TASK SPECIFICATION AND MANAGEMENT

IN THE VLSI DESIGN PROCESS

Valerie D. King

September 15, 1989

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

ABSTRACT

Task specification and management is presented as one aspect of a complete model for managing the VLSI design process. I describe the appropriate role of tasks in the design process, their hierarchical specification, including a detailed description of the grammar used, how they are invoked, and then present a prototype implementation integrated into the U. C. Berkeley CAD environment. The task management model presented includes a Template Manager for handling static operations on task specifications, called templates, and a Tool Navigator for dynamic operations including running, previewing, suspending, and resuming tasks.

CONTENTS

FIGURES

Task Specification and Management in the VLSI Design Process

1. Introduction

1.1. The VLSI Design Process

As VLSI designs have grown larger, CAD tools have been developed to assist VLSI designers by automating some of the steps in the design process. Although some designers were initially skeptical, maintaining that the results of a program would not match those done manually, these tools are now widely relied on to produce good-quality results in a reasonable time-frame. Work has also been done in the area of design databases, such as Oct [SPIC89], developed here at U. C. Berkeley, which provide the structure needed to organize the data and establish common interface formats for the CAD tools, and of version servers, such as the prototype also developed here at U. C. Berkeley [SILV89], which organize the design data with respect to versions and configurations.

These developments lay the groundwork for study of the process of design itself. With an increasing portion of the design completed using CAD tools, and the design objects developed held in the database, the process is visible and can be captured, understood, and perhaps most importantly, assisted and better managed.

The study of design processes is inherently different from the traditional treatment of processes in the manufacturing environment, which deals with the issues of streamlining and improving a particular process which, once developed, will be repeated many times. A *design* process is characterized by uniqueness and unpredictability. It is futile to attempt to determine beforehand the exact sequence of steps in the process, and unproductive to stifle a designer's creativity by applying rigid constraints. Instead, the understanding of VLSI design processes must concentrate on the procedural and tool sequencing behavior exhibited by designers in managing their design efforts manually.

In working with various types of complex software systems, there is a propensity of individual users toward developing their own favorite methods of interacting with the system. There may be many ways to reach a particular goal, but most people find a path with which they are comfortable and rarely deviate from it. Some may keep scraps of paper listing their favorite incantations, while others develop script files and complex aliases for these sequences. A model of the design process, then, must accommodate this notion of common sequences, of low-level activity related by function rather than by association with any particular design object.

Most designers will impose some form of organization on their work, such as keeping folders of drawings related to a piece of the design, maintaining online directories for related design data, or imposing structure through naming conventions, to cite a few examples. Accordingly, the design process model must also include a facility for organizing the design and for carrying out the system equivalent of "getting out the notebook on the ALU design". This *context* facility must be flexible enough to support the various styles and degrees of organization exhibited by designers. An organization scheme based solely on the design data hierarchy (if one can be assumed to always be explicit), for

example, would be inappropriate for a designer who prefers to organize his work along the lines of development (e.g., maintaining a directory for all logic descriptions).

In addition to organizing the design, designers almost lways maintain (or wish they had maintained) a history of their efforts. These efforts include tracking design paths that have been rejected and intermediate data which is non-essential but valuable for understanding the current state of the design. In addition, the history necessary for meeting legal requirements. A provision for gathering the information contained in the history should be integral to the design process model. Ideally, the history information should be accessible along different axes. For example, a designer may wish to walk back within a particular context of the design, or he/she may wish to identify the origin of a specific design object.

1.2. The Activity Model

Although not a full exposition of the nature of the VLSI design process, these perceptions form the basis of the Activity Model developed by the Process Management Group led by Dr. Randy H. Katz at U. C. Berkeley. Through its two major components, task specification and the history model, the activity model proposes a mechanism for process management, which is defined to be the controlled sequencing of design activities.

Task specification corresponds to the common sequences discussed above. It enables the encapsulation of the CAD tools using a common graphical user interface, and hierarchical sequencing specification by means of task templates. My project focused on the development of the task specification model and on the implementation of a prototype task manager to parse task templates and add them to the Oct database, and to preview and run tasks instantiated from the templates. The development of the task manager was influenced by its role in the overall system, and thus, a brief discussion of the rest of the model is presented.

The history model encompasses both the organizational and history gathering facilities of the activity model. Dynamic task invocation corresponds to an *activity*, such as "design a pla controller". Operations are available on activities to support cooperative work, exploratory design, and rolling back the database for reworking an unsatisfactory portion or for iterative design. These capabilities are closely tied to the maintenance of history, the record of events captured with respect to an activity and stored in the database.

Fig. 1.1 illustrates the components of the activity model and the relationships among them. The Template Manager and Tool Navigator are part of task management, and the Activity Manager and History Manager are part of the history model. Through the graphical user interface, the designer interacts with every part of the system depicted in the model, but has no direct contact with the Oct database or with the CAD tools.

The Activity Manager interacts with the designer to establish the context for the current session, the *activity*. After the designer selects a template from the database, the Template Manager retrieves it and delivers an instance of it to the Tool Navigator, which obtains the instance-dependent information (e.g., actual inputs, outputs, and parameters) from the designer, and proceeds to navigate him/her interactively through a sequence of tool invocations. Only the tools themselves access and modify the design data. These

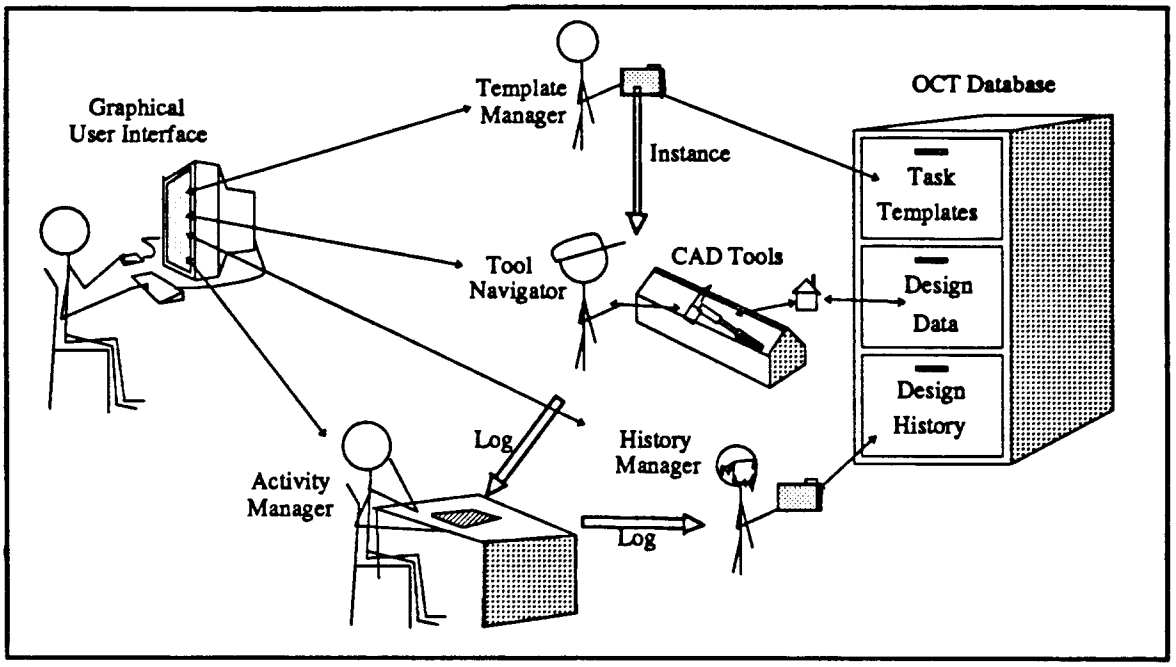


Figure 1.1 - Activity Model

events are recorded and, together with the context, passed to the History Manager. The designer may at any time decide to interact directly with the History Manager for browsing or querying the history of an activity or design object. Outside the realm of design activity, the Template Manager is also responsible for all operations on the templates including creation, deletion, retrieval, and perusal. These administrative operations are independent of any design activity, and need not be part of the design history.

In the next subsection I will examine other work done in this area, with particular attention paid to task specification.

1.3. Previous Work

While no other model of the VLSI design process includes the same features as the activity model, some interesting work has been done in this area. The Ulysses Design Environment [BUSH89] developed at Carnegie-Mellon University, employs a rule-based approach. Design goals are linked with tools through the use of a *scripts* language. The goal is posted on a *blackboard,* and observed by the knowledge source for each tool, which determines whether its capabilities make it a possible candidate. Tool selection is done via inferences made from the codified design methodology, or by direct designer intervention. Sufficient history information is maintained to allow a database rollback capability, but it's not clear how that is accomplished or whether the designer can directly browse history information.

Ulysses defines a *task* as a design methodology described in the scripts language. This notion differs from the task specification in the activity model in that its objects are goals rather than tools, and its semantics presume system knowledge about the tools beyond their I/O behavior and parameters. The rule-based design of the Ulysses system

reflects the AI philosophy of automating the design process, a goal not undertaken, and not necessarily deemed appropriate, by the designers of the activity model.

The Cadweld Design Framework [DANI89] was built on top of the Ulysses Design Environment. It tackles the problem of providing a distributed control mechanism for a large, complex suite of CAD tools by developing a hierarchical object-oriented class structure for the tools. It attempts to hide low-level details of the CAD tools, and to furnish such services as automatic translation when possible. This addition certainly eases the burden on the inference engine in the Ulysses design environment, but it really doesn't change the approach of the system. The class structure's hierarchy is based on the function and abilities of the tools, unlike the task templates in the activity model, which may or may not be *individually* hierarchical in terms of invocation sequences.

Gabbe and Subrahmanyam of AT&T Bell Laboratories [GABB87] present a proposal for an object-based representation of the design process. The design data consists of *modules,* which are made up of other modules, such as transistors. The design work is executed by *taskers,* which are inherited by all objects, and which consist of four elements: (1) *agents* or procedures and *meta-agents,* (2) *controllers,* which contain a list of the agents, (3) *tasker superclasses,* and (4) local data. The selection of agents is done by meta_agents, which have knowledge about the state of the design, the details of the specification, and the goals of the user. The tasker superclasses contain the skeletal structure for the agents, meta-agents, and controllers, which can be spawned dynamically as needed. The agents' and meta-agents' knowledge base may include preconditions, postconditions, computational cost, and an understanding of the meaning of success.

This system resembles the Ulysses and Cadweld systems in its knowledge-based approach. The system does not appear to include any support for history.

