A Planning Model of the Design Process

David W. Knapp

CRI Technical Report 87-06

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4476

December 1986

A PLANNING MODEL
OF THE DESIGN PROCESS

by

David W. Knapp

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

December 1986

Copyright © 1986 David W. Knapp

Acknowledgements

I would like to thank my advisor, Dr. Alice C. Parker, whose support and encouragement saw this research through many frustrating difficulties. I would also like to thank Drs. Melvin A. Breuer and Dennis McLeod, who served on my dissertation committee and provided invaluable advice, help and friendship. I would also like to thank Drs. Barry I. Soroka and John Choma, Jr., who served on my qualifying committee. The TRW corporation has my sincerest gratitude for the gift of the printer with which I produced multiple drafts of this thesis. Last, but definitely not least, I would like to express my gratitude for financial support received from: the National Science Foundation, grants #ECS-8310774 and #DMC-8310774; the Army Research Office, grant #DAAG29-80-k-0083; The Department of the Army, Ft. Monmouth, Contract #DAAK20-80-c-0278; The International Business Machines Corporation, grant #S 956501 Q LX A B22; and the Microelectronics and Computer Technology Corporation.

Table of Contents

1. Introduction 1
1.1. Problems in Digital System Design 1
1.1.1. Cost of Design 1
1.1.2. Goals and Constraints Vary Widely 3
1.2. Difficulties of Current CAD Systems 4
1.2.1. Multiple Design Representations 4
1.2.2. Tool Selection and Invocation 6
1.3. Unified Design Automation Systems 8
1.3.1. Varying Degree of Interaction 10
1.3.2. Automatic Management of Tools 13
1.3.3. Exploration of the Design Space 16
1.3.4. Feedback from Design Results 17
1.3.5. Flexible System 18
1.3.6. Robust System 19
1.4. Scenario 19
1.5. Thesis Statement 24
1.5.1. Unified Data Structure 25
1.5.2. Tool Kit Approach 26
1.5.3. Representing Tool Semantics Explicitly 27
1.5.4. Estimation on Incomplete Information 27
1.5.5. Planning 29
1.5.6. Object Based Computation 31
1.6. The Model 33
1.6.1. Syntactic and Semantic Compatibility 33
1.7. DPE: an Updatable Program 35
1.8. Structure of the Thesis 36
2. Related Research **37**
2.1. Artificial Intelligence 38
2.1.1. Knowledge Representation 38
2.1.2. Planning 38
2.1.3. Meta-Level Reasoning 39
2.1.4. Expert Systems 41
2.1.5. Expert Systems for Design 41
2.2. Design Process Modeling 46
2.3. Large Design Automation Systems 47
2.3.1. Silicon Compilers 48

2.3.2. Estimation 49
2.4. Data Modeling, Data Bases, and Unified DA Systems 49
2.4.1. Hardware Description Languages 49
2.4.2. Data Modeling 50
2.4.3. ADAM 51
2.5. Formal Models of Designs and Design Processes 52
3. The Design Data Structure **55**
3.1. Uses for the Representation 57
3.2. Motivation 58
3.3. The DDS: Overview 61
3.4. The Component 62
3.4.1. Models of a Component 63
3.5. Details of the Four Subspaces 65
3.5.1. The Dataflow Subspace 65
3.5.1.1. A Small Example 71
3.5.2. The Timing Subspace 71
3.5.2.1. Conditional Execution 74
3.5.2.2. Looping 74
3.5.2.3. Example Timing Graph 75
3.5.3. The Logical Structure Subspace 77
3.5.4. The Physical Structure Subspace 78
3.6. Bindings 81
3.6.1. Operation Bindings 81
3.6.2. Implementation Bindings 84
3.6.3. Binding Hierarchical Objects 84
3.6.4. Bindings and Syntactic Checks of Correctness 85
3.7. Looping and Subscripts 86
3.7.1. Unfixed loops and their effect on subscripting 86
3.7.2. Binding Predicates 89
3.8. Timing Predicates 89
3.8.1. Example: Traffic Light Controller 93
3.9. Hierarchies 96
3.9.1. Net Lists 96
3.10. Discussion 100
3.10.1. Factors Influencing the Choice of Subspaces 100
3.10.2. Relations Between Specification, Target and Library 101
3.11. Capabilities of the DDS 102
4. Planning: Overview **107**
4.1. Knowledge Representation Overview 107
4.2. The Design Planning Engine 110
4.3. Definition of Planning 110
4.4. Planner Operation 112

4.4.1. Global Strategy 113
4.4.2. Inputs to the Planner 114
4.4.3. Exploration 114
4.4.4. Planner Output 116
4.5. Operation: What DPE Does 118

5. Modeling the Design Process 125

5.1. Notation 126
5.1.1. Preconditions and Posconditions 129
5.1.2. Operators 130
5.2. Monotonicity 132
5.2.1. Monotonic Operator Estimator Sets 135
5.2.1.1. Boundary Finding Algorithm 137
5.2.1.2. The Blind Path Algorithm 140
5.2.2. Monotonic State Estimators: Discussion 145

6. DPE Implementation 147

6.1. Overview 147
6.1.1. Layers of Interpretation in DPE 150
6.2. Flavors 151
6.3. Rule System 151
6.3.1. The Inference Engine 152
6.3.1.1. Basic Cycle 153
6.3.1.2. Variables and Unification 154
6.3.1.3. Escape Clauses 156
6.3.2. Uses of Rules 156
6.4. Frames 157
6.4.1. Hardware Frames 157
6.4.2. Task Frames 158
6.4.3. Rule-Based Domain Knowledge 160
6.4.3.1. Preconditions 160
6.4.3.2. Postconditions 160
6.4.3.3. Estimators 162
6.4.3.4. Application Rules 164
6.5. Actors 165
6.5.1. Major Actors 165
6.5.1.1. The Planner 167
6.5.1.2. The Frame Expert 170
6.5.1.3. The Manager 172
6.5.1.4. Miscellaneous Actors 172
6.5.2. Operations at the Actor Level 174
6.6. Discussion 176
6.6.1. Layered Interpretation 176
6.6.2. Nonprocedural Estimation 178

6.6.3. Choice of Programming Language 180
6.6.4. Completeness and Correctness 182

7. Test Cases **183**

7.1. Overview 183
7.2. General Capability 184
7.2.1. Search for the Fastest Design 187
7.2.2. Interpolation 188
7.2.3. MAHA and Sehwa 191
7.3. Interpolation and Convergence 193
7.3.1. Effect of Varying Constraints 194
7.4. Unusable Operators 197
7.5. Choice of Chainable Operators 199
7.6. Design States 205
7.6.1. Terminal State Model 207
7.7. Modeling Operators with Frames 212
7.7.1. Task Frame Analyze-DF 212
7.7.2. Maha Task Frame 217

8. Conclusions **223**

8.1. The UDAS Problem: Summary 224
8.1.1. The Fractured Design Space 224
8.1.2. Tool Grain Size 224
8.1.3. Tool Ordering 225
8.1.4. Interaction 225
8.1.5. Verification 225
8.2. The Model 226
8.2.1. Syntactic and Semantic Compatibility 226
8.3. DPE: an Updatable Program 228
8.3.1. Layered Interpreters 228
8.3.2. Rule-based Objects 229
8.3.3. Rule-based Estimators 229
8.3.4. Negative Feedback 229
8.3.5. Frame Representations 229
8.4. Simulated Design and Level of Abstraction 230
8.4.1. Estimation, Hypothesization, and Search 231
8.4.2. Summary 232

References **236**

Appendix A. DDS Details **250**

A.1. Notes on the Hierarchies 250

Appendix B. DPE's Frame Set **259**

List of Figures

Figure 3-1: Example showing four models of a component 64
Figure 3-2: Dataflow graph of the function $y=mx+b$ 67
Figure 3-3: Dataflow graph of the function $f(x)$. 67
Figure 3-4: Dataflow node definition and references. 69
Figure 3-5: Timing and control graph for an instruction cycle. 76
Figure 3-6: Logical structure of the simple computer. 79
Figure 3-7: The simple computer: behavior, structure, and bindings 82
Figure 3-8: Subscripting: Effect of Unfixed Loops 88
Figure 3-9: State diagram of a traffic light controller. 93
Figure 3-10: DDS description of a traffic light controller. 95
Figure 3-11: Example of a nontrivial structured connection. 98
Figure 3-12: Nontrivial structured connection: internal view. 99
Figure 4-1: Architecture of the DPE program. 111
Figure 4-2: Illustration of planning. 113
Figure 4-3: Interpolation technique used by DPE. 121
Figure 5-1: Monotonic and nonmonotonic estimators. 133
Figure 5-2: Illustration of the time complexity argument. 136
Figure 6-1: Architecture of DPE. 148
Figure 6-2: Major actors of DPE. 166
Figure 7-1: Interpolation. 189
Figure A-1: Model Hierarchy 254
Figure A-2: Model Reference Hierarchy 255
Figure A-3: Link Hierarchy 256
Figure A-4: Link Reference Hierarchy 257
Figure A-5: Netlist Types 258

List of Tables

Table 5-1: Notation for states and operators. 127
Table 5-2: Notation for Predicate Calculus. 130

Abstract

This thesis presents a novel approach to the construction of integrated design automation systems, and to modeling the digital design process. This approach allows the design process to proceed automatically from specification to final implementation. The system described here forms part of the ADAM Advanced Design AutoMation system being constructed at the University of Southern California.

A representation of design information is presented. This representation is uniform for target designs, specifications and library components, and contains behavioral, structural and physical design information. The representation is in use by a number of ADAM design automation procedures.

Knowledge about design tasks and about hardware are represented in frames. The design tasks are represented in terms of preconditions and postconditions.

The design process is under the control of the Design Planning Engine (DPE) which is an expert system. It builds a design plan from a set of possible analysis and synthesis tasks, including clocking scheme synthesis, component selection, critical path location, and hardware allocation.

The DPE rules have access to frames describing design tasks which may be performed. Design tasks are first selected, then chained on the basis of their pre- and post-conditions, and on the basis of a heuristic estimate of the advisability of the tasks.

Once a plan is constructed, the DPE iterates using an interpolation procedure until it converges on a plan that it estimates will meet design constraints.

The planner currently has knowledge about register-transfer data path design, and can build plans using such knowledge. Planning knowledge is contained in a collection of rule sets. The separation of planning and design knowledge into rule sets and frames increases the general applicability of the planner and aids in the creation of meta-knowledge to guide the design process.

An experimental prototype of the DPE has been run to produce a number of plans. A formal model of the design process implemented by DPE is described in the thesis.

Chapter 1
Introduction

1.1. Problems in Digital System Design

As digital systems get larger and more complex, the cost of engineering them increases rapidly. The cost of engineering is already a major factor in the area of custom and low-volume products. This trend will presumably continue in the foreseeable future. As product lifetimes and manufacturing costs decrease due to innovations in product definitions and shrinking feature sizes, the relative cost of design as a fraction of total product life cycle costs will increase even in the high-volume markets. Fortunately, however, as computational resources get cheaper, the amount of computer time that can be devoted to design activities will increase.

In order to motivate the ideas and techniques described in this thesis, this chapter first describes some of the factors that contribute to the engineering costs of bringing a new product to the market.

1.1.1. Cost of Design

The design activity is itself expensive. It requires a good deal of highly trained manpower, which can be underutilized when there are gaps in the automation of the design process. Even manual translation of designs from one format to another is time-consuming. Design also requires considerable machine time. As computers become more powerful in proportion to their cost, and human engineering talent remains scarce, computers will increasingly be used to supplement human decision-making.

Design Changes

Design changes are an important cause of high overall design cost. In such cases a complete design might be modified. This can happen for a number of reasons. The design may not meet its performance goals, or it may be too expensive and more powerful than its market will support. The design may have been adequate in its day, but product generations are short and a compatible architecture with increased performance may be required. Some aspect of the machine's design may make it difficult to test, service, or manufacture. Parts that were thought to be available at the time they were selected may not be available at manufacturing time. There may be reliability problems, or unexpected performance penalties. A customer may want optimized performance or increased mechanical ruggedness. The design will almost certainly contain errors that only become apparent during floor testing of prototypes, or even after product delivery begins. All of these occurrences are occasion to redesign the system to some extent, from a global rethinking in the case of unmet performance requirements, to a minor microprogramming change that can be accomplished by field service personnel. Alternatively, sometimes the high cost of redesign leads to situations in which bugs and limitations are tolerated or worked around, with a concomitant reduction in design quality.

Time to Design

There is almost always a marketing window for a new product: if the product appears too soon, money will have been spent too early in the product's life cycle, and the product will be more expensive than it should be. If it appears too late, it may not sell at all, because the competition will be selling something better for less. There are two considerations here: the absolute time a product spends in the design phase of its life cycle, and the predictability of that time. Both of these affect the relationship of a product to its marketing window. As the overall design time is decreased, the effects

of overhead costs and uncertainties in market predictions are reduced. As the design time becomes more predictable, management is able to allocate resources to projects in a properly organized way, with fewer rushes and missed deadlines.

1.1.2. Goals and Constraints Vary Widely

Goals and constraints vary widely from product to product. A product may be part of a military system; such systems tend to have low product volumes, stringent environmental and mechanical specifications, special security precautions during design, manufacturing, and use phases of the life cycle, and an overall goal of high performance. Alternatively, a product may be a "jelly bean" microprocessor. Such a part may have a market volume in the hundreds of thousands of units, moderate performance requirements, and an overriding emphasis on unit cost and adaptability. Other goals and constraints include

- price and performance,
- power consumption,
- fault tolerance and reliability, and
- compatibility with existing software, hardware, and manufacturing facilities.

These categories can be combined and traded off in many different ways, resulting in goal/constraint structures that vary widely. Over the course of time, many different combinations will be encountered by a general-purpose system.

Furthermore, the space of digital designs, even for a single specification, is large and discrete. That is, for a given specification there are usually many possible implementations, differing in discrete ways. For example, one design

might be pipelined, while another achieves the same throughput by using a faster technology.

