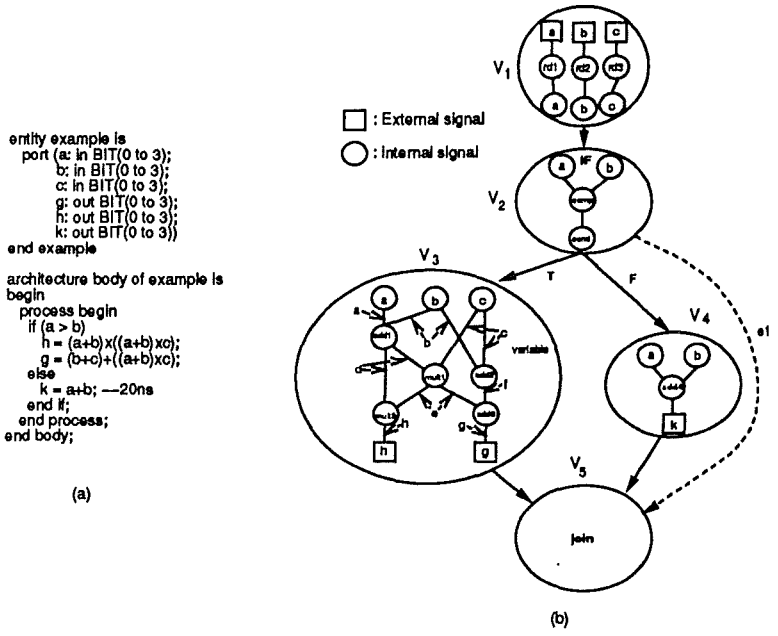


Figure 5: The behavioral model: (a) a VHDL description, (b) the CDFG model.

3.2 Behavioral Model

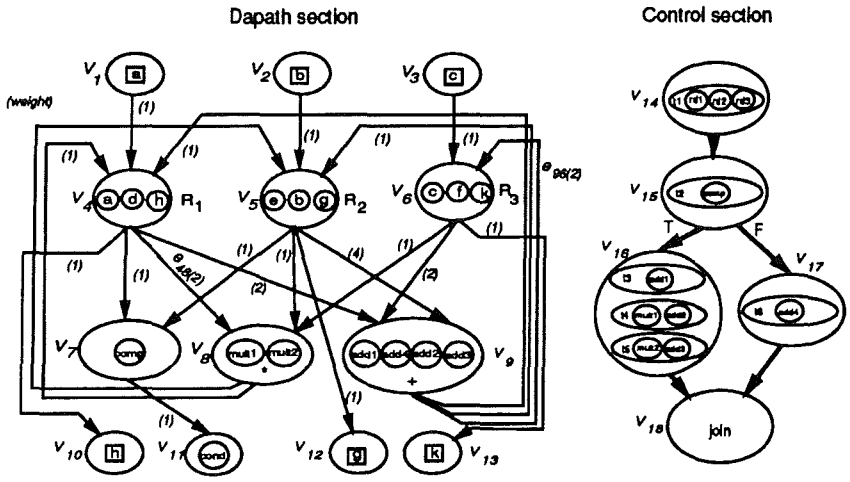
We use a control/data-flow graph to represent behavior. Control flow is represented by supernodes whose edges indicate control transitions. Control-flow supernodes may be conditionals, loops, or data-flow blocks. Data-flow supernodes may be operators, ports, or variables. Edges between data-flow supernodes indicate data flow. In addition, an explicit timing edge between data-flow supernodes indicates a delay constraint.

Figure 5(b) shows a CDFG that corresponds to the VHDL program in Figure 5(a). It consists of five control-supernodes of which V_2 and V_5 are conditionals, and V_1, V_3 and V_4 are data-flow blocks. Edge e_1 indicates a delay constraint of 20ns over data-flow supernode V_4.

