1995

A formal basis for design process planning and management

Margarida F. Jacome
Carnegie Mellon University

Stephen W. Director

Carnegie Mellon University.Engineering Design Research Center.

A Formal Basis for Design Process Planning and Management

Margarida F. Jacome and Stephen W. Director

EDRC 18-52-95

A Formal Basis for Design Process Planning and Management*

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, PA 15213

ABSTRACT

In this paper we present a formalism that allows for a complete and general characterization of design disciplines and for a unified representation of arbitrarily complex design processes taking place in the context of these disciplines. This formalism has been used as the basis for the development of several prototype CAD meta-tools that offer effective design process planning and management services.

1 Introduction

We have reached the point where we need to move beyond the development of CAD tools that only aid designers in solving specific synthesis, analysis, and/or optimization design problems but also aid designers in planning and managing the increasingly complex design process itself. In order to achieve this goal we need to be able to formally characterize and represent the fundamental intentions, strategies, and mechanisms employed during the design process. In other words, we need to be able to formalize the *content* or the *semantics* of design rather than simply allowing this information to remain hidden in the syntactical idiosyncrasies of the design *"form."* Observe that through the realization of such a syntactically transparent and semantically rich design formalism, it becomes possible to articulate the essential concepts of design and model both the *design artifacts* and the *process of design* across multiple design disciplines, in a coherent and precise way. In fact we will show that a design formalism with the above characteristics is immediately useful in guiding the development of general purpose, highly effective design process planning and managements *meta-tools.* Because such meta-tools are capable of capturing the fundamental strategies for controlling complexity embed in traditional design methodologies, and by intelligently making use of such strategies throughout the design process, they allow for the realization of a new generation of powerful CAD environments.

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

In this paper we present a formalism of design that allows for a complete, and general, characterization of design disciplines and for a unified representation of design processes that take place in the context of these disciplines^ 1] This formalism has been used as the basis for the development of several prototype design process planning and management meta-tools (Minerva [2] and Clio [3]). The remainder of this paper is organized as follows. First we discuss the main issues involved in properly planning and managing complex design processes. Then, in Sections 3 to 7, we introduce a formal Characterization of design processes. In Section 8 we present an example to illustrate the application of this formalization in characterizing the hardware/software codesign discipline. In Section 9 we briefly discuss the Minerva Design Process Planning and Management meta-tool, which directly implements the formalism of design described in this paper. Some conclusions are given in Section 10.

2 The Design Process Planning and Management Problem

The design process basically entails a search in a "solution space'[9] for an object that meets a desired "initial specification/' During the course of design, in part due to physical realities, the initial specification may in fact evolve. Since the effort for generating a possible solution to a design problem and evaluating it can be extremely large, searching the entire solution space for the *best* design is impractical. Therefore, designers employ heuristic-based methods to reduce complexity and improve the efficiency of the search for a solution to the design problem. Some of such methods decompose the original, often complex, design problem into a set of less complex sub-problems. Problem decomposition may be recursively performed until the resulting sub-problems are of such complexity that they can be directly solved using the available CAD tools.[1] It is important to note that design decisions made during the solution of one sub-problem may impact decisions that need to be made while solving other sub-problems^]. As a consequence information must be shared among sub-problems and consistency checks must be made during their solution. Choosing which methods and strategies to use for problem decomposition, and then properly applying them for the solution of a design problem, while guaranteeing that consistency is preserved, constitutes the essence of what we call *design process planning and management.*

The development of meta-tools to aid the designer with design process planning and management requires an adequate representation of the design process that goes beyond the concept of "design flows'* [6][7], In particular, such a representation should *unify,* at an *adequate level of abstraction,* all of the different levels and stages of a design process. This unified representation allows for a uniform representation of problem dependencies, and facilitates the free interchange of relevant design information and design decisions between all members of a design team that may be concurrently working on the design problem.

1. Such tools may be fully automated or interactive.

Since the heuristics employed by designers may not always work, the design process may sometimes reach an impasse or dead-end. In such situations, designers may need to backtrack and revisit a previous design state in order to reconsider previous design decisions. A unified design process representation makes it possible to implement not only local but also global backtracking mechanisms, i.e., backtracking mechanisms that impact different phases and/or stages of the same design. As will be shown, in order to provide such a unified design process representation with the necessary semantic content, we also need to have the capability to explicitly characterize the design discipline in which the design process takes place.

3 The Design Space

We begin the formal characterization of the design process by introducing some basic definitions. We will refer to the domain of interest during the design process as the design discipline. Design disciplines may be very broad, i.e. the disciplines of analog VLSI circuits and digital VLSI circuits, or very narrow, i.e. the discipline of operational amplifiers. As will be shown below, design disciplines are defined in terms of *classes of design objects, consistency constraints, design objectives, and operators.* **Classes of design objects are characterized by *specifications* and may** be organized in a *discipline hierarchy.* Such a discipline hierarchy captures the different *abstraction levels* that may be used for describing the object under design (e.g., register-transfer level, and logic level, for digital VLSI circuits). We will also show that at any given moment, a design process may be described in terms of a current *design state* and a *design history,* i.e., the sequence of design states that led to the current design state.

Hie *design space* (i.e., the *problem space* in which the design takes place) is defined in terms of a **knowledge** component and a **data** component. The *knowledge component of the design space* comprises the discipline hierarchy, and also the design objectives, the consistency constraints, and the operators that can be defined for the particular design discipline, and constitutes the formal basis for the explicit characterization of the design discipline. The *data component of the design space* comprises the current design state and the design history, and constitutes the formal basis for creating the unified design process representation.

4 Characterizing Design Objects

A design object is an abstraction of a physical device or process and is characterized in terms of a set of inputs, a set of outputs, and a set of properties that describe its **structure,** i.e., the way it is realized, and/or its **behavior,** i.e., the way it should function. (ALUs and RAMs are examples of classes of design objects for the design discipline of digital VLSI circuits.) We view design as the process of deriving a complete specification, i.e., a set of properties that uniquely characterizes a particular design object, starting with an initial specification, i.e., a sub-set of the properties associated with the complete specification.

Property definitions designate relevant structural or behavioral design object features. A class of design objects is a set of design objects that can be characterized by means of the same set of property definitions, or specification definition. (The specification definition for a class of design objects is, thus, the set of property definitions which are necessary for characterizing each design object in the particular class.) A property instance, or simply a property, describes the feature designated by the property definition. A specification is as a set of properties characterizing a particular design object in the particular class. As an example, consider the class of design objects known as *adders.* A property definition for this class is «ufcfer>Jias_word-length___equaI_to_<va/>. An example of a property derived from this last property definition, and used in characterizing the design object "MY_ADDER", is MY_ADDER Jias__word-length_ _equal_to_ 16 bit.

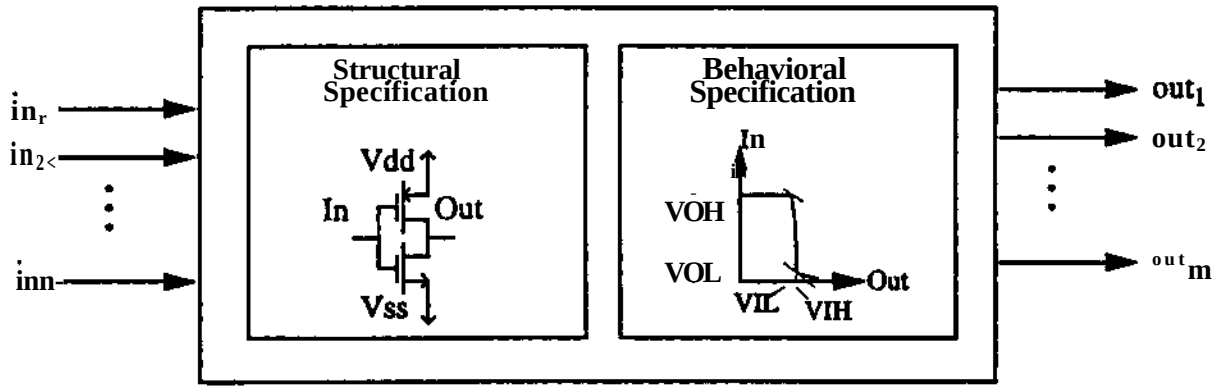

Figure 1 A class of design objects

4.1 Specifications and Specification Definitions

As illustrated in Figure 1, the specification definition of a class of design objects can be partitioned into two general categories: *behavioral specifications* and *structural specifications.* (Observe that the property definitions contained in these specifications will be further partitioned throughout the abstraction levels defined for the design discipline, as will be explained in Section 4.2.)

