Planning and Managing Top-Down, Concurrent VLSI Design Processes

Margarida F. Jacome and Stephen W. Director

EDRC 18-54-95

Planning and Managing Top-Down, Concurrent VLSI Design Processes*

Margarida F Jacome
Electrical and Computer Engineering Dept.
University of Texas at Austin
Austin, TX 78712

Stephen W. Director
Electrical and Computer Engineering Dept.
Carnegie Mellon University
Pittsburgh, 15213

In this paper we discuss the general strategies typically embedded in top-down design methodologies to control design complexity, and show how they can be captured and implemented in a design process planning and management meta-tool. We also desct-ibe such a tool called Minerva. Minerva generates a high-level, problem-based rep-resentation of the entire design process and, based on this representation, provides a set of planning and management services that is sufficient to fully support top-down, design. Such services include: plan generation; plan execution; automatic problem reformulation (i.e., decomposition); support of backtracking for redesign and for problem re-definition; dynamic sharing of design information among designers coop-erating in the same design process; tracking of requirements throughout the various levels of abstraction traversed during the design process; and effective handling of sub-problem interactions in all of the above situations. All of these services are offered assuming the most complex scenario, i.e., a concurrent, distributed design environment.

KEYWORDS: Design, Planning, Management, Top-Down, Methodology, CAD.

1 Introduction

In spite of advances in integrated circuit technologies that allow for higher performance, greater densities, and increasing system complexity, the top-down design paradigm remains virtually the same. In particular, designers primarily employ "point" CAD tools to solve specific design sub-problems, from architectural design to placing and routing of wiring on a chip. To a certain extent, the designers are then responsible for coherently integrating the results across the levels of hierarchy, from the basic devices to the final system. The complexity of such top-down design processes is becoming unmanageable, though — it is not uncommon today to have digital integrated circuits

*** This work is supported in part by the Engineering Design Research Center, Carnegie Mellon University, under contract no. EEC- 8943164.**

(i.c.'s) with several million transistors operating at clock rates of a few hundred mega-Hertz. Changes introduced while addressing one design sub-problem may propagate to other design sub-problems, and across abstraction levels, leading to difficulties in tracking the history, and even the current state of the design process, and ultimately, in performing a coherent exploration of the typically huge solution space. We have thus reached a point where we need to move beyond the development of CAD tools that only aid designers in solving specific synthesis, analysis, and/or optimization problems, to CAD tools that also aid designers in planning and managing the increasingly complex top-down design process itself.

While there has been some research that has focused on the management of data and relatively low complexity design tasks, the issue of true design process management has only very recently been addressed. [1][2] And only limited attention has been paid to design process planning and management. [2] [3] We believe that this has been so because the successful development of high-level, effective design process planning and management services required a formalism for articulating the essential concepts of design and for modeling both the *design artifacts* and *the process ofdesign* across multiple design disciplines, in a coherent and precise way. In [2] [4] we proposed such a design formalism, which allows for a complete and general characterization of design disciplines and for a unified, problem-based representation of design processes. This formalism has been used as the basis for *Minerva,* the design process planning and management meta-tool to be discussed in this paper. As will be seen, Minerva is capable of effectively supporting arbitrarily complex VLSI design processes.

The remainder of this paper is organized as follows.We begin in Section 2 with a discussion on the specific strategies used in top-down design methodologies to control design complexity, and we identify the planning and management needs resulting from the application of such strategies during the design process. Then, in Section 3 we introduce the Minerva Design Process Planning and Management meta-tool. We discuss Minerva's capability for directly implementing top-down design methodologies and we show how a designer, working on a design process planned and managed by Minerva, performs his/her design activity. Important aspects of Minerva's knowledge representation structures are discussed in Section 4.We finish with some conclusions in Section 5.

2 Planning and Management Issues in Top-Down Design

In this section we briefly describe some of the principle issues to consider while planning and managing complex, top-down VLSI design processes. Among these are:

- methods for *problem decomposition* during active problem solving, while guaranteeing that *consistency* is preserved;
- *backtracking* both for *redesign* and for *problem re-definition;* and
- effective information sharing and handling of sub-problem interactions particularly in concurrent design processes.

Top-down design methodologies generally employ three classes of problem decomposition techniques, namely *abstraction, structural decomposition* and *behavioral decomposition,* to control the design complexity. Let us briefly consider each of these.

Through abstraction, the representation of the object under design is simplified and becomes less detailed.[5] In complex design domains, various abstraction levels, representing the design object with an increasing level of detail, are typically defined. "Register-transfer level" (RTL), "logic level," and "circuit level" are typical of the abstraction levels defined for VLSI digital circuits. Through the use of abstraction, designers focus on only on those design issues that are relevant to the simplified representation, or *view,* of the design object. After appropriate decisions have been made for a particular simplified view does the designer consider the next lower level of abstraction, viz. consider the issues associated with a more detailed view of the design object. Thus, while designing an "adder," the engineer might concentrate on the "logic view" of the "adder," and address a set of "logic-level" design issues., such as choosing an appropriate "logic style" (e.g., carry lookahead, manchester carry, etc.) in order to meet a set of performance and cost requirements. More detailed design decisions, such as the specific dimensions of the transistors in the adder, are determined when the "circuit view" of the "adder" is considered.

Structural and behavioral decomposition, on the other hand, transforms an object under design into a set of *less complex components* or sub-objects, that are to be *individually* designed. For instance, a "processing unit" might be structurally decomposed into an "arithmetic logic unit," a set of "registers," and a "control unit." Alternatively, the behavioral description of this same "processing unit" (possibly written in C or Verilog) might be partitioned into smaller chunks of behavior (or "tasks"), thus leading to the behavioral decomposition of the original design object into a collection of sub-objects, each of which would implement a disjoint sub-set of the resulting "tasks." (More examples of structural and behavioral decompositions are provided in Section 3.4)

During top-down design, abstraction, structural and behavioral decomposition techniques are recursively applied to the *design problem* resulting in a collection of *design sub-problems* each of which is "solvable" with the available CAD tools. In general, design problems are only *nearly decomposable,* viz. after decomposition constraining relationships will exist among the sub-problems. For instance, if there is a maximum "area" specified for a particular chip, the sum of the "areas" allocated for all of the components on the chip must not exceed this value. As the design process progresses, if the "area" allocated to one of the components on the particular chip must be increased, then the "area" allocated to the remaining components must decrease. If these sub-problem dependencies are not adequately taken into account, inconsistent sub-problem solutions may be generated and *backtracking* (returning to a previous design state and reconsidering a previously solved design sub-problem) will be required to restore consistency. The later in the design process that such inconsistencies are detected, the higher the cost of backtracking.

This issue becomes even more important when the pressure to decrease time to market motivates the concurrent solution of design sub-problems by different designers. While the desire to achieve shorter design cycles is the primary motivation for adopting *concurrent design,* a concurrent design process in which sub-problem interactions are not properly handled may be less effective than a more conventional, sequential design process. This is so because the team of designers working concurrently may end up trapped into endless backtracking-redesign loops. Thus, being able to guarantee that consistent sub-problem solutions can be generated *simultaneously* by all of the designers engaged in the design process is absolutely crucial for realizing the potential of concurrent design for reducing design cycles.

