A Description Language for Design Process Management*

Peter R. Sutton and Stephen W. Director
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

A language for defining design discipline characteristics is proposed. Design discipline characteristics such as abstraction levels, design object classifications and decompositions, design objectives, and design methodologies can be defined in a simple machine and human readable form. The language, DDDL, has led to the development of a user interface for Minerva II, a design process management tool, which, when configured with a design discipline description, can be used to aid teams of designers in the solution of complex multi-disciplinary design problems. DDDL and user interface examples are presented.

1 Introduction

As the design problems that today's electronic designers face become ever more complex, and the time in which these designs must be carried out becomes ever shorter, the need has emerged for better design management services to aid designers in their task [8]. Initial electronic design (CAD) frameworks provided *design data management* in which design file formats, locations and versions were managed by the design system and *design tool management* in which tools were provided with tool encapsulations [4]. More recent CAD framework systems, both commercial and research systems (e.g. [9],[10] and [11]), have progressed to a higher level of design management - that of *design flow management* or *design methodology management* [8] in which the sequence of tasks a designer must carry out to solve a design problem are managed and possibly automated. These services, however, are no longer sufficient for today's complex design processes due to two main design trends.

One design trend is the move to carrying out design at higher levels of abstraction. For example, digital systems once designed at the gate level using schematics are now designed textually at the RTL level of abstraction, or at the behavioral level of abstraction using Hardware Description Languages (HDLs) such as Verilog and VHDL. Limitations are arising though, because as device sizes decrease and interconnect delays become more important, design at higher levels of abstraction is becoming increasingly dependent on performance characteristics only observable at lower levels of abstraction. Thus, while design is simpler at the higher levels of abstraction, difficulties arise because it is necessary to manage data interactions between the levels of abstraction.

A second design trend is the move towards concurrent multi-disciplinary design so as to reduce design times and hence time-to-market. Examples of this include hardware / software codesign, in which the hardware and software parts of an electronic system are designed concurrently; and circuit / process codesign, in which an integrated circuit and its associated manufacturing process are designed together. It is necessary to better manage design problems of this type, coordinating information between individual designers and the different parts of the design.

To better aid designers in solving these more complex problems, design management services themselves must move to a higher level of abstraction. This level, known as the *design process management* level [1], provides support for, and control of, all aspects of the solution of design problems including conceptual (or exploratory) design, problem decomposition, backtracking, constraint propagation and management, and design history management. It also encompasses support for existing design management services provided at lower levels of abstraction, including data, tool, flow and methodology management.

The provision of these design process management services requires some *knowledge* of design discipline characteristics and how they interact. Typically, much of this knowledge has been kept in designers' heads and on paper. As design processes become more complex, it is becoming necessary for design process management services to be computer supported. In order to do this we must capture the design discipline knowledge. We have developed a language called DDDL (for **D**esign **D**iscipline **D**escription **L**anguage) to accomplish this task.

DDDL has grown out of the work by Jacome [5],[6] which, in proposing a formal theory of design, established that design processes can be described in terms of these characteristics. The Minerva Design Process Manager [7] developed by Jacome, however, did not provide a mechanism for capturing design discipline knowledge and was not able to be configured for arbitrary design disciplines. DDDL was developed to overcome this deficiency. Minerva II, which implements a DDDL parser, is easily configured with design discipline information and can manage design problems in that discipline (or disciplines). DDDL has also allowed for the realization of a more intuitive user interface in Minerva II. The new user interface allows for simpler management of design problems and improves the efficiency of the design cycle.

Characteristics of design disciplines which can be described using DDDL include:

- the abstraction levels relevant to a discipline;
- the types of design objects within a discipline and how these design objects can be decomposed;
- the types of design problems solvable in a discipline and how they can be decomposed;
- the design methodologies that exist for solving problems within a design discipline; and
- the restrictions each design methodology imposes upon a designer.