In the more general area of design processes and design management, work done at Hughes Aircraft Corporation [KONS88] regarding standards for design management systems stresses the need to support exchangeable parts through easy connection and interfacing of tools, to concentrate on the multi-stage process involved in design, to reflect the hierarchical composition of a design, to account for protocols followed in the design process, and to identify the sequence of design steps followed and the tools used. The activity model incorporates the functionality needed to meet all of these goals but with its inherent flexibility, it doesn't enforce strict compliance with them. A designer's individual use of the system may make the multi-stage process somewhat difficult to extract from the history, for example. (This paper also discusses requirements in areas such as product life cycle maintenance which are beyond the scope of the activity model at present.)

An interesting investigation of process specification was carried out at the University of Colorado [DEME87]. Demeure and Osterweil examined recipes and developed a formal language for specifying them. Data representations, parallelism, error handling, and task attributes were tailored to the application. Some of their conclusions are applicable only to work done on physical objects. For example, a *critical section* encloses instructions which must be carried out without intervening pauses, e.g., "after the food is cooked it must be served before it cools off". They also specify *active* vs. *stand-alone* sessions. The active property could be associated with interactive tools which require the presence of the designer. This information is more interesting, however, from the perspective of the project manager who is concerned about scheduling and resource

utilization.

Their discussion of parallelism is also applicable to the VLSI design process. The sequence of operations is non-deterministic and some method of expressing this parallelism must be found. By parallelism, I don't necessarily mean that many things are occurring simultaneously (although I don't preclude that possibility), but rather that the events are independent and different sequences could occur. The task management model uses an identical model of parallelism.

Demeure and Osterweil's approach to tailoring a system for a particular application was to begin with few assumptions and build the system up as new elements are required. This is the approach taken with the activity model as well. It isn't designed to be general enough to accommodate all types of design systems, but rather to efficiently manage the VLSI design process.

In the area of software engineering, one of the earliest and most widely used tools for managing the design process is *make* [FEL79], which automates the compilation process, keeping track of changes made to each module. Source-code control systems, or SCCS [BTL81], also widely used, provide versioning control for software systems. Building on this work, Apollo's DSEE [LEBL84] includes source-code control, configuration management, task management, and dependency-tracking with notification. A DSEE *task* is a title and a list of textual items used for planning and recording low-level steps. A *current* task is specified with *active* items distinguished from *completed* items. When all items are completed, the task becomes a *transcript*, which can be used as a form of *advice* for other software engineers with similar objectives. DSEE also utilizes task templates containing model active items to derive new instantiated tasks.

This more general view of a task and task items is probably well-suited to to the software engineering environment, but doesn't cover many of the features that seem very beneficial in the VLSI design world with its abundance of complex tools; there is no order, no hierarchy, and no system support (in general) for the completion of the items. It's merely a collection of items which, when completed, finish a task. The emphasis in DSEE is on consistency and change notification.

1.4. Report Organization

The rest of this report will be organized as follows. In the next section I will describe the model for task specification and management. Section 3 will describe the prototype implementation, followed by a section detailing the specification grammar. Lastly, section 5 will present a summary and my conclusions. Examples of task specification are found in the appendix.

2. The Task Specification and Management Model

2.1. Motivation

In addition to supporting the natural tendency of CAD designers to use familiar sequences in their work, there are several other reasons for including task specification and management in the activity model.

The increasing number and complexity of CAD tools makes it difficult for an individual to keep abreast of changes and additions to the tool suite. Learning to use many of the tools is a non-trivial task, and many designers may prefer using a familiar-but-less-suitable tool to investing the time and effort in learning to use a new one. One of the goals of the task specification model is to hide low-level detail in a system with a consistent user interface and an internal knowledge of invocation specifics. Many details such as command-line arguments and data formats can be determined from the context and are better maintained in computer memory than human memory.

Another benefit is derived from the difference between what is considered by a designer to be an atomic goal, or a low-level task, and what constitutes a low-level task in the CAD tool environment. An atomic goal, as far as the designer is concerned, may require more than one CAD tool invocation. Accordingly, understanding "what a designer was trying to do" requires more than knowing that a particular tool was invoked on a set of Oct objects. The sequence of tool invocations furnishes a context for understanding what was occurring. Thus, the use of tasks is essential to maintaining a meaningful history.

The use of tasks also establishes breakpoints or "firewalls" for the activity manager and history manager. The end of a task is a convenient and meaningful function-related point for updating the history, garbage collecting temporary data, and for rolling back the database.

Furthermore, the task manager provides a consistent, graphical interface. While there may be CAD tool suites developed with consistent interfaces, the task manager allows a system to be built around tools from different vendors without compromising the homogeneity of the interface.

From the perspective of project management, task specifications supply a means of specifying design methods and policy through constraints on the invocation. For example, a design policy may prohibit placement and routing before the design has been successfully simulated.

2.2. Overview

The two components of the model are task specification, handled by the Template Manager, and task management, performed by the Tool Navigator, which correspond to static and dynamic views of the model, respectively. Static task specifications will henceforth be referred to as *templates*, and named instances of the templates, with stipulated inputs, outputs, and parameters will simply be called *tasks*. A *primitive* task is equivalent to an encapsulated tool invocation, a procedure for invoking a tool. *Complex* tasks are made up of other tasks, called *subtasks*, which may be either primitive or complex. *Ordering constraints* direct the sequence of invocation of the subtasks. A designer invokes primitive and complex tasks in the same manner.

The templates contain all of the information necessary to invoke the tools, if primitive, and to sequence the tools, if complex. The system 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 template database. While a task is running, the Tool Navigator informs the designer of what steps are permissible, but not which one is best. The task manager is designed to facilitate the use of the CAD tools, and to present the designer with a consistent and friendly interface, not to directly influence the design itself. The system itself imposes no particular design style, but the capability exists for a group to adhere to a method or style through the use of invocation and sequencing constraints.

2.3. Parallelism

The term parallelism is used in this model in the context of subtask sequencing. If there are no dataflow or other constraints on the sequence of execution of two subtasks, for example, they are considered to be parallel. Either subtask may be executed first, and neither need wait for completion of the other before commencing. This represents 'AND' parallelism, in which all paths must be followed but their order doesn't matter.

Fig. 2.1(a) illustrates a complex task. Subtask "T2" is parallel with the sequence "T3 followed by T4". Fig. 2.1(b) shows several, but not all, sequencing possibilities for the subtasks of "Complex_Task", assuming for simplicity that each subtask executes for the same time duration. The overlapping bubbles represent simultaneous execution of subtasks.

Note that in each sample sequence, all four subtasks were executed. The graphical representation may mislead some into thinking that any path from start to finish completes the task. Parallelism in that sense, the 'OR' sense, does not exist in this model. Truly separate paths involving different groups of subtasks are represented by distinct templates. The single exception to this rule is that any individual subtask may be characterized as *optional* and may be omitted.

Parallelism in the data domain, i.e., performing the same operations on each set of data, is made available, although not explicitly, through operations available in the

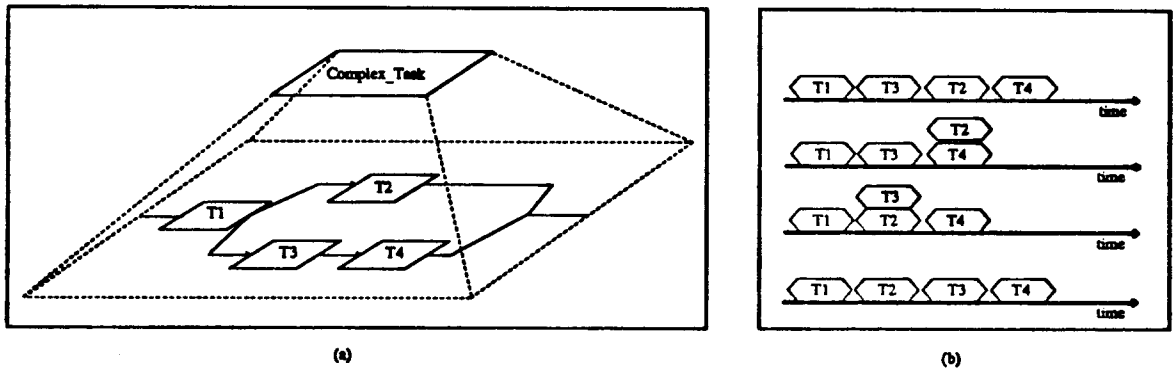


Figure 2.1 - Subtask Parallelism

activity manager. This data parallelism is outside the realm of task specification, which concerns itself with the method of performing those operations.

2.4. Task Invocation

In contrast to the active control used by the VLSI design systems discussed in the introduction, the task manager takes a more passive role with respect to task invocation. This decision is based on the view that design work is inherently novel and unpredictable. The designers continually test the limits of the tools by applying them in new ways for different purposes. This view guided my choice of which actions are appropriate for the system to manage and which properly belong to the designers. This is not to say that there isn't some benefit in having the system provide some form of advice to designers regarding the purpose of the tools. The task manager leaves this up to the individual who creates the task specifications. It is assumed that the designers will have a library of templates at their disposal which have been created by the tool suite manager.

In addition to the sequencing constraints on subtasks, a method of controlling task invocation is available through preconditions. A precondition could be, for example, that computation-intensive tasks may only be run after midnight, or that the task is available only to a restricted group of designers. The conditions must be based on information available to the system, of course, and must be quantifiable. In other words, the system cannot be asked to decide questions such as whether or not a design track is worth pursuing. Since each subtask is also an instance of another task template with its I/O and parameters predetermined, any preconditions associated with a primitive task remain in effect when that task is instantiated as a subtask.

Task invocation is carried out by the *Tool Navigator*, the part of the task manager responsible for navigating the designer through the sequence of tasks. Once a task has been selected by the designer, the Tool Navigator first checks for the existence of preconditions. If they are satisfied, the Tool Navigator obtains all of the information it needs, the actual inputs and outputs and the selection of parameters, from the designer via the graphical interface. If the task is primitive, the tool is invoked. If not, what happens next depends on the sequencing constraints on the subtasks.

Looking at the example in Fig 2.1, task T1 must be invoked first. However, that task may itself be complex. Assuming it to be primitive, the designer now has the option of setting additional tool parameters, if applicable, (other than those set explicitly in the specification) or immediately instructing the Tool Navigator to proceed with the task. Upon completion of T1, the designer is notified that both T2 or T3 are available for execution and prompted for the next action. Let's assume that T3 is selected. T2 is still available and the designer may wish to immediately proceed with its execution. T4 cannot be invoked, however, until T3 has completed, irrespective of the status of T2.

Some subtasks may be automatically invoked. Assume, for example, that T2 is a complex task for timing analysis. The first subtask in T2 invokes a translator to create the appropriate format for input to the timing analysis tool. The translation will always have to be done and its results, assuming successful completion, are not particularly interesting. There is no need for the designer to be involved in the decision to run the translation tool. It is a prime candidate for automatic execution.

