Design Methodology Management Using Graph Grammars

Reid Baldwin and Moon Jung Chung [*]
Department of Computer Science
Michigan State University
{baldwin,chung}@cps.msu.edu

Abstract

In this paper, we present a design methodology management system, which assists designers in selecting a suitable design process and invoking the selected sequence of tools on the correct versions of design data. We introduce a formal graph representation of design methodologies in which nodes represent either tasks or design data. Using a graph grammar, nodes representing abstract tasks are replaced by graphs of less abstract tasks and intermediate specifications. Graph grammars enable us to concisely and flexibly describe a large class of methodologies.

Often there are several alternative methodologies or tools available for some subtasks, leading to different results. Our system utilizes automated control agents, in combination with user interaction, to select among methodologies and invoke tools. Multiple alternatives can easily be explored simultaneously.

1 Introduction

Design methodology management is the selection and execution of an appropriate sequence of tools to produce a design description from available specifications. This can be a daunting task due to incompatible assumptions and data formats among tools. To support a higher degree of automation, *CAD frameworks* must be able to select and execute tools automatically for frequently repeated tasks, enabling designers to concentrate on higher level decisions.

The simplest form of assistance is monitoring designers' actions. In [8], Di Janni describes a *Monitor* for CAD tools which models fixed methodologies using extended petri nets. The VOV system [2] records the sequence as the designer executes tools. When an input file is modified, these systems help the user keep data consistent by invalidating output files or by repeating previous tool executions.

There are several systems which automatically determine what tools to execute. The Design Planning Engine of the ADAM system [9, 10] produces a *plan graph* using a forward chaining approach. Acceptable methodologies are specified by listing pre-conditions and post-conditions for each tool in a lisp-like language. Estimation programs are used to guide the chaining. Ulysses [1] and Cadweld [4] are blackboard systems used to control design processes. A knowledge source, which encapsulates each tool, views the information on the blackboard and determines when the tool would be appropriate. Minerva [7] and the OCT task manager [3] use hierarchical strategies for planning the design process. Hierarchical planning strategies take advantage of knowledge about how to perform abstract tasks which involve several subtasks.

Nelsis [12] provides a graph formalism, called *flowmaps*, for representing sets of methodologies. Each *functional unit* (task) in a flowmap has input and output ports indicating the required data types. Flowmaps have a hierarchical structure in which some functional units correspond to *activities* (atomic operations) and others correspond to more detailed flowmaps. Some flowmaps are flagged to indicate that they should be executed automatically as soon as the input data is available. When tasks are repeated, previous outputs and data derived from them may be invalidated. Flowmaps may offer alternative methodologies, but users must statically specify a preference or choose among them at run time. If two functional units don't have any outputs, flowmaps do not indicate whether they are alternatives or must both be completed. Nelsis does not provide any mechanism to pursue multiple versions simultaneously.

We introduce a formalism called *process flow graphs* to represent individual methodologies. A type of graph grammar called a *design process grammar* is used to document what methodologies are available in a framework. When tasks are repeated, there are multiple versions of its outputs. We use a formalism called a *versioned flow graph* to represent this situation and define exactly when versions of specifications are *compatible*. We believe that our formalisms are more natural than the flowmaps of Nelsis. Alternative methodologies are very explicit, making it easier to indicate which methodologies should be chosen.

We have developed a framework which utilizes design process grammars to assist designers in planning and executing design activities. In our framework, designers build a process flow graph interactively by applying productions from a design process grammar. A set of *manager programs* utilize encapsulated knowledge to guide the designer. Designers may optionally give the manager programs direct control over more detailed decisions, enabling the designers to concentrate on higher level decision making. Our framework is also capable of utilizing idle resources by investigating several alternatives in parallel.

In the next section, we define and discuss our formalisms for describing and manipulating design methodologies. We identify sufficient conditions to guarantee that a process flow graph can be successfully generated. In the following section, we extend these concepts to support the simultaneous exploration of several alternatives. Finally, we describe the architecture of our framework and its implementation.