Before describing DDDL and Minerva II, it is useful to discuss how our work relates to other work in this field. As stated above,

** This work is supported by the Semiconductor Research Corporation under contract # 96-DC-068.*

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

design process management is a level of service above that provided by current commercial and research CAD frameworks. These frameworks (e.g. [9], [10] and [11]) provide flow-based execution management and are able to manage simple methodologies but do not provide the higher level management of decomposition, backtracking, conceptual design and constraint propagation that is essential for more efficient design cycles. The lower level services provided by these systems are still important though, as it is through these services that a design process manager is able to manage tool executions and data.

To our knowledge, there are no existing languages for formally describing the electronic design process at a high level. Previous languages have been restricted to describing lower level details, for example, the task specification language of Chiueh et al. [2], which encapsulates tool details and provides for the sequencing of tasks but, unlike DDDL, doesn't describe the higher level details of a design discipline. Dewey [3] developed structures for representing properties and constraints in a design domain (in this case DSP filters) so as to support conceptual design. Again, information about decompositions, possible problems and methodologies necessary for managing the complete design process is missing.

The remainder of this paper is organized as follows. In Section 2 we define the terms that we use in this paper. In Section 3 we present the details of the design discipline description language, and in Section 4 describe the details of the user interface developed for the Minerva II Design Process Manager. Finally, in Section 5, we draw some conclusions.

2 Terminology

To facilitate our discussion of DDDL, we first define the concepts that are important in describing a design discipline. These definitions are based on the design theory developed by Jacome [5],[6].

Design Discipline

A *design discipline* is a "field of design" encompassing all design objects of some broad type. For example, "digital circuit design" and "software design" are design disciplines. A design discipline may be much broader than this though, encompassing several of what may usually be considered design disciplines. For example, the electronic system design discipline may encompass hardware design - from digital to analog, from integrated circuit to printed circuit board; and software design - from operating systems to application programs. In the context of DDDL, we will restrict the use of the term *design discipline* to describing the field of design which encompasses all design fields of interest - in our case, electronic systems design.

Design Domain

A *design domain* is a design object classification or type. For example, "Hardware System" and "Processor" are design domains. Design objects in a given design domain are characterized by having certain types of *properties*. For example, design objects in the "Processor" design domain may have the properties "instruction set", "addressing modes", "timing specifications" and "unit cost". The design domains in a design discipline can be arranged in a *design domain hierarchy*, where the top level design domain corresponds to the design discipline itself, and lower level design domains inherit the properties of the higher levels. For example, design objects in the design domain "Digital Signal Processor" may have all of the properties of the "Processor" design domain, along with the additional properties "FFT speed" and "Maximum sample rate". We say that the design domain "Digital Signal Processor" is a *specialization* or *subtype* of the design domain "Processor". Fig. 1 shows a (partial) design domain hierarchy for the "Electronic System" design discipline. "Electronic System" (which is the top level design domain) has the specializations "Hardware System" and "Software System". Note that it is possible for a design domain to be a specialization of more than one design domain, for example, in Fig. 1 the design domain "Hardware / Software System" specializes both the "Hardware System" and "Software System" design domains, meaning that it has all the properties of both hardware systems and software systems.

Design Objective

A *design objective* is a design function to be carried out. For example, "Design", "Synthesize", "Verify", and "Optimize" are all design objectives. Design objectives are context dependent. That is, the design domain, target abstraction level and starting point all impact the meaning of a particular objective. This will become clearer below in our discussion of a design problem.

Design Problem

A *design problem* can be specified in terms of a design objective for a target design domain at a target abstraction level for some given input properties. The *target* design domain is the design object type to be created (or operated on), and the *target* abstraction level is the abstraction level at which we wish to represent our final design object. An example of a design problem is: "Design a hardware system at the layout level having a given behavioral description (at the behavioral level of abstraction), using a particular 0.5µm CMOS process and having a required area no larger than $1cm^2$". In this example, the design objective is "design", the target design domain is "hardware system", the target abstraction level is "layout" and the input properties are the behavioral description, the fabrication process, and the required area.