Behavioral specifications contain all behavior related properties for a given class of design objects, i.e., all properties that define how the design object should "behave" or function. Structural specifications, on the other hand, contain all structure related properties for a given class of design objects, i.e. properties that define how the design object will be actually 'realized'. Each of these specifications may be further partitioned into three sub-categories: *requirements, restrictions,* and *descriptions.*

4.1.1 Requirements

Behavioral and structural requirements are properties that define the "givens" of the design problem, in other words, these are the properties that the object under design must meet when the design process is complete. Behav-

ioral requirements are frequently the initial specification for a given system, either when the intended behavior for the system has "ideal characteristics"[2] or when the complete behavioral description of such a system is **too** complex to be directly managed, at least during the initial stages of the design process. For instance, the behavioral requirements for a low pass filter are grouped into the lowpass frequency response specification, and include tolerances for the stop band of the filter and for the maximum deviation between the ideal and real filter amplitudes. Slew rate is an example of a requirement for operational amplifiers. Examples of structural requirements, on the other hand, are area and dissipated power.

4.1.2 Restrictions

Behavioral and structural **restrictions** are properties that are employed by the designer to prune the design space in order to reduce design complexity. Designers typically derive restrictions during the **conceptual design phase** of the design process. For instance, when the intended behavior for the design object has "ideal" characteristics, behavioral restrictions may be used for characterizing a convenient type of approximate behavioral description. Structural restrictions, on the other hand, allow the designer to constrain the structure to specific topologies or to particular fabrication technologies. For example, a designer may wish to constrain the structure of a given digital circuit to CMOS fabrication technology or to a specific layout style, such as gate array.

4.1.3 Descriptions

Behavioral and structural **descriptions** are properties that, respectively, fully define the behavior and the structure of a given design object Specifically, **behavioral descriptions** indicate, via mathematical equations or behavioral description languages, how a design object reacts, or should react, to specific sets of stimuli applied to its inputs. Behavioral descriptions (relating the outputs of the design object to its inputs) can be specified using simply, and directly, the fundamental connectives and/or constructs of these mathematical formalisms and/or behavioral description languages or, alternatively, a behavioral description can be specified in terms of a set of *assumed behavioral sub-descriptions.* A behavioral sub-description is said to be "assumed" in a particular behavioral description, if such a sub-description is *not explicitly represented* in the behavioral description (in terms of the fundamental connectives and/or constructs of the mathematical formalisms and/or behavioral description languages that characterize the particular abstraction level). Each of the assumed behavioral sub-descriptions can, thus, be seen as a "non-primitive behavioral building block," for the particular abstraction level.

A behavioral description incorporates a functional decomposition in the sense that it expresses the (complex) behavior of an entire design object in terms of less complex (primitive or non-primitive) "behavioral building

2. "Ideal" in the sense that it has characteristics that are known to he physically or technologically impossible to achieve or implement, but can be approximated by a real design object

blocks." A behavioral description is said to incorporate a fundamental **functional decomposition** if it contains only "primitive behavioral building blocks", otherwise, is said to incorporate a **non-fundamental functional** decomposition. Non-fundamental functional decompositions are used for controlling the complexity involved with deriving the behavioral description of a complex design object

Let us illustrate the concept of a non-fundamental functional decomposition for the VLSI analog circuits discipline. Assume that u_M, y_M, and G^ are, respectively, the input, the output, and the impulse-response matrix of a given object, M, under design, and represented at the circuit level of abstraction. Assuming that an external description is being used, the behavior of M is given by $y_M(t) = \int^{t} G_M(t, x)\, u_M(x)\, dx$. Observe that at this point we could decide to fully express G_M in terms of the primitive constructs on an external description, which would make the behavior of object M correspond to a "fundamental behavioral decomposition." Alternatively, we could express G_M as $G_M(t, x) = G_x(t, x) + G_2(t, x)$, in which case the behavioral description of object M would contain a non-fundamental functional decomposition, where Gl are G2 are the "non-primitive behavioral building blocks" of the description.

A structural description is an interconnection of a number of *structural building blocks.* Such building blocks can be either *primitive building blocks,* or *non-primitive building blocks.* **A primitive building block cannot be** expressed in terms of other, simpler, building blocks defined at the same abstraction level. Before the design process can commence, structural primitive building blocks must be available for each abstraction level of the specific design discipline. Furthermore, a behavioral description, must exist for each of the structural primitive building blocks. Such behavioral descriptions are called models. An atomic physical device or process (and thus the primitive building block that represents it) may have different models depending on the operational ranges in which it will be used and/or the required accuracy with which we want to reproduce the real physical process.

As in the previous case, a structural description can also be seen as a structural decomposition. A structural description is said to incorporate a fundamental structural decomposition if it only uses structural primitive building blocks. Otherwise, if the structural description uses at least one structural non-primitive building block, it is said to incorporate a a non-fundamental structural decomposition. Non-fundamental structural decomposition is used for controlling the complexity involved with deriving the structure of a complex design object For instance, the structural description of an OPAMP, at the circuit level of abstraction, may be defined in terms of transistors and capacitors, which are among the primitive building blocks of the circuit level of abstraction for the VLSI digital design discipline. This structure would, thus, constitute a fundamental structural decomposition. On the other hand, the structure of the OPAMP could alternatively be represented, at the same abstraction level, using non-primitive

building blocks, such as current sources, differential amplifier stages, and voltage gain amplifiers, which constitutes a non-fundamental structural decomposition.

It is important to note that the descriptions generated during the design process must be consistent with the corresponding restrictions and requirements, whenever they exist. In other words, there is a certain level of dependency among properties that may cause inconsistences. We will return to this subject in Section 5.

4.2 The Discipline Hierarchy

Typically, for each given discipline, it is possible to identify a number of different classes of design objects. For instance, for the discipline of VLSI Digital Circuits, we may design ALUs, Multipliers, and Adders. What creates the notion of a discipline, though, is the fact that different classes of design objects typically share important properties. For example, a design discipline may use the same levels of abstraction to represent several different classes of design objects, which implies the adoption of the same behavioral and/or structural "primitive building blocks" for describing all its classes of design objects. Furthermore, different classes of complex design objects may share important structural and/or behavioral "non-primitive building blocks". This suggests that it is important to properly organize the entire set of classes of design objects that constitute a given discipline. In this section we describe such an organization, called the *discipline hierarchy.*

We can formally define the discipline hierarchy, denoted by A, as a two dimensional structure of ordered sets of design domain facets, or simply facets, denoted by 5_{if} i = 1, 2,..., each of which is a set of property definitions associated with specific classes of design objects. The general structure of a design hierarchy is illustrated in Figure 2, where the vertical dimension is referred to as the *abstraction dimension,* and the horizontal dimension is referred to as the *specialization dimension.*

The abstraction dimension organizes the design hierarchy into abstraction layers, denoted by $\wedge_{i}^{A}$, where "i" identifies the particular abstraction layer. More specifically, the abstraction dimension *partitions* the specification definition of each individual class of design objects. Specifically this partition causes the creation of a *specification definition* for each abstraction level $\wedge_{i}^{A}$, i= 1,2,.... The resulting sub-specifications constitute the facets of the class of design objects. The design abstraction layer that corresponds to the least abstract level of A is called the ground abstraction level, or simply the ground level, and is represented by A_Q. (Note that the least abstract layer has the most detail associated with it). In Figure 2, the direction in which the level of abstraction increases is indicated.

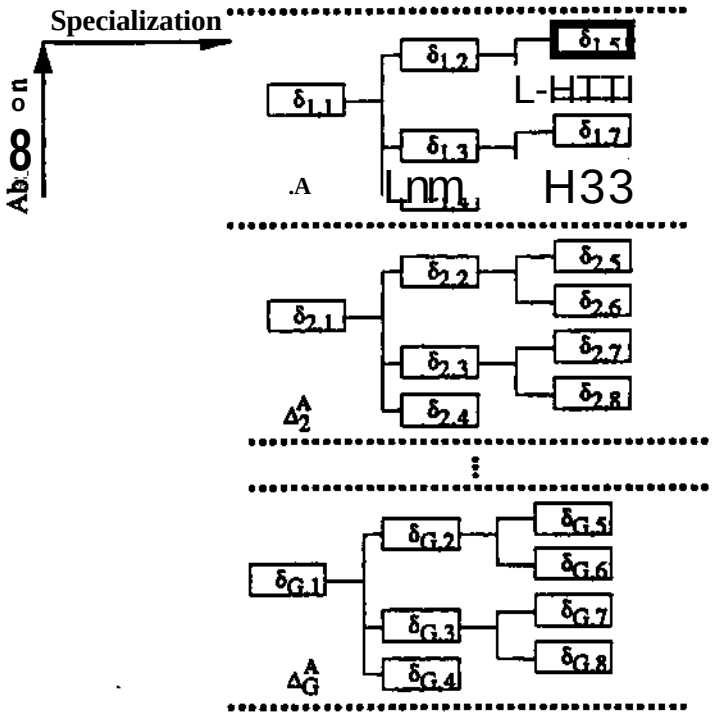

Figure 2 The Discipline Hierarchy A

The principle behind the use of abstraction is to reduce the complexity of finding a design object (in a given class of design objects) that satisfies a given specification. [8] This is accomplished by allowing the designer to begin by considering only those properties contained in the most abstract specification. When this initial specification is met by the object under design, the designer moves down in the abstraction hierarchy, and considers the next, lower level, specification. This process is repeated until the ground abstraction level is reached. Examples of levels of abstraction for the discipline of VLSI digital circuits design are register-transfer level, logic level, and circuit level.