[*]This work supported by the Air Force Wright Laboratory under contract F33615-92-C-1029

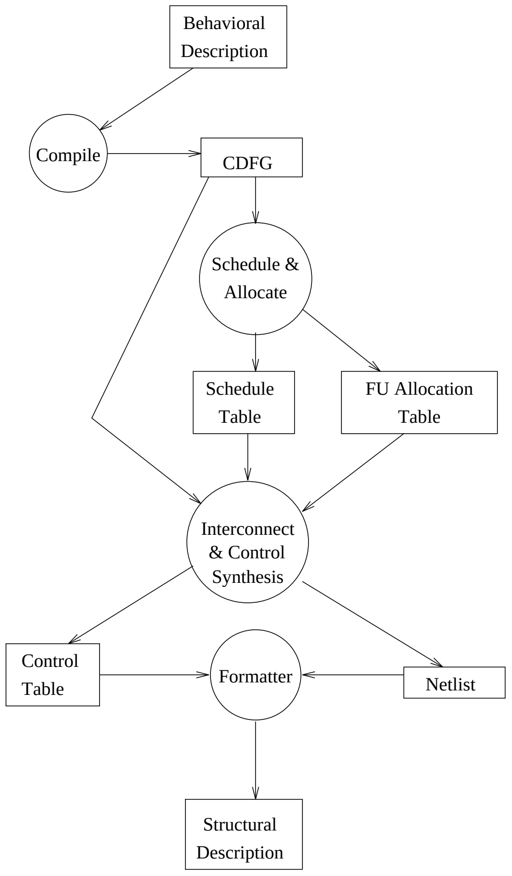

Figure 1: A Sample Process Flow Graph

2 Specifying Design Methodologies

2.1 Process Flow Graphs

Process flow graphs describe the information flow of a design process. Formally, a process flow graph is a bipartite acyclic directed graph of the form $G = (T, S, E)$, where

T is the set of task nodes. Each task node is labeled with a task description. (We draw task nodes using circles.)

S is the set of specification nodes. Each specification node is labeled with a specification type. (We draw specification nodes using rectangles.)

E is the set of edges indicating which specifications are used and produced by each task. Each specification must have at most one incoming edge. Specification with no incoming edges are assumed to be inputs of the design exercise.

Figure 1 shows a process flow graph that describes a possible high level VLSI design process, in which a behavioral description is transformed into a structural description. Unlike Nelsis, which allows cyclic graphs for iterative processes [12], we do not allow cycles. We will discuss iterative processes in section 4.

To address data format incompatibilities, the various specification types form a class hierarchy, where each child is a specialization of the parent. There may be several incompatible children. For example, `VHDL Behavioral Description` is a child of `Behavioral Description`, but is not interchangeable with `Hardware C Behavioral Description`.

Task nodes can be either terminal or non-terminal. A terminal task node represents a run of an application program, which is commonly called a *tool invocation*. We draw terminal task nodes with double circles. Non-terminal task nodes represent abstract tasks, which could potentially be done with several different tools or combinations of tools. Process flow graphs can describe design processes to varying levels of detail. A graph containing many non-terminal nodes indicates roughly what should be done and what information is desired without describing exactly which tools should be used. Conversely, a graph in which all nodes are terminal completely describes a design process.

We use the following notation:

$In(N)$ is the set of input nodes of node N:
$In(N) = \{M \in T \cup S | (M, N) \in E\}$.

$Out(N)$ is the set of output nodes of node N:
$Out(N) = \{M \in T \cup S | (N, M) \in E\}$.

$T(G), S(G), E(G)$ are the sets of task nodes, specification nodes, and edges of graph G, respectively.

$I(G)$ is the set of input specifications of graph G:
$I(G) = \{N \in S(G) | In(N) = \emptyset\}$.

2.2 Design Process Grammars

Graph grammars provide a convenient means for transforming process flow graphs into progressively more detailed process flow graphs. The user specifies the overall objectives by supplying the initial graph, which indicates what input specifications are available, what output specifications are desired, and what abstract tasks should be performed. This graph is progressively modified using a graph grammar which we call a *design process grammar*. The non-terminal task nodes, which represent abstract tasks, are replaced by subgraphs of less abstract tasks and intermediate specifications. The output specification nodes are also replaced by nodes that may have a more specific format.