Design Decompositions

Design problem decomposition is the act of breaking up a design problem into smaller, easier-to-solve problems. Design problem decomposition can take one of two forms - design domain decomposition or design objective decomposition. *Design domain decomposition* is the breaking up of the design object into sub-objects with the same design objective being applied to each. For example, the design problem "design a hardware system" can be decomposed into the sub-problems "design a processor system" and "design a memory system". *Design objective decomposition* involves breaking up the design goal into sub-goals which apply to the same whole design object. For example, the design problem "design a hardware system" could be decomposed into the sub-problems "specify a hardware system", "synthesize a hardware system" and "verify a hardware system".

When decomposing a problem, *design constraints* are often generated. Design constraints relate the different parts of a design object. For example, when decomposing a hardware system into a

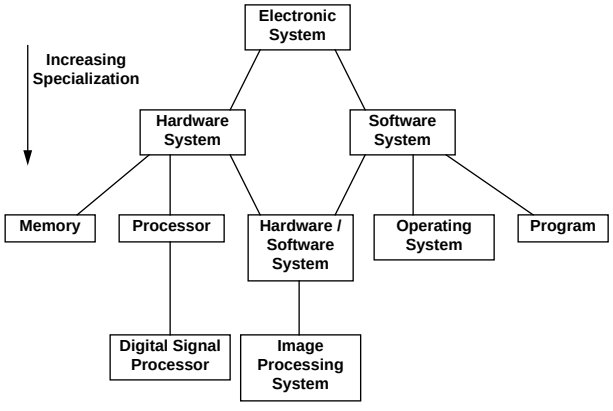

Fig. 1. An example (partial) design domain hierarchy for the "Electronic System" design discipline.

processor system and a memory system, several constraints may be generated. One constraint might apply to the interface between the two parts, ensuring that the two sub-systems will work together. Another constraint might relate the area of the two parts to that of the whole, ensuring that the restriction on the overall hardware system's area is not unknowingly violated.

Design Methodology

A *design methodology* is a specific method, approach and/or set of rules to be followed when solving a given design problem. For example, it may take the form of rules concerning the order of performing certain design tasks, how problems are to be decomposed, or, which particular tools are to be used for a certain task. Typically a company or group will specify a methodology for designers to follow so as to ensure consistent design results.

3 DDDL - A Design Discipline Description Language

In order to better manage the design process, a design process management system must have some knowledge of the design discipline characteristics described above and how they interact. In order to capture this knowledge, we have developed a language called DDDL.

DDDL can be used to describe all of the concepts defined above. Specifically, the language can describe abstraction levels; design domains and their specializations, properties and decompositions; design objectives and their input properties, output properties and decompositions; constraints; and design methodologies. We now consider each of these in turn.

3.1 Abstraction Levels

A design discipline may have many levels of abstraction. For example, the digital circuit design discipline often has behavioral, RTL, gate (or logical), transistor and layout levels of abstraction. Not all design objects in the discipline need to be characterized at all levels of abstraction, thus, a memory circuit may only be represented at the transistor and layout levels of abstraction, but not at higher levels. In fact, it may be impossible to characterize some design objects at some levels of abstraction. This is the case for the electronic system design discipline. This discipline covers both hardware design as well as software design, so software objects can not be represented at any of the abstraction levels for hardware and hardware objects can not be represented at any of the software abstraction levels. Abstraction levels in DDDL are specified with a simple declaration as in this example[1] for the electronic system design discipline depicted in Fig. 1:

```
ABSTRACTION LEVELS { "Behavior", "RTL",
  "Gate", "Transistor", "Layout", "Source
  code", "Object code", "Binary" };
```