2.5. Successful Task Completion

The concept of completion can be examined on more than one level. At the lowest level, completion of a tool means that it is no longer executing. At the other extreme, the designer may not be "done" with a tool until some criterion is satisfied. He/She may not be finished with the logic simulator until a particular bug is located, or done with the timing analyzer until he/she's satisfied that the valid critical path has been located, or done with the layout compactor until it's time to go home. Most tools return an exit code which can be examined for determination of success at that level. Beyond that, it's up to the designer to indicate completion of that step. For that reason, at the end of every task and subtask, the designer is given the option of invoking it again.

Furthermore, any task invoking editors, for both circuits and text, are specified as members of the *editing* class. The Tool Navigator requires designer notification of completion for these tools since editing may extend over multiple sessions.

Postconditions may also be specified for a task. These operate in a manner similar to preconditions and are subject to the same constraints, but also accept instance-specific condition parameters which may be used to describe some measure of success. For example, a layout task may invoke as its final subtask a statistics collector. The designer can supply a maximum area for the layout, which can be compared with the result for a decision on the success of the task.

2.6. Task Suspension

The model provides for suspending a running task for later resumption. The instance-dependent information obtained from the designer is stored in the database, as well as status information for the subtasks. The designer is prompted for a name for the instance by which the status is retrieved. The task can be resumed where it left off with very little overhead and no inconvenience to the designer. There is no limit on the number of times a task can be suspended.

Given the long runtime for many CAD tools, this capability is very useful, if not absolutely essential. It also convenient to be able to maintain the status while a designer is called away temporarily to work on another more pressing problem.

2.7. Error Handling

The automatic and correct handling of errors is an important issue in many management systems. This model, however, isn't designed for self-correction. Designers are working closely with the system and assumed to be much better qualified for handling problems that come up. If, for example, a tool hangs or thrashes, it's up to the designer to detect and correct the problem. The system isn't normally on "automatic pilot" and certainly doesn't become so in response to errors. Control originates in the designer for direction of the design and correction of problems.

The related issue of consistency is also relegated to the users of the system. It is assumed that designers working on parts of the design which interact are working closely enough with each other to communicate directly and not rely on system notification. The Activity and History Managers do furnish some support for the consistency issue, both by recognizing individual and group workspaces, and by providing the means for tracing the history of a design object.

2.8. The Template Manager

As described briefly in the introduction, the Template Manager handles all of the operations related to templates. This includes creation, copying, deleting, editing, and viewing. Copying and editing are available for the easy construction of new variations of a template. Editing a template can be used for replacing a tool with a different tool or a newer version. The change can be made to the primitive task and as long as the inputs, outputs, and parameters don't change, complex tasks which include this task won't need to be modified. If the parameters and I/O do change, the complex templates will need to be updated, of course. The grammar for the specification will be described in detail in a later section.

New templates can be created directly from a text file, or interactively via the graphical interface. Some semantic checks are done to locate specification errors as early as possible, before the template is added to the database. For example, a template cannot be both primitive and complex.

As well as interacting with the user to create templates, etc., the Template Manager retrieves instances of the templates requested by the Tool Navigator. The Tool Navigator also makes use of the Template Manager's access to the database for storing and retrieving task status information.

2.9. The Tool Navigator

The Tool Navigator supports three operations. Designers will use it for interactive guidance through the execution of a task, as described in the sections on invocation, completion, and parallelism, and for requesting suspension of a task, and tool suite managers and anyone else responsible for specifying the templates, can use the Tool Navigator for *previewing* a task, which refers to walking through a task without executing any of the tools.

For either the normal task execution or the previewing operation, after the Tool Navigator obtains an instance of a template from the Template Manager and fills in the inputs, outputs and parameters, some additional correctness checks are done. For example, the subtask sequencing is examined for conflicts with dataflow. This cannot always be done from the template directly since conditional statements may affect the I/O specifications (that will become more clear when the grammar is described). Also, when a complex template is specified, the the subtask information is not compared against the individual templates for consistency. That is done at the time the instance is created. Since these checks are done by the Tool Navigator, the previewing capability should be exploited at the time the template is created. Previewing is also helpful in defining new complex tasks that are combinations of existing tasks. Furthermore, designers will find this operation useful in deciding whether or not to invoke a task.

During execution of the task, the Tool Navigator maintains state information about the task and subtasks. While a task is Running, the subtasks are either **Ready, Running, Not Ready,** or have already **Ran.**

Fig. 2.2 shows the "Complex_Task" of Fig. 2.1 in the midst of execution. Subtask T1 has already progressed through the Ready and Running states. Subtask T2 is **Ready,** Subtasks T3 is **Running,** and Subtask T4 is **Not Ready** since T3 has not yet completed. If, after a subtask completes execution, the designer wishes to run the subtask over again,

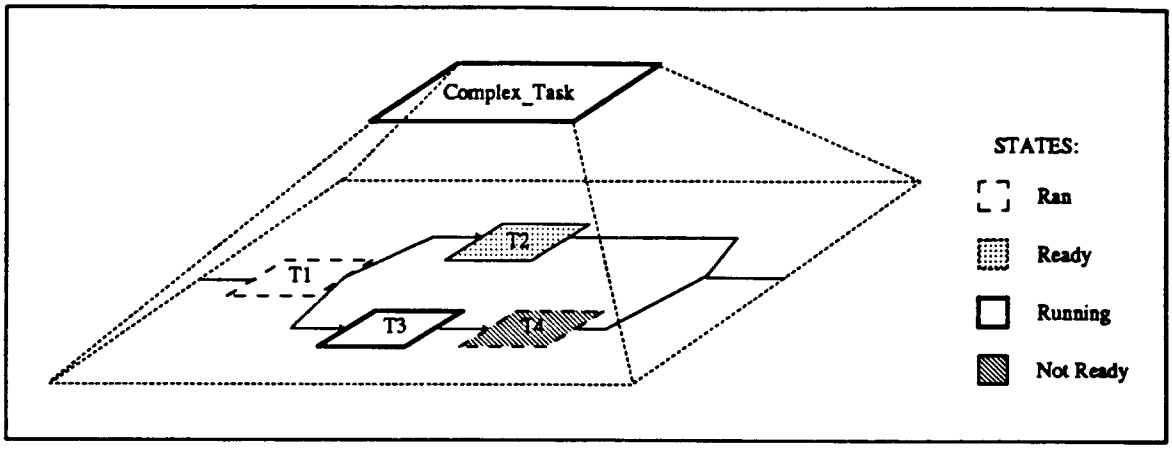


Figure 2.2 - State Model for Tasks and Subtasks

it remains in the **Running** state. When a task is suspended, the state of each subtask at that time is remembered. and a listing of the code are found in the appendices.

3. Prototype Implementation

The prototype implementation of the task specification and management model, *TMGR*, implements a subset of the model. Although the model is applicable to the VLSI CAD environment in general, TMGR is designed specifically for use with the U. C. Berkeley CAD tool suite and the Oct database. Briefly, this environment includes a data manager, Oct, a graphical browser and editor, VEM [HARR89], and a heterogeneous computing environment, RPC [SPIC87], as well as a wealth of CAD tools. The data representation in Oct is sufficiently flexible that task templates can be stored as easily as design data with no loss of information. At this time, the implementation of the Activity and History Managers is not yet operational, so TMGR operates independently.

In general the majority of the model is implemented by TMGR with the exception of the graphical user interface, and the precondition and postcondition specifications. Also, TMGR leads the designer through the tasks as discussed earlier, but doesn't actually invoke any tools, so automatic execution is not supported, and task completion information must be supplied by the designer.

The Template Manager functions of the creation of task templates and their storage in and retrieval from Oct are provided, as well as the display of existing templates. Lex and Yacc are used for parsing the specifications, which are checked for correctness and stored in the database. TMGR implements all of the Tool Navigator functions, including executing, previewing, and suspending and resuming a task, albeit with a crude user interface, and without automatic execution.

Although a graphical interface has not been implemented, the following section on the grammar specification includes examples of a proposed interface which is designed to fit into the VEM model with its hierarchical pop-up menus and dialogue boxes. It is also important that keyboard bindings and console window input be available for all commands, as is the policy with VEM-integrated tools.

3.1. Creation of a Task Instance

After a copy of the template is retrieved from the data, the following steps are taken to create an instance of the task. For both primitive and complex tasks the first step is get the designer's parameter choices and arguments, if applicable. On the basis of the parameter list, conditional statements can be resolved, which can result in additional input, output, parameter, and subtask specification. Then, input and output names are obtained from the designer, if applicable. And finally, subtask instances are built into a subtask tree.

Building the subtask instances involves a similar process, except that parameters and input and output names are obtained from the complex task rather than from the designer directly. Objects and files created and perhaps used by subtasks which don't correspond to input and output of the task itself, are given temporary names.

Identifiers used in a template are required to be unique only within a template. For complex tasks, subtask instance identifiers are explicitly mapped to subtask template identifiers. Thus the same identifier could occur in a template and in its subtask's template. Fig. 3.1 illustrates the hierarchical mapping. When this instance of *Complex_Task* was created, the actual input good_design was mapped to the internal input logic_spec, which in turn was mapped to the input design of its subtask *T4*. *T4* is itself a complex

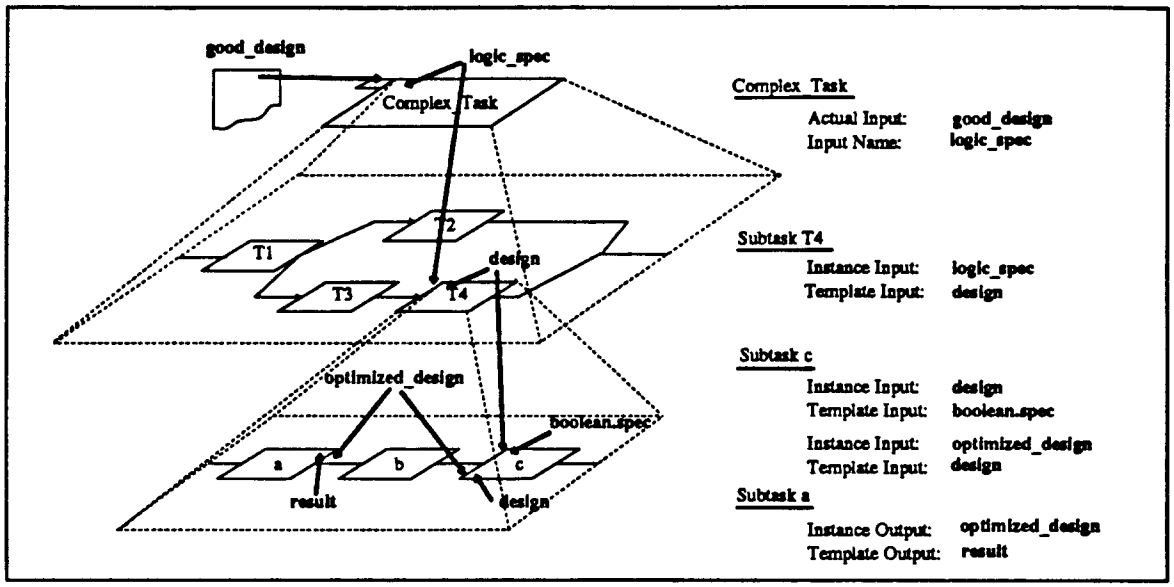


Figure 3.1 - Task and Subtask Input/Output Name Mapping

task and this input is mapped to **boolean_design** in *its* subtask *c*. *c* itself has another input, **design** which is mapped to *T4's* internal identifier **optimized_design.** This internal identifier serves to direct the output of one subtask to the input of another, without being visible outside of *T4*. Note that the identifier **design** is used both by subtask *T4* and its subtask *c* without being mapped to the same object.