The productions in a graph grammar permit the replacement of one subgraph by another. A production in a design process grammar can be expressed as a tuple $P = (G_{LHS}, G_{RHS}, \sigma_{in}, \sigma_{out})$ where

G_{LHS}, G_{RHS} are process flow graphs for the left side and right side of the production, respectively, such that $T(G_{LHS})$ is a single, non-terminal task node representing the abstract task to be replaced.

σ_{in} is a mapping from $I(G_{RHS})$ to $I(G_{LHS})$ indicating the correspondence between input specifications. Types must match exactly.

σ_{out} is a mapping from $S(G_{LHS}) - I(G_{LHS})$ to $S(G_{RHS})$ indicating the correspondence between output specifications. Each output specification must map to a specification with the same type or a subtype.

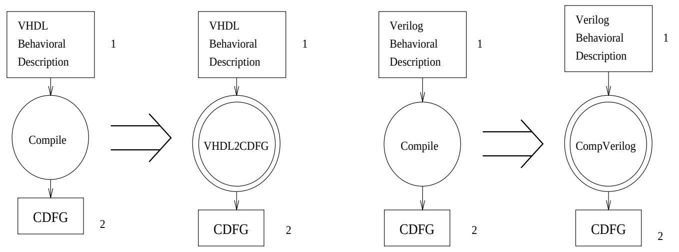
VHDL Behavioral Description
1
Compile
CDFG
2
VHDL Behavioral Description
1
VHDL2CDFG
CDFG
2
Verilog Behavioral Description
1
Compile
CDFG
2
Verilog Behavioral Description
1
CompVerilog
CDFG
2

Figure 2: Alternative productions based on input format

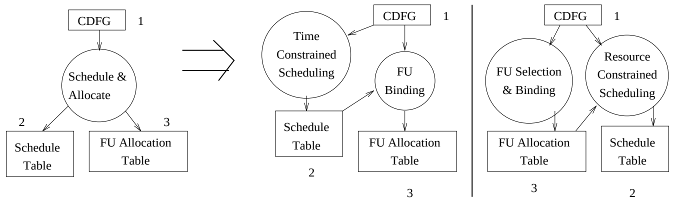

Figure 3: Productions indicating alternative algorithms

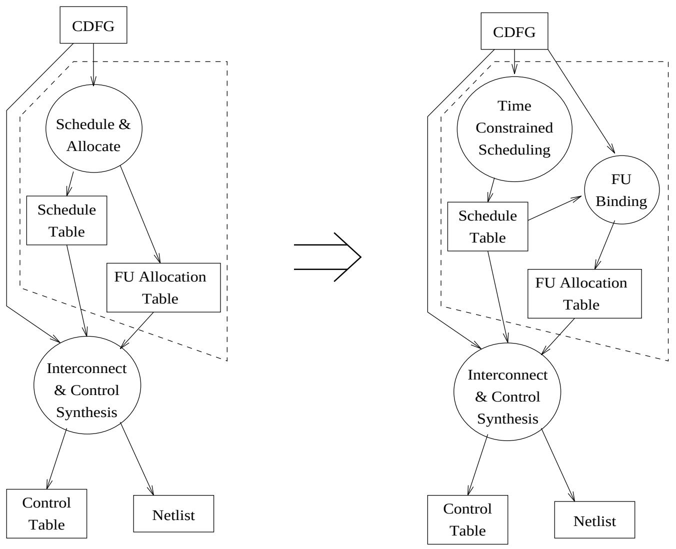

Figure 4: A Sample Graph Derivation

Figures 2 and 3 illustrates some productions for the tasks `Compile` and `Schedule & Allocate`. The mappings are indicated by the numbers beside the specification nodes. The vertical bar is a shorthand notation to indicate multiple rules with the same G_{LHS} but different G_{RHS}'s. Alternative productions may be necessary to handle different formats, or because the right hand sides perform differently in different situations.

Let A be the non-terminal task node in $T(G_{LHS})$ and A' be a non-terminal task node in the original process flow graph, G. Formally, the production matches a node A' if and only if:

i. A' has the same task label as A,

ii. There is a mapping, ρ_{in}, from $In(A)$ to $In(A')$, indicating how the inputs should be mapped. For all nodes $N \in In(A)$, $\rho_{in}(N)$ should have the same type as N or a subtype.

iii. There is a mapping, ρ_{out}, from $Out(A')$ to $Out(A)$, indicating how the outputs should be mapped. For all nodes $N \in Out(A')$, $\rho_{out}(N)$ should have the same type as N or a subtype.

