MANAGING THE VLSI DESIGN PROCESS[1]

Tzi-cker F. Chiueh, Randy H. Katz, Valerie D. King

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: Ways to represent and structure data within the design environment are reasonably well-understood, and have led to a number of proposed and implemented design frameworks. Comparable support for the operational nature of design, i.e., the controlled and disciplined sequencing of CAD tool invocations, are still in their infancy. In this paper, we describe a model for managing designers' work within the VLSI design environment. The model is based on *a task specification language*, for encapsulating CAD tool invocations and arranging the sequencing of such invocations to accomplish specific tasks, and an *activity/history model*, which maintains the history of task invocations and serves as a focus for sharing work results in a cooperative manner. The task specification language and a prototype tool navigator have been implemented within the OCT CAD framework.

Key Words and Phrases: Design Databases, Process Management, Groupware, Task Specification, Activity Model;

1. Introduction and Motivation

VLSI design systems first concentrated on providing computer-aided tools for the creation and verification of the design. Because of the proliferation of design description formats, communication of design data among tools became a serious bottleneck. Design database systems, such as OCT developed at U. C. Berkeley [HARR86, OCT 89], evolved to provide common formats and more structured ways of organizing design descriptions to reduce the communications problem. Design management systems were a further evolution, concerned with organizing the design across time by supporting versions and configurations. An example of such a system is the Version Server prototype also developed at U. C. Berkeley [KATZ87].

Design systems have now reached the point where design representations and data structure are reasonably well understood. The next challenge is to develop better support for the operational aspects of design. By this we mean the controlled sequencing of design activities (process management), and the allocation of resources (people, machines, etc.) to the process, coupled with the monitoring of project progress (project management). The term process, as used here, should not be confused with the usual concept of a manufacturing job shop. Rather, process management is concerned with procedural/sequencing aspects of the design work, while project

[1]Research supported through National Science Foundation Grant Number MIP 8706002.

management deals with resource allocation. In this paper, we shall concentrate on process management, and will propose a model for its description and manipulation. We will describe the two components of process management, *task specification* and the *activity/history model*, in considerable detail.

Our process model adopts a semi-structured view. Some work actions are routine and highly structured, while others represent creative activities, and their sequence cannot be regimented in advance. The basic units of work are *tasks*. These structured work actions correspond to CAD tool invocation pipelines, and represent frequently reoccuring units of work whose sequences can be specified in advance. An example might be a pipelined tool sequence needed to extract a netlist description from a schematic. If tasks represent the portion of design work that can be automated, then *activities* provide the mechanism through which tasks can be interleaved and sequenced in any order. Activities correspond to the dynamic invocation of tasks, maintaining the designer's execution context. An example activity might be "build the standard cell design for a datapath". Obviously such an activity involves several task invocations, probably integrates the work of multiple designers, and spans a fairly long duration. Unless the design style is very structured, describing *in advance* the detailed sequence of steps necessary to complete a complex object like the datapath is impossible.

Activities provide a way of collecting history about how a design has evolved. The model allows designers to query and browse the execution history, and to choose its sections for archive or deletion. By capturing previous task invocations, it is also possible to define new high level tasks from frequently encountered primitive task sequences in the history.

Figure 1.1 illustrates the software components of our process management system and the relationships between them. The Template Manager and the Tool Navigator support task management, i.e., the specification of work units and the sequencing through them. The Activity Manager and History Manager implement the activity/history model, i.e., maintenance of design context and the history of task invocations. Through a graphical user interface, the designer interacts with every part of the system depicted in the figure, being insulated from direct contact with the underlying tools and design database.

The rest of this paper is organized as followed. In the next section we present the user model in more detail. Sections 3 and 4 more thoroughly examine task specification and the activity/history, respectively. Section 5 describes related work, and Section 6 contains our summary and plans.

2. Definitions and User View

The user model's elements, summarized in Figure 2.1, are organized around design projects. Within a project, the top level concept is the *design process*, representing a single designer's history of actions. There is one design process for each designer in a project, and a designer can be associated with a distinct design process for each project in which he participates.

The next level is *design activity*, which denotes the part of a design process corresponding to a coherent unit of design actions and their associated data. An activity is created with a specific goal in mind, such as the implementation of individual design objects. They provide a context within which data relevant to the goal can be gathered and against which design operations can be invoked. While there are no restrictions on what constitutes an activity, they are meant to structure the design process, and so should be used judiciously.