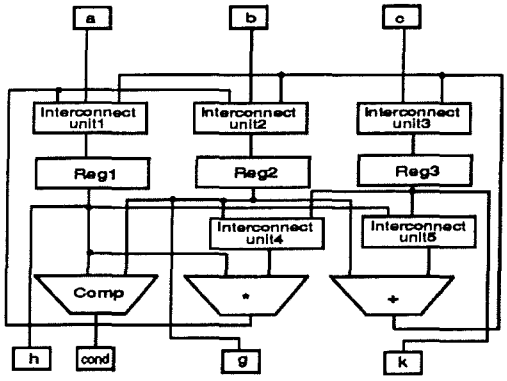
3.3 Structure Model

Structure is represented by supernodes corresponding to ports, datapaths, modules, components, or control units. Supernodes that represent storage components have data-flow superedge children corresponding to storage requirements between time-steps. Supernodes that represent functional components have data-flow supernode children. Superedges between structural supernodes represent physical connections. The children of superedges are data-flow edges between data-flow supernodes. Physical connection superedges may themselves be grouped into interconnect supernodes corresponding to buses or multiplexors. Control-unit supernodes contain data-flow supernodes corresponding to time steps. Each time-step supernode contains some number of data-flow operator children corresponding to execution in that time interval.

Using the example in Figure 5, given an adder, a multiplier and a comparator, a scheduler partitions this CDFG into 6 time steps. The allocator assigns three registers for storage such that variables $\{a, d, h\}$, $\{e, b, g\}$ and $\{c, f, k\}$ are stored in registers R_1, R_2 and R_3, respectively. Figure 6(a) shows the structural model of which V_7, V_8, and V_9 are component supernodes that contain the sets of data-flow supernode children $\{comp\}$, $\{mult1, mult2\}$, and $\{add1, add2, add3, add4\}$, respectively. V_4, V_5, and V_6 are storage



(a)



(b)

Present state	Control ouput														Status	Next step
	Register			Interconnect												
Step	R1	R2	R3	Unit1			Unit2			Unit3		Unit4		Unit5			
	load	load	load	s1	s2	s3	s1	s2	s3	s1	s2	s1	s2	s1	s2	cond.	
t1	1	1	1	0	1	0	0	1	0	1	0	0	0	0	0	0	t2
t2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0/1	t6/t3
t3	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	t4
t4	0	1	1	0	0	0	1	0	0	0	1	0	1	0	1	0	t5
t5	1	1	0	1	0	0	0	0	1	0	0	1	0	0	1	0	stop
t6	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	stop

(c)

Figure 6: The structure model: (a) the structural supernode formation, (b) the structural netlist and control state table.

supernodes that contain the sets of data-flow superedge children $\{a, d, h\}$, $\{e, b, g\}$ and $\{c, f, k\}$. V_1, V_2, V_3, V_{10}, V_{12} and V_{13} are port supernodes. Finally, V_{14}, V_{15}, V_{16}, V_{17} and V_{18} are control-unit supernodes. Each control-unit supernode consists of a set of time steps, and each time step contains a set of data-flow supernode children that can be executed in that time step. For example, the data-flow supernode *comp* in V_2 is assigned to the time step t_2. Thus, the control supernode V_{15} consists of a time step t_2 containing the data-flow supernode *comp*.

We now describe how the structural model can be mapped onto a structural netlist. We first describe the datapath formation. Figure 6(b) shows the resulting datapath structural netlist. The storage supernodes V_4, V_5 and V_6 are mapped to $Reg1$, $Reg2$ and $Reg3$, respectively. The component supernodes V_7, V_8 and V_9 are mapped to a comparator, a multiplier and an adder. Each superedge denotes a physical connection between two supernodes. When a supernode has more than one incoming superedge entering one of its inputs, a selector (e.g., a multiplexer) is needed to select the data input from different sources. For instance, supernode V_4 ($Reg1$) has 3 incoming superedges; therefore, a 3-input selector (*Interconnect unit*1) is needed on its input. Interconnect units may be represented implicitly, or explicitly by grouping superedges into interconnect supernodes