To keep all of this straight, the instance created at runtime for each task and subtask has its own name table. Upward pointers link subtask template identifiers to the internal identifiers of the complex task which instantiated it. For example, the name table entry for the input **design** in the instance of subtask *T4* points to the entry for **logic_spec** in *Complex_Task's* name table.

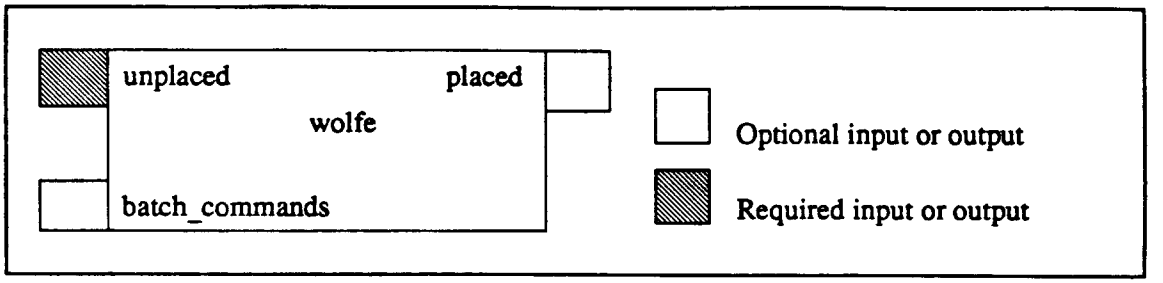
In addition, to avoid following a chain of links to locate the actual name of an input or output of a primitive task, the actual names propagate down the hierarchy. For example, the name table entry for **boolean_spec** in the instance of subtask *c* has a copy of the actual input, **good_design.**

3.2. Task Execution

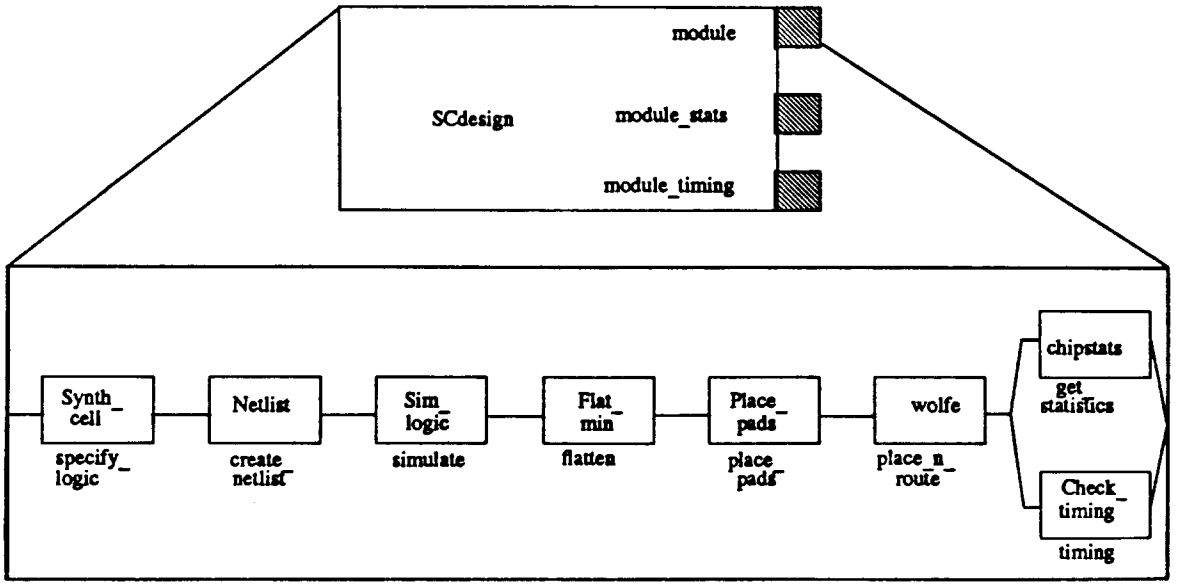
Once an instance is created, the execution routine loops through a series of steps until the task is completed, suspended, or aborted. Looking at the more interesting case of complex tasks, the first step is to find the subtasks which are ready to execute. If there is more than one, the designer is presented with the list and prompted for a selection. In general, once a subtask is in the Ready state, it remains there until it reaches the Ran state. *Optional* tasks, however, can require special handling. For example, assume that subtask *T3* in Figure 3.1 is optional, and that subtask *T1*has Subtasks *T2*, *T3*, and *T4* are all Ready. If subtask *T4* is selected to run next, subtask *T3* is now Not Ready. This special case of a subtask which is Not Ready and can never be ready, is detected and doesn't prevent the task from completing.

4. Grammar Specification

In this section I will describe the grammar for task specification in detail, as used in the prototype TMGR. Two examples will be developed to illustrate the grammar. Wolfe, a tool for placement and routing of standard cell designs, will illustrate primitive tasks, and SCdesign, a sequence of tasks for completing a standard cell design, which includes an instance of wolfe, will illustrate complex tasks. The illustrations will include both the ascii text format for specification, as implemented, and in some cases, a *proposed* graphical user interface.



(a) "wolfe" - Primitive Template Example



(b) "SCdesign" - Complex Template Example With Expanded View Showing Subtasks

Figure 4.1 - Graphical View of Example Templates

Fig. 4.1 (a) and (b) depict the graphical abstractions of these tasks. The small boxes on the right are inputs, and on the left, outputs. In (b), the subtasks of SCdesign are also

shown, which flow from left to right. The instance names of the subtasks are shown below the boxes, and their template names are shown inside.

4.1. Specification Format

The information contained in the template is organized as properties and property values, in s-expression format as outlined in Figure 4.2. The specification must begin with (**task** and be followed by the (**name** and (**class** properties, but the remaining properties may be specified in any order. Any individual task specification will not use all of these properties.

```
(task
        (name  task_name)
        (class class_name_list)
        (executable executable_name)
        (input
                input_descriptor_list)
        (output
                output_descriptor_list)
        (subtasks
                (ordering_constraint subtask_list))
        (parameters
                parameter_descriptor list)
        (cond
                condition_statment otherwise_statement)
        (purpose (string)))
```

Figure 4.2 - Task Specification Outline

Words shown in bold print are reserved words for the properties which cannot be used as identifiers in the specification. More reserved words will be added to this list as each property is discussed. The following examples will not use bold print since it is not used in an actual specification. The reserved words are recognized in both upper- and lower-case, or may begin with an upper-case letter, followed by lower-case letters. For example, task, TASK, and Task are all recognized forms for the reserved word. The newlines and indentation in Figure 4.2 are not required by the grammar, but are added for visual clarity.

Note that the graphical abstraction in Figure 4.1 doesn't portray all of this information, as it would be difficult to present in a clear manner graphically. The additional information contained in the template should be available through pop-up menus, however. The user could select *parameters*, for example, and view the parameters available for the task and those which have been set. For a complex task, *subtasks* could be selected and each individual subtask viewed in greater detail through the selection of properties in the menu.

Figure 4.3(a) shows the selection of the *Create* function from the Template Manager menu, and Figure 4.3(b) shows the secondary menu which lists the available

properties. Each property will be described in more detail in the following sections.

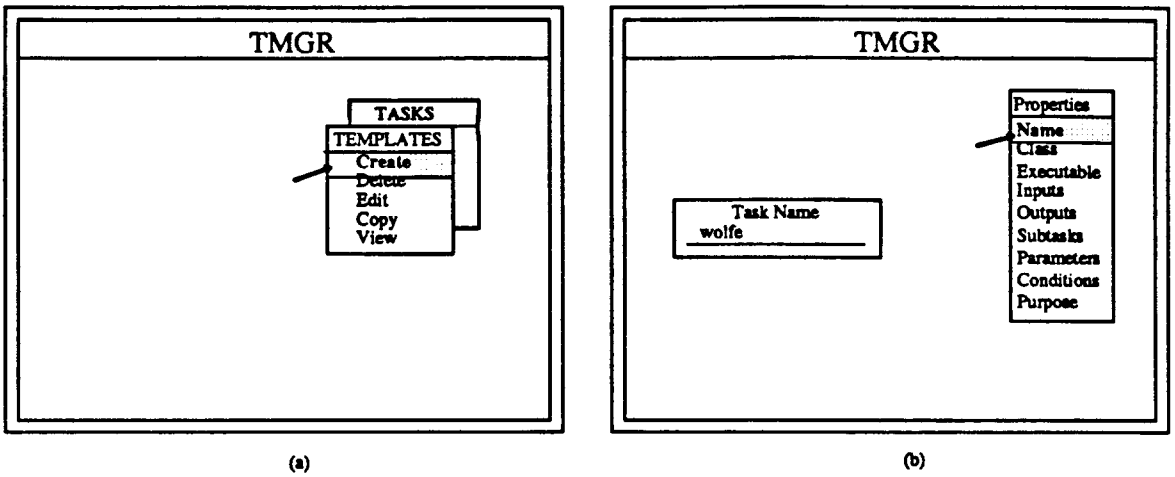


Figure 4.3 - Graphical Representation of Template Creation and Name Specification

4.2. Minimum Requirements

Name, class, and purpose properties must be specified for every task, or the Template Manager will report an error. In addition, primitive tasks must specify the name of the file that implements the tool as the executable, and complex tasks must have more than one subtask. In general, tasks will require more properties than these to be meaningful, but the Template Manager doesn't insist that they do.

4.3. Task Names

A *task_name* must begin with a letter and contain no spaces, parentheses, or double quotation marks, but may include any other printable ascii character. Names are case-sensitive. Also, names must be different from the reserved words shown thus far and in the remainder of the grammar description. These restrictions apply to all user-supplied identifiers in the specification. The name of a primitive task need not be the same as the name of the CAD tool it invokes.