Other problems that tend to raise the overall cost of design are

- it is difficult to evaluate a partially complete design, thus necessitating expensive lookahead where it is possible, and backtracking where it is not,
- subproblems tend to interact in complex and data-dependent ways, making lookahead difficult, and
- a fixed sequence of operations is weak because it does not take advantage of opportunities presented by the specific problem.

1.2. Difficulties of Current CAD Systems

This thesis focuses on solutions to the technical problem of high design cost and design time through the consideration of data and control issues. In the area of data, a major technical problem is the use of multiple representations, which lead to interfacing, consistency, and correctness problems; in the area of control, major problems are those of making choices between large numbers of possible design decisions, selecting appropriate tools, and invoking the tools with proper input data.

1.2.1. Multiple Design Representations

Multiple unintegrated representations for design information are a major cause of design errors. Each such representation does not stand alone because it does not cover all of the design data, but must be used in conjunction with other representations. Hence there are many potential compatibility and correctness problems.

Consistency Problems

First and foremost, there are problems of consistency and correctness. Consistency can be regarded in two senses. In the first sense, a single representation can have internal inconsistencies, as in a logic diagram in which a gate's type is referenced as a NAND gate in one record, and a NOR in another. In the second sense, two representations of the same design can be inconsistent with respect to one another, as in the case where a logic diagram indicates the function NAND but the corresponding layout is that of a NOR. Where multiple representations are used, consistency problems of the second kind can be difficult to detect. For example, the signals controlling an ALU may cause subtraction, when addition is the desired behavior. In that case the behavior implemented in the ALU is inconsistent with the algorithm it is intended to support.

Tool Interfacing Problems

There are many CAD tools available; however, they do not all use common formats. The result is that there are suites of tools that can communicate with one another, but not with members of other suites. Some tool sets enforce a *de facto* ordering of design activities because the intermediate data formats and representations are unique to tool-tool pairs. This reduces the flexibility of the tool set, in particular making it difficult to do incremental design, to allow the user to make isolated design decisions, and to backtrack.

Errors in Translation

Data translation programs (filters) are often nontrivial programming projects; they can themselves be sources of errors. When translations are done by hand, the probability of errors is even greater, and the process is tedious and costly.

1.2.2. Tool Selection and Invocation

An inferior or nonexistent capability for selection of design strategies results in high design cost because human intervention is required whenever tool selection and invocation decisions arise. This human intervention is expensive and is responsible for introduction and propagation of errors. In one such case, an engineer working for a major company failed to use a timing analysis program on his chip design, even though timing analysis was part of company policy. The chip was fabricated, but did not work. In an effort to find out what went wrong, the timing analysis program was applied to the original design, and it successfully "predicted" the problem. This would not have happened if the use of timing analysis had been automatic, or if the system had been able to remind the user of the timing analysis requirement.

The following factors are important in considering the overall problem of automatic or semi-automatic tool selection: any solution must take them into account.

Possible Design Strategies

Selection of a tool is in many cases synonymous with selecting a design strategy, especially when the tool is a synthesis tool. Hence the selection of an appropriate tool depends on design constraints, objectives, previous tool selections, and implementation decisions. For example, the selection of a tool that implements pipeline clocking schemes is tantamount to selecting a pipelined design style. It is not obvious which tool to use in the general case, which includes constraints and objectives derived from previous design decisions as well as from input specifications. The effects of tool selection on design strategy, and the effects of proposed design strategies on tool selection, must both be modeled.

Number of Tools

There are many CAD and CAM tools. They cover a continuum from heat-flow analysis to queueing analysis, and from specification through schematic to layout capture. An automatic selection system should be able to handle a large and diverse tool set. If the tool set is fine-grained, i.e. it is composed of many comparatively simple tools, this factor becomes proportionately more important.

Extension of the Tool Set

Not only is the tool set potentially large, but it is time-varying as well. An automatic tool selection system must be flexible enough to incorporate new tools as they become available. The complexity of adding new tools should be as close to linear (in the number of tools) as possible.

Tool Size and Cost

CAD tools are expensive both in terms of software procurement and in maintenance and upgrading. Hence their flexibility and generality should be maximized. Competing with this goal is that of the quality of results; general-purpose tools tend not to take advantage of special cases. One way to address this tradeoff is by the use of many small tools, as opposed to a few big ones. This enhances flexibility and reuseability, and small tool size can reduce the unit cost of the tools. However, it returns us to the problem of underutilization, because the number of tool selections and invocations grows, increasing the burden on human designers.

Tool Operation is Complex and Detailed

One of the reasons it is difficult to automatically select and invoke tools is that tools are complex and their coverage is incomplete. Some tools are intended to carry out specific design strategies, or can only be used in the context of specific strategies. In such cases, the tools are not appropriate in all cases. Each tool has its own peculiarities, not only of the data it expects,

but of the semantics of the data. For example, a register-transfer allocator may have been constructed under the assumption that MOS dynamic logic would be available for temporary storage; such an allocator could be extremely wasteful if it was invoked in the context of an ECL design. An analog simulator might not operate correctly if substrate potential is not explicitly set, or it might fail to converge if parasitics are not stated to be present. One routing program will do well on a sparse layout and another on a dense one. Some tools are guaranteed to succeed in every case, while others are guaranteed only to terminate with some result, and still others are not guaranteed to terminate at all. Some tools can be guaranteed only under special circumstances, but will function acceptably most of the time. The expense of applying a tool varies widely, from exponential complexity down to constant time. Most tools have flags and other auxiliary parameters that express differences in the underlying algorithm.

All of these factors should be taken into account when the tools are being considered for application; yet the volume of such information for a set of heterogeneous tools may be too large for a designer to make correct and efficient use of the set as a whole.

1.3. Unified Design Automation Systems

The previous sections describe a few of the problems confronting organizations that produce designs for digital hardware. Clearly there is an economic incentive for building design automation systems in which some of the routine supervisorial and interfacing work at the lower levels is taken care of by the system. There is also an incentive to automate some of the design strategy, operational, and supervisorial expertise at the higher levels.

We will call a system which is capable of automating the entire design

process from a rigorous high-level specification to a final layout a *Unified Design Automation System* or *UDAS*. This definition will be construed to permit, but not force, human participation in any amount from no human participation at all to entirely human-made design decisions. The long-term goal of this research is the construction of a general-purpose UDAS, i.e. one that can be used for many product types.

The requirements for a general-purpose UDAS are different than those for specialized systems, because specialized systems can be used only where there is a comparatively large market for a class of designs which is relatively homogeneous. A general-purpose system must be able to implement a wide variety of design strategies to fit a wide variety of applications and constraints.

It is desirable for a for a general-purpose UDAS to have a number of specific properties. These are

- a varying degree of user interaction,
- an ability to reason about design tasks, which will in turn allow
- automatic or semi-automatic management of tools,
- the capability of exploring the design space,
- feedback from design results,
- a flexible implementation, and
- robustness.

These properties will now be discussed in detail.

1.3.1. Varying Degree of Interaction

An ideal general-purpose UDAS would allow a varying degree of user intervention. There are four main reasons for this. First, for important decisions the judgement of a human designer is expected to be superior to that of the system. For routine, trivial, or unimportant decisions human judgement is expected to be less critical.

Second, forcing a human designer to make trivial or routine decisions contributes to underutilization of the designer's time, and may lead to the introduction or non-detection of errors.

Third, it is not clear that any one decision or class of decisions can be declared to be either "important" or "trivial" *a priori*, i.e. at UDAS programming time. Such assignments of importance to decisions should be made in context (i.e. when a specific design, with its potentially unique goal/constraint/functionality combinations, is encountered).

Fourth, it is desirable to use a "fine-grained" tool set. Such a tool set can enhance flexibility and reusability, and cut programming costs, as discussed above. However, such a strategy tends to increase the number of tool selection and invocation decisions that must be made. This increase can be offset by automatic invocation where the decisions do not require judgement. For example, commercial silicon compilers are composed of a number of tools for data capture, module generation, interactive placement, routing, area estimation, and so on. The designer carries the design forward incrementally task by task. Interactive tools carry this one step further, by putting a human in the decision-making loop within a program.

An important parameter is the grain size at which human decision-making can intervene. At one extreme of this parameter, tools are large, i.e.

substantial amounts of computation follow each interaction. Hence the level at which a designer can intervene with large tools is very high. This is advantageous in that he can make a few interventions to control a complex process, but at the same time it means that he does not have detailed control. At the other extreme the tools are small, and the human can intervene at almost any stage of the process. An optional fine level of intervention should, in the ideal case, be combined with a required coarse level of intervention. Hence an ideal UDAS should allow a varying degree of human intervention. At one extreme, the system makes all design decisions from specification to final design output; at the other, a human or humans make all decisions.

Suppose, for example, that a designer has been given a behavioral specification to implement. He decides to synthesize the critical parts of the data path using the Sehwa [Park 85a, Park 85c] pipeline synthesis tool. In order to do this, he must first select a module library. Sehwa finds, for that module library and behavioral specification, the fastest and cheapest designs. The designer then inputs alternating speed and cost constraints to converge on the overall "best" design for the situation. He may repeat this process with a different module library; he might also decide to use the computationally expensive exhaustive search mechanism Sehwa provides as an option.

In the meantime, the constraints he is trying to meet are also changing, due to the activities of other designers, or because other parts of his design (e.g. the controller) do not meet their budgets for area, speed, and so on. He might even decide to transform the input dataflow after gaining experience with it, in order to optimize it for the constraints he is trying to meet. Each time this happens, he must go through the whole process of library selection and running Sehwa again.

And datapath allocation is only one of many steps in the design process. If each is performed in a similar fashion, the designer is going to alternate between confusion about what he has tried before, and boredom as he repeats the same tasks over and over again with only minor variations.

Furthermore, task invocation is not completely routine. At one point he may decide to try to partition the design, to see if it could be put on two chips instead of one. He might run an area estimator to see if a particularly promising design would meet its constraints. The program Sehwa might also fail, because in his boredom he forgot to check that the module library had the right kinds of functions available. Hence while the task selection and invocation process is tedious and detailed, it is not completely routine, and requires knowledge of the design process.

The presence of a dynamic tradeoff between human and automatic decision making can potentially confer the following benefits:

- automatically made decisions can be overridden,
- a structured framework can be imposed on human intervention, resulting in enhanced correctness and consistency,
- completeness does not depend solely on human vigilance and persistence,
- production cost can be traded off against design cost without sacrificing correctness; e.g. high-volume products can be human-tuned in order to cut end-product unit cost, whereas low-volume products can be entirely machine-designed,
- "rough-cut" designs to explore design alternatives and to provide starting points for human optimization can be constructed easily,
- human effort can be concentrated on critical parts of the design,
- a fine granularity of tool size can be achieved without forcing a

human user to manually invoke a large number of small tools one at a time, and

- a large number of tools can be used without a detailed understanding of each tool.

Note that the system is not required to be able to make decisions "as good as" those of a human, although some tools currently match or exceed human design capabilities. The decision-making capability of the system is ultimately intended to be used only where a human has decided not to intervene.

1.3.2. Automatic Management of Tools

In order to implement a capability whereby completely noninteractive use is possible, the system should be able to manage tools. That is, given a set of specific attributes that describe a design at some stage of its development, and a collection of tools, it should be able to choose and invoke the tools in a reasonable way.[1] This is a composite of several capabilities.

First, it should be able to collect a set of applicable tools, that is, tools whose preconditions are met in the current situation. This guarantees that only appropriate tools are used.

Second, it should be able to rank the tools in terms of their advisability in the current situation, i.e. the net long-term gain to be derived from their application. It should be able to do this even in the presence of missing information; if not much is known about the design, the system should still function as long as preconditions are met.

---

[1]In fact, for an interactive system to control proper invocation of tools and to maintain a history of the design process, the system must have essentially the same capabilities and knowledge.

Third, it should be able to invoke tools without human intervention, i.e. through calling them as subroutines, or as coroutines, or as completely separate jobs under the operating system. This implies that tools can be written in different languages, and even run on other machines.

Fourth, it should be able to recombine the results of tool operation with the current design status. That is, the system should be capable of updating the design representation to reflect the application of tools.

Fifth, it should be able to interrupt tools if they fail to terminate, if run-time limits are exceeded, or if the results are clearly useless.

Note that all of these capabilities must take into account the current status of the design process as a basis for the decision-making power of the system.

An automatic tool selection and invocation capability can also help in other ways. One way is with the automatic invocation of estimation and checking tools. Suppose, for example, a designer is working on the critical path of a target design, making parallel-serial tradeoffs. Automatic area (and speed) estimation would be useful because the designer would be warned immediately if a change would cause a violation of constraints.

Note also that an automatic invocation capability is a superset of an intelligent prompting capability. If invocation as such is disabled, the code that would otherwise select an operator could still be used to issue messages of the form "do you want to run the area estimator now?" Invocation could then be made conditional on the user's response, or could be left completely manual.

Another possible use for intelligent prompting would be in the area of ensuring proper tool invocation. A designer, for example, might want to use an unfamiliar tool. If an automatic selection capability was present, it would be straightforward to answer such questions as:

- what tools are applicable?
- is tool T applicable?
- what preconditions must be true of the data, for tool T to be applicable?

In addition, a powerful operator model of the kind discussed in Chapters 4-7 could provide answers to even more questions:

- what tool would make the preconditions of T true?
- what can T be expected to do in the current context?
- what would the estimated run time of T be?
- what is the relative advisability of running T instead of some other applicable tool?
- what effect would using T have on the ultimate design?

It is argued in the next section that an operator model capable of answering such questions is needed to provide an automatic tool selection and invocation capability. Even in the context of a semi-automatic or intelligent prompting system, and even if the answers were not completely reliable, the answers to such questions might be of substantial use to a designer.

Another capability that becomes possible in the context of automatic selection and invocation is the maintenance of a machine-readable design history. Such a history would be composed of tool invocation records, annotated with invocation results. For example, suppose that a register-transfer allocator was called, with a specific set of library modules as one

argument. The name of the module set would be saved as part of the invocation record. If flags, constraints, etc. were passed to the allocator, that information would also be saved in the invocation record. A result record would also be kept, containing high-level information about the run. Such information might include a statement that the program terminated normally, the value of the delay through the critical path, the name of the file containing the allocated structure, and the estimated cost of the design. Such a recording capability could be used to prevent, for example, the use of an obsolete data file as input to the next stage of the design process. Combined with an intelligent prompting facility, it could be used to generate warnings of the form "the file you have specified is not current."