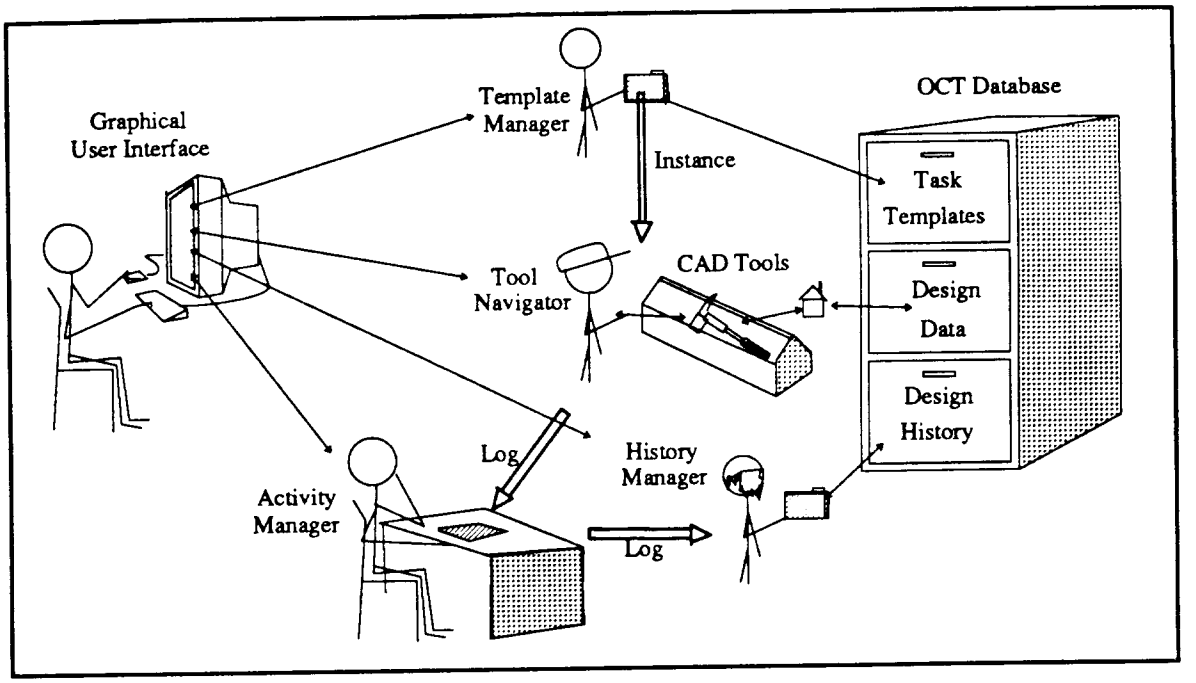


Figure 1.1: Components of the Process Management System

A Project Leader defines new tasks through the Template Manager. The Tool Navigator leads designers through tasks, invoking encapsulated tools on their behalf. The Activity Manager assists the designer in creating and maintaining his design contexts. The History Manager allows him to browse and manipulate the log of work actions he has performed.

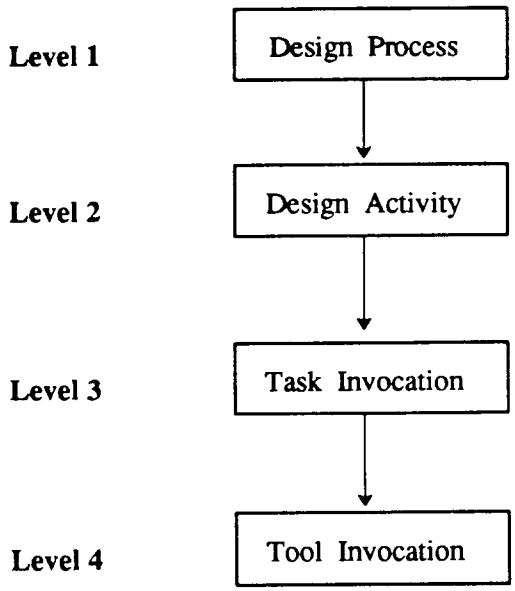


Figure 2.1 The Hierarchy of Design History

A design project consists of multiple design processes, each corresponding to the universe of work performed by a particular designer. Design processes, in turn, consist of a number of goal-oriented design activities, such as build the ALU, build the shifter, etc. A design activity is a thread of multiple task invocations, each of which is realized by actual CAD tool invocations.

The units of work are tasks, and their specification is a key element of process management. They encode commonly encountered tool invocation sequences. Complex task specifications are built up hierarchically as ordered sequences of more primitive tasks. At their most primitive, tasks represent a simple encapsulation of a design tool, describing its inputs, outputs, and default parameters required to complete the function of the task. We call such specifications *task templates*.

Tasks are always invoked within an activity. This causes a task template to be instantiated with the proper substitution of input objects and parameters. Designers can affect the details of task sequencing by their choice of certain task parameter settings. Within the design history, tasks and their effects are normally ordered by their time of invocation, creating a linear history. Section 4 describes how we can reset the environment to an earlier state, permitting the invocation of new tasks, thus creating branches in the design history.

The most primitive action in our model is tool invocation, that is, the execution of a CAD tool with proper inputs and parameters. The sequencing of tool invocations within a task is largely predefined by the task template. If the task does not define an ordering among tool invocations, the designer is presented with a choice of what to execute next.

3. Task Specification

3.1. Model and Components

Tasks are encapsulated units of work. They take input objects, and under the control of selected parameters, produce output objects. They can be described hierarchically in terms of simpler tasks. Task structures are specified through *task templates*. Primitive task templates provide all the information needed to invoke a specific design tool.

Complex tasks are composed from a combination of primitive and complex tasks. They resemble primitive tasks in their description of inputs, outputs, and parameters, but include a description of *subtasks*. These, in turn, are named instances of task templates, primitive or complex, with stipulated inputs, outputs, and parameter settings and arguments. *Ordering constraints*, to be described in more detail below, constrain the invocation sequence of subtasks.

3.2. Primitive Task Specification: Tool Encapsulation

Let's consider the specification of sample primitive and complex tasks to illustrate the graphical and internal forms. Figure 3.1a and 3.1b show the two forms for the same task, which executes the *place pads* tool available in the OCT Design Environment. This particular tool, *padp*, performs a variety of utility operations on pads, depending on parameter settings. Designers deal with the graphical form, while the LISP-like S-expression form is stored directly as OCT internal structures.

The detailed task information is recorded in the internal form. A task has a unique name, a descriptive purpose, and belongs to a class. The latter determines a task's type, such as primitive, complex, utility, editing, etc. The information is exploited by the task management software: utility tasks produce reports rather than design data, and do not effect the flow of data among tools; editing tasks may use the same object for both input and output. Since the example task is "primitive", the property "executable" must also be specified, giving the name of the executable file that implements the tool.

Note that task names and tool names need not be identical. Tools frequently provide alternative functions depending on the parameter settings. For example, the same tool might map from truth tables to equations and vice versa. Such a tool could be encapsulated as two different tasks, with different parameter settings, depending on the work to be done. The fact that it is really the same software is hidden from the user.