In Fig. 4.3(b), the property name is selected from the property menu and the name *wolfe* is entered into the dialogue box.

The textual specification for our two examples is shown in Fig 4.4. In my examples of complex templates, the first letter of the name will be capitalized, to distinguish them from primitive templates, but this is not required.

```
• (task
        (name wolfe))
• (task
        (name SCdesign))
```

Figure 4.4 Example Name Specifications

4.4. Task Classes

Tasks are members of one or more of the following classes:

- **primitive:** A task which has no subtasks and which invokes a CAD tool.
- **complex:** A task which is defined in terms of complex or primitive subtasks.
- **editing:** A task which requires user notification that the session has completed.
- **utility:** A task such as printing or plotting for which history is not maintained.

Primitive and complex classes are mutually exclusive. Specifying a task as a utility plays no role in the task specification model but it informs the activity and history managers that they needn't capture this step. Editing tasks are singled out for two reasons. First, as discussed earlier, the Tool Navigator cannot determine when an editing session has completed. Secondly, editors are the only tools which modify the actual input rather than preserving it, as is the general policy of the system.

A *class_name_list* is a list of class_names separated by spaces or newlines. Class specifications are added to our examples in Fig. 4.5(a). In Fig. 4.5(b) examples of class specification are shown in which a task belongs to more than one class, and Fig 4.6(a) portrays the interactive method for class specification.

- (task
  (name wolfe)
  (class primitive))
- (task
  (name SCdesign)
  (class complex))

(a) Example Class Specifications

- (class primitive editing)
- (class complex utility)

(b) Additional Examples

Figure 4.5 - Class Specification

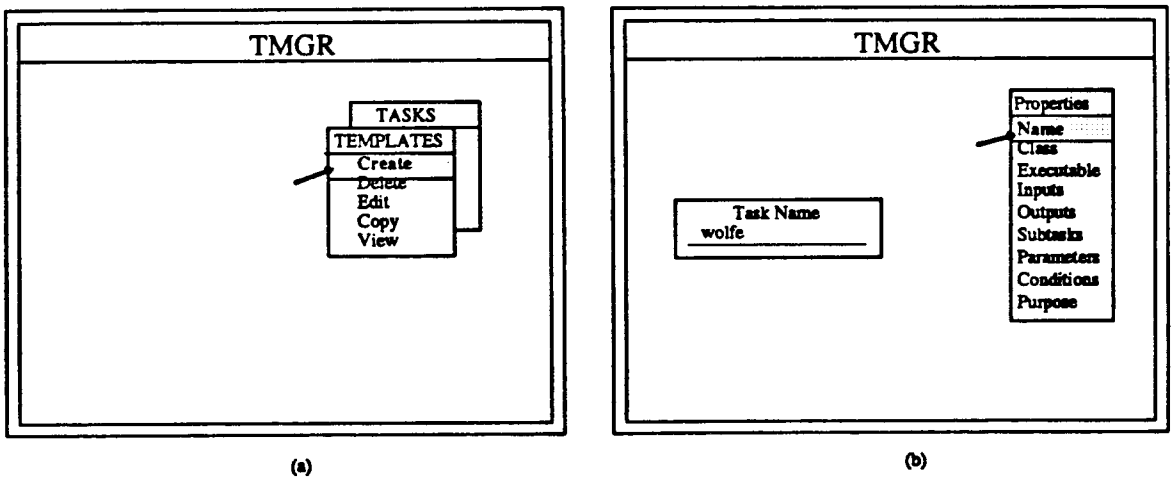


Figure 4.6 - Graphical Class and Executable Specifications

4.5. Executable

This section applies only to primitive tasks. Following the reserved word **executable** is the name of the CAD tool to be invoked. The *executable_name* should be in the appropriate case normally used to invoke the tool. Fig. 4.6(b) shows the interactive method for specifying the executable name, and Fig. 4.7 updates the primitive template example to include the executable.

```
(task
        (name wolfe)
        (class primitive)
        (executable wolfe))
```

Figure 4.7 - Example Executable Specification

4.6. Input and Output

The reserved words **input** and **output** are followed by lists of input and output descriptors. Inputs and outputs which are parameter-dependent are specified in the **cond** section of the property declarations.

Input_descriptor_list items and *output_descriptor_list* items are respectively of the forms:

((**name** *input_name*) (**type** *io_type*) (*input_property*))

((**name** *output_name*) (**type** *io_type*) (*output_property*))

An *input_name* or *output_name* is a one-word identifier; not necessarily the actual name of an object or file. The *io_type* classifies inputs and outputs into one of the following categories:

- **oct_name:** An Oct object which may be either physical or symbolic.
- **oct_physical:** A physical Oct object.
- **oct_symbolic:** A symbolic Oct object.
- **bdnet_text:** A text file in bdnet format.
- **bds_text:** A text file in bds format.
- **padp_text:** A text file in padp format.
- **blif_text:** A text file in blif format.
- **text:** A text file, including bdnet_, bds_, padp_, and blif_text types.
- **crystal:** A file in sim format, readable by crystal.
- **pla_format:** A file in Berkeley pla format.

4.6.1. Input

Inputs are those objects necessary for completing the task, which are not generated by a subtask. In other words, inputs are required objects which come from outside the task. Currently the only input_properties are **optional** and **primary**. The property **optional** is used for inputs which may or may not be needed, depending on how the designer uses the tools, but are not dependent on parameter settings. For example, he/she may wish to create a file of batch commands before running a simulator. That would be an optional input.

When a primitive task has more than one input, one of the inputs must be given the property **primary**. It is this input which is used in the command line invoking the CAD tool. Data created through *standard input* during the execution of a task is not considered to be task input since the task may begin without it. Instead, it is captured as interactive **script** output (see Sec. 4.6.2).

Fig. 4.8(a) illustrates the interactive method for input specification. Note that some of the properties listed in the input dialogue box are in italics. These properties are only for outputs and will be discussed in the next section. The input to wolfe is shown in Fig. 4.9; SCdesign has no inputs.

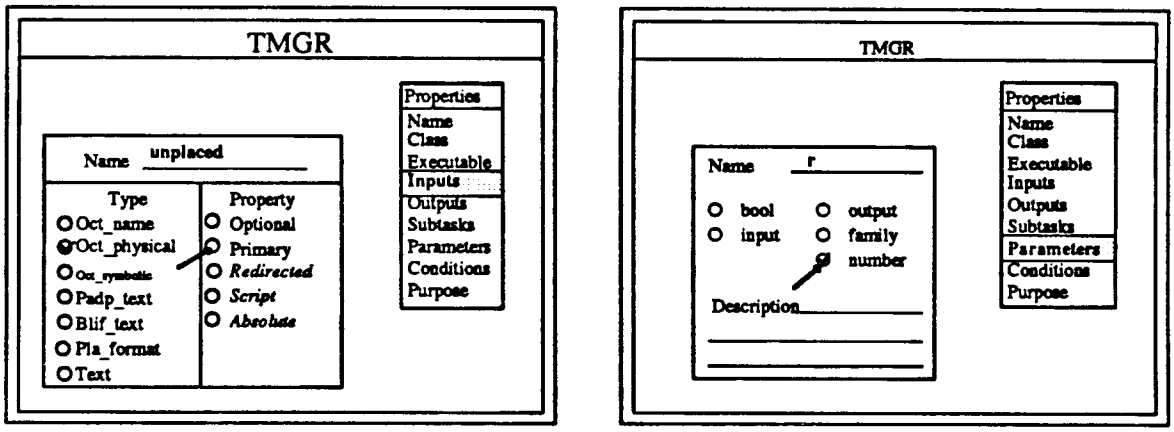


Figure 4.8 - Graphical Input and Parameter Specification

```
(task
        (name wolfe)
        (class primitive)
        (executable wolfe)
        (input
                ((name unplaced) (type oct_physical) (primary))))
```

Figure 4.9 - Example Input Specification

4.6.2. Output

What constitutes the output of a task is at the discretion of the designer of the task. It should include any Oct objects and files which are needed by other tasks. It should also include a transcript of interactive tool execution which would otherwise be lost to the history. For example, interactive execution of *bdsim*, a simulator, includes execution commands by the designer which should be captured. This type of output is included in the *output_properties* listed below:

- **redirected:** Output redirected from standard output.
- **script:** Output generated interactively and captured from the console.
- **absolute:** The name of the output which is some permutation input name. For example, if the input *infile* is instantiated with the actual input *mydesign*, the absolute output *"infile".sim* becomes *mydesign.sim*.

Fig. 4.10 lists the outputs for the examples. Additional files and objects are generated during the execution of SCdesign which are not considered to be output of the task in this example, but may be declared as such, at the discretion of the template designer.

```
• (task
        (name wolfe)
        (class primitive)
        (executable wolfe)
        (input
                ((name unplaced) (type oct_physical) (primary)))
        (output
                ((name placed) (type oct_physical))))
• (task
        (name SCdesign)
        (class complex))
        (output
                ((name module) (type oct_physical))
                ((name module_stats) (type text))
                ((name module_timing) (type text))))
```

Figure 4.10 - Example Output Specifications

4.7. Parameters

Parameters are used to modify the behavior of a task. For primitive tasks, these generally correspond to CAD tool switches. A *parameter_descriptor_list* item is of the form:

(*parameter_name* (type *parameter_type*) (descr (*string*)))

A *parameter_name* is a single, case-sensitive alphabetic character, which does put a limit on the number of parameters available to a task. A parameter's type is related to the existence and semantics of its argument:

- **bool:** A boolean switch; if listed, it's "on".
- **input:** A switch followed by an object name, separated by a space.
- **output:** A switch followed by an object name, separated by a space.
- **number:** A switch followed by an integer with no space.
- **format:** A switch followed by a list of format names enclosed in parentheses and separated by spaces.
- **family:** A switch followed by the name of a technology family, separated by a space.

The **input** and **output** parameter types are differentiated from the **family** type in that they must correspond to input and output specifications, which appear in the **conditional** section, and will be discussed later.

Parameter **descr** is optional. The *string* can be used to describe the use and purpose of the parameter. It must be enclosed in double quotation marks; no double quotation marks should be used within the string.