The mappings are used to determine how edges that connected the replaced subgraph to the remainder of G should be redirected to nodes in the new subgraph.

Once a match is found in graph G, the production is applied as follows:

1. Insert $G_{RHS} - I(G_{RHS})$ into G. The inputs of the replaced task are not replaced.
2. For every N in $I(G_{RHS})$ and edge (N, M) in G_{RHS}, add edge $(\rho_{in}(\sigma_{in}(N)), M)$ to G. That is, connect the inputs of A' to the new task nodes that will use them.
3. For every N in $Out(A')$ and edge (N, M) in G, replace edge (N, M) with edge $(\sigma_{out}(\rho_{out}(N)), M)$ to G. That is, connect the new output nodes to the downstream tasks which will use them.
4. Remove A' and $Out(A')$ from G, along with all edges connecting them.

Figure 4 illustrates a derivation using a production from Figure 3. The dotted lines outline the subgraph that is replaced.

2.3 Guaranteeing Success

In this section, we discuss *completeness* of grammar symbols, which guarantees that a process flow graph with no non-terminal task nodes can be generated from an initial graph. Without this guarantee, it is dangerous to start execution of any of the tasks before completely generating the process flow graph. The designer might reach a dead end where, after investing considerable effort, there are no tools to complete the job from the present state. Being able to start execution before planning is completed is important because information generated by executing some tasks can be very useful in planning others.

A task node is *complete* with respect to certain input and output types if it is possible to produce acceptable output types from any combination of possible input types. Formally, let $I = \{in_1, in_2, ...\}$ and $O = \{out_1, out_2, ...\}$ be lists of types for a task's input and output specifications, respectively. Let $I' = \{in'_1, in'_2, ...\}$ be a list of types such that each in'_i is in_i or a subtype of in_i. A terminal task node N is complete with respect to input types I and output types O if and only if:
for every possible I', there exist an $O' = \{out'_1, out'_2, ...\}$ with each $out'_i = out_i$ or a subtype of out_i such that the tool represented by N can transform inputs with types I' into outputs with types O'.
A non-terminal task node N is complete with respect to input types I and output types O if and only if:
for every possible I', there exists a production P such that

i. N with $In(N)$ having types I' and $Out(N)$ having types O matches P, and

ii. every task node M in $G_{RHS}(P)$ is complete with respect to the types of $In(M)$ and $Out(M)$.

Intuitively, Theorem 1 states that completeness of the nodes in the initial graph guarantees success of design planning. If all task nodes in the start graph are complete with respect to the specification types with which they appear, there are tools available to transform the input specifications into outputs of the desired type. Users should avoid using a production with a task node in G_{RHS} which is not complete with respect to the input and output types with which it appears. Fortunately, algorithms exist to check a set of productions for completeness of non-terminal tasks.

Theorem 1 *If all of the task nodes N in a process flow graph are complete with respect to the types of $In(N)$ and $Out(N)$, the process flow graph can be transformed into one with no non-terminal symbols.*

The proof is an induction on the number of non-terminal task label, input type, and output type combinations that are stated to be complete. As an induction basis, if there are no combinations, than any process flow graph in which all of the nodes are complete with respect to their neighbors already consist of all terminal task nodes. If a new combination is added to the list of complete combinations, then any process flow graph in which all nodes are complete can be converted to a graph containing only the previous set of combinations by applying the appropriate productions. By the induction hypothesis, this new graph can be converted into one containing only terminal nodes. □

3 Handling Multiple Versions

The previous section described how process flow graphs are generated by replacing abstract tasks with graphs of less abstract tasks. However, design involves a search through the space of possible alternatives. As the tasks in a process flow graph are executed, some of the tasks may be executed several times. For example:

- The first execution might not produce an acceptable result, so backtracking occurs and some of the decisions made on the first execution are changed.
- New information may become available which changes some of the decisions made on the first execution, such as approximate characteristics of the final design. In iterative design processes, this new information is a direct result of earlier executions and is refined in later executions.