The *padp* template shows the effects of parameter settings on the function of the tool. If *-l* is set, *padp* creates an output list of terminals and their placements. The *-D* parameter causes the tool to read an annotated terminals list to direct the pad placement. Note that not all inputs and outputs need correspond to design objects stored in files. *Padp* places its *-l* output onto standard output, which must be redirected to a file if it is to be saved for further processing.

3.3. Complex Task Specification: Building Tasks from Subtasks

Figures 3.2a and 3.2b contain an example complex task. It describes the process of (1) listing the terminals, (2) annotating the listing with placement information, and (3) performing the placement by running padp with the annotated list. Thus, it uses two distinct instances of the *padp* primitive task, with different parameter setting to control the tool's behavior.

The key difference between a primitive and complex task specification are the additional properties added for specifying and ordering the subtasks. Each subtask is given an instance

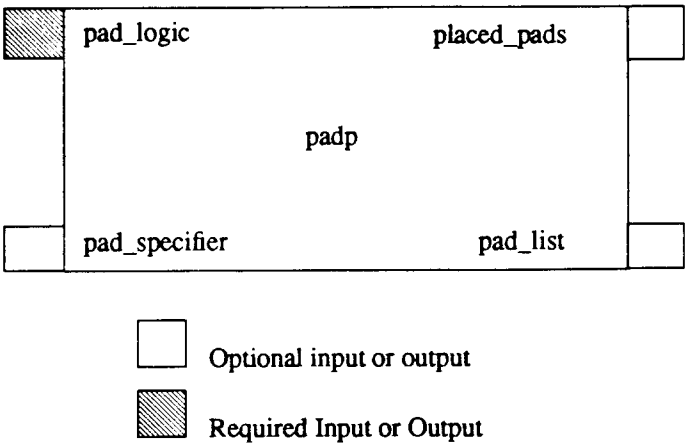


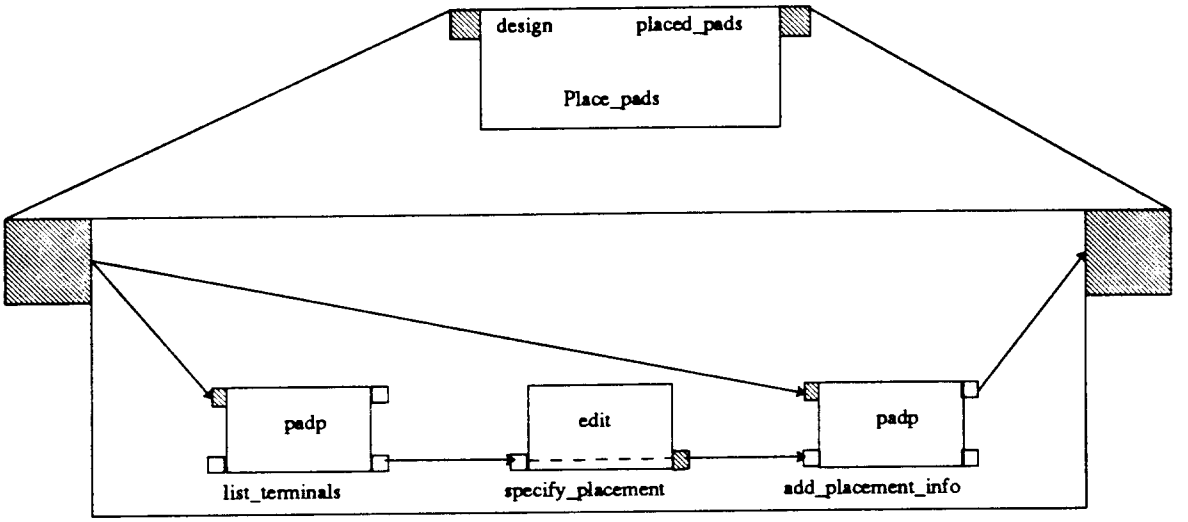
Figure 3.1a - Graphical Task Representation

This is a black box representation with inputs on the left and outputs on the right. Note the use of shading to distinguish optional from required I/O objects. Parameters are selected by interactively "clicking" on the task box. A menu pops up, allowing the parameters to be set. Padp is an OCT tool that places pads around the periphery of a circuit and wires them to internal logic. The only required input is the logic itself. Depending on the parameter settings, different output objects could be created.

```
(task
    (name padp)
    (class primitive)
    (executable padp)
    (input
      ((name pad_logic) (type oct_name) (primary)))
    (parameters
      (l (type bool) (descr ("list pads")))
      (D (type input) (descr ("use placement information from file")))
      (o (type output) (descr ("output file specifier")))
      (f (type bool) (descr ("label pads with property PLACEMENT_CLASS")))
      (u (type family) (descr ("use family for unimplemented terminals")))
    (cond
        (eq? parameter l)(
          (output
            ((name pad_list) (type padp_text) (redirected)))))
    (cond
        (eq? parameter D)(
          (input
            ((name pad_specifier) (type padp_text)))))
    (cond
        (eq? parameter o) (
          (output
            ((name placed_pads) (type oct_name)))))
    (purpose
      ("Run padp")))
```

Figure 3.1b - Internal Form of Task Representation

The internal form follows an s-expression/attribute-value pair syntax that is easy to parse into OCT data structures. The internal form records all parameters understood by the tool and the effect a parameter specification has on determining the inputs and outputs.