3.2 Design Domains

Design domain declarations specify the various characteristics of design domains: defining the abstraction levels for the design domain including the ordering of the abstraction levels; listing which other design domains the domain specializes (or inherits properties from); and, listing the properties specific to the design domain. The following is an example design domain declaration using DDDL:

```
DOMAIN "Hardware System" {
  IS A SPECIALIZATION OF "Electronic System";
  HAS ABSTRACTION LEVELS {
    "Behavior", "RTL", "Gate", "Transistor",
    "Layout"
  };
  HAS PROPERTY "Cost" {
    AT ABSTRACTION LEVELS all {
      REAL, UNITS "$", RANGE ("Cost">0)
    };
  };
  HAS PROPERTY "Behavioral Description" {
    AT ABSTRACTION LEVEL "Behavior" {
      DESCRIPTION IN LANGUAGE "Verilog"
    };
    AT ABSTRACTION LEVEL "RTL" {
      DESCRIPTION IN LANGUAGE "RTL" OR
          "Verilog"
    };
    ...
  };
  HAS PROPERTY "Structural Description" {
    AT ABSTRACTION LEVEL "Layout" {
      DESCRIPTION IN LANGUAGE "Magic file",
          "CIF" OR "GDS II"
    };
  };
  ...
};
```

The example specifies that the design domain "Hardware System" specializes the "Electronic System" design domain, has the given abstraction levels (listed in decreasing order of abstraction), and has several properties.

Abstraction levels need not be specified for a design domain, in which case, the design domain inherits the abstraction levels of the design domain it specializes from (or, in the case it specializes more than one design domain, the intersection of the abstraction levels for each design domain it specializes). It is also possible to specify abstraction levels in addition to those that are inherited, rather than specifying levels absolutely.

Design domain properties can be one of several types: integer, real, string, boolean, option or description. Option properties are like enumerated types and may be restricted to a choice of one value from a set, or the choice of a subset of values. Description properties are for specifying longer textual or graphical information. Most of the property types require additional, possibly optional, information to be specified. Integer and real types can optionally have some units and a range limitation specified. String types can have a unit specification if desired. Option properties require a list of the possible options to be specified and whether one or many can be selected. Description properties require a list of the possible description languages to be specified.

Properties may be restricted to certain abstraction levels and may have different types or details at different abstraction levels.

3.3 Design Domain Decompositions

Many types of design objects can undergo design domain decomposition, indeed it is possible that a design object may have several possible decompositions. Design domain decompositions are specified within domain declarations as shown in this example:

```
DOMAIN "Hardware System" {
  ...
  HAS DECOMPOSITION {
    2+ "Hardware System"
  };
  HAS DECOMPOSITION CPU+Memory {
    1+ "Processor", "Memory", "Interconnect",
    1+ "Logic Block"
  };
};
```

Each decomposition sub-statement specifies the number of design objects of each of the decomposition design domains. This number

1. Note that in all code examples, indenting is for clarity only and an ellipsis indicates omitted detail.

may be an absolute specification (the default is 1) or a range, for example, “1+” meaning one or more, or “1-5” meaning one to five inclusive. Design objects may decompose into others of the same type or domain as shown in the first decomposition specification above. Decompositions are optionally named (e.g. `CPU+Memory`) for use in methodology definitions. This name is only used within the DDDL file.

3.4 Design Objectives

As stated above, design objectives need to be specified within the context of a target design domain and abstraction level, and some input properties. In addition to these characteristics, a design objective declaration specifies the output properties of the design objective and how the design objective may be decomposed. A design objective name (such as “Design”), a target design domain name (such as “Electronic System”), a target abstraction level (such as “Behavior”) and a given set of input properties must uniquely determine the output properties and the available decompositions. It is possible that the set of input properties may be empty; this corresponds to starting a problem from scratch (e.g. some design and specification problems). At least one output property must be specified, even if it is just a boolean value as may be the case with a verification design objective. The following is an example of a partial design objective declaration in DDDL:

```
OBJECTIVE “Design” {
  APPLIES TO DOMAIN “Electronic System”
      AT TARGET ABSTRACTION LEVEL “Behavior”
      WITH INPUT PROPERTIES none {
    HAS OUTPUT PROPERTIES “Behavioral
        Description”, “Behavior verified”;
    HAS DECOMPOSITION partial_design {
      “Specify”, “Verify”
    };
    HAS DECOMPOSITION complete_design {
      “Specify”, “Verify”, “Optimize”
    };
  };
  APPLIES TO DOMAIN “Software System”
  ...
};
```

As in domain decomposition specifications, the objective decompositions specified in the objective declaration may be given a name (e.g. `complete_design`) for later use in a methodology declaration.