We form a control-state table from the control section of the structural model, as shown in Figure 6(c). Using a given component library, we can determine the control pins for each component. For example, if we choose a 3-input multiplexer with 3 select-inputs for *Interconnect unit*1, it has three control inputs, $s1, s2, s3$. Present state, status and next state can be derived directly from the structural model. For example, time step *t1* in V_{14} (Figure 6(a)) consists of three read operations ($rd1, rd2, rd3$) to load input data from ports a, b and c to registers R_1, R_2 and R_3. $rd1$ reads input data from port a and stores it into the register R_1 via *Interconnect unit*1 (Figure 6(b)). Therefore, the control outputs for the load input of R_1 and the select input $s1$ of *Interconnect unit*1 are set to one. Since this control supernode (V_{14}) is not a branch node, the next state is the first time step (*t2*) of its successor (V_{15}). On the other hand, for a conditional branch node V_{15}, the next state depends on the conditional status (row *t2* of Figure 6(c)).

3.4 Floorplan Model

Based on the datapath and control-unit formation, we can directly map the structural model onto a chip floorplan. Using Figure 6 as an example, the structural netlist is divided into four parts (Figure 6(a)): *datapath*, *control*, *input ports* and *output ports*. Figure 7(b) shows the chip floorplan. The datapath section is mapped to a floorplan datapath that can be implemented using a bit-sliced stack or standard cells, while the control section is mapped to a control unit that can be implemented using a PLA or standard cells. The area and dimension of the datapath and control unit can be obtained using layout models [KuRa91, WuCG91, Zimm88] or by running target layout tools. Each port supernode is mapped to a chip pin consisting of an I/O pad and a pad driver. Finally, the superedges crossing the boundaries of the datapath, control unit, and input/output ports are mapped to the routing area of the chip.

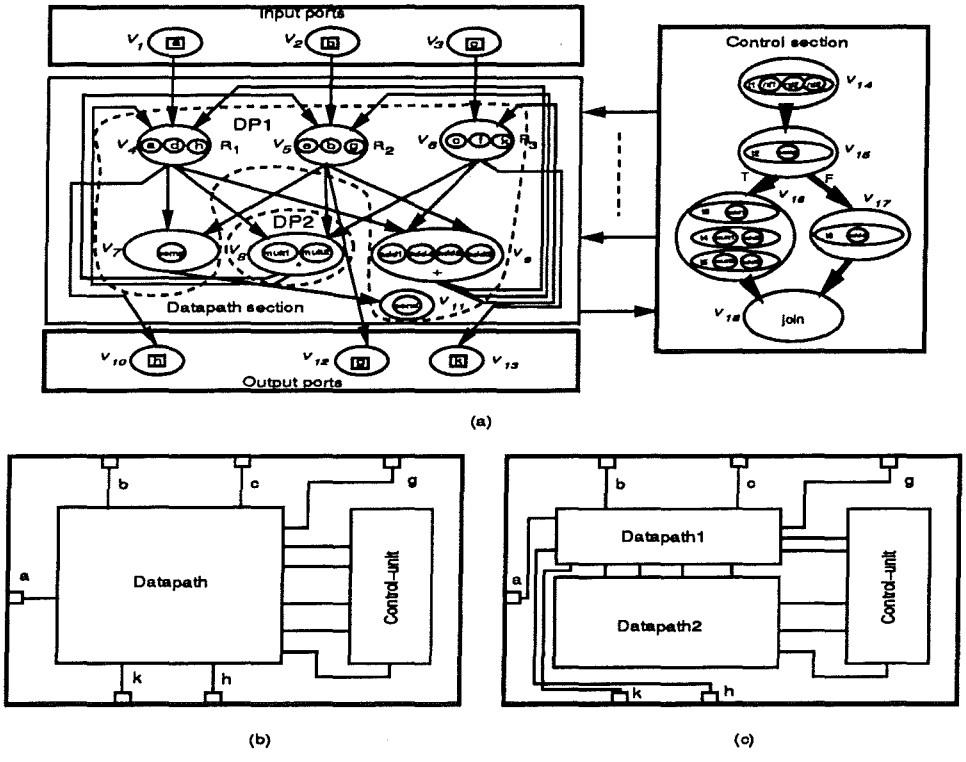


Figure 7: The floorplan model: (a) the structural model, (b) the floorplan with a datapath, (c) the floorplan with multiple datapaths.