"Place_pads" Dataflow

Figure 3.2a - Graphical Task Specification

The place_pads complex task is composed from three primitive subtasks. Padp (with -l) is first invoked on the logic description to extract the terminals. The designer uses a text editor to annotate the terminal list with placement directives. Padp (with -D) is then invoked to perform the placement.

```
(task
    (name Place_pads)
    (class complex)
    (input
      ((name design) (type oct_name)))
    (output
      ((name pads_placed) (type oct_name)))
    (subtasks
      (seq
        ((name padp list_terminals)
          (input pad_logic design)
          (output pad_list terminal_list)
          (parameters (l)))
        ((name edit specify_placement)
          (input edit_file terminal_list)
          (output edit_file terminal_list))
        ((name padp add_placement_info)
          (input pad_logic design)
          (input pad_specifier terminal_list)
          (output placed_pads pads_placed)
          (parameters
            (D terminal_list)
            (o pads_placed)))))
    (purpose
      ("Get a list of terminals and specify their placement")))
```

Figure 3.2b - Internal Form of Task Specification

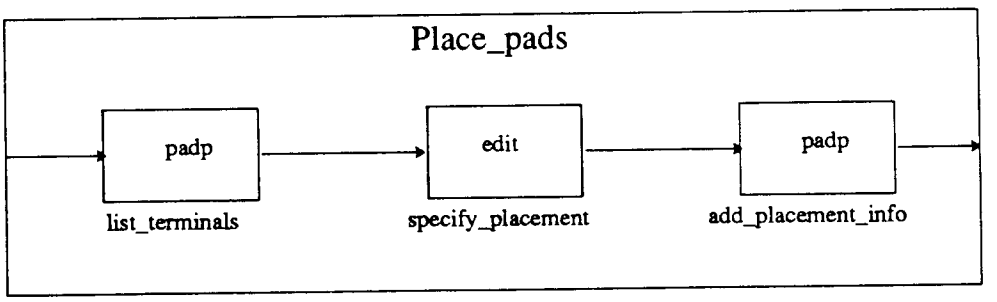
Note the use of subtasks and sequencing clauses. Also, the complex task's inputs and outputs are mapped onto the inputs and the internal subtask instances.

name, unique to this template. A complex task's I/Os are mapped onto the inputs and outputs of its subtasks. Some parameters may be hardwired within the specification to obtain a particular behavior from the subtasks. Although not illustrated by this example, a complex task template may specify parameters and their conditional effects. These can be set by users or higher level tasks. The dotted line in Figure 3.2a from the input to the output in the "specify_placement" subtask indicates that the input itself is modified. Figure 3.3 illustrates the subtask sequencing, which in this case, is completely determined by data flows.

Task templates provide a limited capability for AND parallelism within the task sequencing. Figure 3.4 shows an example sequence constraint network with parallelism among the subtasks. By AND parallelism, we mean that all subtasks must be executed in the course of completing the task, and that all predecessor subtasks must complete before a successor subtask can be invoked. OR parallelism is achieved by using independent tasks.

3.4. Task Implementation and Utilities

The tool most commonly encountered by the designer is the *tool navigator*. It provides the graphical interface for task invocation. Since the system has knowledge of the task structures and the designer's progress through the task steps, the tool navigator can present him with a list of subtasks within the currently active task that can be invoked next. When the designer has selected a task, the tool navigator prompts him for the actual input and output object names, unless these can be inferred directly from the task specification. In addition, he is prompted for parameter settings and other tool arguments. Note that the tool navigator has no intelligence about the objectives of the CAD tools or about the state or merit of the design and thus, makes no choices regarding the selection of tools. Its knowledge is limited to its database of task templates. While a task is running, the navigator informs the designer of what is permissible, not what should be done.



"Place_pads" Sequencing

Figure 3.3 - Order Dependencies Among Subtasks

The sequencing of subtasks is fully determined by data flows in this example.

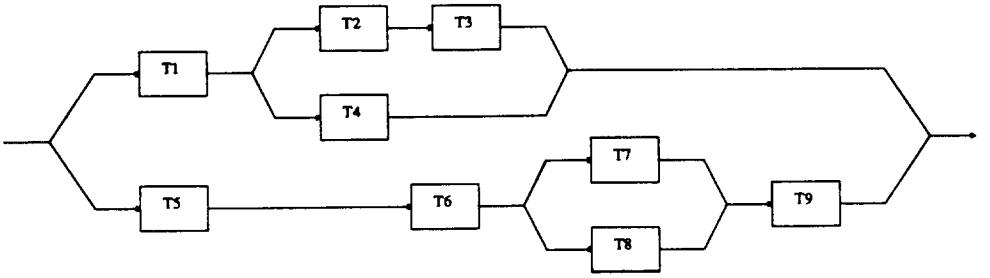


Figure 3.4 - Sequencing Constraint Example

The AND forks and joins impose partial orderings on the subtask invocations. For example, T1 and T5 can be simultaneously in execution, but T1 must complete before T2 or T4 can be invoked and T5 before T6. T7 and T8 must complete before T9 can commence.

A second interface, for task specification, is limited to those designers with responsibility for defining the project's design process. It includes a *task previewer*, useful for debugging task specifications by stepping through a task without invoking any tools. A designer can use the task previewer to try out a task before actually invoking, to see if it performs the work he needs to do.

4. The Activity/History Model

4.1. Conceptual View

A VLSI designer is concerned with three distinct aspects of his design process: the *data space*, the *tool space*, and the *operation space*. The data space is the collection of data objects created or referenced in the course of his design activities. It is his window into the design database. The tool space describes the set of tools he can use. They can be grouped by tool type, such as synthesis vs. analysis tools, by supported design style, such as PLA vs. standard cell, by CAD tool vendor, or by any aggregation criteria that makes sense within his particular design environment. Progress through a task is ultimately represented by references into the tool space to actively invoked tools. The operation space contains the history of all design operations he has performed so far. An operation maps onto an instance of a design tool invoked on some input design objects to produce some output design objects.