Another use for a historical record of this form would be automatic re-invocation when the design process became iterative. In many situations, nearly identical sequences of activities must be carried out. Starting with a sequence of invocation records, one could edit the sequence, for example to change the module library in the allocation example above, and then instruct the system to re-run the sequence with only that change made. Furthermore, a historical trace of this kind would be useful if it became necessary to backtrack to a specific point. This could be accomplished by using the trace to find the latest acceptable checkpoint data, and then to duplicate the sequence of operations from the checkpoint to the backtrack point.

1.3.3. Exploration of the Design Space

For reasons of efficiency, the system should be able to explore the design space on its own. For this it must be able to construct a series of tentative designs rather than a single design. This is in effect a generalization of the decision-making capability to include the construction of "straw-man" designs, which are often a normal part of the traditional practice of design.

More precisely, given a set of behavioral specifications, a UDAS should be able to construct various tradeoff curves, either by estimation or by designing and evaluating trial implementations, or both. The series of "points" need not be composed solely of actual designs, which are detailed and expensive to construct. For example it may be sufficient to construct only an RTL diagram of a system to form an adequate estimate of its performance and cost characteristics. The series of trial designs may also have members that are highly abstract models, for example behavioral models, which do not have a direct mapping into the proposed system structure.

The capability of producing exploratory designs automatically can be valuable in the following ways:

- as an aid in the construction of specifications and the evaluation of tradeoffs between specification choices,
- as a rough predictor of the cost and performance of components yet to be designed, and
- as an aid in evaluating and comparing the effects of implementation choices.

1.3.4. Feedback from Design Results

The system should be capable of using feedback from the results of the design space exploration process to converge on designs that meet constraints and optimize objectives. That is, the system should be able to use the results of its explorations.

It should also be possible for humans to intervene during the exploratory process in order to add new constraints, relax old ones, make suggestions, and so on. This is a direct consequence of the desire for a variable level of interaction. It also reflects a desire to treat the exploration

phase in the same way as the final design process, and permits a process of "negotiation" between what the system can implement and what the designer asks for.

1.3.5. Flexible System

The system should be flexible. That is, it should allow the use of a wide variety of styles and methods, and it should be easy to change. It should support the addition of new tools and the addition of new ways to use old tools. It should also support some kind of interactive tuning whereby the tools and their management methodology can be tuned during the construction of actual designs.

The chief reason to provide flexibility is economic: a flexible and updatable system will not become obsolete very quickly, thus enhancing the recovery of the initial software investment.

Addition of Tools

A UDAS should allow the incorporation of diverse design styles, tools, and procedures. This is needed because of the large and discrete design space; no one style, tool, or procedure is expected to work well in every case. As the state of the art changes and new tools and styles become popular, the system will have to be updated in order to stay current.

Addition of New Techniques of Tool Use

Not only is ease of changing the tool set desirable, but the "judgement" of the system should be easily updatable. This might be done for purposes of fine tuning, but it might also become necessary when new tools are added that change the tradeoffs between older tools. For example, suppose that a horizontal-microprogram synthesizer were replaced by one with better optimization capabilities. The tradeoffs for highly parallel designs would then

change, resulting in differences in the way clocking and allocation choices were made. The breakover point at which the system would decide to use a parallel scheme in preference to a serial one would also shift.

Hence the system should allow tool usage techniques to be updated and modified. This includes all aspects of tool management: the preconditions under which a tool can be used, the ranking functions by which the most advisable tools are chosen, the ways in which tools can be invoked, and the ways in which tool results are reintegrated into the design. Hence not only the tools themselves must be flexible, but the models that describe tools and tool uses must be flexible as well.

1.3.6. Robust System

Programming errors and missing information in an integrated system must not make it useless. That is, the system should tolerate the presence of at least some bugs. This tolerance could be supplied by human intervention where the system's own judgement is faulty or inadequate, or it could be inherent in the system.

1.4. Scenario

In order to make the discussion more concrete, a brief scenario describing how a UDAS might be used will now be adduced.

The scenario begins when a human designer is given a large design task. He decides to pass part of the problem to the system, with instructions to produce a preliminary "rough cut" design. He puts the specification into the form of an HDL and passes it to the design system, which he puts in full-automatic mode so he can think about other aspects of the problem.

The system begins designing by examining the input. It is in the form of an HDL with attached constraints, but the system has no way to operate on HDL descriptions directly, so it invokes a compiler that extracts a dataflow graph and a minimal timing and sequencing graph. The system also sets up criteria for ending the design process, i.e. criteria by which it will judge the completeness of the task it has been assigned. Because the user asked for a preliminary design, it is able to set up completeness criteria that fall short of a fully routed layout.

Now the system has several options. It can

1. analyze the dataflow, e.g. to find out if it contains nested loops, exotic operations like square root, etc.; and to extract a tentative critical path,
2. optimize the dataflow by successive transformation techniques,
3. optimize both dataflow and timing/sequencing by making hardware/firmware tradeoffs,
4. partition the dataflow,
5. implement the dataflow using combinational logic,
6. implement the dataflow using a simple FSM,
7. select a more complex sequential style, e.g. a controller/datapath architecture,
8. choose a technology, or
9. choose a particular module library within a technology.

All of the above operations could be done, given the data. However, they are not equally advisable in the current situation. For example, the idea of direct implementation in combinational logic is probably infeasible, because the compiler reported that the dataflow graph had hundreds of nodes. In fact,

the most advisable thing at this stage is to find out more about the dataflow graph, so the system decides to use the dataflow analysis tool. This tool is executed immediately instead of simply being scheduled, because the system's goal is to gain information.

Note that the list of applicable operators still includes the entire list of tasks listed above; they can still be executed after dataflow analysis. The system then decides, on the basis of node type and graph complexity information, as well as the presence of loops, that a controller/datapath architecture is highly advisable. The sole effect of this decision is an assertion that the design will have such an architecture.

Once the controller/datapath decision has been made, the module library choice becomes the most advisable. Because the system has a wide range of library elements, it simulates the selection by assigning plausible average propagation delays and hardware costs. Because the system does not yet have a clear idea of whether speed or cost will be more of a problem, middle-of-the-road values are chosen.

Now that propagation delays and operator costs are available, it is possible to perform a cost/speed driven allocation of operators to operations. However, because the critical path information currently available was generated before propagation delays were known, another pass of dataflow analysis is preferable Again, the analysis task is actually performed, using the average propagation delays.

Among the results of the critical path analysis is an assertion that the delay through the path is at least twice the allowed constraint, and an assertion that the cost of the path is very low. Because of the mismatch allocation is not particularly advisable, but module library selection again gets

a high advisability score. This leads the system to reschedule the module library choice. Because of a heightened importance attached to speed and a lowered importance on cost, the result of the (simulated) selection has much higher average speeds and costs for modules.

The critical path analysis is scheduled again, because the old critical path may have been invalidated by the new propagation delays. It is not executed this time, however, because only average speeds and costs were changed. Instead, the old results are simply scaled. The simulated analysis returns assertions to the effect that the critical path delay is now close to the requirement, and that the critical path cost is considerably higher, although it is not above the overall cost constraint.

The partitioning tool is now the most advisable, because this is still a very preliminary design, the major problems seem to be with performance, and the system avoids computationally expensive activities when it can, especially when it is constructing preliminary designs. The partitioner is simulated, returning the set of critical path nodes plus a random subset of the rest of the dataflow graph.

Datapath allocation for the partition built around the critical path now becomes the highest priority. The allocation process is simulated, returning assertions stating that the allocation has been performed, and that all of the nodes in the critical path partition have had individual modules allocated to them.

A logical structure of the kind asserted by the datapath allocation is a prerequisite for area estimation, which therefore becomes applicable. The system decides that area estimation is most advisable activity it can carry out, and schedules it accordingly. Notice that it could also have allocated the

remainder of the dataflow graph, or it could have attempted to estimate the speed of the critical path based on (estimated) wire lengths and connectivities, or it could have scheduled actual layout of the critical path, all of which are now possible. The area estimation is simulated, and it returns an answer close to the constraint, but with low reliability. The system then decides that the area estimation must actually be executed. In order to execute the area estimator with sufficient accuracy, however, the preceding design activities must also be carried out. Accordingly, the system backs up to the module library decision and starts executing the plan it has constructed, step by step. When the area estimation is reached in this execution phase, it returns a lower estimated area than it had before.

The system now decides that the most advisable task is that of allocating the non-critical parts of the datapath in the most area-efficient manner possible. It simulates allocation, because the noncritical partition represents a comparatively small fraction of the overall design.

The system can now lay out the datapath, or make speed estimates, or start work on the controller for the datapath it has partially constructed. The highest advisability score is that of controller style selection; a horizontal microprogram is selected.

Because the system's original goal was to come up with a preliminary design, and not with a finished product, it does not construct a microprogram. Instead, it schedules an estimator, which bases estimates of controller size and speed on statistical properties of microcontrollers, the assertion that the controller is horizontally microprogrammed, the number of control ports in the datapath, and the number of states in the timing and sequencing graph.

Because the estimate of controller cost is only a fraction of the overall cost constraint, and because the controller's estimated speed requirement leaves a margin for error, further work on the controller has low advisability.

Because the system now has a fairly reliable estimate of the target's cost under the speed constraints given, and because the user originally asked for a preliminary design, the system now queries the user, informing him that his specifications can probably be met using a nonpipelined, horizontally microprogrammed, standard-cell approach. The human designer, in the meantime, has made implementation decisions about the rest of the target design which make the behavioral specifications used by the system obsolete. He modifies the HDL specification and passes it back to the system.

1.5. Thesis Statement

There are certain attributes a UDAS should have in order to meet even in part the requirements listed above within the constraints of a reasonably economic implementation. Briefly, the essential attributes are

1. the system should be constructed around a single underlying representation for design information,

2. the system should use many small tools (a *tool kit*) rather than a few larger tools, in order to control the decision-making process at a fine grain size,

3. the preconditions and effects of synthesis and analysis tools (*operators*) must be explicitly represented,

4. the system must be capable of constructing estimates of design quantities and attributes in arbitrary situations, and

5. the system should take an approach whereby future states of the design are modeled before they are created.

The justification for these requirements is given below.

It is also highly recommended that an object-oriented, message-passing model of computation and data representation be used in the construction of a UDAS. While not an essential requirement, It is thought to be a great saver of programming effort, having a substantial effect both on the ease with which the application can be programmed, integrated, and debugged, and on lowering the probability of serious bugs remaining in the system for extended periods of time.

1.5.1. Unified Data Structure

The use of a single underlying representation for design information will greatly aid the construction of any system that has a heterogeneous collection of tools and no *a priori* ordering of design activities. Such a representation does not necessarily force all design data into a single abstraction. Instead, where multiple abstractions of entities are constructed and used, it maintains a linkage between the various models. For example, there may be several models of a NAND gate representing different aspects of the gate: a truth table, a schematic symbol, a schematic (transistor level) diagram, a set of labelled pins on a SSI package, a set of rectangles on a chip layout, a timing diagram, or a heat source model. Each of these models has its own uses and purposes, and should be present in the representation of designs; the correspondence between the various models should be maintained as well.

The design representation formalism should also capture the differences and correspondences between the archetypal components to be found in a design library, (e.g. a TTL data book), and particular instances of the components where they are used in a design. For example, the 7400 part discussed in the TTL book is an archetype, whereas the part soldered into a board is an instance of the type 7400.

1.5.2. Tool Kit Approach

The system should regard the collection of tools as a kit, that is as a randomly accessible set of tools, rather than as a sequence that can only be applied in certain orderings. There are three main reasons to do this.

First, design strategy is a function of the specification, the design constraints and objectives, and the available tools. Hence it varies enormously from design to design. Each strategy can involve the use of different tools and tool use orderings. The mix of analysis and synthesis tasks varies from design to design. Many tasks interact, such as those of placement and routing. Hence a tool-kit approach is necessary in order to achieve true generality, and hence to allow the system to produce good working designs in a wide range of cases.

A tool kit approach also allows us to add new tools to the system without rebuilding a fixed set of task orderings.

Finally, the use of a tool kit makes the re-use of tools simpler. For this reason, the tool kit should initially concentrate on simple, general-purpose tools rather than on complex specialized tools. Where a specialized operation can be performed by composing a number of simple tools, this is to be preferred, because the simple tools can potentially increase overall system generality, because it provides more openings for orderly user and system intervention, and because re-use of tools can offer program development cost savings. For example, tasks like partitioning may occur in many places. There is a benefit to be gained from re-using one partitioning program many times, rather than building special partitioners for all of the different situations in which partitioning is called for. As coverage by general-purpose tools becomes more and more complete, and the cost and quality bottlenecks

in the design process become apparent, special-purpose tools can be added one by one to a running system.

The use of a tool kit reinforces the requirement on unified representation of design data, which is a prerequisite for such a tool-kit approach.

1.5.3. Representing Tool Semantics Explicitly

The semantics of tool application should be explicitly represented. This means that in addition to the executable (or interpretable) code that constitutes a tool, information about the classes of situations in which the tool can be applied, the run-time complexity, and the possible effects of running the tool should be represented. These representational requirements correspond to the translation of tool documentation into machine-understandable form, so that the system can reason about the applicabilities and effects of tools. This applies to both synthesis and analysis tools.

1.5.4. Estimation on Incomplete Information

It is a further requirement that a UDAS should be capable of generating estimates on the basis of incomplete information. The ability to make estimates is required because complete information about design properties is often expensive to obtain; complete information about the results of a particular design decision may not be available until the end of the entire design process (or until manufacturing).

It is also required that such estimates be relatively cheap to generate, because if estimates were not cheap, detailed solutions could be generated and compared just as easily.

We distinguish between two kinds of estimation, both of which must be supported. An *operator* estimator is used for operator[2] ranking. That is, when more than one operator is applicable, some ranking function must be used to determine which of the operators should be applied. Operator estimates are only used for the establishment of the relative value ranking of operators, with the ultimate purpose of determining the best operator to apply in the given situation. In algebraic terms, an operator estimate is a mapping from a (*state, operator*) pair to a scalar.