The specialization dimension (horizontal dimension) of the discipline hierarchy serves a dual purpose. First, it discriminates between the different classes of design objects which may coexist for the design discipline, and second it also allows expression of commonalties among different classes of design objects defined for the particular design discipline. In other words, it allows representation of the concept of specialization (or conversely, generalization) by allowing different classes of design objects to share property definitions. The direction in which the level of specialization increases is indicated in Figure 2. An example of a design discipline hierarchy is given in Section 8. Other examples can be found in [2].

5 Consistency Constraints

Values of properties that belong to the same, or different, abstraction levels, and to the same, or different, classes of design objects, may be constrained by arbitrarily complex relations. Examples of relations among properties that may be defined, for instance, for a CMOS inverter, at the transistor level of abstraction are (assuming Vss = 0):

$V_{IL} = (3\ V_{DD} + 3\ V_{TP} + 5\ V_{TO})/8$; and

$V_{IH} = (5\ V_{DD} + 5 V_{TP} + 3\ V_{TO})/8$;

where Vj_H and Vj_L denote the high and low logic thresholds, respectively; V_{DD} denotes the drain voltage source; and Vjp and V_{TO} denote the threshold voltages for the inverter's p-channel and n-channel MOSFETs, respectively.

Consistency constraints, which represent dependencies among properties, are defined in terms of an *independent specification declaration,* a *dependent specification declaration,* and a *relation* involving the properties contained in both specification declarations. (Observe that properties can only be declared or referenced, as opposed to defined, in the consistency constraint definition.) For instance, for the consistency constraints shown above, the set of independent properties could be {Vdd, V-n* V_{TO}}, while the set of dependent properties could be {Vi_L}, for the first case, and {V_{IH}}, for the second case. Observe that, given a particular relation, defining which properties belong to which group may be design methodology dependent. Note also that a property can be a member of an arbitrary number of consistency constraints and may be listed as a member of the independent sub-specification for some of these consistency constraints, and as a member of the dependent sub-specification for the remaining consistency constraints.

An active consistency constraint, is defined by a reference to a consistency constraint, and by an *independent specification* and a *dependent specification.*[3] (As mentioned above, the set of actual properties contained in both specifications can belong to one or more design objects, and can pertain to the same or to different abstraction levels.) For each active consistency constraint, a predicate can be derived, denoted by *"VERIFY(active consistency constraint)"* whose value is "true" if the independent and dependent specifications verify the relation defined in the consistency constraint and "false" otherwise.

Observe that due to the fact that property values are often non-independent, including requirements among them, one of the key aspects of design is to determine appropriate trade-off between non-independent requirement values. [9]

6 Organization of Design Objects in a Design Process

A design process generates a hierarchy[4] of design objects. This hierarchy, called the design process hierarchy, and denoted by D^A, has two dimensions. The vertical dimension of the hierarchy is the same as the hierarchy of abstraction levels defined in the discipline hierarchy and is also called the abstraction dimension.

3. In other words, abstract consistency constraints are "templates" while active consistency constraints are instances of such templates.

4. Not to be confused with the discipline hierarchy, A, defined earlier. A is a knowledge-level structure while the design process hierarchy is a data-level structure.

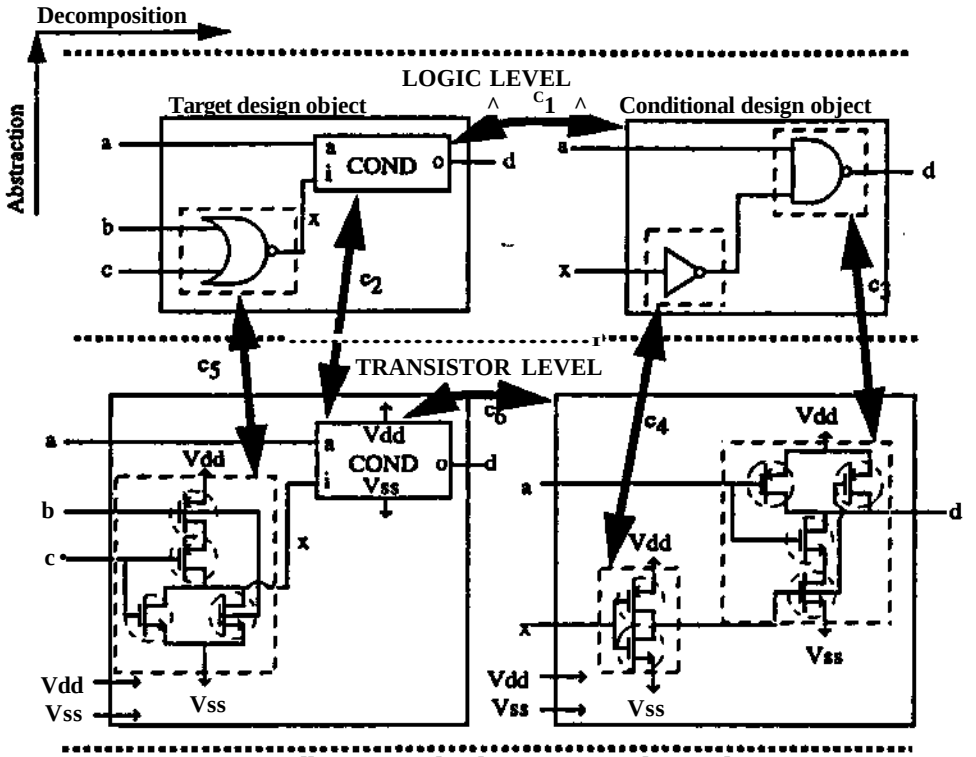

Figure 3 Illustrating the design process hierarchy

Functional and structural decompositions, together, define the horizontal dimension of the design process hierarchy, called the decomposition dimension. Observe that structural and functional decompositions are the only possible mechanisms for creating new design objects and adding them to the design process hierarchy. When new design objects are incorporated into a design process hierarchy, at a given abstraction layer, a set of active consistency constraints, representing the relations among these elements are instantiated in the design process. Such constraints relate the sub-specification of the original design object, or *parent design object,* to the sub-specifications of the *component design objects,* also called *descendant design objects.*

Figure 3 shows a snapshot of a design process hierarchy created while implementing the behavioral description given by d = COND(a, NOR(b, c)).[5] The structural primitive building blocks for the logic level are shown within a square. Observe that the structural description for the target design object constitutes a non-fundamental structural decomposition, since it contains the non-primitive structural building block "COND". In this particular example, COND itself becomes a new design object that has also to be designed. The structural primitive building blocks at the circuit level of abstraction are represented within a circle. As it can be seen in Figure 3, the structural description for the target design object, at the transistor level, constitutes also a non-fundamental structural decomposition, since the structural building block "COND", at the transistor level, is also a non-primitive building block.[6] Note

5. For simplicity we represent the design process hierarchy in terms of the structure description property, yet the concepts illustrated for this particular property are directly applicable to all types of properties.

also the active consistency constraints, denoted $C\backslash$ through c_6, relating properties from the same and from different abstraction levels.

7 Characterizing Design Activity in a Design Process

7.1 Design Objectives

As stated previously, the goal of a design process is to produce a ground level specification for a design object that satisfies all behavioral and structural requirements and restrictions at each level of abstraction traversed during the design process. Active consistency constraints represent the set of arbitrarily complex relations that must be verified among the properties that compose the specification of a given object being designed. The complexity of such constraints makes it virtually impossible in most real world design cases to derive a complete ground-level specification for the design object in just one problem solving step. Hence, the design process consists of an arbitrarily complex, partially ordered, sequence of *generation design steps* interleaved with *test design steps.*

A generation design step typically involves the selection of a particular sub-specification (among those that have been instantiated in the design process hierarchy), the selection of a subset of the associated active consistency constraints, and either: (1) the generation of values for those properties that still do not have values specified, while trying to satisfy the consistency constraints, called a synthesis design step; or (2) the modification of values for properties that already have a value assigned, while trying to satisfy the consistency constraint, called an **optimization** design step. In order to overcome the high complexity of some constraint relations, generation design steps typically take place either by only considering a subset of the relevant active consistency constraints while deriving values for properties and/or by frequently considering only a 'simplified' version of such consistency constraints. Hence it is possible to introduce *inconsistent* property values, i.e. property values that violate some of the relevant active consistency constraints.