The datapath section of the structural model can be further partitioned into multiple

datapaths. For example, the dotted lines in Figure 7(a) show that the datapath section can be divided into two smaller datapaths, *DP1* and *DP2*. In this case, each datapath is mapped to a separate datapath on the floorplan, e.g., *DP1* is mapped to *Datapath1* and *DP2* is mapped to *Datapath2*, as shown in Figure 7(c).

3.5 Design Model for a Linear Topology Architecture

In the previous sections, we described the design model for a random-topology architecture (i.e., point-to-point). In this section we will describe the design model for a multi-bus datapath with a two-phase clock.

We use the example in Figure 5 to describe the structural and floorplan models. The unit/storage binding result is the same as described in Figure 6 except that registers are grouped into a single register file. The register file consists of four ports (one read-only, one write-only and two read/write ports) three register cells, as shown in Figure 8(a).

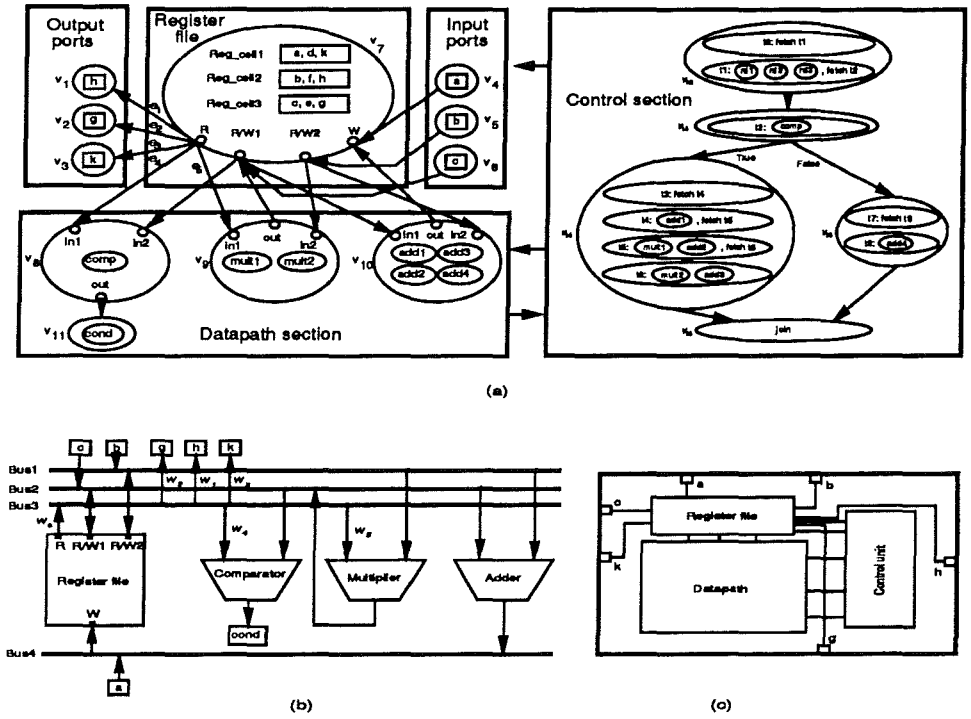


Figure 8: The design model: (a) structural model, (b) datapth formation, (c) floorplan model.