A *state* estimator is one that is used on a design state in order to estimate or evaluate the numeric values of some set of interesting properties, for example area and cycle time. The estimate may be of high or low accuracy, depending on the level of detail in the design and on the algorithm used to make the estimate.

Note that state and operator estimators can potentially be constructed at any level of abstraction. For example, a state estimator could base area estimates on layout, or on a pair of module and net lists, on a logic diagram, on a dataflow diagram, or on a collection of assertions about a hypothetical design state, in decreasing order of accuracy. Similarly, an operator estimator might use as input a state at any level of abstraction. Note, however, that operators do not change during the design process; hence it is unnecessary for operator estimators to accept operator models at any level of abstraction.

It is a requirement on a UDAS that the estimators of both types be capable of making their estimates on the basis of a variable amount of data. The nature of the information available may also be unpredictable because of

[2]Henceforth we will use the word *operator* in the context of the UDAS, in the place of the word *tool*, which has a slightly different connotation. For example, a single tool is capable of capturing many design strategies, each of which would be regarded as a different operator.

the discrete design space; the facts that are present will depend on the part of the space (e.g. pipelined *vs.* serial) being explored and the state of the design, yet estimators must function nevertheless. To offset this rather stringent requirement, there is no requirement that the estimators be particularly accurate, although it will later be shown that certain properties of the estimation algorithms lead to good behavior on the part of the design system as a whole.

For example, the area estimator PLEST [Kurdahi 86] is designed to estimate the area of a chip implemented using standard cells, given cell and interconnection lists. Such an estimator is useful if the requisite data is in hand; otherwise it cannot even be run. It would therefore be useless if the relative areas resulting from two dataflow behavior specifications were to be compared. And yet in many cases the goal of dataflow graph optimization is to minimize the area of the ultimate result; hence an area estimator that operates on dataflow graphs is needed. Similarly, other classes of partially complete design states, including design states in which levels of abstraction are mixed, should be provided with estimation techniques. In the case of operator estimators, this situation is exacerbated by the use of a fine-grained tool set, because the tools will then be applied in extremely heterogeneous situations.

1.5.5. Planning

A general-purpose UDAS should embody some form of *planning* in order to be efficient. For a broad class of tool sets and problem domains, planning is an efficient way to construct estimates of design properties and sequences of tool selection decisions.

Planning is defined as an activity whereby the effect of a sequence of

design operators is modeled in advance of its actual execution. The modeling is done in a *planning space*, whereas the execution of the sequence is carried out in an *execution space*. Planning can be done by means of the following four-step cycle.

First, the conformance of a design or segment of a design to the preconditions of operators in the system's tool set must be checked. This conformance consists of two parts: compatibility of data formats and compatibility of data semantics. For example, a logic minimizer might require a certain input *form* for the description of the network it is to minimize. However, it might not be able to correctly deal with cyclic circuits, e.g. flip-flops. A cyclic logic network could be represented in a compatible form, but the *semantics* of the particular network description would be such that the operator would fail to operate correctly.

Second, the advisability of applying operators must be predicted. In a system where many operators could be applied, and where each operator is potentially expensive to apply, some means of ranking the relative advisability of using each operator must be provided. This advisability ranking should take into account the peculiarities of both the design and the constraints and objectives that pertain to the design.

Third, the run time of the various operators should also be predicted or estimated. For example, an exponential algorithm may be desirable in a case where the problem to be attacked or the space to be searched is small; but for a larger problem a less costly operator should usually be used. The advisability of applying an operator is strongly affected by such run-time estimates and by an overall design-time constraint and/or objective.

Fourth, the result of applying each operator should also be modeled.

This model of the state resulting from the application of an operator will be used as the basis for checking the applicability of tools.

At some stage in the planning cycle, a test for the desirability of executing the plan should be applied. If it turns out that the plan should be executed, the planning process ceases and an execution process takes over. During the execution process, some deviation from the modeled process is expected. It is possible that only details of the execution-space design will differ from those predicted in the planning-space design. It is also possible that the execution sequence can fail altogether, that is it may fail to terminate, it may have to be aborted, it may fail to produce a complete design, or it may produce an incorrect design.

1.5.6. Object Based Computation

It is strongly recommended that the system use an object-oriented model of computation. This should be construed to mean that the data and the procedures that operate on the data should be encapsulated together to form *objects*. Such encapsulation provides a natural way to describe and partition the complex data structures, access methods, and behaviors of a UDAS.

There are two basic subdivisions of the recommendation which divide the objects into *ordinary* objects and into *expert* objects. These will be referred to as *objects* and *actors* respectively.

An ordinary object is primarily a data structuring tool, with emphasis on data storage and on inheritance. The multiple interlinked hierarchies of models that describe a design in a UDAS almost require such a representation. Each of the hierarchies is composed of a large number of

items, each of which in turn is an instance of one of a few types. These types are themselves moderately complex, having ten to twenty attributes. Each type defines a set of access procedures as well. Behavior (i.e. responses to messages) inherited from type to subtype and instance provides a way to deal with this complexity in a clean, extensible, and well-structured formalism. Hence an object **74LS283** might inherit a method to calculate **drive-current** from a generalization **LS-ttl-part**, and a method to infer function from a generalization **generic-adder.** While ordinary objects can be implemented using message-passing semantics, we do not require this, because the message-passing model of computation provides more power than is really needed for this purpose, and it is comparatively expensive.

An expert object, or *actor*, is primarily an object used to capture complex but specialized behavior and data. There are only a few actors in the system compared to the large number of ordinary objects. Inheritance is far less important in an actor because the data structures they encapsulate tend to be unique. For example, a single actor can be used to manage the design data as a whole. This actor could represent an entire data base management system, tuned to the needs of the design representation, and encapsulating a huge quantity of data. The message-passing formalism inherent in object-oriented programming provides a convenient way to regard such enormous collections of data and behavior as single entities. Inheritance is of small importance when dealing with such large and comparatively unique objects; encapsulation of complex behavior and data is the motivation here.

In order to solve the problems discussed above, we propose a program with the characteristics described. The program is based on a formal model of the design process, and is called the Design Planning Engine, or DPE.

1.6. The Model

The model of the design process described in Chapter 5 is based on the idea that a design progresses in steps, from an initial specification to an eventual implementation. Each step is an operation on the design, and is carried out by a tool. The tools then have to be characterized in terms of preconditions and postconditions; this leads to a STRIPS-like model (discussed in Chapter 2), which is both programmable and conducive to analysis.

1.6.1. Syntactic and Semantic Compatibility

The model assumes no *a priori* indication of the ordering of tools. Hence tool ordering is a function of particular design states and objectives. Thus the output of every tool is potentially the input of every other tool; and this in turn implies that tools should use a common representation for design information. The design data structure (DDS) described in Chapter 3 provides this "syntactic" compatibility between tools.

The DDS has a number of important properties as a central organizing model for design data, which augment its capabilities as a syntactically compatible interfacing medium. These include

- the uniformity of specifications, targets designs, and library elements,
- the ability to represent both data-transformation and timing behavior separately from any structure,
- the possibility of embedding the representation in a data base management system (DBMS),
- the ability to separate the design data from the code that implements design transformations, and

- semantic redundancy through which verification and validation techniques can be applied.

Just because the form of the data passed from one tool to another can be assured to be compatible, however, does not mean that the meaning of the data is compatible. Again, we do not know what the details of a design will be until we see them; this means that checking for the appropriateness of operators should be done at design time, using preconditions attached to the operators being considered. This gives us a "semantic" measure of applicability.

In addition to preconditions, we model the results of operator application by creating explicit but abstract planning-space models of design states. This allows us to construct *chains* of operators by matching preconditions against modeled design states. These design state models can be used for estimation and further chaining, and they can often be constructed at a lower cost than the actual execution-space states.

By building a perspicuous model of operators into the system, we can ask questions about execution time as well. This can be important in a domain like that of digital design, where many programs represent heuristic approximate solutions to very hard problems. By predicting run times, it becomes possible to trade off design time and quality.

The maintenance of an abstract representation of the target design also gives us a "reasonable" notation, i.e. one which an expert system can use as its data set without paying a large execution-time penalty. Another benefit of having both a "reasonable" notation and an explicit model of operator action is that high-level facts created by operators do not get buried in a morass of detail, and intent information does not get lost altogether.

1.7. DPE: an Updatable Program

The DPE program is described in Chapters 4, 5, 6, and 7. This program performs operator chaining to construct plans. It forms its own high-level objectives (*importances*) to respond to the idiosyncrasies of the particular specifications it is passed: it iteratively constructs plans in an effort to converge to an acceptable plan.

DPE includes several features that are intended to keep programming costs down, both for front-end system construction and for system expansion. This is in response to the dynamic nature of the VLSI environment.

Layered Interpreters

The use of nested layers of interpretation allows programming tasks to be carried out at an appropriate level. These layers are the following:

1. the *rule* level, which captures high-level planning strategy and estimation algorithms,

2. the *frame* level, which captures models of operators and hardware structures,

3. the *flavors* (object) level encapsulates data structures and the operations on data structures, and

4. the LISP level, which is the lowest level of interpretation.

Rule-Based Objects

DPE includes rule-based objects such as a *planner*, a *manager*, and a *frame expert*. The use of rule-based interpretation of these complex data structures means that representations and operations can be changed with a minimum of programming effort.

Rule-based *estimators* are by far preferable to estimators based on nested case statements for two reasons. First, it is conceptually simple to deal

with discrete differences in style, structure, and intent; second, rule-based estimators are comparatively robust in the presence of programming errors, incomplete data, and changing representations.

Another robustness-enhancement strategy is the use of two-layers of operator selection, whereby the choice of an operator is based on (1) its preconditions, and (2) an advisability-estimation heuristic. Under this strategy, if one operator is considered to be inapplicable due to operator characterization bugs or a bug in the initial (specification) state, other operators may still be usable.

Knowledge-based representations for operators and hardware structures are comparatively easy to change. It is also easy to add new ones as new tools and hardware-implementation styles are added to the system. Knowledge-based representations are also comparatively easy to understand, so they can be constructed and debugged easily by users with little knowledge of the underlying LISP code.

1.8. Structure of the Thesis

The remainder of this thesis is organized as follows. Chapter 2, which follows immediately, covers research related to the subject matter of this thesis. Chapter 3 describes the DDS (Design Data Structure), a formalism for representing design information in a unified way. Chapter 4 gives an overview and general introduction to the planning model. Chapter 5 covers the formal model. The program DPE (Design Planning Engine) is described in chapter 6. Chapter 7 describes the run-time behavior of DPE. Finally, Chapter 8 summarizes the main points of the thesis. Appendix I gives details of the DDS that were not deemed suitable for inclusion in Chapter 3, and Appendix II gives the domain knowledge base of the planning system.

Chapter 2
Related Research

The research described in this thesis has roots in more than one area. The two major areas from which this research drew inspiration were Artificial Intelligence (AI) and Design Automation (DA). Recently there has been increasing interest in the combination of the two, with the sub-area of AI called Expert Systems becoming increasingly prominent in the area of DA. A major part of this thesis lies in bringing another sub-area of AI, that of Planning, into the intellectual purview of DA. In particular, the application of some planning ideas to the problem of hardware synthesis is investigated.

This chapter attempts to bring together the references scattered throughout the other chapters into a consistent and coherent framework. The structure of this chapter is as follows. Section 2.1 contains a review of relevant work in the AI subareas of Knowledge Representation, Meta-Level Programming, and Planning. In Section 2.2 research on models of the design process will be discussed. Section 2.3 covers large design automation systems (which either explicitly or implicitly embody models of the design process). In Section 2.4 design object modeling, data bases, and systems that have underlying common representations will be addressed. Section 2.5 concerns formal models of the design process, which are antecedents of the model underlying the design planning engine, and their relationship to each other and to DPE.

2.1. Artificial Intelligence

Four areas of Artificial Intelligence are of interest in this context: those of knowledge representation, planning, meta-level reasoning, and expert systems.

2.1.1. Knowledge Representation

The question of Knowledge Representation is fundamental to AI. The representations used in DPE are a mixture of three basic kinds of representations: *frames*, *rules*, and to some extent *semantic nets*. Frames were originally proposed in a 1975 paper by Minsky (excerpted in [Minsky 81]) as a way of collecting multiple pieces of information about a single thing in a single structure. Semantic nets were pioneered by Quillian with the SIR and TLC systems [Quillian 69]. Woods [Woods 75] and Brachman [Brachman 85] both contributed to the understanding of these structures by pointing out some of the subtle bugs that can result from incautious or promiscuous use of semantic links. Rule-based knowledge is found in most current expert systems (see, e.g., Hayes-Roth [Hayes-Roth 83], or Barr and Feigenbaum [Barr 81]).

2.1.2. Planning

Research on planning and problem solving goes back to Newell and Simon's General Problem Solver (GPS) [Ernst 69]. Green [Green 69] developed important ideas in the use of formal reasoning for problem solving by means of a theorem-proving technique; Hewitt at about the same time developed a language PLANNER [Hewitt 69] which was intended to serve as a general-purpose problem-solving language. Munson [Munson 71] brings up many of the fundamental problems of dealing with a physical execution space; he was dealing with planning for a robot. Concurrently with

Munson, Fikes [Fikes 71] and others at Stanford were developing the STRIPS robot programming/problem solving language. STRIPS eliminated some of the context and notational problems encountered by Green through the use of add- and delete- sets instead of a predicate-based notation for successor states. The next extension of STRIPS was the addition of hierarchical levels of detail to the effects of operators, in the language ABSTRIPS [Sacerdoti 73]. Sacerdoti then constructed NOAH [Sacerdoti 75], [Sacerdoti 77]; NOAH assumed quite general partially ordered task decompositions, and was capable of refining the orderings of the tasks as intertask constraints became apparent in the course of the planning process. Stefik [Stefik 80] followed that research with the idea of *constraint propagation* [Steele 78], [DeKleer 78], within a plan; his system MOLGEN was capable of selecting which of a set of alternative task decompositions would be selected based on constraints generated by other steps of the plan.