Test steps are intended to detect such inconsistencies. Accordingly, test steps typically consist of selecting a particular sub-specification in which all properties have been assigned values, and verifying the active consistency constraints associated with this particular sub-specification (i.e., calculating the value of the predicate **VERIFY** for each of the active consistency constraints). If a constraint is not satisfied, either an optimization or a **backtracking** step is taken. As a general rule, backtracking occurs when at least one value associated with an independent property referred by the active consistency constraint being violated has to be undone. Otherwise, optimization is being undertaken.

6. This does not necessarily need to occur. In fact, the use of a different synthesis operator could have lead to a different outcome for the same design process.

Each design step accomplishes a specific *design objective,* More specifically, each design objective defines a category of *design problems* whose solution is achieved by means of a design step. Accordingly, design objectives define design goals to pursue in design processes and also define the control knowledge associated with achieving these design goals. Examples of design goals are synthesis, optimization, and test In addition, design objectives may also have an associated set of property definitions, aimed at characterize the specifics of the particular problem solving methods that may be used to accomplish the design objective and/or conveying any additional information that may be needed for the problem solving process. Examples of objective related properties are stimuli for test objectives and stopping criteria for optimization objectives. Observe that consistency constraints can also be defined for objective related properties. Actually, such consistency constraints can freely intermix objective related properties with design object related properties.

In summary, then, a design objective is defined by a design *goal,* by a body of *control knowledge* defining how the particular goal ought to be achieved in an arbitrary design process, and by a set of objective related property *definitions.* An design objective instance, on the other hand, is defined by a reference to the particular design objective being instantiated in the design process, a status (indicating if the goal associated to the objective instance can start being pursued or, if it started already being pursued, indicating the stage of its achievement), [7]a reference to the set of sub-objective instances generated by the current objective instance, if any, and a set of objective related properties.

Design operators are design functions, implemented by means of algorithms and/or procedures, that perform synthesis, optimization, and test design steps. In order to accomplish specific design objectives, design operators are applied to particular sub-specifications in a given design process in order to derive, modify, or extract property values, preserving the relations associated with the set of associated active consistency constraints. Note that the definition of design objectives is necessary, in addition to the definition of consistency constraints, since in a design process the exact same set of consistency constraints may be involved in synthesis, optimization, and/or test

7.2 Design State and Design Problems

The design state, or the current state of a design process, is defined by: (1) the design process hierarchy; (2) the set of all design objectives instantiated in the design process; (3) the set of all active consistency constraints relating the properties contained in the design process hierarchy and/or associated with in the set of objective instances; and (4) the set of all predicates of type "VERIFY" that can be defined for the set of active consistency constraints.

Given a current design state, each design objective instance in this design state defines a design problem, as described below. In simple terms, each active objective in the design state's set of active objectives defines a sub-

7. Possible objective status are: [t4]not-ready$_t$" "ready," "being-directly-achieved," "being-indirectly-achieved," and "achieved."

set on the three remaining components of the design state, i.e., properties, consistency constraints, and verify predicates. Design problems can thus be seen as design sub-states, containing all of the property instances relevant for achieving the problem's active objective, and all of the consistency constraints associated with these properties.

The design state can thus be seen as a hierarchy of design problems. The set of all problems that can be derived from a design state thus has cardinality equal to the cardinality of set of active objectives in the design state. Any new problem (or design objective) added to the design state will, in principle, remain in the design state, until the end of the design process, eventually reaching "achieved" status. Backtracking is the only way of removing a problem from the design state.

A design step is thus a *design state transition* from the current design state into a new design state. Since a design step is always the result of the achievement of a design objective, the transition function can be implemented by any operator that *may be* suitable for accomplishing any one of the active design objectives in the design process, among these whose status is currently "ready."

Observe that we might be led to think that there is some redundancy in separately specifying objectives, operators and consistency constraints. Indeed, this would be the case if design knowledge was complete and ideal, and if we always had available a set of operators, i.e., CAD tools, that properly implemented all conceivable constraint satisfaction value transformations and/or propagations directly derived from such consistency constraints. In an "ideal context", given a set of constraints, the CAD tool able to properly address the specific constraint problem would be uniquely determined. Unfortunately this is not generally the case, and frequently CAD tools implement only simplified versions of such constraint relations, yet still produce acceptable results. Furthermore, we may have different CAD tools that implement different methods for addressing the same class of problems. So, the notion of having a limited set of available operators, that may or may not contain one adequate for solving the specific design problem at hand, constitutes quite an important characteristic of design processes, since it may strongly influence their course of action.

7.3 Design Objectives and Control

We will now define more carefully the control knowledge that should be provided in the definition of a design objective. In general an active objective is directly achievable if there exists an available operator that is able to directly solve its corresponding design problem in the design state. If such an operator is not available, the active objective is indirectly achievable.

When an objective can only be indirectly achieved, the control knowledge should specify how the design problem associated with the particular active objective should be decomposed into smaller, less complex, subproblems, in order to resolve the impasse generated by the non-availability of a suitable, direct, design operator. In other words, the control knowledge should specify how to generate subsets for the three components of the original design prob-

lem (i.e., property instances, active consistency constraints, and verify predicates), and how to arrange these subsets in terms of new design problems residing in the design state. Moreover, for each new sub-problem resulting from the subsets defined by the control knowledge, a new design objective should be created and subsequently incorporated into the design state.

If we look at the definition of the various components of a design problem, though, we conclude that in creating sub-sets of property instances (based on the set of property instances associated with the original design problem), all of the remaining sub-problem components become uniquely defined. Thus, only a criteria from which the specification of such subsets can be made needs to be specified in the objective's body of control knowledge. Possible criteria include: (1) create subsets by different classes of design objects; (2) create subsets by different abstraction level; (3) create subsets by different general categories of specifications (behavioral versus structural); (4) create subsets by different subcategories of sub-specifications (description, requirement, restriction); and (5) create subsets by property.

For a given current design state, the next set of problems that may be *concurrently solved,* in parallel, or independently, is the subset of those problems: *(1)* whose associated active consistency constraints have disjoint dependent specifications; and (2) whose independent sub-specifications, defined for the active consistency constraints, are composed by properties which already have a value.[8]

Finally, the history of a design process contains the ordered sequence of all design states visited so far in the design process, together with the operators used to modify such design states. For the complete definition of the design formalism see [9].

8 Example

We now illustrate the adequacy and completeness of the design formalism introduced above, by applying it to the design discipline of hardware/software (h/s) codesign. While space preclude an exhaustive discussion of such a complex design discipline, we will illustrate how some of the more complex steps of h/s codesign and high-level synthesis methodologies can be easily and coherently described by this formalism.

The h/s codesign discipline is concerned with the design of systems that consist of both hardware and software components. A fundamental issue in h/s codesign is to decide which parts of a particular behavioral description are best realized by hardware and which by software. The initial specification of a h/s codesign problem typically includes a set of independent, interacting, sequential processes described in an hardware description language, such as VHDL, or Verilog, or in a programming language, such as C. The initial specification may also include a set of

8. All such problems will have their associated active objective with status = 'ready*.

requirements, such as *cost* and *speed.* Although formal strategies for h/s codesign have not been completely developed, different techniques for *behavioral partitioning* and *style selection* (hardware vs. software) have been demonstrated. In this example we adopt the techniques and methodology described in [10].

The first step in this methodology is task derivation, in which each of the algorithms provided in the initial specification is partitioned into a number of *tasks,* i.e., smaller chunks of behavior. While undertaking partitioning, the codesigner is trying to identify (in each algorithm) those chunks of behavior that are best suited for hardware implementation and those chunks of behavior that are best suited for software implementation. After task derivation is concluded, style selection is undertaken. Style selection involves looking at the "pool" of resulting tasks and selecting in final terms which tasks will be implemented in hardware and which tasks will be implemented in software, and properly regrouping such tasks into a final number of "hardware processes" and "software processes". These hardware and software processes then have to be mapped into a particular hardware/software system architecture. (These architectures typically consists of some application specific hardware on the system bus of either a general purpose or an embedded computer system running an appropriate operating system.) This step is called architecture mapping. After the mapping to an architecture is performed, the performance of the system can be globally evaluated, in order to asses the actual adequacy of the current partition and style selection. This is done in the methodology being described using a Verilog-C[9] cosimulator. [10] If the current solution meets the requirements, the C software is then compiled using cross compilation techniques, and the Verilog is also compiled, using high level synthesis tools such as SAM. [11]

Let us now consider the formal characterization of the discipline, and in particular, of this h/s codesign methodology. Figure 4 shows the algorithmic and register-transfer abstraction levels of a possible discipline hierarchy for h/s codesign.[10] (Recall that each of the "rectangles", or facets, shown in the figure is basically a container of property definitions, as defined in Section 4.2.) Deriving this hierarchy is the very first step that should be undertaken in characterizing a given design discipline an the corresponding methodology of interest Observe that an adequate discipline hierarchy is crucial to being able to deal, in an integrated and homogeneous fashion, with design objects as diverse as hardware components and software processes.