Each time a task is repeated, it produces new versions of its outputs. Multiple executions of an abstract task often involves using a different production from the design process grammar. In Nelsis, only one version of each specification may be considered at a time. [12]

An extension of the process flow graph, called a *versioned flow graph*, captures this dynamic nature of design processes. Like a process flow graph, a versioned flow graph is a bi-partite acyclic directed graph of the form $G = (T, S, E)$ with the same definitions for T, S, and E. However, the rules for applying a production are changed slightly. When a production is applied in a versioned flow graph, the task node being expanded, A', and its outputs are not removed. A production can be applied to A' again indicating that the task is to be repeated. Each time a production is fired, new specification nodes are generated

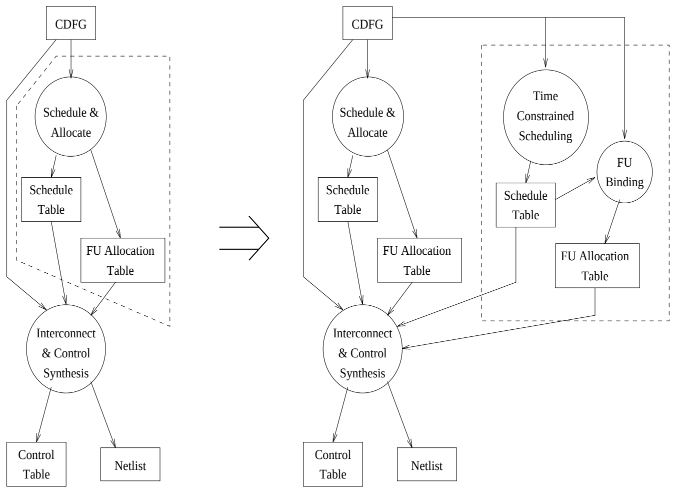

Figure 5: Sample Derivation in Versioned Flow Graph

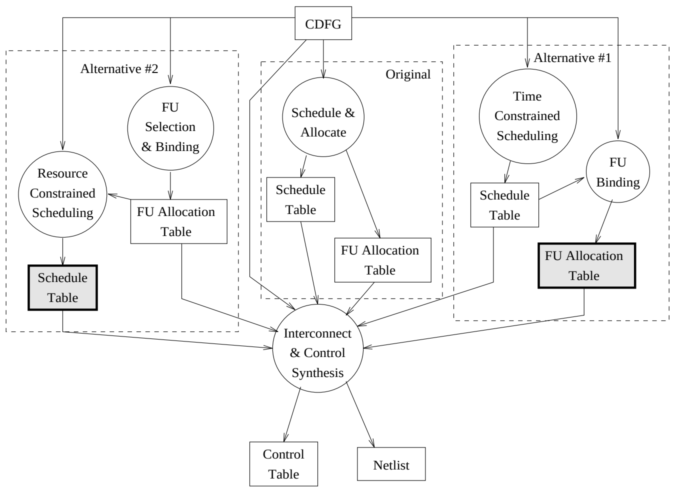

Figure 6: Incompatible Specification Nodes

for the outputs of the abstract task represented by A'. These nodes represent alternative versions of those specifications. The new task nodes are called *subtasks* of A', even if there is only one. Figure 5 shows how the derivation of Figure 4 would be carried out in a versioned flow graph.

In versioned flow graphs, specification nodes resulting from different assumptions can co-exist, as shown in Figure 6. `Interconnection & Control Synthesis` must not be performed using the `FU Allocation Table` of alternative #1 and the `Schedule Table` of alternative #2. Prior to applying a production in a versioned flow graph, the non-terminal task, input specifications, and output specifications must be checked for *compatibility*.

Definition 1 *Two or more nodes are called* compatible *if and only if a non-versioned process flow graph could be constructed that contains all of them.*

The above definition is not very practical in efficiently determining whether certain nodes are compatible. To develop an efficient algorithm for determining compatibility, we must first define the sequence of production firings for a node. A production firing can be characterized by the non-terminal task replaced, the production used, and the input and output mappings. The sequence of firings for a node is defined recursively as the firing in which the node was added to the graph concatenated to the sequence of firings for the task node replaced by that firing.

Theorem 2 *Two nodes are NOT compatible if and only if*

i. their sequences of firings contain different firings for the same non-terminal task or

ii. the sequence of firings for one node includes a firing applied to the other node (if it is a task) or its source (if it is a specification).