Activities are our mechanism for threading together task invocations to organize the design history and to provide the designer with the context for his work. As such, an activity combines three components derived from the spaces described above: the *Activity Workspace*, the *Task Status*, and the *Activity History*. As illustrated in Figure 4.1, these components open windows into the data, operation, and tool spaces, respectively. The Activity Workspace is simply that subset of the underlying design database containing objects created and referenced by the tasks associated with the activity. Since a designer may have multiple activities in progress, the union of their activity workspaces corresponds to the Private Workspace of the Version Server model [KATZ87]. The Task Status records the designer's progress within the current set of task

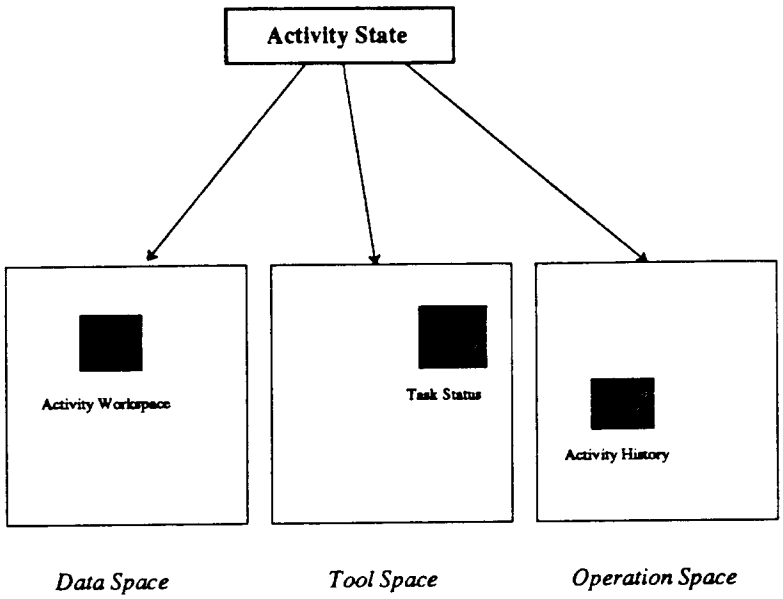


Figure 4.1 The State of a Design Activity

The three components of an activity state: the collection of data objects produced and consumed by the activity, the tools currently actively invoked within the activity, and the set of recorded invocations since the activity began.

invocations. Each activity has an associated hierarchical set of currency pointers that point from active task to active subtask and ultimately to the description of the invoked tool. The Activity History is a possibly branching sequence of history records, which logs a task invocation and its associated input and output objects. The history is organized hierarchically to reflect the hierarchical structure of tasks. Unique states of the Activity Workspace are identified by *design points*; there is one created for each task invocation, representing the state of the Activity Workspace before the task was invoked.

4.2. Operational View

The following assumptions underlie the use of the facilities of our activity/history model:

[1] Every task invocation must be issued within a certain activity, which can be specified explicitly or inherited implicitly from the previous invocation. A *current cursor* identifies the history record of the last subtask completed within the activity. A new history record is appended after the current cursor when its associated task completes, and the cursor is advanced. This assumption requires that a designer be in the proper context by selecting the appropriate activity and the position within the design history from which to invoke a task.

[2] History operations are limited to a single activity. To manipulate a different activity's history, a designer must select that activity and be positioned within it. If a designer wants to work with more than one activity at the same time, he must combine them into a single new

activity and proceed.

[3] When a designer invokes a task template with inputs and parameters, he commits to following the prescribed sequence of operations. A user steps through the tool space under the guidance of the tool navigator, subject to the partial ordering imposed by the task template. Thus, the system not only navigates for users, but also checks their operations, reporting errors when sequencing constraints are violated.

The model supports two distinct classes of design operations: *Task Invocation Operations* and *Control Operations*. The former support the real work by starting and suspending tasks, while the latter help organize the design process. As we have already mentioned, the activity mechanism forces designers to impose certain structures onto their design process. The control operations allow designers to group their activities in a meaningful way or override the strict temporal ordering of task invocations within an activity. They can be further characterized as those that apply within (internal) a design activity or outside (external) it.

Internal control operations are issued within a specific activity. Tasks are invoked from the current cursor of the current activity. Thus, the primary mechanism for creating alternatives is to reposition the current cursor to a previous design point. Repositioning the cursor makes visible only those objects in the Activity Workspace that existed when the design point was created. We call the collection of such design objects the *data scope* of the design point.

Repositioning supports data reuse, by making it possible to apply new task sequences to a previous state of the Activity Workspace. It is also possible to reuse operations: sequences already in the history can be "replayed" with new data objects. Frequently reused sequences can even be promoted to tasks in their own right.

The following are the internal control operations supported by the model:

(1) POSITION the cursor at the desired design point in the current activity. A version of POSITION, called REWORK, supports "tear up and start over". The path between the rework design point and the previously current design point is scanned to find objects to delete from the Activity Workspace. Obviously, this should be used sparingly because of its impact on the process data structures.

(2) FORK from the desired design point. This creates a parallel derivation path rooted at the specified design point.

(3) ACTIVITY-BROWSE: examine the entire design history of the current activity up to the current cursor.

(4) DATA-BROWSE: examine the data scope of the current cursor.

(5) OBJECT-BROWSE: examine the derivation history of a selected design object.

(6) ITERATE-START and ITERATE-END: these operations mark the start and end of an iterated sequence of task invocation where only the final iteration produces objects of interest to downstream tasks. Once an iteration is marked as being complete, all but the last sequence's objects and history records are eliminated.

(7) ANNOTATE: add annotations to an activity, a design object, or a portion of a design activity.

(8) SHOW-TASK-STATUS: the system shows the in-progress tasks within the current activity, and allows one of these to be made the current task.