In other areas, planning has been used as a vehicle for understanding human behavior [Miller 60] and natural language [Schank 81], [Wilensky 83]. Closely related to planning proper is the mechanism known as a *script*; these were originally introduced by Schank [Schank 81] as a way to represent default knowledge about common classes of activities. Schank then used this default knowledge to make inferences about the meaning of natural language texts which would make no sense without the default knowledge.

2.1.3. Meta-Level Reasoning

Planning is only one form of meta-level reasoning. It is particularly convenient where operators in the form of programs are the subject of reasoning, but there are other ways to organize meta-level architectures [Rosenschein 83]. A widely known and influential project was the HEARSAY effort [Reddy 73]. In that system, a number of co-operating "knowledge

sources" each had a separate area of expertise, all organized around a communication "blackboard" and working on the same problem, that of speech recognition. The HEARSAY architecture was later applied to a similar problem, the Distributed Vehicle Monitoring Testbed (DVMT), [Lesser 83], which was composed of a few HEARSAY-like systems, thus extending the hierarchy of experts one level. Questions of coherence of behavior across nodes of the DVMT were addressed by Durfee et al., [Durfee 85]. Other questions of coherence, having to do with the behavior of a single HEARSAY-like node, were addressed by Hudlicka [Hudlicka 84]; in that research *parameters* were adjusted by monitoring the behavior of rule sets in aggregate. These adjustments were made as a response to perceived faults in the system, e.g. of a sensor and hence of a confidence parameter.

In another system, DeKleer's NEWTON [DeKleer 77], a layered set of representations was used to decide what qualitative choices could be made during the process of analyzing a problem in mechanics. For example, a block on a ramp will either slide or not; if it slides, it may strike another block. In NEWTON the qualitiative set of possibilities was constructed, and subsequently the correct analysis methods were applied to determine which of the qualitative possibilities would take place. The qualitative analysis in such a system corresponds to the planning-space choices in DPE, with the quantitative analysis that occurs later corresponding to the execution space. However, NEWTON was concerned with the analysis of essentially one-dimensional trajectories of point objects in the area of Naive Physics; furthermore, no attempt was made to reason about the properties of the constructs of the qualitative space for any other purpose than to determine appropriate quantitative analysis techniques. This aspect of the NEWTON system was similar in concept to the precondition modeling of operators in DPE, but there is no analog of DPE's postconditions.

2.1.4. Expert Systems

The area of expert systems has been expanding rapidly in recent years. The early successes of expert systems tended be in diagnostic tasks of kind or another, e.g. diagnosis of bacterial infections in the MYCIN system [Shortliffe 76].

A particular case of interest is that of the methodology and system proposed by Utt [Utt 85]; in this system the method of tuning a directed search by means of a curve fit and feedback mechanism was described. This is a variant of both heuristic search and learning by parameter adjustment: it is quite similar in concept to the strategy propounded by Brewer and Gajski [Brewer 86] and to the strategy used in DPE.

2.1.5. Expert Systems for Design

The problems of synthesis by expert system have proved to be less tractable than those of analysis, perhaps because of the large numbers of detailed solutions to each synthesis problem. Nevertheless, a number of systems have appeared in the literature. We will concentrate on the digital hardware design area.

An early effort was the Palladio project at Xerox PARC [Bobrow 81], [Brown 82], [Brown 83], [Stefik 81]. This effort began with the construction of tools and exploration of ideas under which the construction of the Palladio design system was to be undertaken; so far tools and ideas have been the most important output of the project. One of the most important ideas brought out in Palladio was that of an integrated environment in which knowledge-based and algorithmic tools could be freely combined. Palladio also used multiple languages for programming and representation, much the same way as DPE does. Palladio's languages include object-oriented, knowledge-

based, and LISP formalisms. Hence the programming language framework of Palladio is similar to that of DPE.

Among the tools developed for Palladio are LOOPS, the base programming language, which provides the rule-based and object-oriented capabilities mentioned above; editors for interactively entering graphical design information; editors for knowledge base representations; and tools for constructing new representational schemas.

The most successful large ES for design, originally called R1 [McDermott 82] and later XCON [Bachant 84], [Brug 85], solves a tightly restricted problem, that of minicomputer system configuration. In this task domain, there are a comparatively small number of components (e.g. boards and cabinets) to be dealt with, and highly structured ways of assembling them. The system uses a simple control structure under which only a small fraction of the knowledge base (in the form of rule sets) is active at any one time, and which does very limited backtracking.

Another well-known system is the DAA system developed by Kowalski [Kowalski 83a], [Kowalski 83b], [Kowalski 84], [Kowalski 85]. This system was directed at the hardware allocation problem at the high levels, from a modified data-dependency graph to an RTL structure. A simple control mechanism involving strictly forward-chained inference without backtracking, and a collection of program-counter-like state variables, in combination with a highly structured set of modules and interconnection rules, characterize this system. This system was originally developed at Carnegie-Mellon University.

Also at Carnegie-Mellon there are currently a number of other applications of expert systems techniques to the problems of hardware synthesis: in [Birmingham 84] the MICON system which deals with the

problem of designing single-board computers at a high level is discusssed; in [Joobbani 85] routing by expert system (WEAVER) is attacked; in [Kim 83] the TALIB layout design assistant is described. Again, each of these deals with a highly restricted task domain so the recognize-react strategy is usable.

Bushnell and Director [Bushnell 86] are currently in the process of building a system based on scripts, called ULYSSES. In this system the design process is assumed to have a number of discrete, ordered and well-defined phases, each of which is represented as a step in a script. Each step is further broken down into alternative ways in which it can be performed; a recognize-react procedure is used to choose from among them. Hence Bushnell's system is a combination of the *scripts* and the *recognize-react* systems. As such the approach can be expected to be limited in flexibility. As of the date of writing no results on the efficacy of this procedure are available.

Abadir and Breuer [Abadir 86] have attacked the problem of embedding built-in test (BIT) hardware in an existing RTL design. This work is not rule-based. Instead, it uses a frame-oriented representation for knowledge. Furthermore, it is capable of analyzing the properties of designs before they are actually constructed, using a meta-level system. In another area, the embedding of BIT into PLAs has been explored by Breuer and Zhu [Breuer 85].

At Rutgers a project currently under way includes criticism of existing designs [Kelly 82] and redesign of existing designs [Mitchell 83], [Steinberg 84]. This work incorporates ideas developed by Steele and Sussman [Steele 78] as well as de Kleer [DeKleer 78], and used also by Stefik [Stefik 80] in the areas of planning, constraints and constraint propagation. The Rutgers group, however, is propagating constraints along structural features, e.g. within a floor plan or along an interconnect bus; whereas Stefik propagated constraints

through a plan along the dimension of planned time, and Steele and Sussman used them to analyze and predict circuit behaviors in the analog domain. The plans used by the Rutgers group also are at a lower level than those of DPE; in the Rutgers systems operators are directly related to single modules, e.g. ALUs, while in DPE the operators are capable of designing entire RTL datapaths.

Ackland and colleagues [Ackland 85] attacked the structure-to-physical mapping problem with their CADRE system. In this system a set of co-operating experts collectively embody the knowledge needed to perform the translation. The primary means of communication between experts in this system is through *constraints*, which are similar in concept to those of the Rutgers group. The experts are co-ordinated through the good offices of a *manager* expert, which decides when and to whom constraints should be propagated. The metaphor is therefore very different from that of the Rutgers group, and from that of Stefik; specific task classes are mapped onto experts, which then make the detailed decisions as to what design activities are to be carried out. In contrast, DPE organizes task classes in a frame system, which can also be regarded as a form of semantic net. The semantic net has links that indicate which tasks are relevant to particular classes of hardware. DPE chooses tasks from the relevant set on the basis of task models, which can be used to determine which tasks are applicable and which are advisable.

Davis et al. [Davis 82] also quite successfully use constraints, but more in the manner of Steele and Sussman's original work. Davis et al. use constraints to infer the location of physical faults in malfunctioning hardware, so it is useful for them to be able to infer the input, for example, of an adder from its observed output. This work does not strictly fall into the category of expert systems as such; it is more along the traditional AI lines of finding a

good knowledge representation (in this case that of functional and structural descriptions of circuits) and then using it in its special task domain without structuring its procedures into a production system. The work of Davis et al. differs from DPE in that Davis et al. consider the problem of troubleshooting rather than design, so they assume that the problems addressed by DPE have been solved. Their representation of structure and function is roughly analogous to the division between logical and physical structure in the DDS; the power made available by the use of such a division is an important part of their work.

James et al. [James 85] address the problem of semi-automatic design of analog control systems using mixed symbolic and numeric techniques. In the CACE-III system the symbolic layer is "in charge" of the operation of the numeric layer. CACE-III has a number of rule sets for user interface, initialization, tool selection, updating, and validating. CACE-III does not construct plans in the manner of DPE; it has a built-in sequence of the form {specification, design, validation}, and within each of the stages it functions in the recognize-react mode.

The SCHEMA system [Clark 85] [Zippel 83] is an integrated CAD system, where the integrating idea is that of objects (strictly speaking, flavors) which represent modules in terms of representations like schematics, layouts, and waveforms. There are three very interesting aspects of this project from the point of view of DPE: first, SCHEMA is capable of handling analog as well as digital behavior; second, it embodies the SLICES method of scoping the data contained in a module so that only properties relevant to the task at hand are visible; and third, it is implemented with an underlying object-oriented model of design entities, whereby the entities can either be generated on the fly, or stored and retrieved, with the difference being transparent to the rest of the system. The focus of SCHEMA seems, however,

to be concentrated on the interface between the analog and digital worlds; it is at a far lower level of design abstraction than DPE. It is also a highly interactive system in which the user makes virtually all decisions above the module level and does not have to make any below that level: this sharply stratified model of user interaction is a form of rigidity that DPE was intended to avoid.

2.2. Design Process Modeling

Batali [Batali 83] suggests a model of the design process based on decisions and the dependencies between decisions; in such a system the tasks of backtracking and documentation could be made much easier and more efficient. Batali makes the point that this is a retrospective tool; it does not generate decisions, or give criteria for making decisions, but merely documents them. This is symmetric to DPE, which makes and records decisions and models their effects, but does not record the reasons, which remain implicit in the plan/history structure.

Rammig [Rammig 85] discusses a feedback model of a different kind, wherein design steps are followed by checking steps, and then proceeds to tie this idea to a multilevel system where levels of abstraction and the transitions between the levels are major determiners of information flow topology. In this model, however, no effort is made to determine how decisions are made. Instead, Rammig's model ascertains the general nature of the information upon which decisions are based, and the general nature of the information that they produce. The question of operator applicability and advisibility, which forms a major part of DPE's active decision-making capability, is present in Rammig's model insofar as it is stated to exist, but no way to calculate data-dependent applicabilities and advisabilities is provided.

2.3. Large Design Automation Systems

Large design automation systems that do not use AI techniques as a unifying strategy, as opposed to individual programs that use AI techniques, have been reported by a number of groups. Perhaps the most well-known is the complex of programs called CMU-DA, [Barbacci 79a], [Barbacci 79b], [Director 81], [Leive 81], [Hafer 81], [Hitchcock 83], [Kim 83], [McFarland 83], [Kowalski 84] [Bushnell 86], [Joobbani 85], [LaPotin 85], [Walker 85]. This is not a unified system in the sense that there is no single program capable of overseeing all of the other programs; the closest is the ULYSSES system discussed above, which unifies a few representative programs.

SARA, the System ARchitect's Apprentice [Estrin 78], [Estrin 86] is a unified system built around a set of common representations, the "UCLA Graphs", a library that includes facilities for managing the representation, and a set of tools that operate on the representation. However, there is no way for SARA to make decisions on its own; this is part of the philosophy under which SARA was designed. Hence SARA is a bookkeeping, program communication, and consistency-maintenance tool that strongly differentiates between "human" decisions and "program" decisions.

The CADDY project at Karlsruhe is another example of a large DA system, [Camposano 84], [Camposano 85], [Rosenstiel 85], but again there is no unifying program: it is a collection of implicitly linked separate programs.

The MIMOLA system [Marwedel 79], [Marwedel 84], [Zimmermann 79] is under development at Kiel. This system is based on a method that starts with a description of software to be run on the desired hardware; thence to the hardware itself. This allows substantial freedom in the architectural-design phase. The MIMOLA system has a relatively inflexible

view of its target architecture: for example, a horizontal controller with maximum parallelism is always used. This is in part a consequence of the fixed order in which activities are carried out by MIMOLA, which is more characteristic of silicon compilers than of systems such as SARA and CMU-DA.

2.3.1. Silicon Compilers

Silicon compilers were pioneered by Johannsen [Johannsen 79] with the "Bristle Blocks" compiler. This compiler had a fixed target architecture consisting of a "core", a "decoder", and "pads"; these corresponded to the datapath, controller, and I/O of the target chip. The highly structured architecture was mirrored by a fixed order of activity, and the silicon compiler used cells which were generated by hand with a common bus pitch and control routing strategy. While this fixed architecture is useful in some cases, it is by no means universal. Johannsen makes the comment that Bristle Blocks is only one possible compiler, and that more than one such is needed to cover the design space where radically different styles are targeted.

Another effort with a similar underlying implementation style was developed by Fox et al. [Fox 83]. The MacPitts system, and its current successor SILC [Blackman 85] were targeted toward telecommunications applications rather than the general-purpose market envisioned by Johannsen, and started with an input description in a high-level language rather than the explicit structural connections of Johannsen. The use of a high-level language allows the automatic choice of cells rather than their explicit specification. MacPitts did, however, partition the target into a datapath/controller architecture similar to that of Johannsen. SILC can put multiple paths having diverse bit widths onto a chip. It can also generate more complex clocking schemes, and it implements resource sharing.

2.3.2. Estimation

The literature on design estimation is largely populated with estimators that make many assumptions about the data which they will encounter. An example of such an estimator is the PLEST [Kurdahi 86] program. This program assumes a standard-cell layout, an average wire length, and the total aggregate cell width, and estimates the area taken by the circuit, including routing in channels and feed-throughs between channels.