Another equally important issue in effectively planning and managing complex top-down VLSI design processes is the support of *backtracking,* either for *redesign* (for re-accessing and modifying previous design decisions), or for *problem re-definition* (for exploring different trade-offs for the design object's requirements). Backtracking for redesign may be necessary because the *heuristics* used to guide the search for a satisficing solution Mo not always work. Specifically, in certain situations, heuristics may lead designers to make inappropriate design decisions. For instance, while working with a simplified view of a design object, the engineer may choose a specific circuit topology, for its simplicity and thus small 'area," expecting that it will meet the remaining requirements. However, it may turn out that the chosen topology leads to a design object that fails to meet another specific requirement, such as "speed," and/or "power." When backtracking for *redesign* occurs during the design process, i.e., when a designer decides to revisit a previous design state in order to reconsider previous design decisions that have proven to be inadequate, he should be provided with all potentially *relevant* information being generated outside the scope of the particular sub-problem where the failure was first perceived. Furthermore, the team of engineers working concurrently on sub-problems that may be impacted by the new decisions must be properly notified, and the design state must be properly updated to reflect the effects of the backtracking step. (For instance, if a particular topology is abandoned, the design of its components may become useless.)

Backtracking for *sub-problem re-definition,* on the other hand, may be necessary because there is never a guarantee that for every design problem there will always be a satisficing solution. It may happen that during the course of the design process an engineer may wish to redefine a design sub-problems by altering one of the requirement values. For instance, it may be necessary to increase the "area" allocated to a particular component of a chip. Backtracking for problem re-definition is achieved by allowing designers to return to the design state(s) where the specific set of requirements *was first stated,* in order to undertake the exploration of alternative trade-offs that may be possible for a particular set of requirement values. Developing alternative trade-offs may benefit from taking into account the context of several sub-problems, and may even benefit from negotiation among design team members.

I. The typical situation in design is to look for good or satisfactory solutions, as opposed to optimal ones. A satisficing solution is thus one that meets the problem requirements, typically stated in terms of some figures of merit of performance. [6]

3 The Minerva Design Planner and Manager

In this section we introduce the Minerva design planning and management meta-tool. Minerva employs a uniform, problem-based representation of the entire design process that encompasses all levels of abstraction traversed during the design process. This representation allows Minerva to capture all relevant data and control information from the initial, conceptual phase of design through the final manufacturing-oriented design phase. Through the use of this global view of the design process, individual members of a team of designers involved in a concurrent design process can, at any time, quickly grasp the status of the entire design. Moreover, individual designers can use this global view of the design process to identify the set of design sub-problems available for solution and be confident that all design information relevant for the solution of these design sub-problems will be provided to them at the appropriate time.

Minerva does not substitute for, or in any form preclude, the use of "point" CAD tools. Rather, Minerva directs designers (engaged in solving synthesis, verification, and optimization design sub-problems) to appropriate CAD tools, and presents the results produced by these tools to the designer within a sound, integrated representation of the particular design process. We begin this section by briefly describing Minerva's architecture and then follow with definitions of some of the key concepts employed by Minerva. Throughout this discussion we use the *hardware/software (h/s) codesign* discipline for illustration. [7] We chose this discipline in order to demonstrate the generality of Minerva in the sense that it is not limited to the domain of single VLSI chips, or even to the design of only hardware.

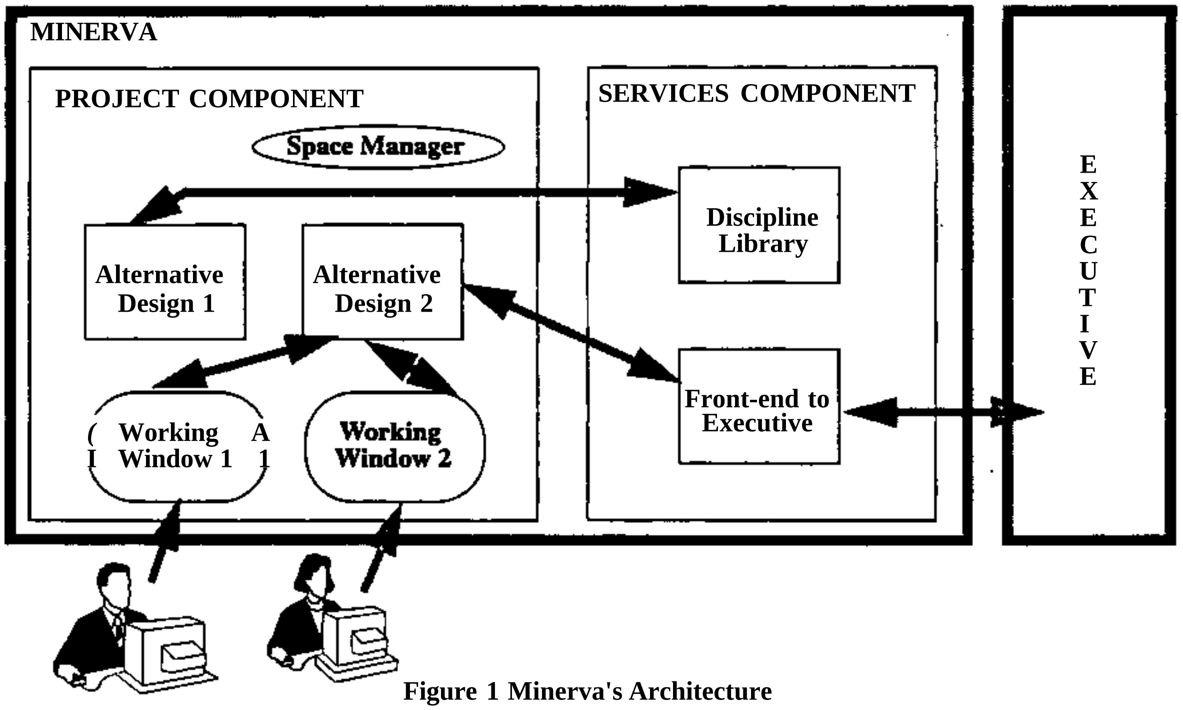


Figure 1 Minerva's Architecture

3.1 Introduction to Minerva's Architecture

Minerva, as illustrated in Figure 1, is comprised of one or more "project components" [2] and a "services component," and is designed to interact with an executive system that is responsible for actually executing design activities. Each *project component* represents an on-going design process. The *services component,* which contains the *discipline library* and a *front-end to the executive,* simultaneously serves all project components.

The original design problem (in the context of a given project) is designated as the *principal project problem.* A *project* contains all of the *alternative designs* that are simultaneously being developed as candidate solutions to the principal problem. A set of *working windows* support, and simultaneously control, the designer's participation in the *concurrent design process* that is being planned and managed. In other words, there is one working window for each designer actively working on any of the project's alternative designs. The discipline library, which contains information on the design disciplines of interest for a given design environment, allows Minerva to be customized to different design disciplines, and to different design methodologies in the context of these disciplines. (The specifics of this library will be briefly discussed in Section 5.) Figure 1 illustrates the situation in which there are two designers simultaneously working on the "alternative design 2" of a specific project.

Minerva uses the "front-end to the executive" sub-component to communicate with an *existing* executive that controls the set of "point" CAD tools and all other resources available in the particular design environment. Such an executive may be a task or workflow manager, such as [8] or [9].[3] (We discuss the interaction of Minerva with the executive in some detail in Section 4.)