If each sequence contains a different firing applied to non-terminal task N, then adding the first node to a non-versioned process flow graph would delete N, making it impossible to add the other node. If a production is applied to a task node N with output specification node S, then both N and S would be deleted from the non-versioned process flow graph. Any node with that firing in its sequence would be incompatible with N and S. If neither of the above conditions hold, then a non-versioned flow graph can be constructed by applying the sequence of firings for the first node and then applying any firings in the sequence for the second that have not already been applied. □

The set of compatible nodes produced by a sequence of firings is called a *design state*. In order to apply a production, all of the nodes involved must be included in the same design state. Applying the production removes the task node and its outputs from the design state, but not from the versioned flow graph. To pursue an additional alternative, a new design state is created. Productions fired in the new design state have no effect on other design states and vice versa.

4 Implementation of a Design Process Manager

Our software architecture is shown in Figure 7. The focal point is a program called Cockpit which keeps track of the versioned flow graph, applies productions, and executes tasks. Upon start-up, Cockpit reads a file indicating what tasks are to be done (in the form of an initial graph), what productions are to be considered, and what constraints apply. Filenames are associated with the input and output nodes of the initial graph to indicate where the actual specifications reside. The cockpit program communicates with a set of manager programs using UNIX message passing. The manager programs determine when to apply productions or help the user decide when to apply productions by assigning ratings. These programs allow domain specific knowledge to be used in control decisions. They may be written by end users, system administrators, tool vendors, or others. The details of manager programs will be discussed in section 4.2.

The manager programs need access to information from other managers in order to make appropriate decisions. For example, to evaluate a production which is known to optimize area at the expense of latency, a manager would need to gather information about which criteria is most critical in the current context. To accommodate this, we support a query protocol in which Cockpit routes queries and replies among manager programs. One application of queries is in iterative design, where managers send queries about previous executions of the same task and utilize the information to incrementally improve the design. This mechanism is used for the data that would come to an *optional input port* in the Nelsis formalism. Any cycles in their flowmaps due to an iterative process may be broken and replaced by queries in our system.

4.1 Cockpit Program

Cockpit is a general purpose program which contains no application specific knowledge other than what is in the input file. It determines when to apply productions or execute tasks by interacting with the user and with a set of manager programs. For each task node in the graph, the program determines what productions could be applied, as well as computing the input and output mappings. It asks the corresponding manager program to assign a rating indicating the production's usefulness in the current context. The user indicates which design state he is interested in at the moment, and the program displays the non-versioned process flow graph for that design state. When the user clicks on a non-terminal task node, the program displays a list of available productions for that node, along with their computed ratings. By clicking one of the production names, the rhs of the production is also displayed. The user may choose to apply the production by pressing a button. Another button allows the user to execute the task, which for a terminal task node invokes the corresponding tool. For non-terminal tasks, the execute button starts an automatic mode in which manager programs decide which productions to use and execute the subtasks on their own. The user may call up an editor to view the data associated with a specification node. The user can backtrack (undo the application of a production) at any time. By defining new design states, he can pursue a new alternative without suspending the exploration of other alternatives.

4.2 Manager Programs

In addition to taking commands from users, Cockpit communicates with manager programs which can issue similar commands. The user indicates in the input file which manager programs should be used for which tasks and which productions. In this way, tasks which can be automated effectively are performed by manager programs with little user intervention. Generally, manager programs would be responsible for lower level (less abstract) tasks, while the user directly manages high level tasks. Users are always allowed to override the decisions made by manager programs.

Manager programs have the following functions:

Manage_Task This function is called when a task is executed. For terminal tasks, it is responsible for executing the tool. This may include determination of options and parameter values. For non-terminal tasks, it is responsible for deciding when to apply productions. The Cockpit program informs it what productions may be applied, as well as how each production was rated. The Manage_Task function can create new design states if necessary to pursue different alternatives in parallel.