2.4. Data Modeling, Data Bases, and Unified DA Systems

One of the key requirements for a UDAS is the use of a single design representation formalism. In ADAM, the DDS (Design Data Structure) [Knapp 83a], [Knapp 83b], [Knapp 85], [Afsarmanesh 85a] performs this function.

2.4.1. Hardware Description Languages

Examples of hardware description languages that explicitly cover behavior as well as structure are ISP [Barbacci 79a], [Barbacci 79b], and VHDL [Waxman 86].

The EDIF [Crawford 84] standard for design information interchange provides a way to transfer designs between different computers and organizations. The information it can represent primarily concerns physical organization and layout; while it does provide a standard for the expression of behavior, this behavior is described at a very low level, i.e. that of truth tables and state diagrams. As such it is not suitable for the description of complex behavior specifications, but it is more suitable for the description of implementations.

2.4.2. Data Modeling

Hardware description languages are intended as interchange forms or as user interfaces, usually for input. The DDS, on the other hand, is intended to be used as an internal representation; conceptually the DDS forms an intermediate layer below abstractions specifically designed for the application programs, and above the physical data management layer. Hence while an HDL could conceivably be designed to cover the same information that the DDS does, it should serve as a way to get data into and out of the system rather than as an internal form for storage and interchange between programs.

Foster [Foster 84] describes a system based on a single common file, the Common Design File (CDF). This file contains all of the schematic and physical information for the target design, but does not capture higher-level behavioral and timing constructs; hence its utility is limited to the physical design process. It is also limited in that none of the advantages of concurrency, version control, and scoping that accrue from using a DBMS approach are available.

Hsu et al. [Hsu 84] describe a system whereby tool attributes as well as design data are integrated into a common database built on top of the INGRES relational DBMS. This approach is conceptually similar to that taken in the ADAM system, but the data models are very different: for example Hsu et al. do not explicitly model circuit behavior, which is done in the DDS.

The Value Trace (VT) [Snow 78] is a dataflow representation originally developed as part of the CMUDA system. It provided much of the inspiration for the dataflow model of the DDS, to which it is similar. The VT is a

hierarchical single-assignment graph in which constructs called "vtbodies" are used for the representation of both procedures and loop bodies. This issue will be returned to in the next chapter: it is one of the most important differences between the DDS dataflow model and the VT. The VT also provides no explicit linkage between time, structure, and behavior, with the result that all of the data transformation behavior of a target system is captured in the VT.

The Design Intermediate Form (DIF) [Bushnell 83] is a hardware-description formalism intended to serve as a common representation for a number of CAD tools. It is based on hierarchical modules, similar to the modules of the DDS logical structure. The modules have *attributes* attached that establish correspondences between them and, e.g., VT structures; however, the attributes are essentially arbitrary text strings, and hence provide only a minimal interspace relation capability.

2.4.3. ADAM

The ADAM system has been under development at USC since early 1983. An overview of the knowledge-based synthesis part of the ADAM system is given in [Granacki 85]. This part of the system is discussed, in slightly different form, in [Knapp 83a].

The DDS was constructed as the underlying model of data for ADAM; it is described in [Knapp 83b] and [Knapp 85]. The task of embedding the DDS into the object-oriented database called the 3 Dimensional Information Space (3DIS) [Afsarmanesh 85a] is described in [Afsarmanesh 85b].

Techniques for timing scheme synthesis developed for ADAM are described in [Park 85a], [Park 85b], and [Park 85c]. The data path scheduling and allocation problems are attacked in [Park 85c] and [Parker

86]; the problem of area estimation for standard cells is discussed in [Kurdahi 86].

Knapp [Knapp 86] presents a shortened version of this thesis.

2.5. Formal Models of Designs and Design Processes

The basis for the model of program behavior was originally developed by Floyd [Floyd 67]. This model was transformed and re-used in a different way when Green described the planning process in the idiom of predicate calculus [Green 69]. Fikes [Fikes 71] developed the STRIPS approach to expressing the preconditions and postconditions of operators which has been adopted by DPE. In the process of adapting Floyd's model of program behavior, Green and Fikes adapted the "operations" carried out in Floyd's "programs" to be steps of plans: that is, where Floyd was describing an underlying program to be executed on some computer, Green, Fikes and their successors were describing actions to be carried out by a robot. Hence an operator in Floyd's model might be an addition; in Fikes's, an operator might be the action of opening a door. Floyd was interested in describing behavior of programs, whereas Green and Fikes were interested in finding operation sequences. At roughly the same time as Green, Manna and Waldinger [Manna 81] were addressing problems of program synthesis through similar techniques; in their case, the operator sequences were actual programs rather than plans of actions to be carried out in the physical world with a view more to their side effects than to the sequences themselves. Hence Fikes did not worry about conditionals, for example, because the plans constructed by STRIPS were ad-hoc affairs to be used only once; whereas Manna was concerned with programs that were to be re-used, and hence had to pay a great deal of attention to control structures like conditionals.

McFarland [McFarland 83] developed a model of hardware behavior that was much closer to that of Floyd or Manna than that of Fikes, except that the operators he was representing were essentially dataflow operations ultimately to be implemented by hardware resources such as adders and ALUs; by demonstrating equivalences between dataflow graphs using McFarland's model of hardware behavior, transformations of the graphs could be guaranteed to preserve correctness while improving the quality of the hardware implementations. So while the operators that McFarland modeled were similar to those of the program-synthesis operators of Manna and Waldinger, his intention was closer to that of Floyd.

The model used in DPE represents operators that synthesize hardware; hence the intention is a hardware analogy of Manna's program synthesis, the construction of a system capable of being re-used many times in many different situations. The technique, however, is closer to that of Fikes: the operators of DPE are not ALUs and muxes, but code that constructs ALUs and muxes. For example, Fikes's operator "open a door" was a means to achieve an end rather than an end in itself, whereas in Manna's system the operator "add" was an integral part of the program his system was constructing.

This closely parallels the difference between Kelly's CRITTER system, which propagates constraints between design objects like adders, and Stefik's MOLGEN, which propagates constraints between steps of a plan, e.g. the operation of choosing an antibiotic to select bacteria populations, and the operation of choosing the bacteria to be used for the experiment. Hence the Kelly-Manna idea is to operate on a general-purpose target directly, whereas in the Fikes-Stefik idea the goal is to construct a special-purpose sequence to solve a specific problem; in DPE the special-purpose sequence (plan) is directed at solving the specific problem of constructing a piece of general-

purpose hardware (i.e. a digital system), so DPE is closer in spirit to the work of Fikes and Stefik. The formal model DPE embodies is therefore not a model of hardware, but rather of sequences of operations that construct hardware; the model of hardware itself is expressed in the DDS and abstractions of the DDS.

Chapter 3
The Design Data Structure

The first requirement for the successful construction of a UDAS is that of a single uniform representation for designs. This chapter describes the motivation and the details of such a representation, called the Design Data Structure or *DDS*. The DDS is primarily intended to support register-transfer level synthesis and verification activities, but is also capable of supporting logic-level design descriptions. It has a partial capability for representing physical structure (i.e. chip and PC layout, with a rudimentary capability for three-dimensional structures such as card cages), as well as for switch-level designs. It is not intended for use as an analog design representation medium, although it does provide for representation of some analog quantities, e.g. resistance and capacitance. The DDS supports synchronous timing disciplines, and provides a limited form of asynchrony. It does not directly support stochastic performance analysis, and it could be used for interface analysis (i.e. protocols, liveness analysis, etc.) after translation into a more convenient form.

The DDS is intended to serve as an internal representation schema rather than a user interfacing formalism. Hence it is not a "design language" in the usual sense, being more akin to the compiled internal form of a design language. There are two possible classes of user interface to the DDS, exemplified by hardware description language (HDL) compilers, and DDS editors. These are discussed in Section 3.11.

The DDS is intended to serve as a single, unifying data model for a

collection of synthesis and verification tools. The collection is not yet complete, and probably will never be complete. Hence the DDS cannot be said to be a response to a set of necessary and sufficient conditions, because such conditions are a function of the available tool set, and of current design goals, both of which are subject to change. Instead the DDS is a result of a consensus-building process in which hardware designers, tool programmers, and database experts participated.

The representation for designs is one data structure among many required for the operation of a UDAS. Other information includes the knowledge base that describes design tools, documentation of the design process (i.e. design histories), accounting and management information, collections of test vectors, simulation traces, and so on.

A good representation for design information can facilitate program-driven verification, incremental design, user interaction, and backtracking. The representation described here is designed for exactly those capabilities. It has the following properties:

1. the design data is unified,
2. the representation is executable,
3. it facilitates analytic detection of design errors,
4. it supports the evolution of families of architectures,
5. it provides a framework for translating between the data formats of different software tools, hence facilitating their use and construction,
6. it allows the design effort to be partitioned in a clean and well-structured way,
7. it facilitates hardware synthesis under program control, and

8. it minimizes unnecessary redundancy in the design data.

Each of these properties will be discussed in detail.

3.1. Uses for the Representation

There are three major ways in which a design representation can be used. It can be used for specifications, implementations of a target design, and design libraries.

A *specification* is a description of the system a customer would like to have. As such it is primarily behavioral information. Some information concerning the form factor, packaging, pin count, environmental considerations and other constraints on the physical and electrical organization of the target design is usually given. Electrical interfaces are usually specified in terms of voltages and currents, timing, and pinouts or bus interfaces; often the interface specifications are merely references to commonly accepted standards such as the IEEE-488 bus standard.

An *implementation* is a description of a system being designed, or a system for which the design process has been completed. It differs from a specification in that the specification cannot usually be changed. It also differs in that the specification primarily exists as a standard for comparison, and as a starting point for design. Hence, for example, a finished target and the specification may have nonidentical behaviors, which must be shown to be compatible if not equivalent. An incomplete target can be compared to the specification in order to find out in what ways the target is incomplete. Implementations can further be divided into a special *target* design or designs, which are design(s) under construction, and members of a *design library*, which are components that can be used to implement target designs.

The distinction between specifications, design library elements, and target designs depends on the point of view of the observer. For the purposes of this thesis, the specification and the library will be regarded as inputs that cannot be changed by the system, and the target will be regarded as a unique output design that must be synthesized.

The DDS is intended to represent specifications, target designs, and library elements. The reasons why a single schema should be used to represent all three of these, and why that schema should itself be unified, will now be outlined.

3.2. Motivation

There are two questions of motivation to be discussed here.

1. Why is it important to use the same internal representation for the three different purposes described above?

2. Why is it important to use a unified internal representation for each?

The distinction can be clarified by considering two examples. First, consider a representational system in which there are special models for specification, target, and library elements. In such a system, the specification might be described in terms of a compiled VHDL behavior description, the implementation in a state transition graph, and the library in the form of EDIF lists. In this system, the data models underlying the specification, target, and library are different, and it is not clear that any rigorous comparison between representations can be made in a practical system. Furthermore, it is unlikely that relationships between the different representations could be easily maintained.

Now let us consider a system in which the same representation is used

for all three purposes, but the representation itself is fragmented. Such a representational system might consist of a collection of PMS diagrams, Petri nets, DDL textual descriptions, logic diagrams, net lists, semiconductor processing instructions, and equations of state. The same system would be used for library elements as for specifications and targets: but there would be no explicit relationship between the different representations, for example between Petri nets and DDL descriptions, both of which might describe the same target design.

Neither scenario inspires a great deal of confidence. Either way, many questions of completeness, consistency, correctness and extensibility will arise. Consider the following three problems.

First, in a system with different models for specifications and implementations, what is the guarantee that an implementation captures the desired semantics of the specification? Would it be possible to construct a rigorous test for equivalence? If not, how much would it cost to develop and run the software to make a nonrigorous check for equivalence, i.e. to validate the correspondence?

Second, what if the specification contained artifacts, i.e. attributes and features that made it easy to write the specification, but which were not necessary in the implementation, and indeed might complicate it needlessly? What if, on the other hand, certain side effects of the specification were necessary to the correct implementation? How complex would a language be that explicitly stated which effects were necessary and which were artifacts of the description formalism? This problem is bad enough without considering an additional layer of translation.

Third, in a system with multiple ad-hoc representations, how can the

completeness of any description be guaranteed? How can consistency between representations be guaranteed? In general, the number of translators needed will be on the order of the square of the number of representations; some of those translations will have to add information, while others will have to abstract it away. How much is that capability going to cost in run time, programming effort, and program validation? How much intelligence will be required of a UDAS in which no explicit correspondence between a layout, its logic diagram, and the behavior it is intended to implement is maintained? What will happen when somebody tries to extend such a system?

The way to avoid these situations is to use a single, unfragmented base information model for all three purposes. This is not to be interpreted as meaning that all users and all programs must use the raw information model; for example, that a user must examine the entire contents of a library NAND gate in order to find its propagation delay. Some scoping and abstraction capability must be provided, so that users and tools can access only data of interest, and not be buried under a mass of irrelevant detail. But underlying the scoped-down and abstracted representations should be a central, unified information model.

The DDS, an information model of this kind, will now be described. In the course of the description, various choices and motivations will be discussed as well, in an effort to provide a sense of the conceptual basis for the details of the DDS.

A formal definition of the DDS in terms of a database information model is given in [Afsarmanesh 85b] and in Appendix I. An overview and some details of this model will now be presented.

3.3. The DDS: Overview

The DDS (Design Data Structure) is a multilevel design representation [Afsarmanesh 85b], [Knapp 83b], [Knapp 85]. It partitions design information into *subspaces* such that there are no implicit relationships between the objects of one subspace and the objects of another. The four subspaces represent respectively the dataflow behavior, the control and timing behavior, the logical structure, and the physical structure of a design. A *component* is described in terms of these subspaces and a set of relations across the subspaces. The subspaces are near-orthogonal in the sense that decisions taken in one subspace affect decisions taken in another subspace weakly across the region of interest. For example, a single behavioral specification can be mapped into several different implementations with varying speeds, power dissipations, and costs.

The entities of the four subspaces are explicitly related to one another by means of *bindings* (for example, between operations, time intervals, and hardware resources). Such bindings have a number of uses in the synthesis, verification, and validation of designs.