(9) EXIT: put the current activity to sleep.

External control operations can be issued only in the special *system activity*. Control operations are used to compose new activities from existing ones or to change their status. Activities are either active or asleep, and a single distinguished active activity is *current*. Users can change the current activity simply by selecting among the active activities. CASCADE and JOIN operations are used to compose a larger activity from smaller activities. Designers use these operation to interrelate activities in any way they desire. Activity composition is further supported by the ability to create a new activity with the same workspace as an existing activity. Thus, the exploration of large granule alternatives is supported by creating a new activity for each alternative, with each obtaining the same initial workspace from a common parent activity.

The external activity operations are:

(1) ACTIVATE a certain activity by explicit commands or mouse clicking.

(2) CASCADE two activities, as shown in Figure 4.2.

(3) JOIN two or more activities at the end or at the beginning, as shown in Figure 4.3.

(4) CREATE a new activity from scratch.

(5) CREATE a new activity by copying the context of another activity.

(6) DATA-BROWSE: examine the collection of design objects formed from the union of all activity workspaces.

(7) ACTIVITY-BROWSE: examine the unioned design histories of all activities within the current design process.

4.3. An Example

In this section, we illustrate the concepts and operations presented above. Figure 4.4 shows an example design flow, whose goal is to synthesize an ALU. First the designer creates an activity with a descriptive name, such as *synthesize-ALU*. At this initial design point, the associated data scope is empty because nothing has been generated or referenced. To create the logic description of the ALU, the designer next invokes a complex task called *create logic description*. Inside this task, there are actually three subtasks: *enter-logic*, *format-translation*, and *logic minimization*. The tool navigator leads the designer through this sequence step by step. The designer is only responsible for setting proper parameters and preparing inputs appropriately.

After *create-logic-description* has completed, the designer invokes the primitive task *logic simulator*. Because the outputs of the simulation might indicate some design errors, the designer may have to reinvoke the first task to correct the errors. This is an iterative process: many invocations may be needed to obtain the correct logic description. But only the final result is needed in the following design steps. The designer issues ITERATE-START and ITERATE-END operations before and after this sequence of steps to identify the iterated task sequence. Only the objects that are actually used later on in the process flow are maintained.

After the logic is finally made correct, the designer then invokes the *standard-cell-place-and-route* task, followed by the *place-pads* task (which was shown in detail in Section 3) to complete the activity. At this point the current cursor is at design point 5.

Suppose the designer is not satisfied with the result of the standard-cell approach and wishes to try another alternative, such as an implementation in a PLA design style. He then resets the current cursor to design point 3, and invokes the *PLA-generation* task followed by

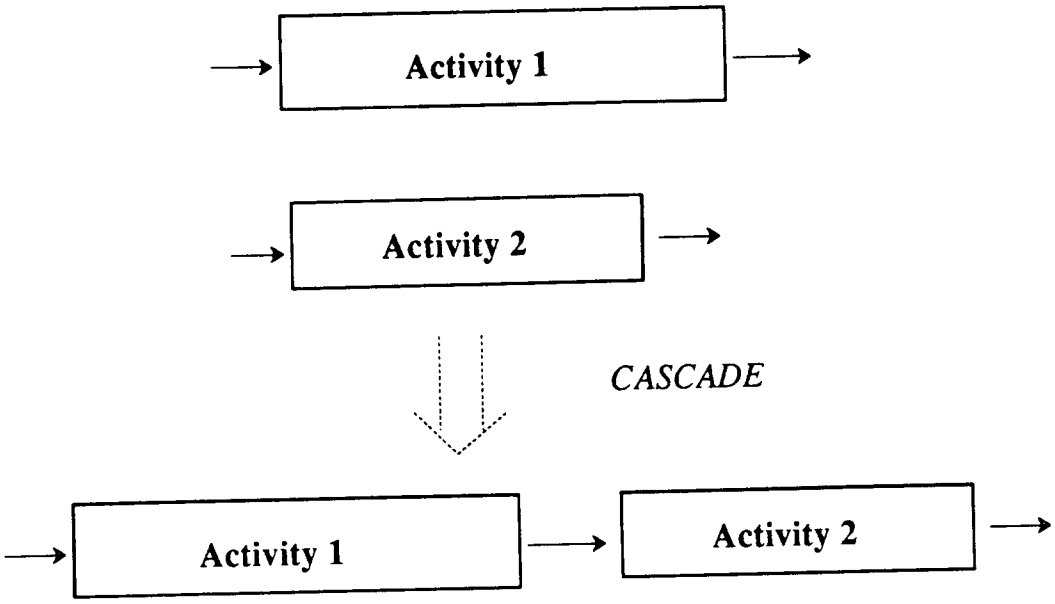


Figure 4.2 Cascading Two Activities

Two activities are combined in series through the cascade operation.