In Figure 4, the root facets at the algorithmic abstraction level and at the register transfer level (RTL), are designated simply "algorithmic level *object'* and "register-transfer level *objecf* respectively. These are indeed very general facets, in the sense that their properties must be meaningful in characterizing *any* design object represented at the particular abstraction level.[11] Structural and behavioral descriptions are thus typically included in these

9. Verilog is the hardware description language adopted in [10].

10. Additional classes of design objects, and specializations of the current ones, could certainly be added to this discipline hierarchy. Unfortunately, space limitations preclude us from being exhaustive. In this example we thus concentrate on illustrating the fundamental abstractions defined in the formalism, rather than in being exhaustive in characterizing the particular discipline at hand.

11. Observe that these properties will be inherited by all remaining classes of objects defined at this abstraction levels.

generic facets, since defining how to describe the structure and the behavior of an object is actually the very essence of what an abstraction level is. (Recall that these property definitions merely specify the languages and/or mathematical formalisms that would later be used in describing the actual structure and behavior of the object under design, at the particular abstraction level.) Moreover, very fundamental requirements, such as "cost," "area," "power," and/or "speed," may be also associated with these general (root) facets.

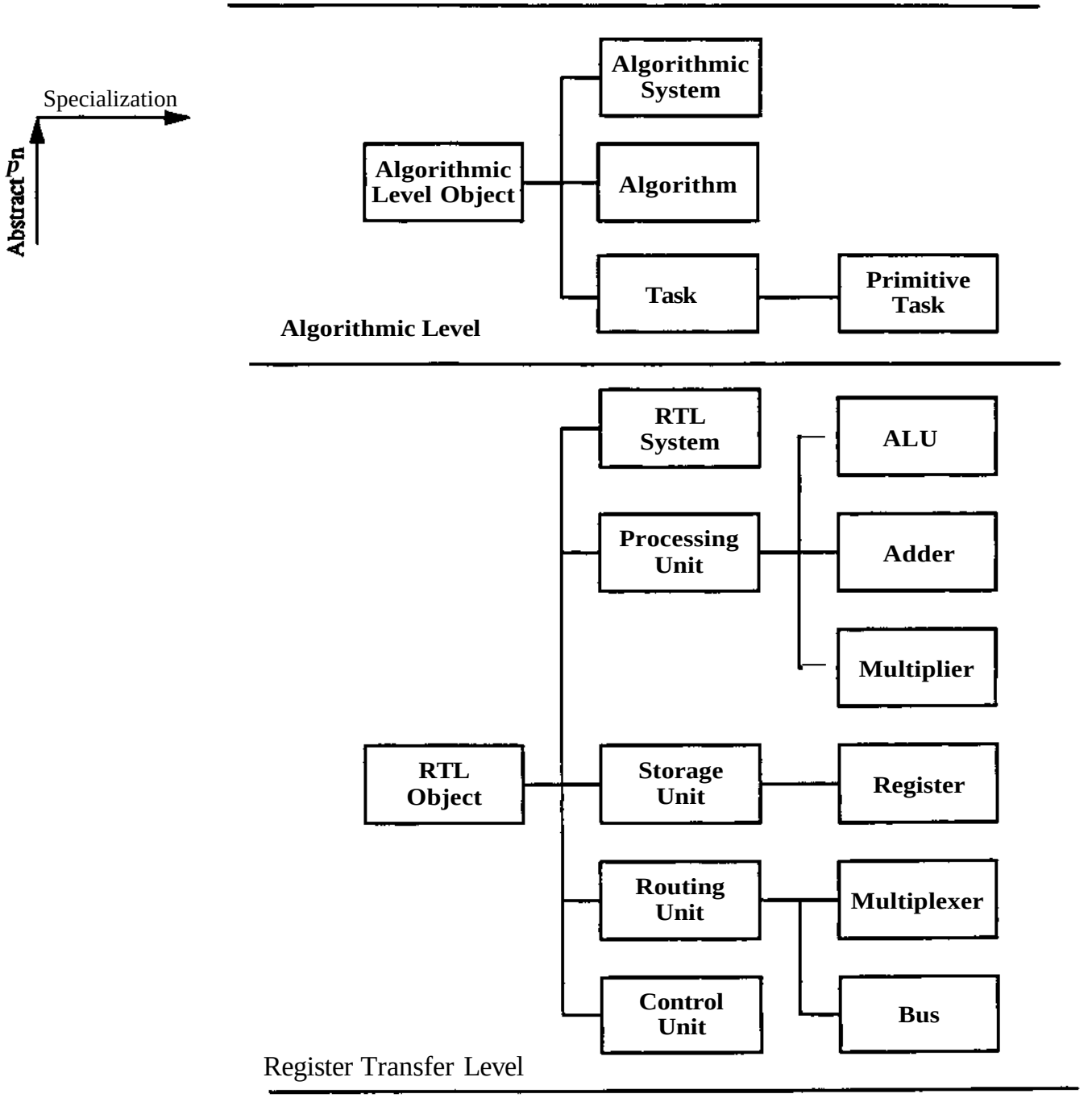

Figure 4 Partial representation of the discipline hierarchy for H/S Codesign

The first level of specialization for the algorithmic abstraction level, contains three facets: "task", "algorithm", and "algorithmic system." Thus, a generic "algorithmic level object" can be: (1) a "task," or chunk of behavior; (2) an "algorithm," or process;[12] and (3) an "algorithmic system", which is a collection of algorithms, or communicating processes. We may wonder at this point why "algorithm" is not defined as a specialization of "algorithmic system," since the former may be intuitively perceived a "simpler case" of the later. To understand this it is important to clarify the difference between two important relations among classes of objects: the *isja* relation, that leads to specialization, and the /?arr_0/relation, that leads to discrimination, i.e. creation of sibling object classes. The "algorithm" (i.e., sequential process) facet is being classified as a sibling of the "algorithmic system" facet (i.e., set of communicating, sequential processes) because some important characteristics of an "algorithmic system" (e.g., communication and synchronism protocols) are simply not present in a single sequential process, or "algorithm." Observe that in a specialization chain, by definition, features can only be added to the more specialized classes.[13] Therefore, an "algorithm" is *not* a specialization of an "algorithmic system", or conversely, an "algorithmic system" is not a generalization of an "algorithm." Let us know look at the specialization chain defined by the classes "task" and "atomic task," also shown in Figure 4. In this case, the second facet is indeed a special case of the first, because all features associated with "task" are shared (and relevant) to "atomic task." [14]

Let us now discuss the register transfer level (RTL) of abstraction. The derivation and organization of facets at this abstraction level follows the exact same general principles discussed before. So, as shown in Figure 4, we also define an "RTL *system"* facet and several other facets, such as "processing unit", and "storage unit," that may actually be a *part of such* a systems (i.e., an RTL system, in the general case, is *not,* simply, a "processing unif \ or a "storage unit"). ALU and Adder facets, on the other hand, have an *isja* relation with processing unit, i.e., they share all of the specifics of a "processing unit", and are therefore classified in the discipline hierarchy (see Figure 4) as specializations of this last facet.

Observe, finally, that these facets (and their possible specializations) define the *only* meaningful object classes at each abstraction level. This means that the property definitions contained in these facets must allow designers to fully characterize any design object represented at the particular abstraction level.

Let us now turn to the characterization of *objectives.* In simple terms, objectives characterize the sub-problems that have to be solved while addressing a general design problem in the context of the methodology of interest and define how each of such sub-problems should be sequenced during the problem solving process. To capture the h/s

12. In deriving the discipline hierarchy, we should be only as general as it is necessary for adequately capturing the fundamental concepts embedded in the discipline methodology. Since in our methodology of interest the concept of an algorithm and the concept of a process are equivalent, the discipline hierarchy should reflect that fact.

13. This is the reason "algorithmic system**, and "algorithm** can be both specializations of 'algorithmic level object*'

14. *"Atomic"* tasks constitute the primitive behavioral building blocks at this abstraction level. A set of such atomic tasks could thus be included at this abstraction level, as specializations of the generic "atomic task" facet Or, instead, directly as specializations of the "task** facet.

codesign methodology we need to define an objective that represents the central design problem being addressed by the methodology, as well as at least one (sub)objective for each of the design steps defined by the methodology.