Figure 8(b) shows the datapath structural netlist. The supernode V_7 has been mapped onto a register file that consists of three register cells, *Reg_cell*1, *Reg_cell*2 and *Reg_cell*3 and four ports, R, R/W_1, R/W_2 and W. The component supernodes V_8, V_9 and V_{10} have been mapped to a comparator, a multiplier and an adder. The input-port supernodes V_4, V_5 and V_6 are mapped to input ports a, b and c, and the output-port supernodes V_1, V_2, V_3 and V_{11} are mapped to output ports h, g, k and *cond*. Using the multi-bus architecture, all superedges that share the same connection are grouped into a bus. For instance, superedges e_1, e_2, e_4, e_5 and e_5 are mapped to wires w_1, w_2, w_3, w_4 and w_5 which are connected to the $Bus3$ sharing the common source (R port of the register file) via the wire w_s. Finally, a control-state table can be derived directly from the structural model in the same manner described in the previous section.

The final structural netlist is divided into five parts: *datapath* , *control*, *register file*, *input ports* and *output ports*. The corresponding floorplan model is shown in Figure 8(c).

4 Complexity Analysis of Basic Operations

To support the interactive synthesis tasks described in Section 2, six basic operations are required: (1) initial setup, (2) adding an edge or link, (3) deleting an edge or link, (4) adding a node, (5) deleting a node, and (6) path searching.

In the following sections, we describe the space requirement for the design model and the time complexities of each operation.

4.1 Space Requirement

Let n_1 and n_2 be the number of data-flow nodes and edges, and let n_3 and n_4 be the number of control-flow nodes and edges in the CDFG. Let m_1 be the number of components in the structural hierarchy and m_2 be the number of modules in the floorplan hierarchy.

We will analyze the space requirement in two parts: (1) the space complexity based

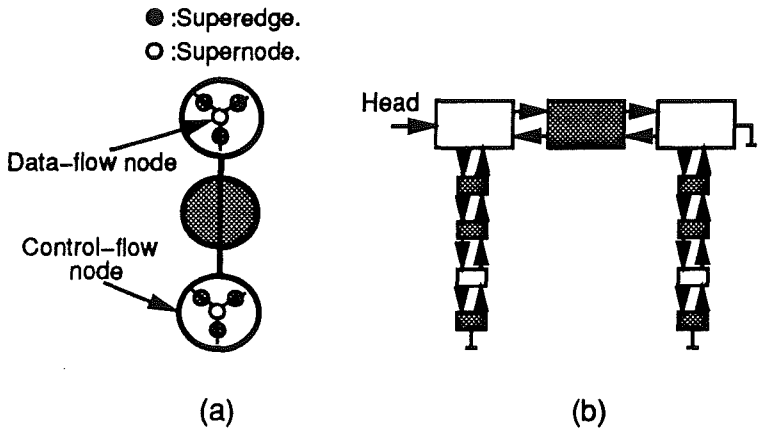


Figure 9: A linked list example.

on the number of entities (i.e., supernodes and superedges) used in our design model and (2) the space complexity based on the number of links between the entities in the same and different hierarchies. Our data structure maintains a linked list that contains all the entities in the same hierarchy. In addition, each entity contains a linked list of its parents and children. For instance, Figure 9(a) shows a CDFG example that contains two control-flow supernodes and one control-flow superedge. Each control-flow supernode contains one data-flow supernode and three superedges. Figure 9(b) shows the corresponding linked list.

In the CDFG (i.e., leaf-level), one supernode is needed to represent a data-flow node as well as a control-flow node. Also, one superedge is needed to represent a data-flow edge as well as a control-flow edge. Therefore, the space complexity of the CDFG is $O(n_1 + n_2 + n_3 + n_4)$.

In the structural level, one supernode is needed to represent a component. Since interconnect units are represented implicitly in our structural model, components only include functional and storage units. Furthermore, the control section of the structural netlist is identical to the control-flow graph; hence, no additional entities are needed in the structural level to represent the control section. Since in the worst case each behavioral superedge is mapped onto one structural superedge, the space complexity of the structural model is $O(m_1 + n_2)$. Similarly, in the floorplan level, one supernode

is needed to represent a module but the connections between modules are represented implicitly. Therefore, the space complexity of the floorplan model is $O(m_2 + n_2)$.