Fig. 4.8(b) shows the interactive method for specifying a parameter, and Fig. 4.11(a) adds parameters to the wolfe specification, all of which correspond to the tool's command-line switches. SCdesign has no parameters. Examples of parameter types not used in wolfe are shown in Fig. 4.11(b).

```
[wolfe] (parameters
            (f (type bool) (descr ("run in fast_mode")))
            (o (type output) (descr ("output oct:cell:view specifier")))
            (t (type input) (descr ("read commands from this file")))
            (r (type number) (descr ("number of rows")))
            (e (type bool) (descr ("extend pins beyond cell bounding box")))
            (h (type bool) (descr ("allow hierarchical input")))))
```

(a) Example Parameter Specifications

```
(parameters
        (t (type format (pla blif oct)))
        (f (type family)))
```

(b) Examples of Other Parameter Types

Figure 4.11 - Parameter Specifications

4.8. Subtasks

This property applies only to complex tasks, naturally. There are two aspects of subtask specification: constraining the sequencing order for all the subtasks, and specifying instance information for each individual subtask.

The two *ordering_constraints* available for implementing correct data flow and process-dependent sequencing are **seq** and **par**. Seq indicates that the subtasks must be completed sequentially in the order listed, while par indicates that the listed subtasks are sufficiently independent that any ordering is acceptable. These constraints may be nested to yield any desired ordering. A *subtask_list* item is either an *ordering_constraint* followed by a *subtask_list*, or a *subtask*.

The properties specified for a subtask are:

```
((name template_name subtask_instance)
        (subtask_property_list)
        (input input_name input_instance)
        (output output_name output_instance)
        (parameters (parameter_list)))
```

Whether a subtask itself is primitive or complex cannot be detected from the specification.

The general scheme for mapping identifiers used in properties of the subtask instance to the corresponding items in the template is the triplet:

```
(property template-identifier instance-identifier)
```

```
(task
        (name SCdesign)
        (class complex))
        (output
                ((name module) (type oct_physical))
                ((name module_stats) (type text))
                ((name module_timing) (type text))))
        (subtasks
          (seq
                ((name Synth_cell specify_logic)
                  (output oct_logic module_logic))
                ((name Netlist create_netlist)
                  (output oct_netlist module_netlist))
                ((name Sim_logic simulate)
                  (input design module_netlist)
                  (output sim_results sim_results))
                ((name Flat_min flatten)
                  (input hierarchical_logic module_netlist)
                  (output minimized_logic module_min))
                ((name Place_pads place_pads)
                  (input design module_min)
                  (output pads_placed module_placed))
                ((name wolfe place_n_route)
                  (input unplaced module_placed)
                  (output placed module)
                  (parameters
                        (o module)))
                (par
                  ((name chipstats get_statistics)
                        (input oct_cell module)
                        (output statistics module_stats))
                  ((name Check_timing timing)
                        (input design_file module)
                        (output timing_results module_timing))))))
```

Figure 4.12 - Example Subtask Specification

4.8.1. Subtask Names

Template_name refers to the *task_name* used in the task specification of the subtask. In Fig. 4.12, which specifies the subtasks for SCdesign, the first few template_names are "Synth_cell", "Netlist", and "Sim_logic". *Subtask_instance* refers to the name of this instance of the subtask. These names are not required to be different; nor are they required to be the same. They are useful for relating the subtask to its role in the task.

Some subtasks may be **optional**, and this would be listed as a *subtask_property*. At runtime the Tool Navigator interacts with the designer to determine whether or not an optional subtask will be run.

4.8.2. Subtask Inputs and Outputs

Input_name and *output_name* refer to the names of inputs and outputs used in template specification of the subtask. *Input_instance* and *output_instance* refer to the names of the inputs and outputs for this *instance* of the subtask. As with the subtask names, these names are not required to be different; nor are they required to be the same. Note that the reserved words **input** and **output** are used with each input and output. If a subtask input is optional and is not used in this instance, it should not be listed.

All required inputs and outputs for each subtask must be specified. In some cases, the designer of the task is not interested in saving an object which is created by one subtask and used by another and thus does not specify that object as an output of the task itself. To perform the mapping, however, an internal identifier must be created, and the subtasks' input and output mapped to it.

In SCdesign, for example, the output of the subtask "create_netlist" isn't mapped to any outputs of SCdesign, but is mapped to "module_netlist" which, in turn, is mapped to inputs of the subtasks "Sim_logic" and "Flat_min".

It may appear that the output of the subtask "specify_logic" is not used at all. However, it is assumed to be used by "create_netlist" without being specified explicitly. This must be understood by the designer of the task.

4.8.3. Subtask Parameters

The *parameter_list* is used to set parameters for the subtask. The format is

(parameter_name parameter_argument)

Parameter_names are restricted to one case-sensitive, alphabetic character [A-Za-z]. This corresponds to the informal standard in the Oct world, but the specification could easily be extended to accommodate CAD tools from other environments. The parameter_argument is only appropriate for some types of parameters, as discussed in the previous section.

Fig. 4.13 illustrates the interactive method for specifying some of the subtask properties. (As is clear by now, the graphical user interface proposal follows a similar format for each property, so no further interactive examples will be shown.)

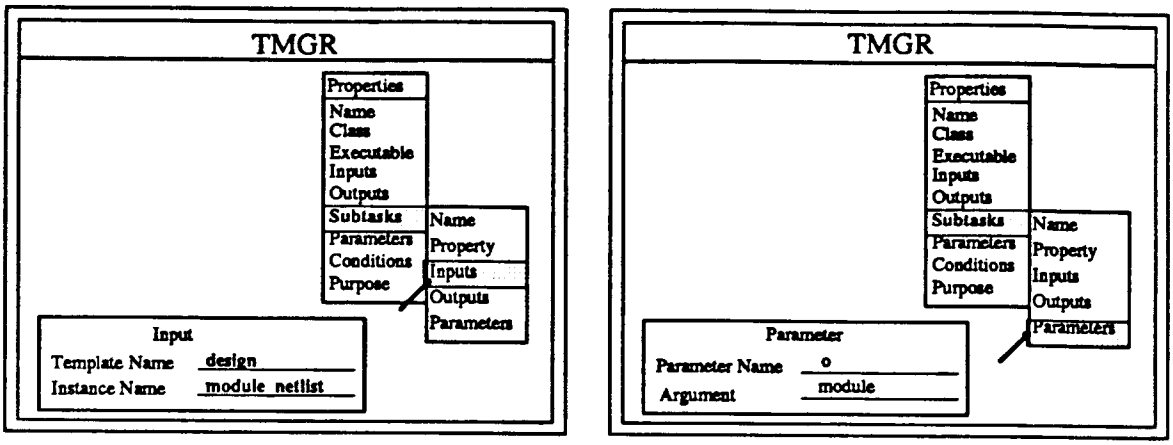


Figure 4.13 - Graphical Subtask Input and Parameter Specification

4.9. Conditional Properties

Conditional statements are used to specify additional properties based on parameters settings and arguments. Additional inputs, outputs, and parameters can be specified. Also, additional properties can be added to a subtask which is listed in the subtask specification section. Conditional statements cannot be used to nullify properties already specified. The only way in which the information in a conditional statement can affect the flow of subtasks is by adding the property **optional** to an existing subtask.

Conditional statements may be nested. The only condition tested is equality. If the condition evaluates to *true* the listed properties are added to the specification. If the condition does not evaluate to *true*, a default list of properties may be specified, subject to the constraints mentioned above. The format of a conditional is:

(**cond** *condition_statement otherwise_statement*)

The otherwise_statement is optional. It corresponds to the "ELSE" in in the commonly understood "IF-THEN-ELSE" structure. The format of a *condition_statement* is:

(*condition* (*task_property_list*))

in which the *task_property_list* format for inputs, output, parameters, and conditions is identical to that already described. The format for subtasks is the same as well, excluding the ordering constraints.

The format of the *condition* is:

(**eq?** *var_name instance_value*)

In Fig. 4.14, the parameters for wolfe which specify an input or output have a corresponding condition property. For example, the first condition specifies the input

"batch_commands" if the parameter *t* is set. Although not shown in this example, several properties can result from a true condition.

```
(task
    (name wolfe)
    (class primitive)
    (executable wolfe)
    (input
      ((name unplaced) (type oct_physical) (primary)))
    (parameters
      (f (type bool) (descr ("run in fast_mode")))
      (o (type output) (descr ("output oct:cell:view specifier")))
      (t (type input) (descr ("read commands from this file")))
      (r (type number) (descr ("number of rows")))
      (e (type bool) (descr ("extend pins beyond cell bounding box")))
      (h (type bool) (descr ("allow hierarchical input"))))
    (cond
      (eq? parameter t) (
        (input
          ((name batch_commands) (type text)))))
    (cond
      (eq? parameter o) (
        (output
          ((name placed) (type oct_physical))))))
```

Figure 4.14 - Example Condition Specification

A condition which is not nested must test for the setting of a specific parameter; in this case the *var_name* is the word **parameter**, and the *instance_value* would be the name of the parameter, e.g.:

```
(cond
      (eq? parameter a)
```

A nested condition may test for the setting of a specific parameter, or may test the value of a parameter, in which case the *var_name* would be either the name of the parameter, or the parameter type, for all but the **bool** parameters. In the latter case, the *instance_value* is the value being tested. In some cases, such as with a parameter of type format, the allowable choices for the *instance_value* are given in the task specification, e.g.; assuming a is of type format:

```
(cond
      (eq? parameter a) (
            (cond (eq? format oct)
```

For others, such as with a parameter of type family, it is up to the designer of the specification to provide a reasonable family name for the *instance_value*. If parameter a is of type number, for example, the following specification could be used:

```
(cond
        (eq? parameter a) (
                (cond (eq? a 2)
```

In the nested condition example shown in Fig. 4.15, if the parameter *t* is set, the format argument is tested against two possibilities, and the input properly specified accordingly, and if neither of them is set, the default input type is specified.

```
(cond
        eq? parameter t) (
                (cond (eq? format pla) (
                        (input
                                ((name pla_logic) (type pla_format) (primary)))))
                (cond (eq? format oct) (
                        (input
                                ((name oct_logic) (type oct_symbolic) (primary))))))
                (otherwise (
                        (input
                                ((name blif_logic) (type blif_text)(primary))))))
```

Figure 4.15 - Example of a Nested Condition

4.10. Purpose

This final, required property permits the task designer to indicate the purpose of the task. The string is subject to the same restrictions as the parameter-description string.

The purpose specifications for are our examples are shown in Fig. 4.16 (without reiterating the entire specification).

```
• [wolfe]       (purpose
                        ("Place and route a standard cell design"))

• [SCdesign]    (purpose
                        ("Complete standard cell design"))
```

Figure 4.16 - Example Purpose Specifications

5. Summary and Conclusions

Task specification and management plays an important role in the the activity model for the VLSI design process. This model allows the system to ease the burden on the designers of learning and using a multitude of complex tools by managing the low-level details and guiding the interaction of cooperating tools. Unlike other models which have been proposed or implemented, however, the system takes a passive role and is not involved in decisions affecting the design. The principles used in developing the model of the design process are applicable to design processes in general but the model itself is tailored to the VLSI CAD design environment, and the prototype *TMGR*, to the Berkeley Oct environment.