3.2 Design Objects

The goal of a design process is the realization of an artifact, called a *design object,* that performs in some desired way. Design objects are characterized in terms of *collections of properties,* also called a *specification.* There are three basic types of properties: *descriptions, requirements,* and *restrictions. Requirements* are those properties that the design object must exhibit in order to be considered a satisficing solution to a design problem. These are usually specified prior to the start of a design process and are therefore considered the "givens" of the design problem. *Restrictions* are the properties "imposed" by the designer through decisions made during conceptual design for the purpose of pruning the solution space and enhancing design efficiency. Finally, *descriptions* are properties that *completely* characterizing the behavior and/or the structure of a design object, at a particular abstraction level. (Sometimes, behavioral descriptions are also provided in the original design problem, in which case they are also considered as "givens" of the problem.) Typically, these different property types are considered in distinct phases of the design process, as discussed below.

2. For simplicity, only one of such project components is shown in Figure 1.

3. Observe that a key aspect in implementing a meta-tool like Minerva consists of making the best possible use of technology already in place, namely in the areas of CAD task and workflow management.

Properties can be further partitioned in terms of their relevant levels of abstraction and each of these partitions corresponds to a possible *facet* (or abstract view) of the particular design object. Figure 2(a) illustrates a set of properties grouped into two facets (algorithm and register-transfer levels) of a design object that belongs to the class of "processing elements". (A *class* of design objects is a category of design objects that can be organized into a *discipline hierarchy[2]* [4] which is contained in Minerva's discipline library. More on this later.)

3.3 Design Problems

In general, the current state of a design process can be represented in terms of a *hierarchy of design sub-problems.* Each design sub-problem has three fundamental components: a *set of properties* (that may belong to the same or to different design object facets); an *objective* (the particular activity, e.g. synthesize, simulate, etc.) to be undertaken in the context of the previous set of properties; and a *set of consistency constraints,* which express relations among properties contained in the set of properties. Since some of the properties contained in these consistency constraints may not belong to the property set of the problem (see example below) we can say that consistency constraints "tie" together (in most cases *weakly)* the solution spaces for the design sub-problems generated during a design process.

Figure 2(a) illustrates the elements of a design problem. In this problem, the objective is *behavioral synthesis.* The goal of behavioral synthesis is to generate a new property value (i.e., a *behavioral description,* for the *register-transfer level* facet of the *processing unit),* from (or consistent with) the set of existing property values (i.e., *behavioral description, area, speed,* and *cost,* specified at the *algorithmic level* facet of the *processing unit).* Figure 2(a) also illustrates a consistency constraint, denoted by *Relation 1,* relating these properties. A second consistency constraint is partially shown, denoted by *"Relation* 2", that relates the *area* requirement of this particular *processing unit* and the *area* requirements of other design objects instantiated in the design process. (Note that these design objects are not explicitly shown in the figure). Other properties and consistency constraints would certainly exist in the context of the particular design object shown in Figure 2(a) yet, for the purpose of clarity we omit them.

Unfortunately, representation of design problems as illustrated above can become fairly involved making them difficult to deal with. This is particularly true for complex design processes that require the simultaneous representation of numerous sub-problems, as discussed in the next section. In order to make design problem representation more accessible to the designers, we can abstract it further, by "hiding" properties inside design object facets, as illustrated in Figure 2(b). This results in a more simplified view of a design problem, where design objectives are shown "connected" to specific facets of design objects, instead of being directly connected to sets of properties. (Observe that each design objective is still associated with the set of properties that is relevant for its achievement, we just do not show this association explicitly.). Since properties are no longer immediately visible in this simplified representation, the consistency constraints relating these properties are also not immediately visible.

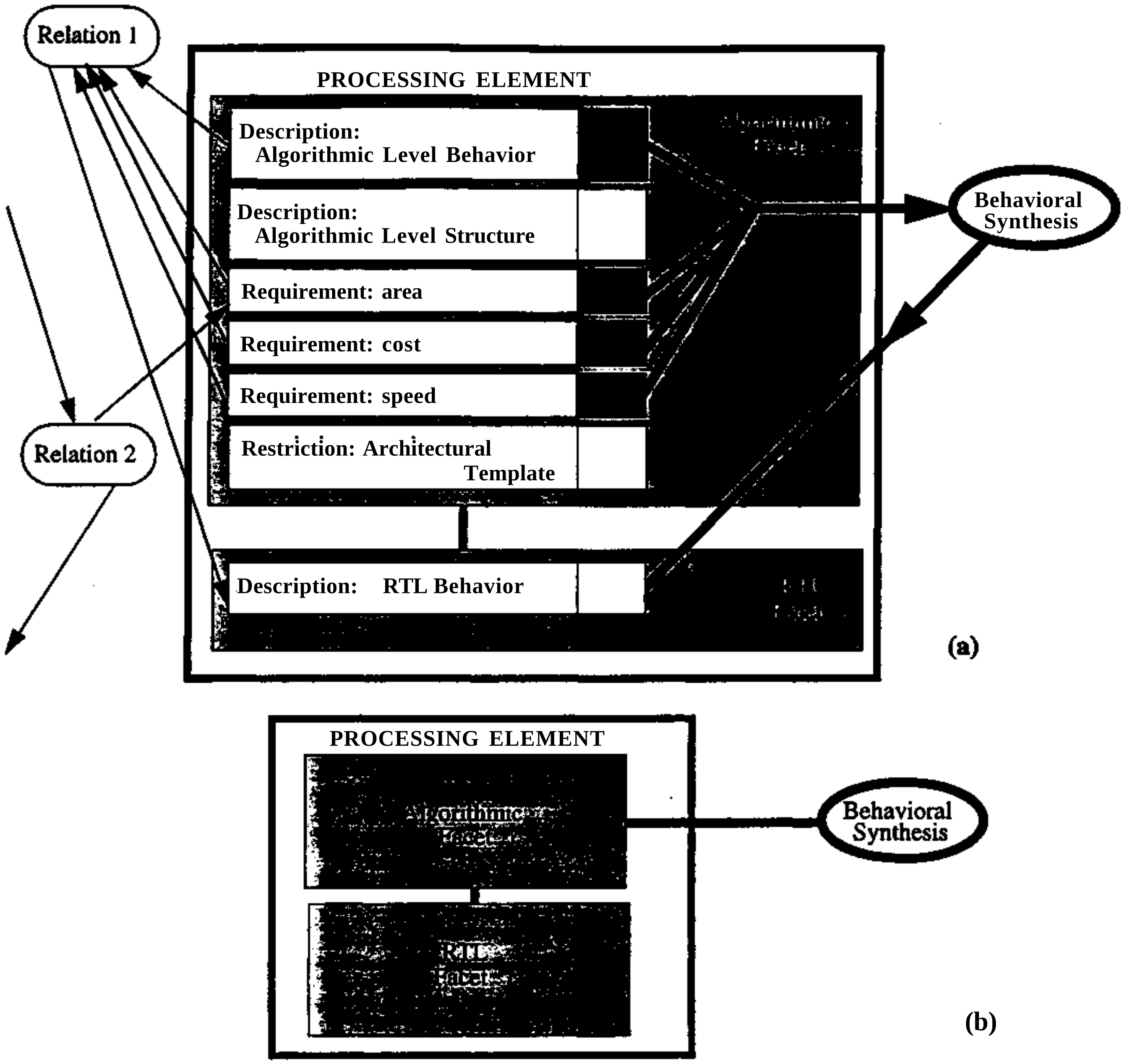


Figure 2 A Design Problem, (a) Complete view (b) Simplified view.