Thus we can define one objective to be "codesign." Since "codesign" is a very complex objective that it is unlikely to be directly achievable (i.e., achievable in just one problem solving step, by direct application of an operator), we must specify how it can be decomposed. Specifically, in the control knowledge body of "codesign," we will indicate that it can be decomposed into the sub-objectives: "task derivation", "style selection", "architecture mapping", "cosimulate", and "design." We must also specify a valid sequencing for these sub-objectives, indicating what action should be taken both upon success as well as failure of each of the sub-objectives.

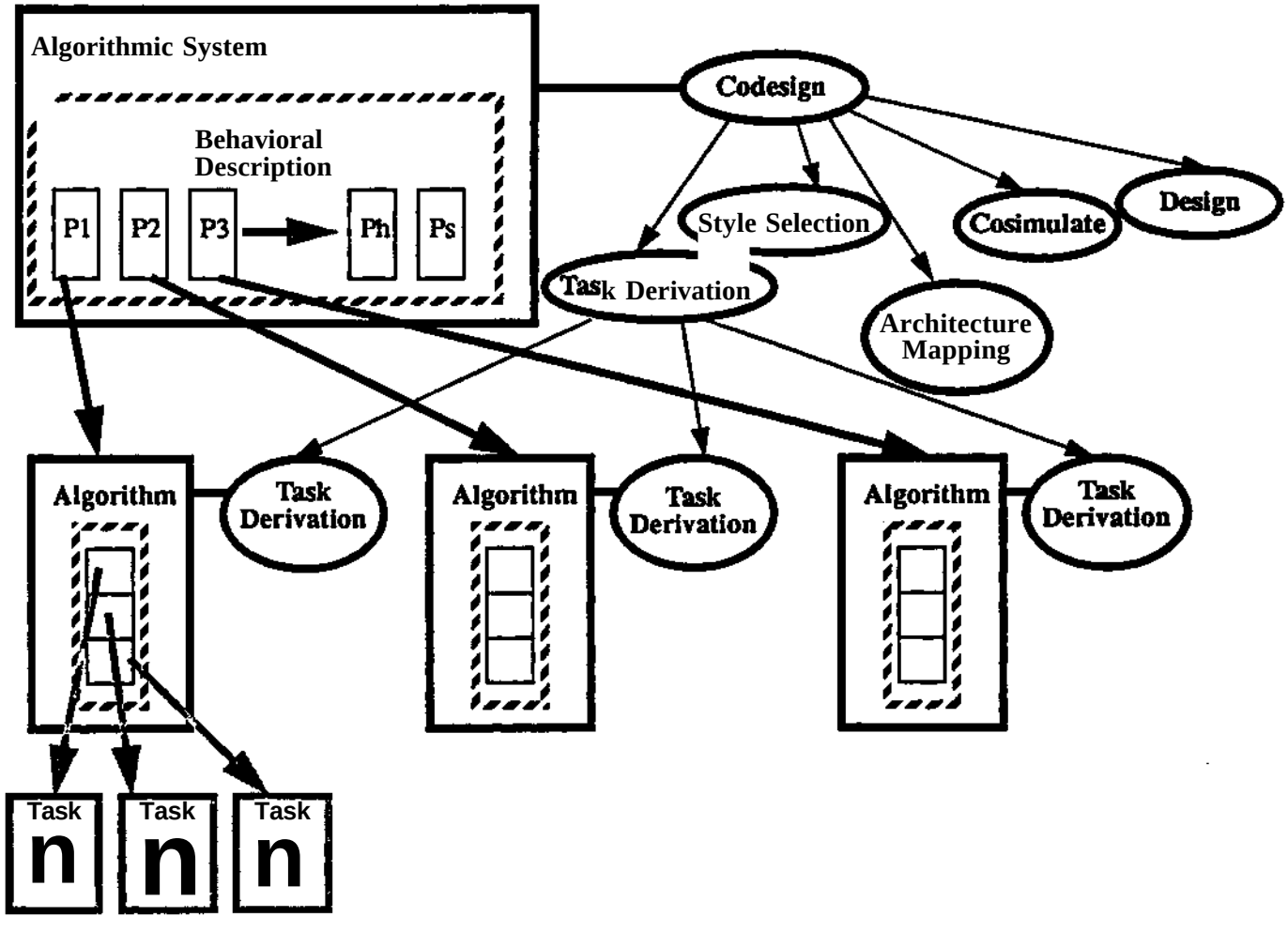

Figure 5 Representing the design state for a codesign process

Recall that upon problem decomposition, each resulting sub-objective focus attention on a particular sub-set of the specifications defined for the target design object (or objects) associated with the problem defined by the parent objective. To better illustrate how these concepts map into actual objective definitions, consider Figure 5 which shows a high-level, problem-based representation of an advanced stage of the codesign process. In this symbolic representation of the *design state,* rectangles represent design objects and bubbles represent design objectives. (In

this simplified view we have omitted the individual properties that are targeted by each individual (sub)objective, and the network of consistency constraints that relate these properties. Furthermore, only the behavioral descriptions are explicitly shown in the boxes, even if other properties reside within these facets.)

To see how the design might have progressed to result in the state shown in Figure 5, recall that design is a problem solving process aimed at solving one main design problem. As discussed earlier, a design problem is defined in terms of a set of properties (that can be associated to one or more design objects) and by an objective, that defines the specific design goal to be achieved in the context of such properties. We thus start by identifying the main problem that lead to the design process being shown in Figure 5. Since the initial specification given for the object under design included a behavioral description given at the algorithmic level, and consisting of a *set* of independent, interacting, sequential processes, the main object under design was identified (according to our discipline hierarchy) as a member of the class "algorithmic system." Since the goal was to design such an object through the h/s codesign methodology, the objective *codesign* was associated with this algorithmic system object, thus defining the main design problem to be solved through the design process.[15] As mentioned before, the initial specification of the design object may also include a set of global requirements, such as *cost* and *speed.* Values for these requirements may also be given at the time the main problem is defined. (See Section 8 for a brief discussion on how can the definition of the main design problem to be addressed in a design process be effectively implemented in a meta-tool such as Minerva.)

As soon as the design process commences, an impasse is reached by the objective "codesign," since there is no tool (operator) that can directly solve the codesign problem. According to the control knowledge associated with the objective "codesign," the objective is decomposed into "task activation", "style selection", "architecture mapping," "cosimulate," and "design," (see Figure 5). "Codesign" then activates the first of these sub-objectives, i.e., "task derivation."

'Task derivation" focuses on the behavioral description of the "algorithmic system object." The goal of task derivation is to decompose algorithms into chunks of behavior, or tasks. Since the current behavioral description for the target object consists of a *set* of algorithms, "task derivation" prepares for achieving its goal by performing an "initial" design object behavioral decomposition. In short, it instantiates (in the design process hierarchy) a new design object for each of the algorithms (or processes) contained in the original behavioral description, and then decomposes itself into one "task derivation" sub-objective for each of these new design sub-objects, as shown in Figure 5. All "task derivation" sub-objectives will then be simultaneously activated (by the parent "task derivation" objective), creating an opportunity for concurrent design. As shown in Figure 5, the resulting descendent design sub-

15. The set of active consistency constraint (and also the set of related 'Verify'* predicates) associated with the main problem directly derives from these two components, i.e., does not need to be explicitly "stated."

objects will still be represented at algorithmic level, but will now be of the class "algorithm" since their behavior encompasses only a single algorithm.

Each of the task derivation sub-problems should now be directly, and independently (maybe even concurrently) solved. Each time a task derivation sub-objective reports completion to the parent task derivation sub-objective, the process of decomposing one of the original "algorithms" into a set of tasks has been completed. (Figure 5 shows that each such "task" object could also be instantiated in the design process hierarchy, but there is no apparent need for such an instantiation in the codesign methodology being described, and therefore it is not actually performed.) When all "task derivation" sub-objectives report successful achievement to the parent "task derivation" objective, the parent objective collects the behavioral descriptions from all of the objects related to the sub-objectives and stores them in the behavioral description associated with the main "algorithmic system" object, so that the overall consistency of the design state is restored. The "task derivation" reports success to the "codesign" objective, upon which the "style selection" objective is activated. (Recall that in the control knowledge component of the definition of "codesign," if should be specified what action to undertake in case of success and in case of failure of each of the sub-objectives).

"Style selection" focuses on the now reformulated behavioral description of the "algorithmic system". The general goal of this objective is to determine which tasks will be implemented in hardware and which tasks will be implemented in software, and properly group these tasks into processes. Since the h/s codesign methodology requires that all tasks be considered at the same time, no problem decomposition is performed at this time, meaning that the problem is to be solved directly, possibly using an interactive CAD tool as operator. Thus there is no need to specify objective decomposition in the control knowledge of the "style selection" objective. The outcome of this design step is a new behavioral description for the "algorithmic system" object. In Figure 5 this is symbolically represented by PI, P2, and P3, the three original processes, being "transformed" into two distinct processes, Ph and Ps. Observe that requirements, such as "cost" and "speed" may play an important rule in determining an adequate solution for the "style selection" problem and therefore these properties will be made available to the "style selection" operator (i.e., the CAD tool that will be used in addressing the problem). When the "style selection" objective reports completion to the "codesign" objective, the sub-objective "architecture mapping" is activated.