We now analyze the space complexity of links between nodes in the same hierarchy. To maintain a linked list for all the entities in the same hierarchy, we need $n_3 + n_4$ links for the behavioral hierarchy, m_1 links for the structural hierarchy, and m_2 links for the flooplan hierarchy. Therefore, the space complexity is $O(n_3 + n_4 + m_1 + m_2)$.

The links between hierarchies may have a many-to-many relationship. The links between the data-flow and control-flow have a space complexity of $O((n_1 + n_2) \times n_3)$. Similarly, the space complexity of the links between the CDFG and structural level is $O(m_1 \times (n_1 + n_2 + n_3 + n_4))$. Finally, the space complexities of the links between structural/floorplan and floorplan/chip levels are $O(m_1 \times m_2)$ and $O(m_2)$, respectively. The above complexities considers the worst-case scenario. In practice, it is reasonable to assume that each data-flow node/edge will map onto a single component and each component will map onto only one module. Therefore, control-flow supernodes contain at most n_1+n_2 children, structural supernodes contain at most $n_1+n_2+n_3+n_4$ children, and floorplan supernodes contain at most m_1 children. Thus, $O(2(n_1 + n_2) + n_3 + n_4 + m_1 + m_2)$ space is needed to link the entities between different hierarchies.

4.2 Time Complexity Analysis

This section discusses the time complexity of the six basic operations.

1. *Initial setup.* Given a CDFG and an initially complete or partial design with a schedule, a module set, and structural bindings, this step constructs the links between the different hierarchies. To map the CDFG onto the structure requires a search through the structural component linked list for each CDFG node and edge. Therefore, the time complexity is $O(m_1 \times (n_1 + n_2 + n_3 + n_4))$. Similarly, it takes $O(m_1 \times m_2)$ time to map the structural components onto the modules and and $O(m_2)$ time map the modules onto the chip.

2. *Adding an edge or link.* Adding a link between two entities of different hierarchies or an edge between two entities in the same hierarchy requires a search through the entity's linked list. Thus, it takes $O(n_1 + n_2 + n_3 + n_4 + m_1)$ time to insert a link between the CDFG and structural levels, or to insert an edge in the CDFG. Similarly, it takes $O(m_1 + m_2)$ time to insert a link between the structural and floorplan levels, or to insert an edge in the structural level. Finally, it takes $O(m_2)$ time to insert a link between the floorplan and chip levels, or to insert an edge in the floorplan level.

3. *Deleting an edge or link.* Deleting a link consists of two steps: locating the node and locating the edge or link. It takes $O(n_1+n_2+n_3+n_4)$, $O(m_1)$, and $O(m_2)$ time to locate a node in the CDFG, structural, and floorplan hierarchies, respectively, and the same time complexity to locate the edge or link.

4. *Adding a node.* Since each hierarchy maintains a linked list of all the entities, adding a node to that hierarchy takes constant time.

5. *Deleting a node.* Deleting a node requires a search through the entity's linked list. Thus, it takes $O(m_1)$ and $O(m_2)$ times to delete a node in the structural and floorplan hierarchies, respectively. If the child links of the deleted node are not empty, it takes constant time to reassign the node and parent hierarchy links.

6. *Path search.* The main feature of this operation is to identify the data-flow path and its corresponding layout orientation path. To locate a path in the CDFG takes $O(n_1 + n_2 + n_3 + n_4)$ time complexity. However, only constant time is needed to trace the corresponding path in the structural and floorplan hierarchies.