To summarize, a design problem in this simplified view, is uniquely determined by a particular design object facet and by an associated design objective. Properties and consistency constraints become visible to the designer only when the particular sub-problem is addressed.

3.4 Problem Decomposition

Decomposition is a key strategy employed by designers to transform complex design problems into simpler design sub-problems that can be solved. In this section we discuss the symbolic representation of three kinds of decomposition: *behavioral decomposition, structural decomposition,* and *transformational synthesis,* each of which can be performed across the abstraction levels defined for a discipline. To facilitate this discussion, consider Figure 3 which shows the data abstractions that might have been generated at an advanced stage of a h/s codesign process. In this figure, design objects are shown symbolically as boxes drawn in bold, while design object facets are shown symbolically as regular boxes drawn inside the design object box. Observe that for the design state illustrated, all design objects have just one facet, except for the design object represented by shaded region (e), which has two facets. In order to make the discussion that follows clearer, some properties (behavioral and/or structural descriptions) are explicitly shown inside the design object facets. Structural and behavioral decompositions are shown symbolically by bold lines relating parent and descendent facets of design objects. (Actually, those constitute the only type of consistency constraints explicitly shown in this simplified view.) So, in Figure 3 we see that the h/s codesign methodology comprises several (behavioral and structural) decompositions made for the target design object, as indicated by shaded regions (a), (b), (c), and (d). In what follows we briefly discuss the context in which these decompositions take place.

We assume that, at the beginning of the h/s codesign process, that the designer has descriptions for the desired behavior for a set of processes, denoted by ∑P. These descriptions, usually written in C or Verilog constitute the behavioral description for the principal object under design. Observe that, as shown in the upper left corner of Figure 3, this principal design object is of the class "processing element" and was first instantiated at the algorithmic level of abstraction. Further, a *behavioral decomposition* is implicit in this initial collection of processes, since they are "non-atomic" elements contained in the behavioral description of the principal design object.[4] Each process in ∑P is thus represented as the behavioral description of a new design sub-object, as shown in region (a).

Each of these original processes is then further 'decomposed' into a collection of tasks, or smaller chunks of behavior. This new set of behavioral decompositions is symbolically represented in shaded region (b). The collection of tasks derived for the entire set of processes is denoted ∑T.

Through further analysis, these tasks are then partitioned into a new set of processes, labeled]^P» some of which will best suited to be realized in software while others will be best suited to be realized in hardware. Thus ^P is a new behavioral decomposition for the main design object, as indicated in shaded region (c).

After ^P is derived, an architectural template, corresponding to a predefined *structural decomposition*, should be selected. The objective now becomes that of mapping the new processes (^ P) onto the components of the partic-

4. When a behavior or structural description of a design object is specified, each non-atomic element of the description is represented by a new design object. An atomic element in a description is one that cannot be further expressed in terms of the constructs of the mathematical formalisms or the language adopted to express the description, as described in [4].

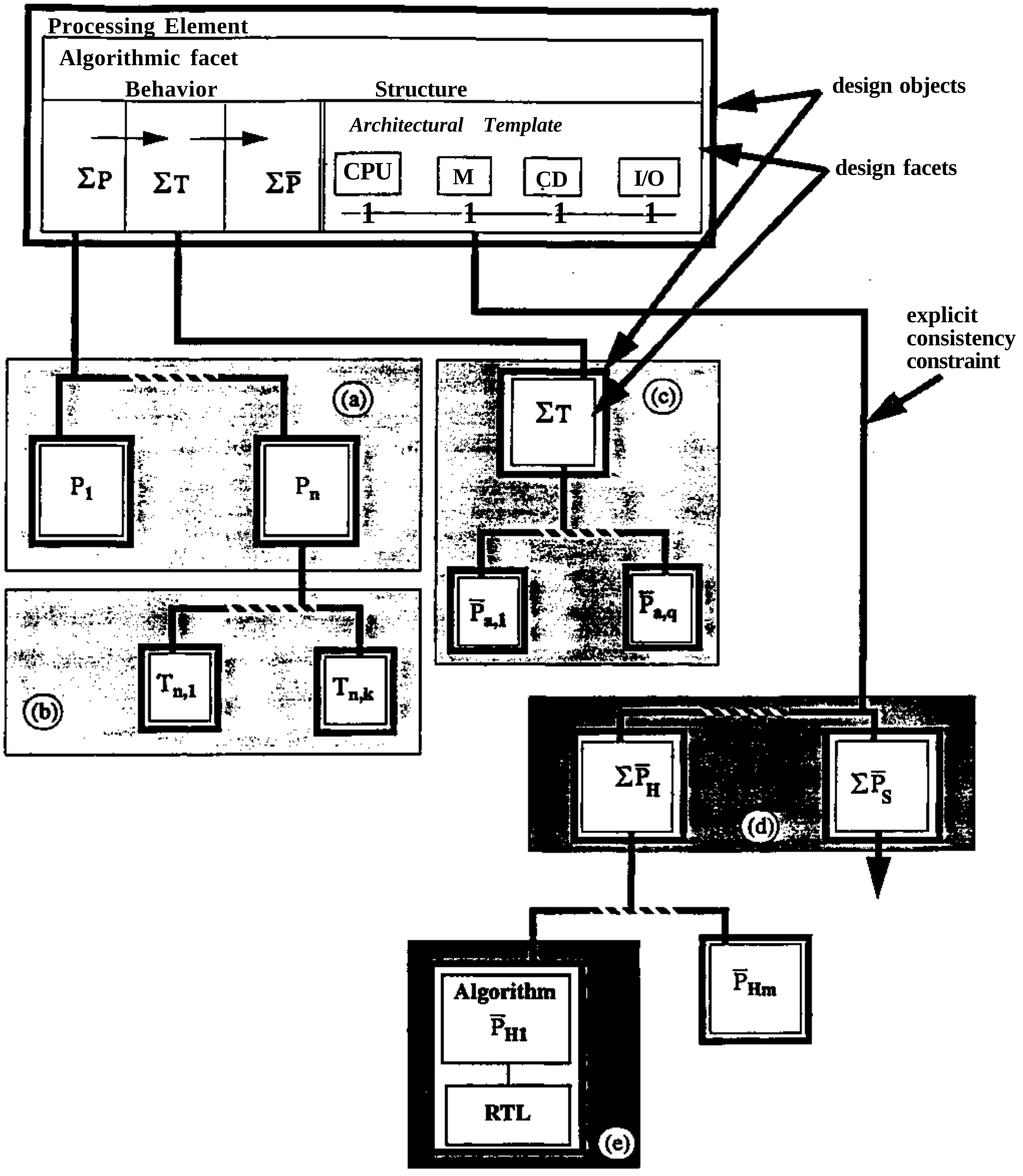


Figure 3 Abstracting design process data

ular architecture, i.e., defining the expected behavior for the components of the structural decomposition. This is shown (in a simplified form) in shaded region (d).

Note that all design objects resulting from the set of decompositions just described belong to the class of "processing elements"[5] and are initially instantiated (in the design state) at the algorithmic level of abstraction.[6] The dash lines indicate that some branches of the described behavioral and structural decompositions have been omitted.

Transformational synthesis steps are symbolically represented as "facet" transitions made within the context of a particular design object. At the moment the snapshot of the codesign process shown in Figure 4 was taken, the behavior of one of the hardware components, labeled P_{H1}, originally specified at the behavioral level, had already been synthesized to the RTL level, as indicated by the facet transition shown in the shaded region (d).