Architecture mapping is responsible for defining the particular *hardware/software system architecture* that will be adopted in realizing the algorithmic system. It focuses on generating the algorithm level structural description for the "algorithmic system." According to our methodology, such an architecture will typically consist of some application specific hardware on the system bus of either a general purpose or an embedded computer system running an appropriate operating system. (These various architectural alternatives can be pre-compiled and presented to the designer, for selection. In such a case, the pre-compiled options constitutes the set of possible alternative values for a property of type restriction, as discussed in Section 4.1.2. These options could completely or just partially determine the "value" for the actual structural description property for the target "algorithmic system." A discussion on

this and other kinds of "strategic'* decisions concerning the details of the discipline's characterization is beyond the scope of this paper, though.) Again, requirements, such as "cost" and "speed," may play an important rule in determining an adequate architectural solution for the particular "algorithmic system."

After "architectural mapping" is concluded, "codesign" activates the "cosimulate" sub-objective, which goal id to asses the performance of the solution under development, i.e., to evaluate the correctness and adequacy of the current partition, style selection, and architecture. Observe that this is the very first "test" problem encountered in this methodology. The operator used by "cosimulate" is a Verilog-C cosimulator. [10] If the performance is considered satisfactory by the designer, "cosimulate" reports success, and the final objective, "design," is activated.

Similarly to what happened with the "task derivation" objective, "design" starts by performing a structural design object decomposition of the "algorithmic system" object, by instantiating (in the design process hierarchy) a new design object for each of the architectural components that have to be designed. Then "design" transfers to each of such sub-objects the particular behavior they should implement Finally, "design" decomposes itself into one "design" sub-objective for each of these sub-objects, and activates all of such "design" sub-objectives simultaneously, thus creating a new opportunity for concurrent design. (For simplicity reasons, these "design" sub-objective and their related design sub-objects are not shown in Figure 5. Instead, one of such components is individually shown in Figure 6.)

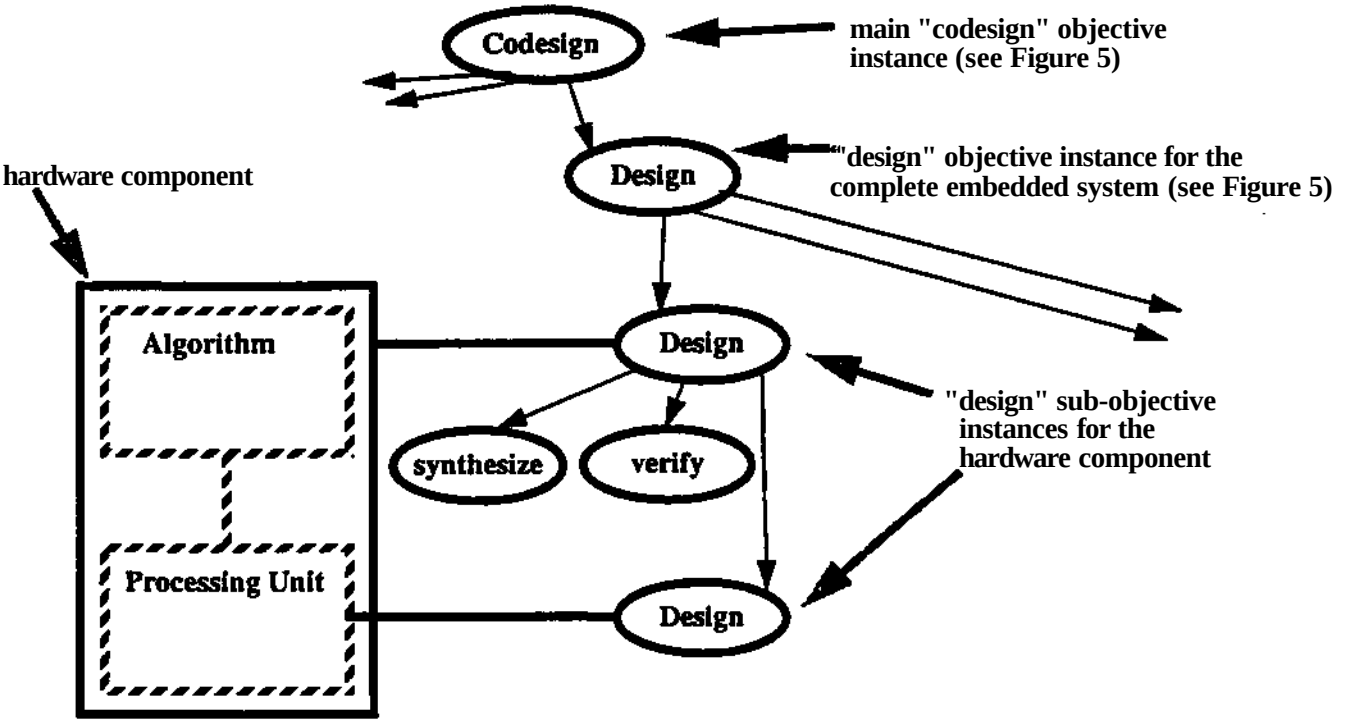

Figure 6 A transformational synthesis step, from algorithmic to register-transfer level

The C software is then compiled (i.e., "designed") using operators implementing cross compilation techniques. The process of designing the hardware components can be significantly more complex, though, since a number of transformational synthesis steps (e.g., from algorithmic to register-transfer level, and from register-transfer level to logical level) may be required. Moreover, these steps must be interleaved with verification steps, as discussed below.

Figure 6 shows in some detail how can transformational design be recursively modeled, mainly by adequately defining the objective "design." So, the objective "design" associated with each one of the hardware components has first to be decomposed into a "synthesize" and a "verify" sub-objectives, and "synthesize" should be first activated. (This decomposition is shown in Figure 6, for one of the hardware components.) The goal of a general "synthesize" sub-objective is to generate a more detailed description of the object under design. Specifically, in our example the "synthesize" objective shown in Figure 6 will focus on the Verilog behavioral description for the particular component, and will try to generate a register-transfer level description consistent with it, using operators implementing behavioral synthesis techniques. When such a representation is created, "synthesize" creates a new facet for the object under design, representing the abstraction level at which the object is now being described, and stores the new description in it. After that, the objective "synthesize" reports completion to the parent "design" objective, and this last activates the "verify" sub-objective. The goal of the "verify" sub-objective is to check if the RTL solution just generated meets the set of requirements stated for the particular component If the cost and performance of the solution are considered satisfactory, "verify" reports success to the parent "design" objective. The objective "design" then checks if the current representation of the design object is at the ground level of abstraction. If yes, "design" reports completion to its own parent "design" objective. Otherwise, "design" instantiates a new "design" sub-objective and associates it with the new, more detailed representation of the design object, as shown in Figure 6. (Observe that such a design object representation is now stored in a different facet of the same design object.) The transformational design process then proceeds, recursively, exactly as defined for this first step, until the ground level is reached, unless otherwise is decided by the designers. (For a detailed discussion on how to represent alternative or non-default control decisions in a design objective definition see [9].)

To this point in our example we have been addressing the active problem solving aspects of the h/s codesign methodology. Since h/s codesign tends to be highly iterative in nature, let us know briefly address how to model the aspects of the h/scodesign methodology that deal with iteration, i.e., those related with deciding on effective backtracking strategies given specific failures during the h/s codesign process. It is important to note that choosing the most adequate backtracking strategy in each case is crucial for a *controlled* and *coherent* exploration of the solutions space, assuring that a satisfycing solution, if existent, will be found in efficient time. Or, conversely, that if a solution does not exist for a particular problem, the situation will be identified in reasonable time.

In order to illustrate the discussion, assume that during the verification step of one of the "algorithmic" objects, say the one represented in Figure 6, it became clear that the RTL solution just generated for the particular component does not have the potential to meet a specific sub-set of the object's requirements. Three fundamental backtracking

strategies can be considered at this point: (1) re-synthesize the specific component, by trying to impose adequate restrictions on the behavioral synthesis process; (2) modify the behavior for the particular component being designed, by partially re-doing style selection; or (3) modify the set of tasks, by partially re-doing partitioning. Observe that partially re-doing partitioning allows the designer to generate significantly different solutions, while re-synthesizing just the specific hardware component only allows the designer to move very locally in this space. Yet re-synthesizing is much less costly than partially re-doing style selection, which in its turn is much less costly than re-doing partitioning. Clearly, a more costly backtracking step should only be tried if the immediately less costly one proves to be incapable of generating a potentially satisfycing solution. As will be explained below, active consistency constraints, by explicitly capturing relations among properties, play a decisive role in determining adequate and effective backtracking points given a particular failure.