3.5 Constraints

Constraints are declared to relate properties of various design objects. Multiple design objects come into existence only during design domain decompositions, so, it is within DDDL design domain decomposition specifications that constraints are declared. The following is an example DDDL constraint specification relating design object areas and delays across a design domain decomposition:

```
DOMAIN “Hardware System” {
  ...
  HAS DECOMPOSITION CPU+Memory {
      1+ “Processor”, “Memory”,
      “Interconnect”, 1+ “Logic Block”
    }
    WITH CONSTRAINT (“Area” = SUM(ALL)),
    WITH CONSTRAINT (“Delay” = MAX(ALL))
  ;
... };
```

The first constraint specifies that the “Area” property of the hardware system is equal to the sum of the “Area” properties of the component parts. The second constraint specifies that the “Delay” property of the hardware system is equal to the largest “Delay” property among the component parts. More complicated constraint definitions are possible, but space precludes us from giving further details.

3.6 Design Methodologies

DDDL allows several types of methodology specifications. The following may be specified for a given methodology:

- the design problems the methodology is available for;
- the particular design domain decompositions which are to be used;
- the particular design objective decompositions which are to be used;
- the default design options (e.g. fabrication technology); and
- the tools which must be used (if available) for certain tasks.

A methodology declaration in DDDL specifies all of the above. The following is an example declaration:

```
METHODOLOGY “Company X’s methodology” {
  APPLIES TO PROBLEM “Design”
      DOMAIN “Hardware system”
      WITH TARGET ABSTRACTION LEVELS {
        “Layout”, “Transistor”
      };
  REQUIRES “Hardware System” DOMAIN
      DECOMPOSITION CPU+Memory;
  REQUIRES “Design” OBJECTIVE DECOMPOSITION
      complete_design;
  HAS DEFAULT OPTION “CMOS” FOR PROPERTY
      “Fabrication style” OF DOMAIN
      “Integrated circuit”;
  HAS DEFAULT LANGUAGE “Verilog” FOR PROPERTY
      “Behavioral description” OF DOMAIN
      “Hardware system” AT ABSTRACTION LEVEL
      “Behavior”;
  REQUIRES TOOL “HSpice” FOR OBJECTIVE
      “Simulate” DOMAIN “Integrated circuit”
      AT ABSTRACTION LEVEL “Transistor”;
  ...
};
```

4 A Design Process Management User Interface

The development of DDDL has led to the development of Minerva II, a new version of the Minerva Design Process Manager [7] with a completely revised user interface. Specifically, problem definition, selection, backtracking and decomposition are now handled in a more intuitive way. In this section, we illustrate, through user interface screen-shots, how Minerva II, configured with a DDDL design discipline description is able to manage a design process.

It is important to note that Minerva II operates at the *design process level* - a level of abstraction above that of today’s CAD frameworks. Minerva II, however, can use the executive and data management services of most existing CAD frameworks. This is accomplished through a “framework encapsulation” interface enabling Minerva II to use the services which the executive provides. A prototype framework encapsulation of the Hercules Task Manager [9] has been implemented. We forego a discussion of this type of encapsulation so as to concentrate on the designer’s view of design process management.

4.1 Design Problem Definition

Designers using Minerva II are led through the problem solving cycle shown in Fig. 2. This figure illustrates the case in which several design problems can be considered concurrently.