3.5 Design Objectives and Control of the Design Process

All actions performed during the design process are controlled by design objectives. Thus, unlike design objects which are passive (data) entities, design objectives are *active entities.* Just as sub-problems are created during the design process, due to decomposition, so too are design objectives. (Recall that an individual design objective is uniquely associated with each design (sub)problem defined in the design process.) Design objectives actually drive these decompositions. Specifically, if decomposition proves to be necessary in the context of a given problem, the associated design objective is responsible for actively decomposing itself into the appropriate set of sub-objectives thereby creating a new set of design sub-problems. The design objective is also responsible for instantiating any design sub-objects that may result from this decomposition, as well as for activating and properly sequencing the resulting sub-objectives. (See detailed example below.) Some of the actions that have to be performed by objectives to control their corresponding sub-objectives can actually be quite complex. For instance, sub-objective sequencing must take into account, among other things, "reports" generated by completed sub-objectives. If a given sub-objective reports failure to its parent (sub)objective, the follow-on action that the parent objective pursues could be quite different from the one it would pursue if success was reported. (In order to facilitate this process, design objectives are organized into a hierarchy that allows them to directly *communicate* with their immediate parent and descendent objectives.)

To illustrate these concepts consider Figure 4, which displays the complete representation of the design process, including the hierarchy of design objectives, for the exact same stage of the hardware/software codesign process discussed in Section 3.4. The relation between parent and descendent objectives is symbolically represented, as by an arrow pointing from the parent to the descendent. Observe that by inspecting the objectives hierarchy we can quickly grasp the design methodology being used. Here the complex top-level objective, *codesign,* is first decom-

5. For simplification purposes, the class names of most design objects have been omitted in Figure 3and Figure 4.
6. Being more precise, their highest level facet is the algorithmic one.

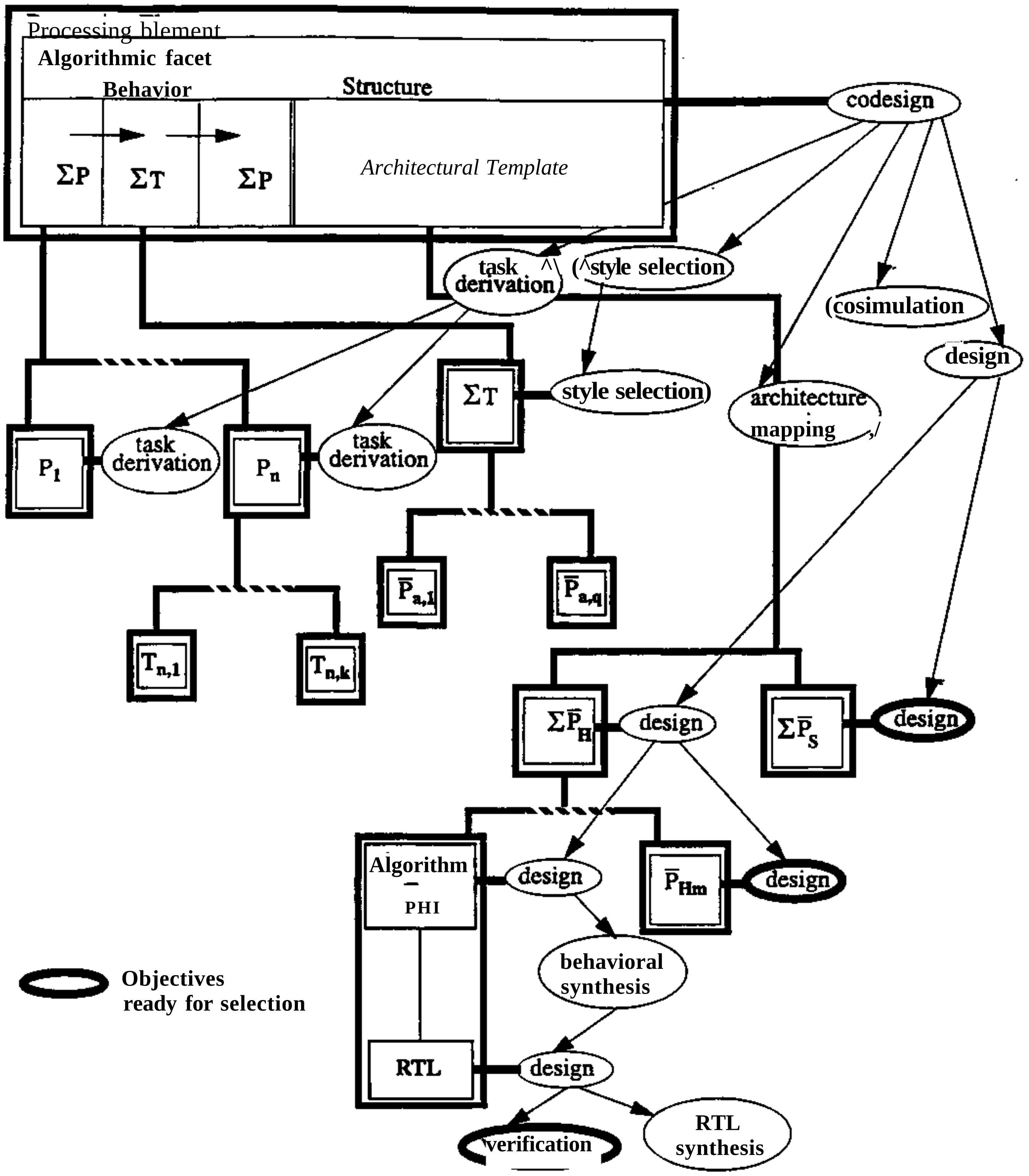


Figure 4 Snapshot of a hardware-software codesign process represented in Minerva.

posed into the sub-objectives *derive tasks, style selection, architecture mapping, cosimulation* and *design.* The first sub-objective to be activated, "derive tasks", is responsible for controlling the process of partitioning the original set of processes into tasks, given a specific set of criteria. "Style selection" controls the process of first regrouping the resulting set of tasks into a new set of processes, again based on a set of criteria and on the requirements for the design object, and on deciding which processes will be implemented in hardware and which processes will be implemented in software. The next objective, "architecture mapping," controls the mapping of the new set of processes onto a particular, previously selected architecture. Then, "cosimulation" is responsible for verifying if the current partition matches the requirements for the principal design object (the processing element). Observe how the recursive decomposition of design objectives ends when the sub-problems generated are such that they can be addressed by the CAD tools at hand. After cosimulation is complete, and if the results are satisfactory, the design of the individual software and hardware components can commence. In other words, the status of the sub-objective "design" becomes "ready". At the moment the snapshot in Figure 4 was taken, we can see that all "design" sub-objectives associated with implementing each process, except for $\vec{P}_{H1}$, together with the "verification" sub-objective associated with the design object that is intended to realize $\overline{P}HI*$ can now be concurrently addressed. In Section 4, we briefly explain how design objectives that are capable of exhibiting behavior patterns as complex as the ones just described can actually be defined in Minerva.