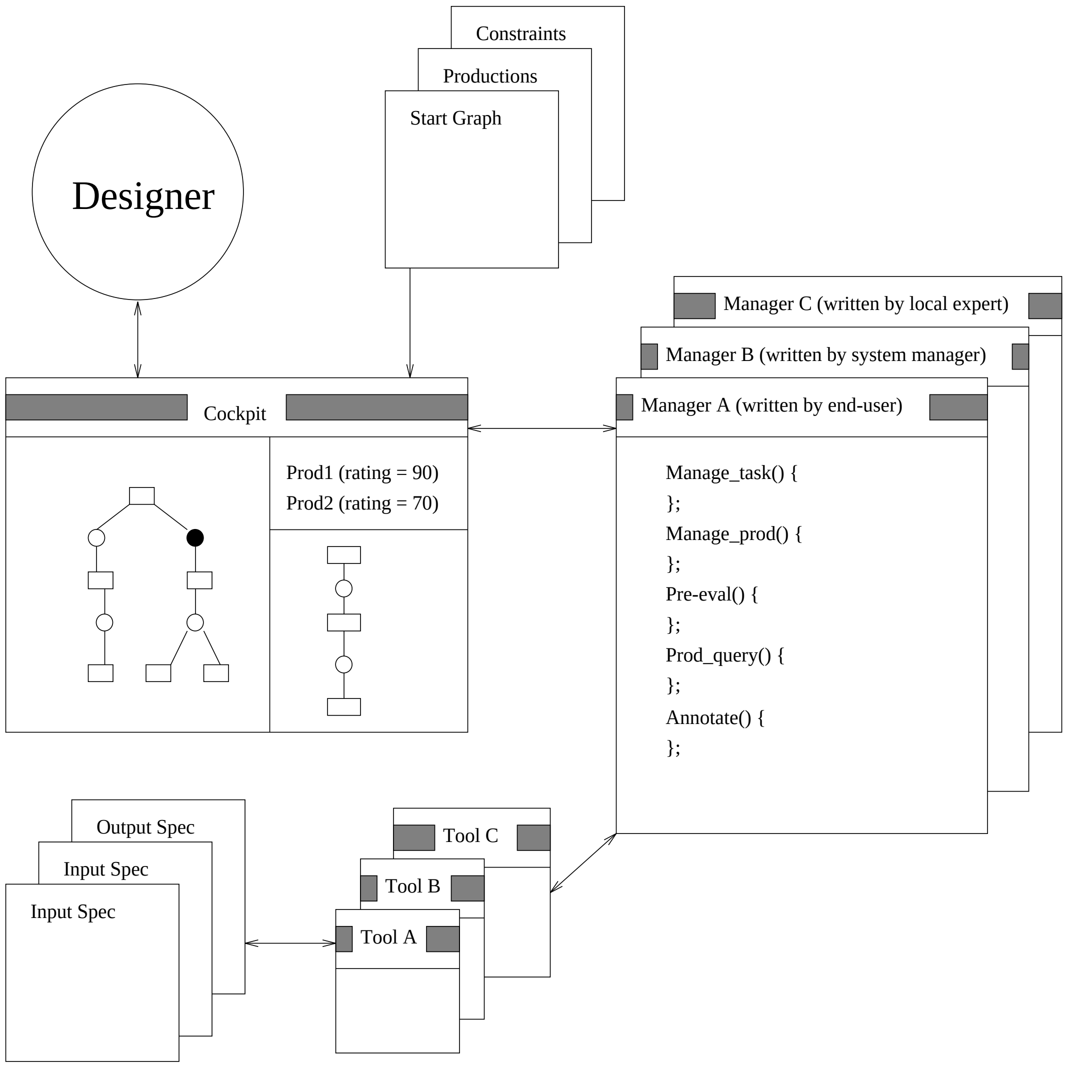

Figure 7: Block Diagram of System

Manage_Prod This function is called immediately after applying a production in automatic mode. It decides when each of the subtasks should be executed. It is informed when the data in any of the specification nodes is modified, so tasks which use that data can be executed.

Pre_Eval This function assigns the rating indicating how suitable the production would be in the current context. It is called whenever the Cockpit program identifies that the production can legally be applied. Users and Manage_Task functions can also explicitly ask for a production to be re-evaluated if new information becomes available that might change the rating.

Task_Query This function answers queries for tasks specific information which may be useful in evaluating a production or managing a task. Computing a response often requires sending queries to one of the productions or to a parent task in the task hierarchy.