The first step of the problem solving cycle, called *problem definition*, involves the selection of a target design domain and a design objective. This is achieved through a problem definition window as shown in Fig. 3. On the left side of the window, the hierarchy of design domains declared for the design discipline is shown. This

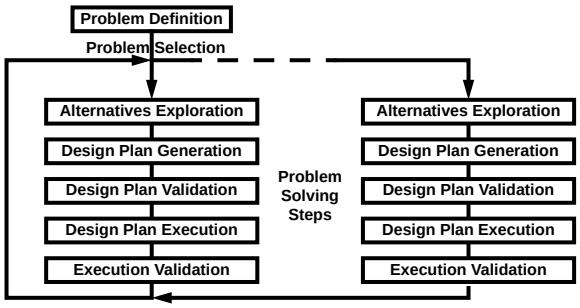

Fig. 2. Minerva II's problem solving cycle. The dashed line indicates that several design problems may be being solved concurrently.

hierarchy is directly derived from the DDDL description used to configure Minerva II. On the right side of the window is a listing of all the design objectives defined for the design discipline. The problem definition window is constructed in such a way that if a design domain is selected first, only the design objectives compatible with that design domain become available for selection by the designer. Alternatively, if a design objective is selected first, only the design domains compatible with that design objective are made available for selection by the designer. Once a design domain and design objective have been selected, the target abstraction level may be specified (as shown in Fig. 4) or, by default, chosen to be the lowest abstraction level for that design domain/design objective combination. It is also possible for the designer to specify a design methodology to follow. If a design methodology is specified, any restrictions contained within the methodology will be enforced by the system during the problem solving process.

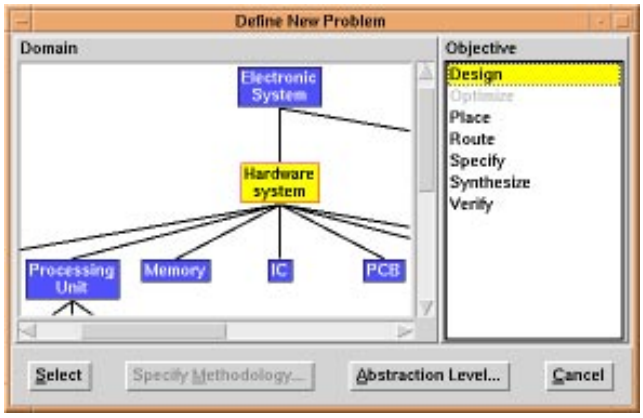

Fig. 3. Problem definition. A design domain and design objective must be selected in order to define a design problem. In this example, the design domain "Hardware system" and design objective "Design" have been selected.

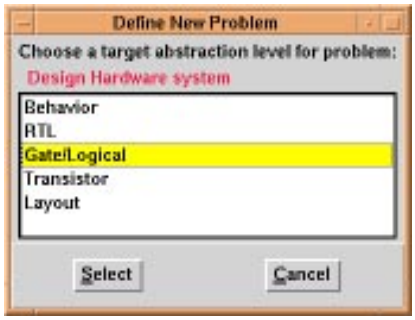

Fig. 4. Problem definition - abstraction level selection. A target abstraction level may be specified for the given design domain and design objective. Only those abstraction levels at which the design objective is defined for the design domain may be selected.

4.2 Problem Selection

Minerva II's problem status and selection window, shown in Fig. 5, is used to select a problem to solve (or to join a solution process in progress). All active problems are displayed and sub-problems of any given problem can optionally be displayed. Adjacent to each problem description is the current status of that problem. Problems can either be in progress, that is, in one of the five problem solving stages shown in Fig. 2, ready to be addressed, solved, or waiting for the solution of some other problem - either a sub-problem or a sibling problem (a different sub-problem of the same parent). Any problem that is ready to be addressed can be selected for solution and any problem whose solution is in progress can be continued. Multiple problems (or sub-problems) can be addressed concurrently (by the same or different users) if desired.