My guiding philosophy in developing the model was to determine what would make the design process easier for VLSI CAD designers. The task manager could be viewed as yet another tool to be mastered, and, if it required too much overhead for each step or were difficult to understand, would be rightfully ignored by designers. I believe the same fate awaits any design system which behaves too much like a black box. Designers want to know, and indeed, need to know what's going on, so as to correct and improve their use of the CAD environment.

Despite my bias toward the designers' concerns, task specification and management, especially when coupled with activity and history management, offer project managers the information they need to reach their objectives of resource management and maintaining realistic schedules. They can obtain direct data about the design process and progress rather than having to rely on the observations and reports of team members.

The elements in the model, e.g., the mechanism for specifying hierarchical task templates with sequencing constraints on the subtasks, for previewing, running, and suspending the tasks, and a graphical user interface, represent what I consider to be the minimum functionality needed for a task management system. Before other features are added, feedback from groups of designers using the system should be obtained and carefully considered. One area that could perhaps benefit from additional flexibility is the specification of subtask parallelism. Specifically, a feature such as an exclusive-or constraint, i.e., "choose only one of this group of subtasks", could be incorporated into the parallel-sequential constraints.

In conclusion, given that VLSI design is increasingly accomplished using growing suites of CAD tools, the need for a system to manage these tools will also grow. A manager such as TMGR can facilitate both the process of design and the management of the overall design effort.

6. References

[BTL81] "Source Code Control System User's Guide," *UNIX System III Programmer's Manual*, October 1981.

[BUSH89] Bushnell, M. and Director, S.W., "Automated Design Tool Execution in the Ulysses Design Environment," *IEEE Transactions on CAD*, Vol. 8, No. 3, March 1989.

[DANI89] Daniell, J. and Director, S.W., "An Object Oriented Approach to Distributed CAD Tool Control," *IEEE Proc. 26th Design Automation Conference*, (July 1989).

[DEME87] Demeure, I.M. and Osterweil, L.J., "What We Learn About Process Specification Languages, From Studying Recipes," *CU-CS-373-87*, Department of Computer Science, University of Colorado, Boulder, Colorado, (August 1987).

[FELD79] Feldman, S.I., "Make - A Program for Maintaining Computer Programs," *Software Practice and Experience*, April 1979.

[GABB87] GABBE, J.D. and Subrahmanyam, P.A., "An Object-Based Representation for the Evolution of VLSI Designs," *Artificial Intelligence in Engineering*, Vol. 2, No. 4, October 1987.

[HARR89] Harrison, D.S., "VEM: Interactive Graphics for Oct," Master's Thesis, University of California, Berkeley, 1989.

[KONS88] Konstantinow, G., "Emerging Standards For Design Management Systems," *Proc. of (COMPSTAN) Computer Standards Conf. 1988*, Washington, D.C., (March 1988).

[LEBL84] Leblang, D.B. and Chase, R.P., Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proc. ACM SIGPLAN/SIGSOFT Conf. on Practical Software Development Environments*, (April 1984).

[SILV89] Silva, M., Gedye, D., Katz, R.H., and Newton, A.R., "Protection and Versioning in Oct," *Proceedings of the Design Automation Conference*, pp. 264-269, June 1989.

[SPIC89] Spickelmier, R., editor, *Oct Tools Distribution 3.0*, University of California, Berkeley, CA, March 1989.

[SPIC87] Spickelmier, R., "RPC: Remote Procedure Call Package (For Use with VEM 5)," In Rick Spickelspmier, editor *Oct Tools Distribution 2.0*, University of California, Berkeley, CA, November 1987.

APPENDICES

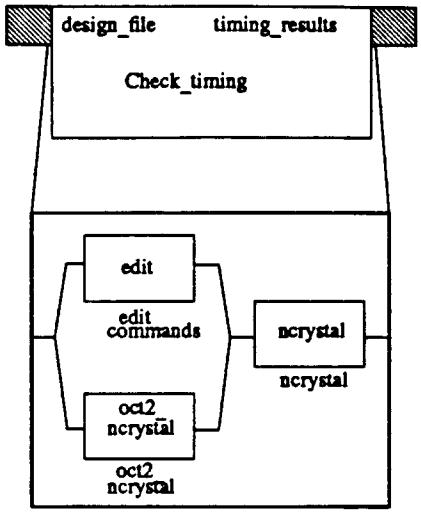
Appendix A: Technical Notes

- The template examples in Appendix B are based on the Oct tools as they existed during Fall 1988, since I did not have access to the current versions until the end of my project. The template "components" was included to provide a script for scanning bds and bdnet files to make component information visible to the history and activity managers. The script was never written and is used in the templates as an example only.

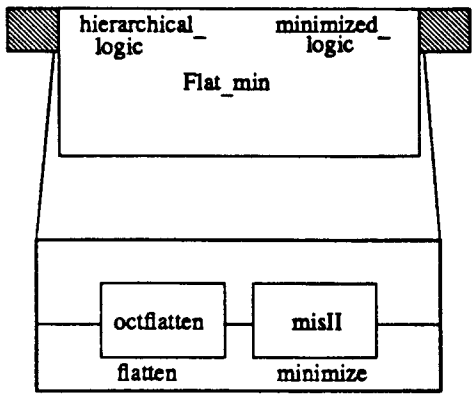
- The use of the Oct database itself uses the current version.

- TMGR assumes that a directory called "OCTDB" exists in the current directory for storing and retrieving the Oct templates.

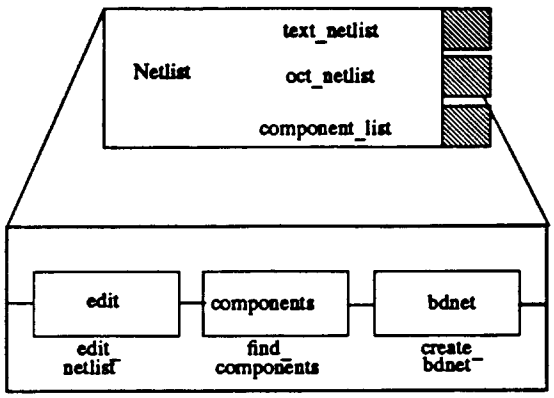
Appendix B: Template Examples



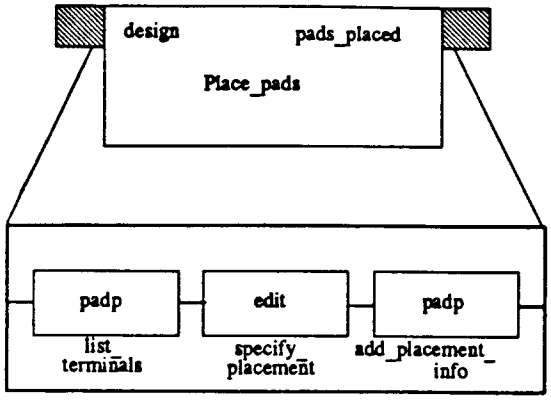

```
(task
        (name Check_timing)
        (class complex)
        (input
            ((name design_file) (type oct_name)))
        (output
            ((name timing_results) (type text)))
        (subtasks
                (seq
                   (par
                        ((name edit edit_commands)
                           (optional)
                           (output edit_file crystal_commands))
                        ((name oct2ncrystal oct2ncrystal)
                           (input translate_file design_file)
                           (output 'translate_file'.sim crystal_file)))
                   ((name ncrystal ncrystal)
                        (input crystal_file crystal_file)
                        (input crystal_commands crystal_commands)
                        (output timing_results timing_results))))
        (purpose
                ("Translate file to crystal, do timing analysis, save results")))
```



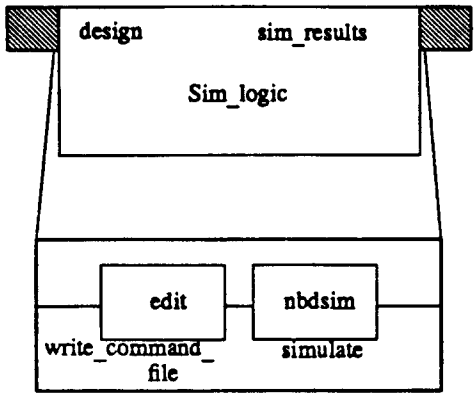

```
(task
        (name Flat_min)
        (class complex)
        (input
                ((name hierarchical_logic) (type oct_symbolic)))
        (output
                ((name minimized_logic) (type oct_symbolic)))
        (subtasks
                (seq
                  ((name octflatten flatten)
                        (input hierarchical_logic hierarchical_logic)
                        (output flat_logic flat_logic))
                  ((name misII minimize)
                        (input oct_logic flat_logic)
                        (parameters
                                (t oct)
                                (T oct)
                                (o minimized_logic))
                        (output oct_min minimized_logic))))
        (purpose
                ("Flatten hierarchical oct design and minimize")))
```



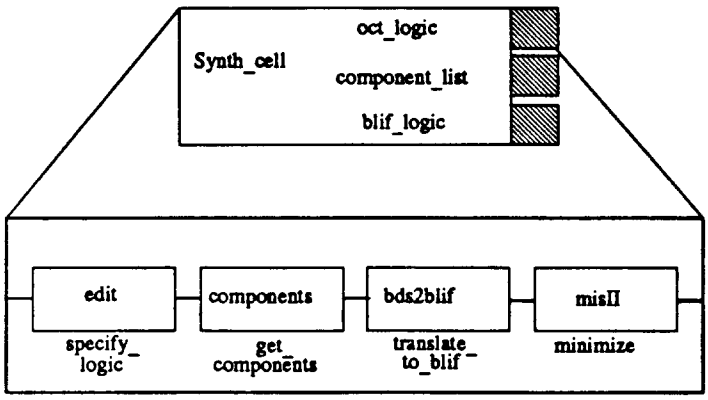

```
(task
        (name Netlist)
        (class complex)
        (output
                ((name text_netlist) (type bdnet_text))
                ((name oct_netlist) (type oct_symbolic))
                ((name component_list) (type text)))
        (subtasks
                (seq
                   ((name edit edit_netlist)
                        (output edit_file text_netlist))
                   ((name components find_components)
                        (input container text_netlist)
                        (output component_list component_list)
                        (parameters
                                (n)))
                   ((name bdnet create_netlist)
                        (input bdnet_logic text_netlist)
                        (input component_list component_list)
                        (output oct_netlist oct_netlist))))
        (purpose
                ("Create a netlist, find the components, and create the oct representation")))
```