Active consistency constraints stating relations among *local* requirements should be the ones to be first considered, in order to access the adequacy of the cheapest backtracking strategy, i.e., strategy (1). In order to better illustrate these ideas, let us again consider our example. Assume that the three fundamental requirements to be met by the object in Figure 6 are "cost", "delay" and "path." ("Cost" provides a measure on how expensive the solution will be, in terms of its required resources, i.e., registers, multiplexers, adders, etc. "Delay" provides the critical combinatorial path for each control step. Finally, "path" measures the total number of control steps between specific operations.) As part of the discipline characterization, the set of consistency constraints (informally) shown below could be defined among such requirements. Observe that these consistency constraints state *qualitative inverse proportionalities* among the requirements "cost", "path", and "delay", i.e., they explicitly indicate that trade-offs are possible among these requirements. Observe also that each consistency constraint is annotated with comments on how to actually implement the specific trade-off.

(1) Cost

- Qualitative inverse proportionality with *Delay*. Trade-off: try to use less costly (slower) resources.
- Qualitative inverse proportionality with *Path*. Trade-off: try to share resources.

(2) Delay (the critical path is too long)

- Qualitative inverse proportionality with *Cost*. Trade-off: try to use fast (more expensive) resources, or try to diminish resource sharing on critical path.
- Qualitative inverse proportionality with *Path*. Trade-off: try to "break" critical path, by introducing extra control steps.

(3) Path (too many controls steps in between two specific operations)

- Qualitative inverse proportionality with *Cost*. Trade-off: try to diminish resource sharing.
- Qualitative inverse proportionality with *Cost&Delay*. Trade-off: try to chain together operations in target path, maybe using faster resources.

If some of the trade-offs expressed by these consistency constraints happen to be possible among the solution's current unsatisfied and satisfied requirements, then the current candidate solution may be quite close to a satisfycing one. In other words, the most adequate backtracking strategy would be to resynthesize the particular component, imposing the trade-offs indicated by the set of relevant active consistency constraints. For instance, if the requirement "cost" is not being meet by the object under design, but if there is some margin for trade-offs on the requirements "path" and/or "delay," i.e., if these requirements are being over achieved by the current solution, then the first set of consistency constraints shown above (tagged (1)) determine the most effective backtracking strategy to pursue, and maybe even using the tactics suggested by these consistency constraints.

On the other hand, if no local trade-offs are possible among these local requirements, it may be that the particular region of the solution space being explored does not hold any acceptable solutions. The network of active consistency constraints, relating all of the relevant properties in the particular h/s codesign process (e.g., each of the original algorithms to the resulting pull of tasks, each of such tasks ask to the particular hardware and software process where it ended up being implemented, and so forth) should then be accessed again, and the next least costly backtracking strategy should be considered, as discussed before. In conclusion, then, the network of consistency constraints captured in the design state provides the "causality chain"[9] needed for effectively implementing a controlled and effective search of the global and also of all of the "local" solution spaces that can be defined on a design processes of this level of complexity.

9 Minerva

Due to its inherent complexity, the h/s codesign problem must be decomposed into smaller subproblems, and CAD tools should be developed to efficiently address each of these sub-problems. However, *conventional CAD tools cannot implement a h/s codesign methodology*, or, for that matter, cannot implement any complex design methodology. For instance, the history of the design process — the set of iterations already made in the particular codesign process — cannot be captured by conventional CAD tools. In addition, the semantics of these iterations — the reasons for backtracking to specific points and what these iterations mean in terms of the *coverage of the solution space* — would also be missing. Moreover, handling of sub-problem interactions, during active problem solving and during backtracking, cannot, again, be handled by these conventional, "local" CAD tools. In summary, then, any high level strategic decision that needs to rely on some sort of global view of the entire design process, cannot be achieved by just using "local," conventional CAD tools. If we consider the current trend in industry of moving towards total product development, including manufacturing, and the adoption of concurrent design practices, sometimes distributed among geographically separated sites, it is highly improbable that all planning and management strategic decisions associated with these highly complex, cross-disciplinary, and distributed design processes, can continue fundamentally relying on "informal mental pictures" maintained by individual designers. It is highly inefficient, and even more important, too risky.

In this paper we have presented a formalism of design that allows for the explicit characterization of design disciplines and provides a unified, problem based representation of design processes. This formalism is the basis for the creation of a new generation of powerful CAD meta-tools that can implement arbitrarily complex, cross-disciplinary design methodologies, and offer advanced design process planning and management services based on such methodologies, answering to the industry needs. One such tool is Minerva. [2] Minerva is capable of dynamically generating a global, unified, semantically rich representation of the entire design process, and based on such a representation, provides designers with a set of high-level design process planning and management services that 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; and effective handling of problem interactions in all of the above situations. All of the above services are offered assuming the most complex scenario, i.e., a concurrent design environment

Minerva is discipline independent and can be customized to any design discipline. It is therefore highly suitable for supporting total product development.[16] One of Minerva's key components is a knowledge base where the explicit characterization of the discipline of interest, as defined in the formalism (i.e., relevant design objects, organized into a discipline hierarchy, and the set of design objectives and consistency constraints necessary for capturing the specifics of a particular methodology or set of methodologies of interest) can be declaratively defined. So, designers may browse the *discipline hierarchy,* as defined in the knowledge base, in order to select the target design object for a given design process. Designers may also browse the *"library" of objectives,* in order to select the class of design problem that is to be solved in the context of the selected design object Observe that selecting the top level objective ultimately means selecting the methodology that will be used in addressing the particular problem. (In Minerva, nothing preclude the definition of two or more "codesign" objectives, each one embodying a different codesign methodology, though. Alternative objective decompositions, offering (on the fly) alternative methodologies, are also supported by Minerva. For a complete discussion on Minerva's knowledge base see [9].)

Figures 5 and 6 show the global view of the design process exactly as it is symbolically represented by Minerva. Designers working concurrently in a design process through Minerva are shown snapshots of this view, whenever they wish to select a new sub-problem to work on. After the sub-problem selection process in concluded, designers are allowed to fully concentrate their effort on solving the particular sub-problem at hand (using specific, conventional CAD tools), and Minerva will automatically provide them with all of the *potentially relevant* information being developed outside the context of the particular problem. (This information sharing is implemented in Minerva through the active consistency constraints). Also, when a failure occurs, Minerva will not only provide design-

16. It is important to note that this is radically different from developing a design environment from scratch that embodies the specifics of a particular discipline and methodology, as is the case, for instance, of the environment for hardware-software codesign described in [12]. Clearly, this kind of dedicated, "hardcoded solution" is not suitable for total product development, since each product would ultimately require the development of a dedicated environment capturing its specific needs.

ers with a global, semantically rich view of the entire design process, thus allowing the particular failure to be put in perspective, but will also suggest adequate backtracking strategies, as discussed before.

In closing, we feel that a meta-tool such as Minerva, if properly used, will ultimately allow for design processes to be *repeatable* and subjected to a systematic accessmenr. Unfortunately, space preclude us from discussing Minerva in more detail. For a detailed description of the Minerva design process planning and management meta-tool see [2][4].

10 Conclusions

We have presented a formalism of design that allows for the explicit characterization of design disciplines and provides a unified, problem based representation of design processes. This formalism is the basis for the creation of a new generation of powerful CAD meta-tools that can implement arbitrarily complex, cross-disciplinary design methodologies and offer advanced design process planning and management services based on such methodologies. Such services include: (1) full support to concurrent design, by allowing for the integration and optimal (partially ordered) sequencing of all of the sub-activities comprised in arbitrary product development processes, and by promoting an adequate handling of sub-problem interactions in such a complex scenario; and (2) full support to "global" backtracking strategies, i.e., backtracking strategies that may traverse an arbitrary number of levels of abstraction, and possibly relate decisions concerning to objects of radically distinct natures. Observe that in the context of complex, cross-disciplinary methodologies, which tend to be highly iterative in nature, this last feature may have a dramatic impact on productivity, by allowing for a controlled and effective exploration of the (typically huge) solutions space. Observe, finally, that this formalism, may also be useful in helping designers to develop and to evaluate, in a systematic form, complex design methodologies and available, conventional CAD tools.

11 Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] M.F. Jacome, and S.W.Director. "Minerva: A Meta-Tool for Design Process Planning and Management." To appear.

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

[6] P.R. 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.

[7] 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.

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

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

[10] 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.

[11] R. Cloutier, and D. Thomas, "Synthesis of Pipelined Instruction Set Processors" In Proceedings of the ACM/ IEEE 30th Design Automation Conference, ACM, 1993

[12] T.B. Ismail and A. A. Jerraya. "Synthesis Steps and Design Models for Codesign." *Computer,* pages 44-52, February 1995.