4.3 Problem Solving

After problem selection an alternatives exploration step is undertaken. During this step the designer explores possible trade-offs for the design problem at hand and specifies any restrictions on the problem (e.g. maximum power and area). Once this step is finished, design plan generation occurs. It is during this step that the framework executive system is queried as to which tools are available to directly solve the design problem under consideration. If appropriate tools are available, then Minerva II tries to validate the design plan by querying the designer about which of the generated design plans are acceptable. If any are acceptable, Minerva II instructs the framework executive system to execute one of the alternative plans and, when finished, presents the result(s) to the designer for validation. If at any time an impasse occurs (for example, no tools are available to solve the design problem directly, or the user rejects the generated design plans or execution result), Minerva II facilitates either decomposition of the design problem into design sub-problems and starts the design problem selection/design problem solution cycle again; or backtracking to an earlier design state. We discuss each of these in the next two sections.

4.4 Problem Decomposition

Design problem decomposition can be achieved either through design domain decomposition or design objective decomposition. When opting to decompose, the user is shown all available decompositions in a window as shown in Fig. 6, and may choose any one. Possible decompositions are determined from the DDDL design discipline description that is used to configure Minerva II, with the choice being limited if a methodology was specified during design problem definition. When a design domain decomposition is chosen, the sub-problems created appear in the Minerva II problem

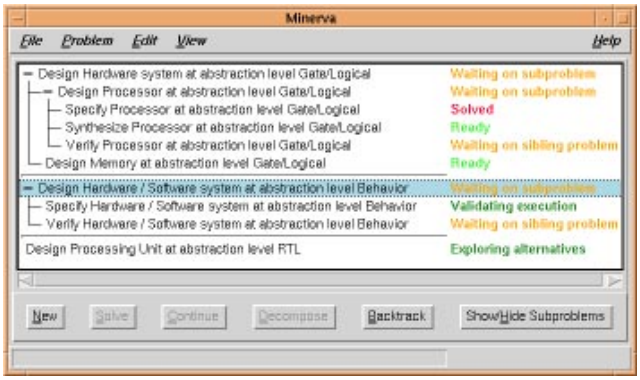

Fig. 5. Minerva II's Problem Status and Selection Window. All active design problems (and their sub-problems) in the system are shown.

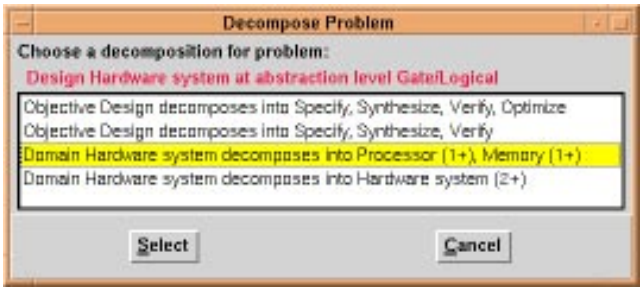

Fig. 6. An example problem decomposition selection window. The designer may choose one of the available decompositions for the current problem.

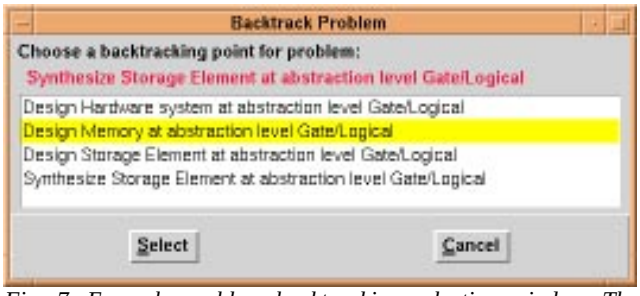

Fig. 7. Example problem backtracking selection window. The designer may choose a problem to backtrack to.

status and selection window. Sub-problems created through a design domain decomposition will all be ready to be addressed. If a design objective decomposition is chosen, the sub-problems must be solved in the order specified in the design objective decomposition declaration. Only the first sub-problem will be ready to be addressed, the others will be waiting on the solution of sibling problems.

When a design domain decomposition is chosen, an implicit recomposition sub-problem is created also. This sub-problem is able to be addressed when all of the other sub-problems have been solved.