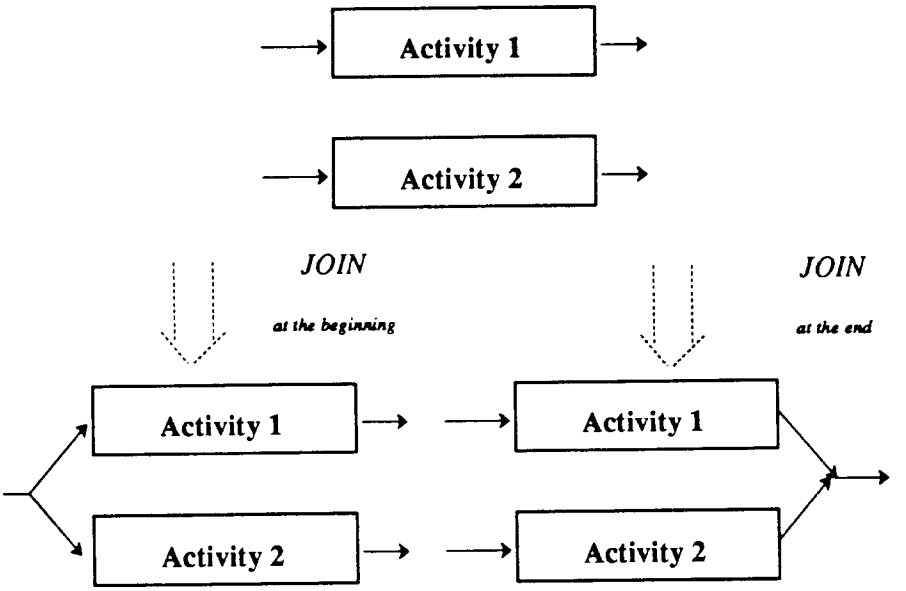


Figure 4.3 Two Ways of Merging Two Activities

Two activities composed in parallel, either at the beginning or the end of the activities, through the join operation.

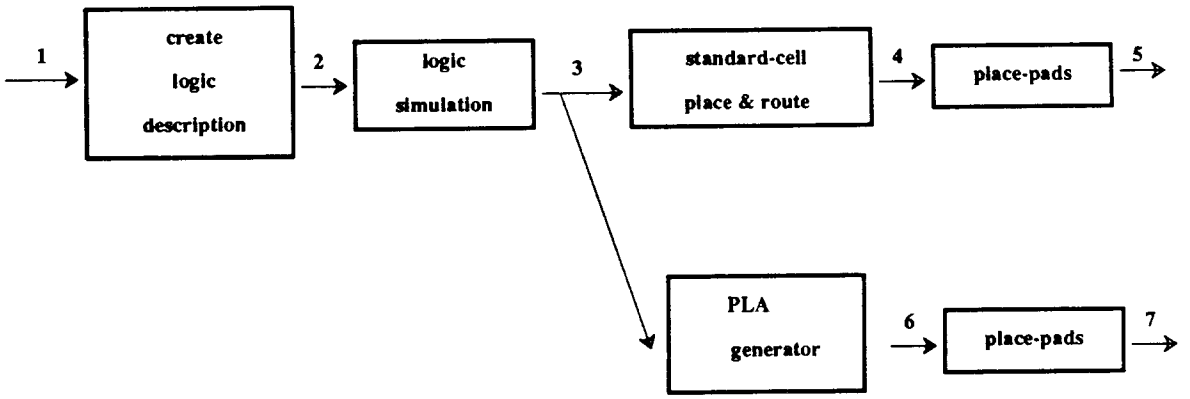


Figure 4.4 The Synthesize - ALU Design Activity

The process flow used in the example. The path is to create a logic description, simulate the logic, and generate lay-out using both the PLA design style and the standard cell design style as alternatives.

place-pads. The effect of changing the current cursor gives the designer the data scope that existed before the standard-cell design flow was started. Note that it is not necessary for the

designer to remember the correct versions of inputs: these are automatically tracked by the history manager. Specifying the roll-back point is relatively easy, especially when the designer can browse the design history and change the current cursor with mouse clicks. This facility for repositioning within the design history is the principal mechanism through which we support exploratory design.

If a design activity is a linear sequence of task invocations, the data scope of every design point is uniquely determined. For example, the data scope of design point 2 is the data used as input or produced as output by *create-logic-description*. However, if an activity exhibits branching structures, the definition of the data scope is ambiguous. Does design point 6 contain data objects from design points 4 and 5, which preceed it in time? In our model, data scopes are defined in terms of paths to the initial data scope. Thus we treat 1-2-3-4-5 and 1-2-3-6-7 as two independent development paths. The data scope of design point 4 (6) is the data set associated with task *create-logic-description*, *logic simulation*, and *standard-cell-place-and-route* (or *PLA generation*). The data scope of design point 5 (7) is similarly determined.

Now suppose another designer is working on control logic. When both the control logic and the ALU are completed, these two efforts should be merged and continued by one of the designers. As shown in Figure 4.5, a designer can merge these two activities (at the end) into one activity called *chip activity*. The data scope of design point N is the union of the activity workspaces of these two activities. The histories of the two activities are also unioned. Moreover, this combined activity is as if it had been created by the current designer: he can reposition to any design point in this new activity and modify the history structure. This facility of combining small activities into larger ones supports an hierarchical design style.

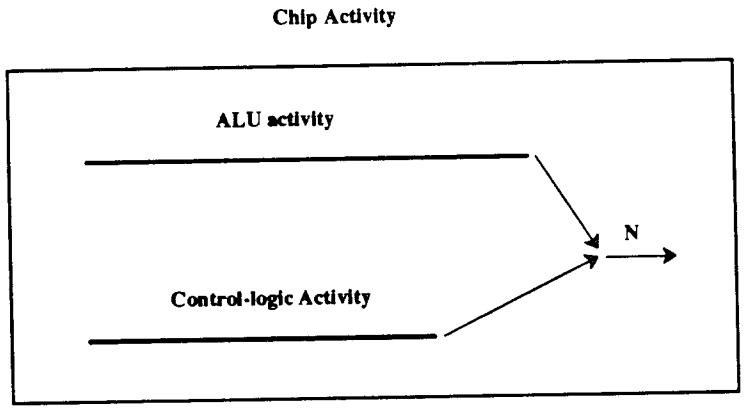


Figure 4.5 Merging Two Activities into One Activity

When related sub-activities are complete, they can be merged and carried onward through a new activities. Merged activities follow the hierarchical structure of the design data.