The subspaces are each hierarchies in their own right. For example, one of the subspaces is used to represent schematic (structural) information; this subspace is a hierarchy with block diagrams at the top, registers and ALUs at the middle, gates a little lower, and transistors at the bottom. An important attribute of the four subspaces is that the hierarchies with which entities are described are not isomorphic. This is illustrated in Fig. 3-1.

3.4. The Component

The fundamental structure of the DDS is the *component*. A component represents a single physical entity. That entity may be a specification, a target, or a library element. Examples of components are an entire system, a CPU, a board, a chip, and a layout cell.

A specification is represented as an incomplete component; that is, it contains partial information about the target design, represented in the same way as the completed design. The information that is present in the specification must at a minimum represent the operations the target must perform and the constraints it must meet.

A design in progress (*target*) is an incomplete component. As the design process progresses, the target should gradually become more and more complete, until finally it can be manufactured. In the initial stages of design, the target component contains primarily dataflow and timing information; in the later stages it will contain more schematic information, and finally most of the information will be physical layout. The original dataflow, timing, and schematic information are preserved for documentation, verification, and validation purposes.

A *design library* contains both procured components and 'in-house' components. When a library element is used in a target design, a reference is generated from the physical model of the target to the design library element being used. Because the target has an explicit representation of the intended behavior of the library component, and the library component has a model of its intrinsic behavior, both specified and implemented behaviors can be represented. For example, a specified *add* behavior can take place in an ALU which is also capable of subtraction.

3.4.1. Models of a Component

Each component is described in terms of four *models*, which correspond to the subspaces described above, and a set of *bindings*, which are explicit relationships between the four models. The four models are the projections of the component onto the four subspaces.

The four models of a component are the dataflow behavior, timing behavior, logical structure, and physical structure models. Each of these models is hierarchical. For example, the physical hierarchy of a large computer might have a cabinet at the top level, a card cage or two at the next level down, some boards at the next level, and so on. The logical structure at the top level might consist of CPU and a memory; at the next level the CPU might consist of a controller and a datapath; the datapath might consist of some registers and operators.

The hierarchies embedded in the four models are not isomorphically decomposed: that is, there is no one-to-one and onto relation between elements of one model and elements of another implicit in the hierarchical decomposition mapping. For example, the physical structure of a computer might contain a 7404 TTL part. This single physical part can be decomposed into six inverters. In the logical structure, one of the inverters might form part of an ALU, another might figure in address translation, another in the interrupt vectoring hardware, and the remaining three might not be used at all. A single, isomorphic decomposition for both the logical and the physical structure would then lead to a poorly organized hierarchy for either the logical model or the physical model. Similarly, the definition hierarchy of dataflow operators might have a many-to-one mapping of additions onto a single logical adder: this might be the case where both addressing and data operations were done in the same ALU in order to reduce cost.

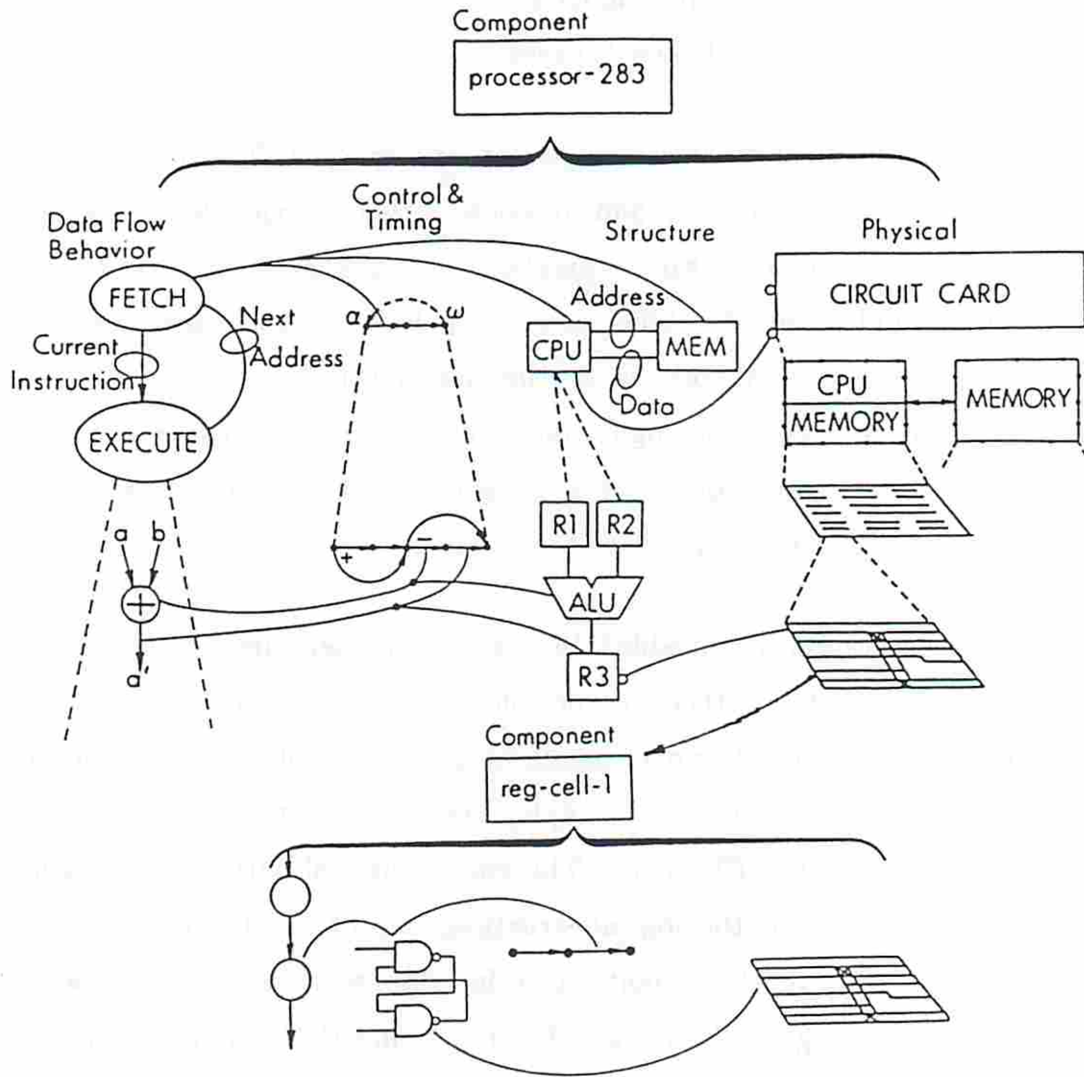


Figure 3-1: Example showing four models of a component

The lack of a clear implicit relationship between the elements of the four subspaces leads to explicit *bindings* between subspaces. Bindings give a formal representation for relationships between elements of different subspaces: for example, between the addition (dataflow behavior), a specific timing range, and the ALU (a logical-structure element) in Fig. 3-1.

3.5. Details of the Four Subspaces

This section discusses the four subspaces and their corresponding models in detail. Note that there is a strong syntactic similarity between the models of the four subspaces. Each is modeled as an annotated graph, where edges and vertices both have meaning, and extra information can be attached to both. The ways in which hierarchical decompositions are described are also highly similar. This syntactic similarity is an artifact, useful only because it makes the DDS easier to understand and use. In each subspace different names have been given to the vertices and edges.

3.5.1. The Dataflow Subspace

The dataflow subspace is used to represent the data-transformation behavior of hardware. It is modeled using a dataflow graph similar to those used in optimizing compilers. This graph is a directed acyclic bipartite graph (N, V, E), where N is a set of dataflow operators (i.e. *nodes* in traditional dataflow parlance and vertices in the graph), V is a set of dataflow values, also represented as vertices of the graph, and E is a set of directed edges that connect nodes and values.

The dataflow graph is a *single-assignment* graph. That is, each node is unique, a symbol that represents a single function application. A value is the result of function application; it is not the same as a variable. A value can be generated only once, and two values are identical only if:

- they are both constants, and the constants are equal, or
- the value names are synonyms[3].

Where a (possibly infinite) loop is represented, *subscripts* are attached to the graph to distinguish between different passes through the loop. A detailed discussion of subscripts is given in Section 3.7.

Both nodes and values are hierarchically defined. An analogy can be drawn to function and data type definitions in Pascal: a function can be defined in terms of other functions, which are either Pascal primitives, or they are explicitly defined in terms of other functions and procedures. Similarly, a node is recursively defined in terms of its **kind**, which is either a primitive, or it represents a dataflow graph, composed of further nodes and values. Similarly, values also have a **kind**, which is analogous to the **type** of a Pascal variable: it may be a primitive type, or it may be a structured type.

Figure 3-2 shows the dataflow graph definition of a function $y=mx+b$. There are two nodes, representing addition and multiplication, and there are five values, representing the input x, the output y, two constants m and b, and an intermediate value connecting the product and the sum.

This graph can be used as the *node definition* of a new node f, with input p and output q, as shown in Fig. 3-3. The values m and b are hidden inside the definition of f, as are the multiplication and addition nodes. The definition can be referred to whenever the function $f(x)=mx+b$ is needed.

The multiplication and addition nodes in Fig. 3-2 are *node references*: that is, they contain references to definitions of multiplication and addition.

---

[3]Synonymous value names normally arise in the case of subscripted values; e.g. X_i and X_j are synonymous when $i = j$.

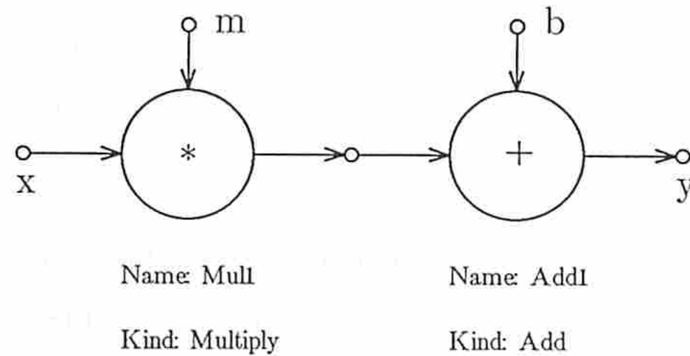


Figure 3-2: Dataflow graph of the function $y{=}mx{+}b$

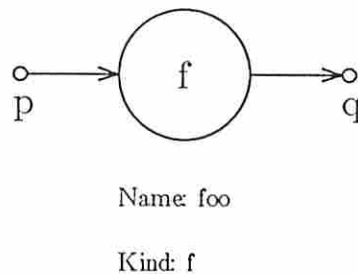


Figure 3-3: Dataflow graph of the function *f(x)*.

Hence the node reference *mul1* has **kind** *multiply*, which could either be a primitive or a further dataflow node definition. Such a definition might be needed, for example, if an exotic representation for numbers was to be used, or if the system had no built-in primitive for multiplication.

In Fig. 3-4 the graphs of Figs. 3-2 and 3-3 are given in the form they would take internally, with field names on the left, and field contents on the right. Arrows represent references. The figure has been simplified so that not all arrows are shown pointing to structures; some arrows point to names in parentheses, which means that the actual references are not shown. Where square brackets are used, a list of entries is indicated. Where single quotes are used, the item is a string.

A dataflow node reference contains the following information:

1. the *name* of the reference, e.g. the name *mul1* in Figs 3-2 and 3-4,

2. the name of its *kind*, e.g. the kind of *mul1* is *multiply*,

3. an *intended function*, which is an English text describing the function the reference is to fulfil, and

4. a *screen position*, which gives the position in which the node reference is to be displayed when the graph is plotted. This field is only a token representative of other data that would be easy to store but difficult to calculate. Many other possibilities, e.g. a schematic symbol shape, can be defined.

A dataflow node definition contains the following information:

1. the *name* of the definition, e.g. *f* in Fig. 3-2,

2. the *function* the definition describes, which is an English text; in the case of Fig. 3-2 the text might be "straight line generator",

3. the name of a *designer*, i.e. the person who actually constructed the definition,

4. a *complete* bit, indicating that the definition is either complete or not,

5. a set of *value references*, which are the internal values of the graph describing the definition, and the connections of those values,

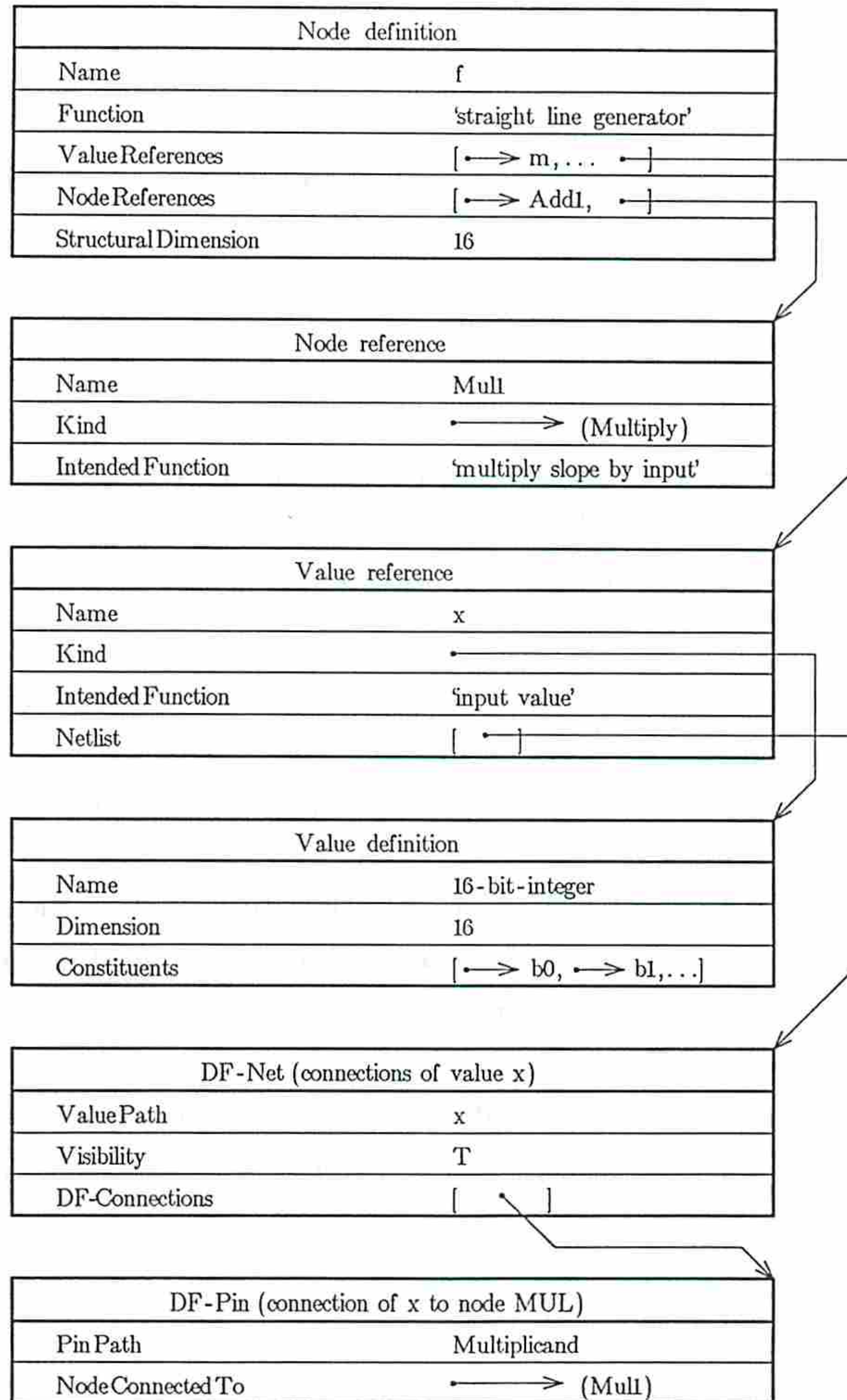


Figure 3-4: Dataflow node definition and references.