From the above analysis, we observe that the worst time complexity among the operations is proportional to the number of nodes and edges in the CDFG. However, for a large design the number of nodes and edges of the CDFG can be considerable. Therefore we use a two-level hierarchy CDFG (Figure 9) which reduces the average search time substantially by permitting the identification of the control-flow node first followed by the data-flow node. For example, consider a CDFG containing 1,000 data-flow nodes

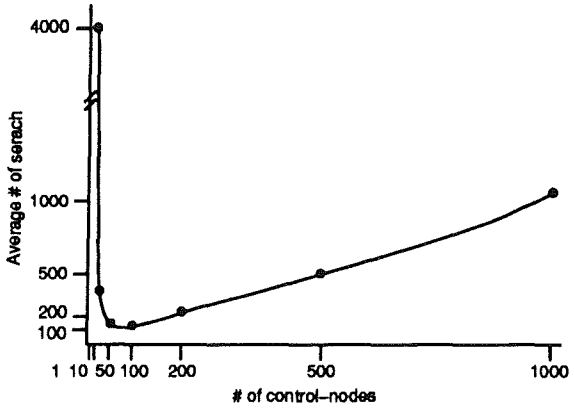


Figure 10: The hierarchical search time complexity example.

and 3,000 data-flow edges. Figure 10 shows the relationship between the average search time and the number of the control-flow nodes. If the CDFG has 100 control-flow nodes in which each node contains in average of 40 data-flow entities, the worst-case search time will be reduced from 4,000 to 140 by first identifying the control-flow node and then the data-flow node.

5 Conclusion

We have presented an efficient multi-view design model for real-time interactive synthesis supporting interactive tasks at the behavioral, structural, and floorplan levels. The hybrid data structure implementing our design model combines all of the design data generated in the synthesis process into a single unified view. For incremental design changes, the space and time complexity analyses shows that updating the design model takes either linear or constant time without excessive memory overhead. These features make the design model ideal for user-controlled synthesis systems which require fast updating capabilities for incremental design changes. Another important feature of the design model is the simplicity of the data structure which allows easy implementation, maintenance, and upgrading capabilities.

We have implemented an interactive synthesis system prototype on top of the design

model proposed which currently supports a subset of the described interactive tasks. The interactive graphical displays are implemented using OSF/Motif and the X11 Window System.

References

[BlTK88] R.L. Blackburn, D.E. Thomas, P.M. Koenig, "CORAL II: Linking Behavior and Structure in an IC Design System," *in Proc. 25th DAC*, pp. 529–535, 1988.

[BuEl89] O. A. Buset and M. I. Elmasry, "ACE: A Hierarchical Graphical Interface for Architectural Synthesis," *in Proc. 26th DAC*, pp. 537–542, 1989.

[CPTR89] C.M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications", *in Proc. ICCD–89*, pp. 432–435, 1989.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Acedemic Publishers, 1992.

[Jenn91] G. Jennings, "GRTL – a Grpahical Platform for Pipelined System Design", *in Proc. EDAC-91*, pp. 424–428, 1991.

[KnPa85] D.W. Knapp and A.C. Parker, "A Unified Representation for Design Information," *in Proc. of the 7th CHDL–85*, pp. 337–353, 1985.

[Knap89] D.W. Knapp, "An Interactive Tool for Register-Level Structure Optimization," *in Proc. 26th DAC*, pp. 598–601, 1989.

[KuRa91] F. J. Kurdahi and C. Ramachandran, "LAST: A Layout Area and Shape Function esTimator for High Level Applications," *in Proc. EDAC-91*, pp. 351-355, 1991.

[McPC88] M.C. McFarland, A.C. Parker, R. Camposano, "Tutorial on High-Level Synthesis," *in Proc. 25th DAC*, pp. 330-336, 1988.

[WRJF92] R.A. Walker, S. Ramabadran, R. Joshi, S. Flatland, "Increasing User Interaction During High-Level Synthesis", *in Proc. Micro–92.*

[WuCG91] A. C-H Wu, V. Chaiyakul and D. D. Gajski, " Layout Area Models for High-Level Synthesis," *in Proc. ICCAD*, 1991.

[Zimm88] G. Zimmermann, "A new Area and Shape Function Estimation Technique for VLSI Layouts," *in Proc. 25th DAC*, pp. 60-65, 1988.