4.5 Backtracking

When an impasse occurs (or at any time if desired), the designer may backtrack to a previous design state. When backtracking, the designer will choose a problem to backtrack to, as shown in Fig. 7. Any design problem in the hierarchy above the current design problem may be backtracked to - in which case the system returns to that design problem and makes it ready to be addressed again. If the current design problem is chosen to backtrack to, the designer may return to any previous problem solving step for that design problem (e.g. alternatives exploration).

4.6 DDDL Advantages and Limitations

Advanced design process management, as provided by design process managers like Minerva II, has several advantages over traditional approaches. Working at the design process level, designers are able to focus on the design problems being solved, rather than on low level tool and data manipulations. Interactions between designers working in different disciplines (e.g. hardware and software design) can be managed. Consistency constraints can be monitored so that inconsistencies can be detected earlier and backtracking commenced sooner. Both backtracking and problem decomposition can be managed. All of these features can lead to shorter design cycles.

There are few limitations to providing these extra services. One problem which may arise is incompleteness in the design discipline description, for example, in the case where a new type of artifact is being designed. In this case, the designer may have to consider the artifact at a higher level in the design domain hierarchy (e.g. as a hardware system instead of the specific system type), so the design process management system may not be able to assist the designer with all aspects of the design problem solution. Some useful services, such as decomposition and backtracking management may still be available, however.

5 Conclusions

We have presented DDDL, a language for describing design discipline characteristics such as abstraction levels, design object classifications or design domains, design domain properties and decompositions, design objectives and their decompositions, constraints, and, methodologies. DDDL allows the capture of design discipline knowledge into a simple human and machine understandable form. The description, once captured, can be used by design process management software to help designers manage the solution of their design problems. Support can be provided for managing constraints, backtracking and problem decomposition among other things. As design problems become more complex, these capabilities will be increasingly important for the success of electronic design processes. We have also illustrated how the Minerva II Design Process Manager, with an improved user interface, can be used to manage the problem solving process. Although DDDL was developed as a textual language, graphical representation of many of the concepts is possible and indeed helpful. A graphical editor is currently under development to allow the simpler creation of design discipline descriptions.

References

[1] J. B. Brockman, T. F. Cobourn, M. F. Jacome, and S. W. Director, "The Odyssey CAD Framework," *IEEE DATC Newsletter on Design Automation*, Spring 1992.

[2] T. F. Chiueh, R. H. Katz, and V. D. King, "Managing the VLSI Design Process," in *Proceedings of Computer-Aided Cooperative Product Development, MIT-JSME Workshop*, pp. 183-199, Nov. 1989.

[3] A. M. Dewey and S. W. Director, "Yoda: A Framework for the Conceptual Design VLSI Systems," in *Proceedings of 1989 IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 380-383, Nov. 1989.

[4] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes, "Electronic CAD Frameworks," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 393-417, Feb. 1990.

[5] M. F. Jacome, *Design Process Planning and Management for CAD Frameworks*. Ph.D. Thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering (Report CMUCAD-93-65), Nov. 1993.

[6] M. F. Jacome and S. W. Director, "A formal basis for design process planning and management," in *Proceedings of 1994 IEEE International Conference on Computer Aided Design* (ICCAD), pp. 516-521, Nov. 1994.

[7] M. F. Jacome and S. W. Director, "Design process management for CAD frameworks," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, pp 500-505, June 1992.

[8] S. Kleinfeldt, M. Guiney, J. K. Miller, and M. Barnes, "Design Methodology Management," *Proceedings of the IEEE*, vol. 82, no. 2, pp. 231-250, Feb. 1994.

[9] P. R. Sutton, J. B. Brockman, and S. W. Director, "Design Management Using Dynamically Defined Flows," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 648-653, June 1993.

[10] K. O. ten Bosch, P. Bingley, and P. van der Wolf, "Design Flow Management in the NELSIS CAD Framework," in *Proceedings of 28th ACM/IEEE Design Automation Conference*, pp. 711-716, June 1991.

[11] P. van den Hamer and M. A. Treffers, "A data flow based architecture for CAD Frameworks," in *Proceedings of 1990 IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 482-485, Nov. 1990.