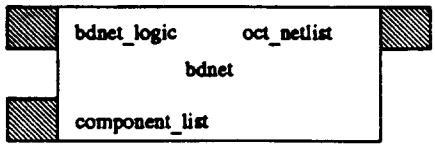

```
(task
        (name Place_pads)
        (class complex)
        (input
                ((name design) (type oct_name)))
        (output
                ((name pads_placed) (type oct_name)))
        (subtasks
                (seq
                   ((name padp list_terminals)
                        (input pad_logic design)
                        (output pad_list terminal_list)
                        (parameters (I)))
                   ((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 list of terminals, and specify their placement")))
```



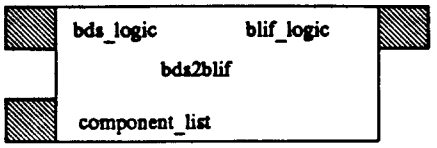

```
(task
        (name Sim_logic)
        (class complex)
        (input
                ((name design) (type oct_symbolic)))
        (output
                ((name sim_results) (type text)))
        (subtasks
                (seq
                   ((name edit write_command_file)
                        (output edit_file sim_commands))
                   ((name nbdsim simulate)
                        (input network design)
                        (input sim_commands sim_commands )
                        (output sim_result sim_results))))
        (purpose
                ("Simulate an oct logic network with an optional command file")))
```



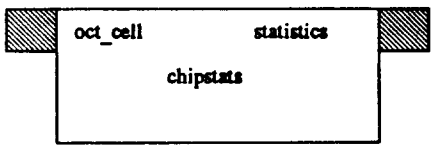

```
(task
        (name Synth_cell)
        (class complex)
        (output
                ((name oct_logic) (type oct_symbolic))
                ((name component_list) (type text))
                ((name blif_logic) (type blif_text)))
        (subtasks
                (seq
                   ((name edit specify_logic)
                     (output edit_file bds_logic))
                   ((name components get_components)
                        (input container bds_logic)
                        (output component_list component_list)
                        (parameters (l)))
                   ((name bds2blif translate_to_blif)
                        (input bds_logic bds_logic)
                        (input component_list component_list)
                        (output blif_logic blif_logic))
                   ((name misII minimize)
                        (input blif_logic blif_logic)
                        (parameters
                                (o oct_logic)
                                (t blif)
                                (T oct)
                                (f SCscript))
                        (output oct_min oct_logic))))
        (purpose
                ("Specify combinational logic; map to standard cells")))
```



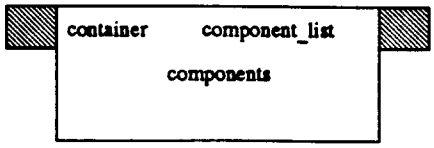

```
(task
        (name bdnet)
        (class primitive)
        (executable bdnet)
        (input
                ((name bdnet_logic) (type bdnet_text) (primary))
                ((name component_list) (type text)))
        (output
                ((name oct_netlist) (type oct_symbolic) (redirected)))
        (parameters
                (n (type bool) (descr ("write to standard output")))
                (c (type bool) (descr ("use with -n; show connectors")))
                (i (type bool) (descr ("print inverted netlist")))
                (s (type bool) (descr ("use with -i; for nets with 1 or 0 connections"))))
        (purpose
                ("Check for existence of components; create a netlist using bdnet")))
```



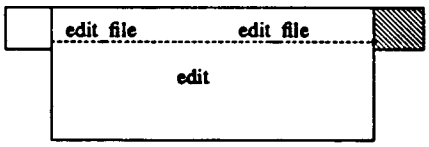

```
(task
        (name bds2blif)
        (class primitive)
        (executable bdsyn)
        (input
                ((name    bds_logic) (type bds_text) (primary))
                ((name component_list) (type text)))
        (output
                ((name blif_logic) (type blif_text) (redirected)))
        (parameters
                (b (type bool) (descr ("disable cleanup evaluation")))
                (c (type number) (descr ("specify amount of collapsing")))
                (n (type bool) (descr ("print table of variables assumed at logic 0")))
                (s (type bool) (descr ("change SELECTALL's to SELECTS's")))
                (u (type bool) (descr ("provide periodic updates")))
                (z (type bool) (descr ("assign DON'T CARES to logic 0"))))
        (purpose
                ("If all components exist, generate a logic network in blif format from the bds description, using bdsyn")))
```



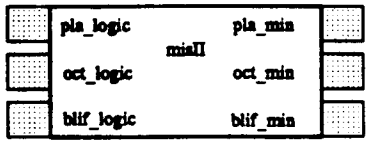

```
(task
        (name chipstats)
        (class utility primitive)
        (executable chipstats)
        (input
                ((name oct_cell) (type oct_physical)))
        (output
                ((name statistics) (type text) (redirected)))
        (purpose
                ("Collect statistics on an oct physical representation")))
```




```
(task
        (name components)
        (class primitive)
        (executable components)
        (input
                ((name container) (type text)))
        (output
                ((name component_list) (type text) (redirected)))
        (parameters
                (l (type bool) (descr ("input is a bds logic description")))
                (n (type bool) (descr ("input is a bdnet netlist"))))
        (purpose
                ("Search bds and bdnet descriptions for component cells")))
```



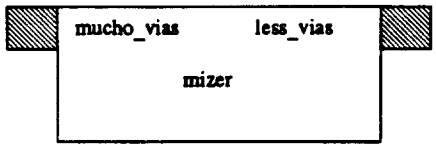

```
(task
        (name edit)
        (class editing primitive)
        (executable current_editor)
        (input
                ((name edit_file) (type text) (optional)))
        (output
                ((name edit_file) (type text)))
        (purpose
                ("Text editor as specified in environment")))
```



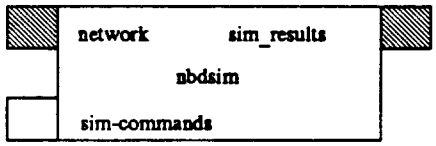

Input/Ouput Controlled
by Parameters

```
(task
        (name misII)
        (class primitive)
        (executable misII)
        (parameters
                (t (type format (pla blif oct)))
                (T (type format (pla blif oct)))
                (o (type output) (descr ("output object specifier")))
                (c (type family) (descr ("run in batch mode with commands from file")))
                (f (type family) (descr ("run in batch mode with commands from file")))
                (n (type bool) (descr ("interactive; do not read in network")))
                (x (type bool) (descr ("suppress final output")))
                (b (type bool) (descr ("run in batch mode"))))
        (cond
                (eq? parameter t) (
                        (cond (eq? format pla) (
                                (input
                                        ((name pla_logic) (type pla_format) (primary)))))
                        (cond (eq? format oct) (
                                (input
                                        ((name oct_logic) (type oct_symbolic) (primary))))))
                        (otherwise (
                                (input
                                        ((name blif_logic) (type blif_text)(primary))))))
        (cond
                (eq? parameter T) (
                        (cond (eq? format pla) (
                                (output
                                        ((name pla_min) (type pla_format)))))
                        (cond (eq? format oct) (
                                (output
                                        ((name oct_min) (type oct_symbolic)))))
                        (cond (eq? format blif) (
                                (output
                                        ((name blif_min) (type blif_text)))))))
        (purpose
```

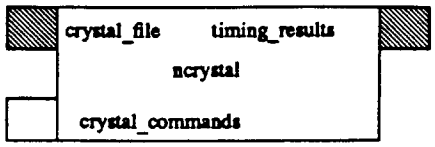
("Run misII")))



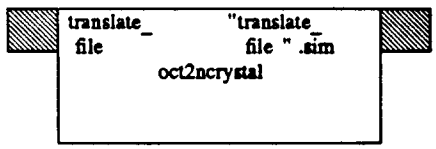

```
(task
        (name mizer)
        (class primitive)
        (executable mizer)
        (input
                ((name mucho_vias) (type oct_physical)))
        (output
                ((name less_vias) (type oct_physical) (redirected)))
        (purpose
                ("Minimize vias")))
```



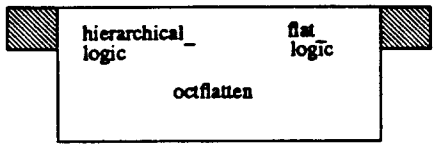

```
(task
        (name nbdsim)
        (class primitive)
        (executable nbdsim)
        (input
                ((name network) (type oct_name) (primary))
                ((name sim_commands) (type text) (optional)))
        (output
                ((name sim_result) (type text) (script)))
        (purpose
                ("Use nbdsim for interactive or batch simulation")))
```



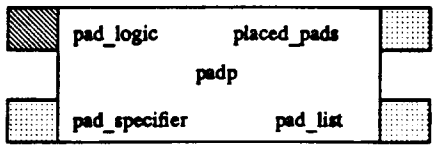

```
(task
        (name ncrystal)
        (class primitive)
        (executable ncrystal)
        (input
                ((name crystal_file) (type crystal) (primary))
                ((name crystal_commands) (type text) (optional)))
        (output
                ((name timing_results) (type text) (script)))
        (purpose
                ("Use ncrystal for interactive timing analysis")))
```




```
(task
        (name oct2ncrystal)
        (class primitive)
        (executable oct2crystal)
        (input
                ((name translate_file) (type oct_name)))
        (output
                ((name 'translate_file'.sim) (type crystal) (absolute)))
        (purpose
                ("Translate oct format to 'sim' format for ncrystal's use")))
```




```
(task
        (name octflatten)
        (class primitive)
        (executable octflatten)
        (input
                ((name hierarchical_logic) (type oct_name) ))
        (output
                ((name flat_logic) (type oct_name) (redirected)))
        (purpose
                ("Flatten hierarchical oct logic network")))
```




```
(task
        (name padp)
        (class primitive)
        (executable padp)
        (input
                ((name pad_logic) (type oct_name) (primary)))
        (parameters
                (l (type output) (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")))
                (S (type bool) (descr ("add constraints for sparcs")))
                (a (type bool) (descr ("place pads according to floorplan")))
                (c (typ    e bool) (descr ("place pads clockwise; order generated by oct")))
                (C (type bool) (descr ("use FLOORPLANNER's constraints")))
                (g (typ    e bool) (descr ("use as module generator")))
                (F (type bool) (descr ("add property MOBILITY with value FIXED")))
                (r (type bool) (descr ("place pads randomly")))
                (P (type bool) (descr ("tell padp chip is pad limited")))
                (v (type bool) (descr ("verbose output"))))
        (cond
                (eq? parameter D)(
                        (input
                                ((name pad_specifier) (type padp_text)))))
        (cond
                (eq? parameter l)(
                        (output
                                ((name pad_list) (type padp_text) (redirected)))))
        (cond
                (eq? parameter o) (
                        (output
                                ((name placed_pads) (type oct_name)))))
        (purpose
                ("Run padp")))
```