A fundamental piece of information associated with an objective is the problem status. Observe that since the only element *unique* to a design problem is the design objective, it must actually store and properly update the status of the problem. Problem status indicates, for instance, if a particular problem is ready to be addressed, or if it has already been solved. Objectives that are ready to be addressed (and thus selectable by. the designer) are shown in bold in Figure 4. The problem state is also the conceptual tool used for supporting most of the synchronization issues in Minerva, yet space precludes us from addressing such issues in this paper. For a detailed discussion of synchronization in Minerva see [4].

4 Interacting with Minerva

We are now ready to describe how designers interact with Minerva during the design process. For reference purposes, we summarize Minerva's high-level, distributed control cycle in Figure 5. In what follows we introduce and justify each of the individual steps shown in this figure.

4.1 Problem Definition

A new project-session in Minerva begins with a *problem definition* phase, in which the principal project problem is defined. During this phase the designer is shown all *classes of design objects* contained in Minerva's discipline library and is expected to select one. The designer in then shown all of the possible *objectives,* contained in the discipline library, and is expected to choose one, thus completing the specification of the principal project problem.

After the principal project problem has been defined, each designer wishing to work on a particular alternative design has first to "connect" to the particular "project component", and then select the "alternative design" of interest.[7] The designer then engages in a *problem-selection, problem-solving* cycle. (Observe that any number of

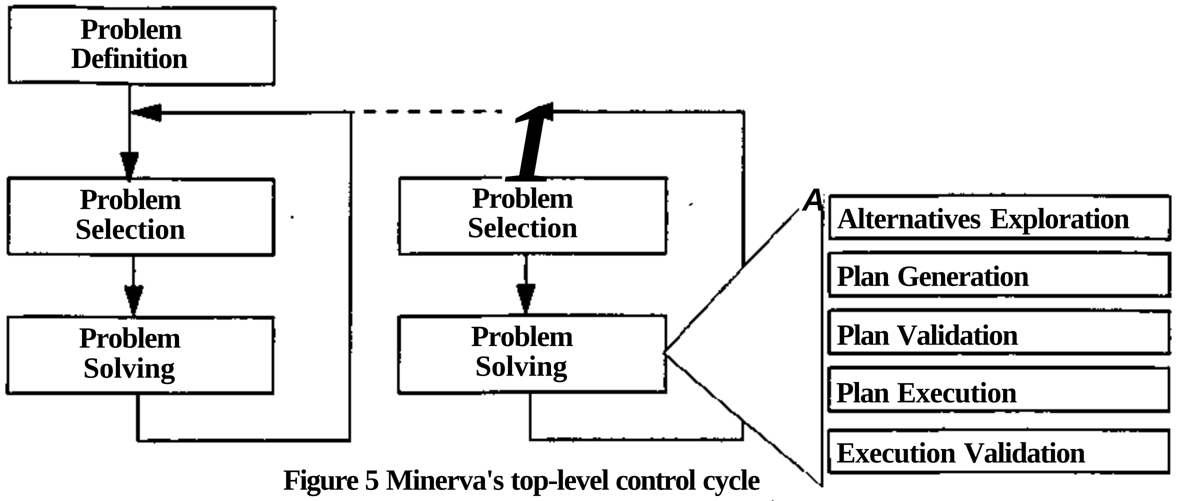


Figure 5 Minerva's top-level control cycle

designers can simultaneously be working on a given alternative design. This is represented by the multiple *"problem selection-problem solving"* loops shown in Figure 5, and is supported by the "working windows" assigned to the designers.)

4.2 Problem Selection

During *problem selection* the designer selects the design problem to work on. In order to facilitate problem selection (and also reduce the amount of information flowing between the corresponding alternative design process and the designer's working window process), the designer is first shown the hierarchy of design objects generated in the particular alternative design (an example of such a hierarchy was shown in Figure 3). After selecting a particular design object and facet in this hiearchy, the designer is presented with their associated hierarchy of objectives. As mentioned before, in the design objective hiearchy associated with a particular design object facet, only those objectives whose status is *ready* can be selected by the designer. For instance, in Figure 4, the objectives shown in bold are the only ones that can be addressed at the present stage of the design process. (Observe, though, that these problems can be addressed concurrently by the designers assigned to the project.)

The designer may be also asked to enter requirement and description values during the final phase of problem selection. (Observe that requirements and descriptions are the "givens" of the design problem, in the sense that they allow the designer to specify, more or less completely, the intended behavior and/or the structure for the design object.) Examples of requirements, for a processing element represented at the algorithmic level, are shown in Figure 2. The designer may also be requested to specify values for objective dependent properties, such as a "stimuli" for a verification objective. A key observation should be made at this point. While specifying each requirement value, the designer is shown all *consistency constraints* that exist relating the particular requirement with any other

7. Immediately after the principal project problem is defined only one alternative design will exist. In what follows we discuss how can different alternative designs be generated in Minerva.

properties in the design process. In other words, the designer receives (on-line) *a filtered* version of the state of the overall design, containing only that information that is *relevant* for him/her at the particular moment of the process. (This provides the basis for propagation and tracking of requirements throughout the various levels of abstraction traversed during the design process.)

Note that designers may sometimes be interested in postponing a commitment to a specific set of property values. Postponing value assignment to requirements and descriptions, for instance, guarantees the possibility of supporting complex designs, since, sometimes, it may be difficult to completely define the intended behavior and/or structure of complex design objects at the initial stages of the design process. (Postponing value assignment to restrictions, on the other hand, allows for the possibility of delaying specific design decisions to later stages of the design process). In order to deal with this, in Minerva, properties (of any type) may classified as *mandatory* or *optional.*

When the designer finishes entering values for requirements and/or descriptions, the problem selection phase is considered concluded. The new property values, if any, arc returned by the working window to the proper alternative design, and the particular problem enters its solving phase.

43 Problem Solving

Design problems in Minerva can be solved either *directly* or *indirectly.* We discuss each case separately.

4.3.1 Direct Solution of Design Problems

A problem is *directly solvable* if CAD tools exist that can be used to solve it. (In other words, if each CAD task contained in the design plan generated for solving the design problem can be implemented by a particular CAD tool.) When a design problem is directly solvable, we say that it has a corresponding design objective that is *directly achievable.* Directly solving a design problem typically involves five steps: *alternatives exploration, design plan generation, design plan validation, design plan execution, and execution validation.* (These steps are controlled by the design objective associated with the particular problem.) *We* discuss each of these steps and illustrate them using the design problems shown in Figure 2 and Figure 3.

• Alternatives exploration

The goal of alternatives exploration, also known as *conceptual design* [10], is to efficiently prune the solutions space by focusing on that area of the solutions space that is mostly likely to yield a satisficing solution to the design problem. Thus, alternatives exploration involves choosing values for restrictions (or design issues) associated with a specific design problem, if any. Examples of restrictions for a processing element are "architectural template," "fabrication technology," and "layout style."

Alternatives exploration typically requires the *investigation* of alternative values for specific restrictions in terms of their consequences on the most critical problem requirements, so that adequate design decisions can be made. Towards this end, during conceptual design, it is again crucial to inform the designer about any consistency constraints that may exist relating the restrictions being considered to any other properties in the design process (typically requirements). Thus, designers receive a filtered version of the state of the overall design, containing only that information that is relevant for him/her at the specific phase of the problem solving process.

A "special" type of consistency constraint that only relates restrictions (i.e., design issues), is additionally available in Minerva to characterize *inconsistency* among specific design decisions. For instance, we know that certain layout styles are incompatible with certain fabrication technologies. When consistency constraints exist relating two or more restrictions, *ordering constraints* imposing a certain sequencing in assessing those restrictions can be also established in Minerva.