5. Previous and Related Work

To the best of our knowledge, there is no single system that integrates the features that our system possesses: task specification for structured work actions, an activity mechanism for unstructured work, and history management to support an exploratory design style. There are, however, a number of works in the literature that represent previous attacks on the problem from the perspective of VLSI CAD, software engineering, and office "groupware."

Professor Steven Director and his group at Carnegie Mellon University have done pioneering work on supporting the VLSI design process. Their systems, Ulysses [DANI89] and Cadweld [BUSH89], provide a "script" language for specifying high-level design activity in terms of low-level primitives. It is similar in intent to our task specifications, but differs in philosophy. The script language specifies goals for the design activity, which are posted to a control structure called the blackboard. Tools are like agents that have knowledge about their ability to satisfy goals. More than one one tool can qualify, and it is up to the designer to select among them the one to invoke. The tools are organized in an object-oriented class hierarchy and a rich set of relationships among tools are supported. The work done at CMU concentrates on tool encapsulation and the flexible maintenance of the complex tool environment. Our work focuses more on navigating tasks and history, and supporting the exploration of design alternatives. We provide decision making support for the designer by helping to organize his work steps and their effects on the design database.

Much of the work in the software engineering literature concentrates on collecting an operational history in order to support undo and redo commands. [LEE 88] provides an extensive literature survey on the uses of history in software environments. [LINX86] describes a specific history manager for the Gandalf system at Carnegie Mellon. In general, these systems capture users' actions, usually in terms of the primitive operations supported by the environment, annotated with how they have changed its state. They allow the actions to be *undone* to roll their environment back to an earlier state, or *redone* to roll them forward, perhaps after a crash. Note the lack of any notion of *hierarchical* tasks or activities. The emphasis on primitives rather than higher level tasks is in part due to the nature of the environment: tools at the level of complexity of the VLSI design environment are not frequently found in the software design domain.

The Apollo DSEE provides a more complete mechanism for supporting activities [LEBL84], although the emphasis is primarily on change propagation. DSEE *tasks*, are the detailed work actions necessary to make related representations consistent after a change. *Tasklists*, are generic ways of doing things, and are used to generate specific tasks. Note that tasklists are not meant to represent the design process, but rather the high-level procedure for making the design consistent after a change. They have nothing to do with tool encapsulation, nor are they meant to support exploratory design.

Researchers in cooperative work environments, such as at UC San Diego [BANN83] and Xerox PARC [CARD87], have proposed activity models similar to our own. Their motivation is to provide mechanisms to support multiple, interleaved, and simultaneous threads of activities. They share with us the idea that an activity offers a context whereby users can focus only on things that are relevant for the work action at hand. But their models do not provide operations for activity manipulation, nor do they offer a clear semantics for the relationships between operations, objects, and activities.

6. Summary and Implementation Status

We have a presented a model for process management based on task specification and history management. Key elements for task specification are a graphical format for describing tool encapsulations and a way to create complex "tasks" from the composition of more primitive tasks. The activity/history model is founded on the concepts of design projects, design processes, activities, and task invocations. Its key feature is the notion of an activity: the focus for collecting the history of task invocations including the consequences of such invocations in terms of the design objects produced and consumed. A primitive Template Manager and Tool Navigator are now operational and have been implemented on top of the U. C. Berkeley's OCT Data Manager. These are described in more detail in [KING89]. The implementation of a more sophisticated system incorporating activity and history management is currently underway.

We view design process management as a natural outgrowth of the CAD community's emphasis on design data management over the last few years. The emphasis is now shifting away from data concerns to those which are more operational in nature. Beyond process management, we see a rich area for future work in project management: that is, the exploitation of knowledge about the design process in order to better utilize the resources dedicated to completing a project in a timely fashion.

7. References

[BANN83] L. Bannon, A. Cypher, S. Greenspan, M. L. Monty, "Evaluation and analysis of users' activity organization," Proceedings of the CHI'83 Human Factors in Computer Systems, Boston, MA, (1983), pp. 54-57.

[BUSH89] M. Bushnell, S.W. Director, "Automated design tool execution in the Ulysses design environment," *IEEE Transactions on CAD*, V. 8, N. 3, (March 1989).

[CARD87] S. K. Card, D. A. Henderson Jr., "A multiple, virtual-workspace interface to support user task switching", Proceedings of the CHI'87 Human Factors in Computer Systems, Toronto, Ontario, (1987), pp. 53-59.

[DANI89] J. Daniell, S.W. Director, "An object-oriented approach to distributed CAD tool control," in IEEE Proc. 26th Design Automation Conference, Las Vegas, NV, (June 1989).

[HARR86] D. Harrison, P. Moore, R. Spicklemeier, A. R. Newton, "Design Management and Graphics Editing in the Berkeley Design Environment," Proc. ICCAD Conference, Santa Clara, CA, (November 1986).

[KATZ87] Katz, R. H., R. Bhateja, E. Chang, D. Gedye, V. Trijanto, "Design Version Management", *IEEE Design and Test*, V 4, N 1, (February 1987).

[KING89] King, V. D., "Task Specification and Management in the VLSI Design Process," Computer Science Technical Report No. UCB/CSD 89/533, (September 1989).

[LEBL84] D. B. Leblang, R. P. Chase, "Computer-aided software engineering in a distributed workstation environment," Proceedings ACM SIGPLAN/SIGSOFT Conference on Practical Software Development Environments, (April 1984).

[LEE 88] A. Lee, "Use of history for user support," Technical Report CSRI-212, Computer Systems Research Institute, University of Toronto, (1988).

[LINX86] C. Linxi, A.N. Habermann, "A history mechanism and UNDO/REDO/REUSE support in ALOE" Dept. of Computer Science Technical Report CMU-CS-86-148, Carnegie Mellon University, Pittsburgh, PA.

[OCT 89] OCT Manual, U. C. Berkeley EECS Department Technical Report, 1989.