Prod_Query This function also answers queries for task specific information. It is separated from Task_query because the method of computing the information may depend upon which production was applied.

All of these functions have access to the input and output filenames and to any global information such as constraints.

These functions can implement a range of control strategies. A simple Manage_Task function might simply select the production with the highest pre-evaluation rating. A more advanced Manage_Task function might decide to create several design states to apply several productions in parallel. It should then decide what computing resources should be devoted to each. A Manage_Prod function might execute each subtask as soon as the inputs were ready. On the other hand, a Manage_Prod function for `Chip Design` might insist that `simulation` be performed before `synthesis`, even though there is no data dependency. More advanced Manage_Prod functions would adjust their behavior depending on what computing resources are available.

Pre-evaluation functions must encode task specific knowledge to be useful. For the production in Figure 3 which replaces `Schedule & Allocate` with `Time Constrained Scheduling` and `FU Binding`, the pre-evaluation function would assign a high rating if a time constraint was specified, especially if the constraint seemed difficult to satisfy. The alternative production, which uses `Resource Constrained Scheduling`, would receive a high rating if an area constraint was specified.

Query functions make more information available for decision making. For example, the pre-evaluation function described above might send a query to the `Schedule & Allocate` task asking if any other productions have been applied and, if so, why they failed. The rating would be higher if other productions failed due to time constraints. Task managers may use queries to determine which computing resources are available before deciding how many productions to pursue simultaneously.

5 Conclusion

The primary advantages of our system are:

Formalism We have developed a strong theoretical foundation for our system. This enables us to analyze how our system will operate with different methodologies.

Parallelism Our system allows several alternatives to be explored simultaneously. This enables designers to make better use of idle computing resources.

Flexibility Many different control strategies can be implemented by manager programs. The user is not forced to encode knowledge using pre-defined methods.

Our framework will be most successful if there are many small grain tools that use common formats for intermediate specifications. Unfortunately, we have found that tool sets that exist now for high level synthesis do not satisfy these requirements. We expect this to change due to users' demand for interoperability among tools, as indicated by the strong support for CFI.

References

[1] Michael L. Bushnell and S. W. Director. VLSI CAD tool integration using the Ulysses environement. In *23rd ACM/IEEE Design Automation Conference*, pages 55–61, 1986.

[2] Andrea Casotto, A. Richard Newton, and Alberto Sangiovanni-Vincentelli. Design management based on design traces. In *27th ACM/IEEE Design Automation Conference*, pages 136–141, 1990.

[3] Tzi-cker F. Chiueh and Randy H. Katz. A history modle for managing the VLSI design process. In *International Conference on Computer Aided Design*, pages 358–361, 1990.

[4] James Daniell and Steven W. Director. An object oriented approach to CAD tool control. *IEEE Transactions on Computer-Aided Design*, pages 698–713, June 1991.

[5] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph Grammars and their Application to Computer Science and Biology*. Springer-Verlag, Berlin, 1979.

[6] H. Ehrig. Tutorial introduction to the algebraic theory of graph grammars. In *Graph Grammars and their Application to Computer Science*. Springer-Verlag, Berlin, 1987.

[7] Margarida F. Jacome and Stephen W. Director. Design process management for CAD frameworks. In *29th ACM/IEEE Design Automation Conference*, pages 500–505, 1992.

[8] Alberto Di Janni. A monitor for complex CAD systems. In *23rd ACM/IEEE Design Automation Conference*, pages 145–151, 1986.

[9] David Knapp and Alice Parker. The ADAM design planning engine. In *Artificial Intelligence in Design, Volume II*, pages 263–285. Academic Press, 1992. reprinted from IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 10, No. 7, July 1991.

[10] David W. Knapp and Alice C. Parker. A design utility manager: The ADAM planning engine. In *23rd ACM/IEEE Design Automation Conference*, pages 48–54, 1986.

[11] M. Nagl. A tutorial and bibliographic survey on graph grammars. In *Graph Grammars and their Application to Computer Science and Biology*. Springer-Verlag, Berlin, 1979.

[12] K.O ten Bosch, P. Bingley, and P. van der Wolf. Design flow management in the NELSIS CAD framework. In *28th ACM/IEEE Design Automation Conference*, pages 711–716, 1991.