Notice, finally, that it is mainly during the alternatives exploration phase that alternative designs are created. Being more specific, each time a designer chooses more than one value for a given restriction (for instance, two different architectural templates for realizing a particular behavioral description), new alternative designs are created in the context of the particular project. The resulting alternatives will be similar among themselves except for the fact that each of them will contain one of the selected design options for the particular restriction. From that point on, the new alternative designs will evolve separately.

Alternatives exploration is carried entirely out by Minerva i.e., no communication with the executive is needed. However, design assistance tools, such as Clio, [11] can be invoked by Minerva, upon a users request.

- **Design Plan Generation**

Design plan generation is the step during which a design plan (i.e., a partially ordered sequence of CAD tasks) is created for solving the particular design problem, containing all relevant property values. After generating a design plan, Minerva sends it to the executive to determine the availability of the CAD resources required by the plan, and this concludes the step. (Observe that this dialog with the executive occurs in a form that is totally transparent to the designer. The designer may actually be working on a different sub-problem while this is happening.) When the executive reports back on the executability of the plan, this particular design problem enters the next direct solution step.

- **Design Plan Validation**

Design plan validation is the step in which the designer is informed about the availability of resources (i.e., CAD tools) required by the individual CAD tasks in the plan. A plan is executable if such resources exist. If the plan is executable, the executive should also report back the list of selected CAD tools, and the designer is then allowed to reject specific CAD tools from that list.

If proper CAD tools are not available, or the designer rejects the available CAD tools, the plan is considered non-executable. If the plan is non-executable, an *impasse* is generated and two alternatives are provided by Minerva to resolve it: *backtracking* to a previous design state or *problem decomposition.* In either case, the direct solution of the particular objective is suspended. (We discuss this situation later.)

- **Design Plan Execution**

Given a valid design plan, the next step is *design plan execution.* This step allows designers to control the moment of execution of a specific plan. When the designer requests plan execution, Minerva sends a message to the executive requesting the execution of the particular design plan. (Observe that while the plan is executing, the designer may actually be engaged in such an execution, if the selected CAD tool is interactive. Otherwise, if the selected tool is non-interactive, the designer would be free to work on a different sub-problem. Minerva does not interfere with the plan execution.) When the executive reports back on the results of plan execution, will this specific design problem enter the next problem solving step.

- **Execution Validation**

The final step in the direct solution of a design (sub)problem is *execution validation,* in which Minerva displays the results of the execution of the plan. A design plan may fail, and thus an impasse be generated, if for instance, the selected CAD tools crashed or failed during execution. Moreover, the designer is allowed to reject the results of a plan whose execution was concluded successfully, in which case an impasse is also created.[8]

As with impasses during plan validation, impasses during execution validation may also be solved by backtracking, or alternatively, by problem decomposition. In either case the direct solution of the problem is suspended. On the other hand, if the plan was successfully executed, and the designer decides to accept its results, the data resulting from the plan execution is properly abstracted in the design alternative (for instance, as a new property value residing in an existing or in a new object facet), and the design problem is marked as solved.

4.3.2 Indirect Solution of Design Problems

When an impasse is resolved through problem decomposition, the particular design problem where the impasse occurred will be *indirectly solved,* i.e., will be solved throughout the solution of its sub-problems. We say then that the corresponding design objective will be *indirectly achieved.* For instance, for the case shown in Figure 4, the problem associated with the objective "h/s codesign" is being indirectly solved through the sub-problems associated with the sub-objectives "derive tasks", "style selection," "architecture mapping," "cosimulation" and "design".

8. This last situation may occur, for instance, if a designer is working, say, with an interactive synthesis tool, and after a while concludes that a satisficing solution is impossible to achieve. So, when the designer quits the tool he already knows that the results produced by the tool should be discarded.

The design plan for a problem being indirectly solved is thus a composition of the design plans generated for all its sub-problems, and this composition applies recursively, until the leaf design sub-problems are reached. This is why, in the hierarchy of design problems representing the alternative design, *only the leaf sub-problems* (i.e., those not yet decomposed), are still (potentially) directly solvable, and thus directly selectable by the designer. If an impasse occurs while solving one of these leaf sub-problems, then the direct solution of the particular sub-problem is suspended, as described above. If problem decomposition is chosen to address the impasse, and the objective decomposes itself into a particular set of sub-objectives, its state changes into "waiting on sub-objective achievement", and it would not be directly selectable by the designers from that point on.

4.4 Backtracking

As mentioned before, backtracking can be chosen to solve a particular impasse. In such a case, the designer is allowed to return to a previous state, and the direct solution of the design problem where the impasse occurred is suspended. Backtracking can also be directly invoked by the designer, from outside of the "problem selection-problem solving cycle", in which case Minerva displays the global representation of the design process and allows the designer to choose the specific problem to backtrack to. Problems are always restarted at their initial direct solution phase.

Consistency constraints are used in Minerva for guiding the selection of convenient backtracking points given specific failures. This is so because they allow designers to trace the chains of design decisions that lead to a particular dead end. Consistency constraints are also used in implementing the backtracking steps themselves, i.e., in propagating their effects throughout the entire, on-going, design process.

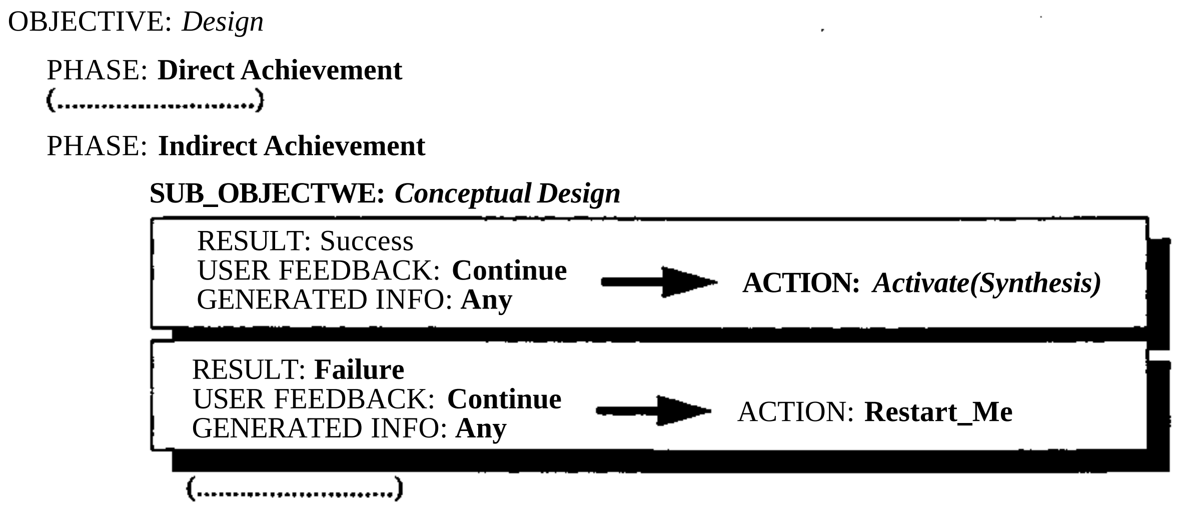
5 Minerva's Discipline Library

A key issue in Minerva's implementation was to guarantee that it would be easily customizable to different design disciplines and/or methodologies. This was accomplished by deriving suitable *declarative* representation structures for the various elements contained in the "discipline library" (see Section 3.1). In the process of deriving adequate representation structures for those elements of the library that fundamentally abstract data, such as classes of design objects and their properties and consistency constraints, we choose to emphasize modularity and simplicity and arrived at declarative structures that are quite simple and self-explanatory. (See [4].) A greater challenge was finding suitable declarative structures that would allow for proper specification of the complex behavior patterns that can be exhibited by design objectives, the "active" elements of the Minerva's design process representation. Note that design methodologies are captured in the way design objectives decompose themselves, sequence their sub-objectives, etc., and therefore having the ability to customize objectives was quite important.

The proposed solution for characterizing and representing design objectives explores the potential for recursion (and thus regularity) created by the network of communication links that is established among the design objectives in a design process (see Figure 4). Specifically, it allows the control of a given objective over its own sub-objectives, in case of decomposition, be specified and treated similarly to the control used if direct achievement is possible. Accordingly, even if the control part of an objective specification is described in terms of two basic bodies of knowledge (that for direct achievement and that for indirect achievement, as illustrated in Figure 6) in both cases such specification is done in terms of similar *condition-action pairs.* [9]As can be seen in Figure 6, such condition-action pairs (shown in a shadowed rectangle) are quite simple, being composed of a condition tuple and a primitive action.

In order to illustrate how complex behaviors such as the one described in Section 3.5 can be easily specified in Minerva's discipline library, we now discuss the objective definition shown in Figure 6.[10] (Observe that Figure 6 illustrates just part of a possible declarative specification for the objective "design".)

In Figure 6 two condition-action pairs are shown. Such pairs specify what the objective "design" should do (i.e., what follow-on *action* to take) given two possible "returning reports" (i.e., *condition tuples)* coming from the sub-objective "conceptual design". The first condition-action pair corresponds to the situation in which the descendent sub-objective (i.e., conceptual design) reports, both, success of its solving process and the designer's acceptance of Minerva's default follow-on action for success. In the example, the specified default follow-on action for this particular tuple is activation of the "synthesis" sub-objective (i.e., the action **Activate** *(Synthesis))..*



OBJECTIVE: *Design*

PHASE: **Direct Achievement**
(..........................)

PHASE: **Indirect Achievement**

SUB_OBJECTWE: ***Conceptual Design***

RESULT: Success
USER FEEDBACK: **Continue**
GENERATED INFO: **Any**
→ **ACTION:** ***Activate(Synthesis)***

RESULT: **Failure**
USER FEEDBACK: **Continue**
GENERATED INFO: **Any**
→ ACTION: **Restart_Me**

(..........................)

Figure 6 Declarative specification of a design objective

9. In other words, in specifying the direct achievement and the indirect achievement of a design objective, we use condition-action pairs that have *precisely the same structure* and use the *exact same primitive constructs.*

10. The "fields" of the declarative template provided by Minerva for defining the control of an objective are shown in capital letters. Reserved works (enumerated types or primitive actions) are written in bold. Finally, objective identifiers (i.e., their names) are shown in italic.

The second condition-action pair corresponds to the situation in which the descendent sub-objective reports failure and user acceptance of the default action for failure. In the example, the specified default follow-on action for failure is a backtracking step, i.e., restarting of the descendent objective that just failed (i.e., the action **Restart_Me).** Observe that the designer is also allowed to select (at run-time) a follow on-action different from the default one. Such alternative actions are offered to the designer for selection (by Minerva) before finishing with any "problem-selection problem solving cycle".

It is important to point out that all primitive actions provided by Minerva for specifying sub-objective follow-on actions (such as "Activate" and "Restart_Me") are *general* and *discipline independent.* A list of the set of primitive actions built-in Minerva and a discussion on their functional coverage is beyond the scope of this paper and can also be found in [4].

Finally, observe that Minerva allows alternative objective decompositions to be specified in the definition of a particular design objective. For instance, observe that in Figure 4 the objective "design" can be decomposed into three different forms during the design process. Such different decompositions can be indexed in several ways in the objective definition, yet space preclude us from discussing the subject in this paper. A complete discussion on specifying design objective decompositions in Minerva can be found in [4].

6 Conclusions

In this paper we described the Minerva design process planning and management meta-tool. By implementing the design formalisms described in [2] [4], Minerva is capable of providing effective design process planning and management services supporting arbitrarily complex top-down design processes. Such services include: plan generation; plan execution; automatic problem reformulation (i.e., decomposition) in case of impasse during plan generation or during plan execution; support to backtracking for redesign and for problem re-definition; dynamic sharing of design information among designers cooperating in the same design process; tracking of requirements throughout the various levels of abstraction traversed during the design process; and effective handling of sub-problem interactions in all of the above situations. All of the above services are offered assuming the most complex scenario, i.e., a concurrent, distributed design environment.

Minerva is discipline independent and can be customized to any design discipline. Moreover, Minerva can be customizable to any desired design methodology in the context of the discipline of interest. Currently Minerva supports the design of datapaths, and the executive interacting with Minerva currently has available the Berkeley Synthesis Tools for logic synthesis. [12] Expansion of the discipline library in order to cover the hardware/software codesign discipline is underway.

7 Bibliography

[1] M.F. Jacome, and S.W.Director. "Design Process Management for CAD Frameworks." In Proceedings of *29th ACM/IEEE Design Automation Conference.* ACM Press, 1992.

[2] M.F. Jacome, and S.W.Director. "A Formal Basis for Design Process Planning and Management." In Proceedings of *International Conference on CAD,* ACM/IEEE. November 1994.

[3] D.W. Knapp and A.C. Parker. "A Design Utility Manager: The ADAM Planning Engine." In *Proceedings of the 23th ACM/IEEE Design Automation Conference,* ACM Press, 1989, pages 48-54.

[4] M.F. Jacome. *Design Process Planning and Management for CAD Frameworks.* PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, September 1993.

[5] E. D. Sacerdoci. "Planning in a Hierarchy of Abstraction Spaces." *Artificial Intelligence,* 5:115-135,1974.

[6] H.A. Simon. *The Sciences of the Artificial.* The MIT Press, 1981.

[7] D.E.Thomas, J.K. Adams, and H. Schmit. "A Model and Methodology for Hardware-Software Codesign." *IEEE Design and Test of Computers,* pages 6-15, September 1993.

[8] PR. Sutton, J.B. Brockman., and S.W. Director. "Design Management Using Dynamically Defined Flows." In Proceedings of *30th ACM/IEEE Design Automation Conference,* ACM Press, 1993.

[9] K.O. ten Bosh, P.Bingley, and P. van der Wolf. "Design Flow Management in the NELSIS CAD Framework." In Proceedings of *28th ACM/IEEE Design Automation Conference,* ACM Press, 1991.

[10] A.M, Dewey and S.W.Director. "YODA- A Framework for the Conceptual Design of VLSI Design Systems." In Proceedings of *26th ACM/IEEE Design Automation Conference.* ACM Press, 1989.

[11] J.C. Lopez, M.F. Jacome, and S.W. Director. "Design Assistance for CAD Frameworks." In Proceedings of *First GI/ACM/IEEE/IFIP European Design Automation Conference.* ACM Press, 1992.

[12] R.L.Spickelmier, and P. Cohen. *Examples of Tool Use in Octtools 3.5.* Technical Report, University of California, Berkeley, Electronics Research laboratory, 1990.