6. a set of *node references*, describing the graph's nodes, and

7. a *structural dimension*, describing the bit width of the node; in Fig. 3-2 the multiplication might be a 16 by 16 multiplication.

Dataflow values are similar to dataflow nodes, except in the matter of value references, which refer to both a vertex $v \in V$, but a collection of edges $\{e_1, e_2 ...\} \subseteq E$ as well.[4] A value reference contains the following information:

1. a *name*, which is the name of the reference, e.g. x in Fig. 3-2,

2. the name of its *kind*, which is the name of a value definition; e.g. *sixteen-bit-2's-complement*,

3. the *intended function*, an English text describing the intent of the value,

4. the *screen position* in which the reference should be displayed, and

5. a *netlist*, which is the list of edges that connect it to nodes.

The netlist of a dataflow value is actually not as simple as it sounds here, in part because of the hierarchical description used in the DDS. Netlists are further discussed in Section 3.9.

A dataflow value definition consists of the following:

1. the *name* of the definition, e.g. *sixteen-bit-2's-complement*,

2. the name of its *designer*, i.e. the person responsible for it,

3. a *complete* bit,

4. its *dimension*, i.e. its bit width, and

---

[4]The reader will recall the bipartite graph (N, V, E) of Section 3.5.

5. its *constituents*, which are further value references.

3.5.1.1. A Small Example

The logical and physical structure subspaces are syntactically similar to the dataflow behavior subspace. The timing and sequencing behavior subspace, however, is substantially different.

3.5.2. The Timing Subspace

Timing and sequencing models are used to describe the timing and ordering of events in hardware. Such a model is constructed of *ranges* and *points*. A range represents a constraining or derived duration of time, an ordering relationship, or a causal relationship between points. Points represent events of infinitesimal duration, which separate ranges. A range may have either a known, an approximately known, or a wholly unknown duration; and any two points may or may not be linked by a range. Ranges are directed: i.e. they indicate the direction of the flow of time. A nonhierarchical graph of points and ranges is a directed acyclic graph, corresponding to a partial ordering of events (hierarchical graphs can appear to be hypergraphs). The partial ordering can convey both conditional branching and concurrent execution, both of which are important in hardware design. An example timing graph is shown in Fig. 3-5.

Parallel and conditional sequences of events are represented by sets of ranges falling between pairs of *fork* and *join* points. A pair of parallel sequences will be described as a pair of ranges with a common *and-fork* source point and a common *and-join* sink point; the *and* denotes that all branches will be traversed in parallel. Such a pair of parallel sequences falls between points t_3 and t_5 of Fig. 3-5. A pair of conditional (*i.e.* exclusive) branches is represented by a pair of ranges connected at both ends by *or-fork*

and *or-join* points; a predicate[5] is attached to each of these ranges, describing the conditions under which they become the taken paths. Conditional branching is exemplified by points t_2 and t_7 in Fig. 3-5.

Ranges are defined hierarchically in terms of other ranges and points, in a manner similar to that of dataflow nodes. A range definition has the following contents:

1. a *name*,

2. a *complete* bit,

3. a *designer*,

4. a set of *durations*, which are constraining or achieved values, and which may be either constants or numeric expressions,

5. a *causality* bit, which distinguishes ranges that describe causal (as opposed to measuring) time relationships,

6. a set of *point references*,

7. a set of *range references*, and

8. a *subscript*, which is the name of a symbolic or constant subscripting variable, and which is used in loop description, as discussed in Section 3.7.

A range reference has the following constituents:

1. a *name*, i.e. the name of the reference,

2. a *kind*, i.e. the name of the range definition to which the reference refers,

3. a *role*, which is an explanatory text in English,

---

[5]The predicates of the timing subspace are actually more complex than this suggests; they will be further discussed in Section 3.8.

4. a set of *predicates*, which are discussed in detail in Section 3.8, and

5. a set of *asynchronous predicates*, also described in Section 3.8.

Points, on the other hand, are primitives and have no explicit definition within the model. Point references have the following contents:

1. a *name*,

2. a *kind*, which is one of a predefined set of primitives,

3. a *role*, which is an English text,

4. a set of *sinks*, which are ranges to which the point is connected, and which are "later" than the point (i.e. out-directed),

5. a set of *sources*, which are ranges to which the point is connected, and which are "earlier" than the point,

6. a *subscript*, whose function is similar to the subscript of a range reference, and

7. a *visibility* bit, which allows the point to be "seen" from outside the range definition (the analogous construct for dataflow values, etc., is discussed in Section 3.9.)

The set of point **kinds** is enumerated as follows:

1. *Simple* points represent connections between range pairs.

2. *Alpha* (α) points are used to denote the points to which control is returned when a loopback is made.

3. *Omega* (ω) points are used to denote the points from which control is transferred when a loopback is made.

4. *Fork* points are used to denote conditional (*or*) and concurrent (*and*) branch points.

5. *Join* points are used to denote the re-merging of a control stream

after the end of a concurrent execution phase or the end of a conditional action. They are labeled either *or* or *and*.

These primitive point kinds are the same as defined in [Knapp 83b]; however, see [Granacki 86] for a different view of the timing subspace.

3.5.2.1. Conditional Execution

The representation of conditional execution can cause a good deal of confusion. Consider two cases:

1. The function $f(x)$ is calculated in two different ways depending on what part of its domain x falls into.

2. The machine M undergoes radically different sequences of states depending on the value of a single bit *interrupt*.

In the first case, it is natural to express $f(x)$ in terms of two dataflow graphs joined by a select node. In the second, extra timing and sequencing information is present, which may have some bearing on the overall design problem. While it is possible to express both of these situations in either an augmented control or an augmented dataflow graph, and in theory freely transform the one to the other, in practical fact these are different kinds of choices. Some kinds of conditionals can be classified either way. Work is in progress [Granacki 86] on defining these ambiguous cases more rigorously, and on defining the valid classes of transformations between the two representational styles.

3.5.2.2. Looping

The special points α and ω are used in the representation of loops. The point α is the initial point of a loop, and the point ω is the point at which the loop recycles to α. The α and ω points are given symbolic subscripts. These subscripts are used in distinguishing values and nodes in different iterations of the loop.

Looping and conditional execution are closely related. It should therefore not be surprising that loops can appear in more than one form in the DDS. For example, a loop can be modeled in the DDS dataflow space through the use of a subscripted variable whose (*i-1*)st instance is consumed by a function that produces the *i*th instance. Similarly, a loop can apply different functions to the elements of a vector, or to arbitrary unrelated data elements. In such a case the only evidence of looping is in the timing subspace.

A loop can be generated in any of the following ways:

- it can be compiled directly from an HDL that supports looping,
- it can be derived from a dataflow graph,
- it can be input to the DDS directly, and
- it can be the result of transforming a behavior that might or might not have contained loops before transformation.

3.5.2.3. Example Timing Graph

The example of Fig. 3-5 represents the basic instruction cycle of a simple computer. It illustrates the following points.

1. Attached to the interval *RESET* is an *asynchronous* predicate (*VRESET*, α_{reset}). The purpose of the predicate will be discussed in detail in Section 3.8; for the moment it will suffice to say that it describes the reset behavior of the system.

2. The points α_i and ω_i together represent the beginning and ending of the instruction cycle. Loops are further described in Section 3.7.

3. The point α_i is a parallel-fork point denoted by the adjacent dot, and signifying that both of α_i's successor ranges will be active at the same time. In this case, one branch increments the PC while the other interval describes an instruction fetch and decode. These are described at a high level: none of the internal details are given.

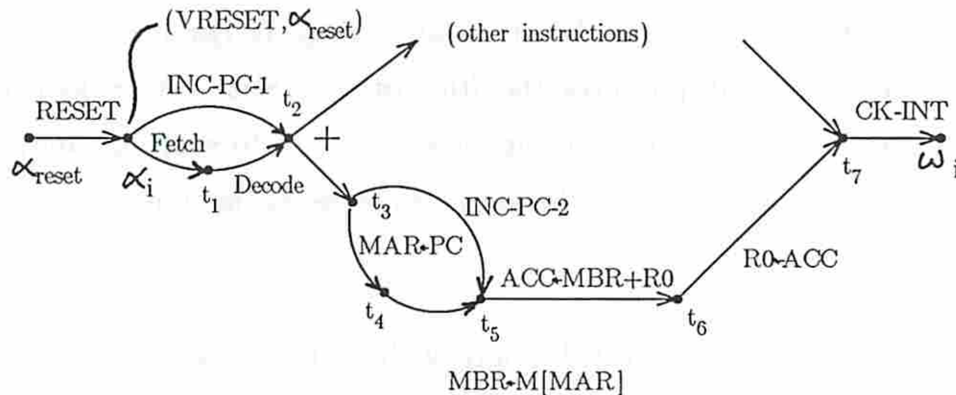


Figure 3-5: Timing and control graph for an instruction cycle.

4. When both *INC-PC-1* and *DECODE* are finished, a parallel join point (denoted by another dot) merged with a conditional fork point (denoted by the symbol "+") is encountered. The merged point is called t_2, and its meaning is that after the decode and increment are both done, a number of different sequences that exclude one another are possible. Only one of those sequences is shown here; it corresponds to the operation of adding a directly addressed word to the contents of a register *R0*. Each of the possible sequences is characterized by a predicate which describes the conditions under which that sequence will be "taken".

5. The first operation along the illustrated branch is a "dummy" arc inserted to establish the ordering of the parallel and conditional forks t_2 and t_3 unambiguously.

6. Following t_3 are two parallel sequences. One is a copy of the PC increment interval; the first range of the other is named *MAR←PC*. (Note that the names attached to ranges are only descriptive; they do not necessarily reflect the operations that are be bound to the interval. The names in this example have been chosen to suggest an underlying set of operations. The DDS uses a much more formal way to express the relationship between operations and time: this is described in Section 3.6.)

7. Following the memory address register load, a directly addressed operand is fetched from memory, during the interval *MBR←M[MAR]*.

8. Following completion of the parallel instruction execution sequences at t_5, the addition is performed during the interval *ACC←MBR+R0*.

9. The result is then stored in *R0*. At this point the instruction execution phase is finished; hence the conditional execution streams forked in t_2 can be merged (*R0←ACC*).

10. Following the merge an interval named *CK-INT* is traversed; this name was chosen to suggest checking for an interrupt. After this interval the point ω_i is encountered, signifying that the next range to be traversed is that following α_i, i.e. the parallel ranges *FETCH* and *INC-PC-1*.

Note that this timing graph may not describe a correct machine: if the PC increment and memory-fetch timings are too far from some set of correct values, then a race condition will occur, because the PC is updated without waiting for completion of the memory operation. A program that detects this kind of problem is mentioned in Section 3.6.

3.5.3. The Logical Structure Subspace

The logical and physical structure subspaces are syntactically similar to the dataflow subspace. The differences are primarily in the detailed composition of the entities; Appendix I contains the full type/subtype hierarchy definitions.

The **logical structure** model of a component is analogous to a traditional schematic diagram. Note, however, that a traditional schematic diagram contains a substantial quantity of implicit and explicit functional information, e.g. the use of special symbols for OR and AND gates. In the

DDS, such functional information is expressed in the dataflow model and hence is not regarded as structural information. The logical structure model is used to represent abstract hardware *modules* and connecting *carriers*. Modules are interconnected by carriers to form a description of electrical topology. Each module references a component used to implement the module, by means of an implementation binding to a physical model, which in turn corresponds to a library component. The llibrary component contains a set of four models, as shown in Fig. 3-1.

A carrier is a structural object to which values can be bound; i.e. it has some kind of storage or transfer function. A module can store data if it has internal carriers capable of storing that data.

The example of Fig. 3-6 shows a simple two-bus CPU architecture capable of implementing the timing graph of Fig. 3-5. The buses *B1* and *B2* are not truly edges in the graph-theoretic sense; they are actually second-class vertices in a bipartite directed graph. This situation occurs also in the physical and dataflow models, and will be further discussed in Section 3.9.

3.5.4. The Physical Structure Subspace

The physical structure model of the DDS is used to represent layout topologies, distances, sizes, and layers. The physical properties of a design are represented in terms of *blocks* and *nets*. Blocks can be used to represent several variants:

1. cabinets laid out on a floor,
2. three-dimensional coordinates of cages, drives, auxiliary and backplane boards and substructures,
3. one- or two-dimensional cage slot layouts,

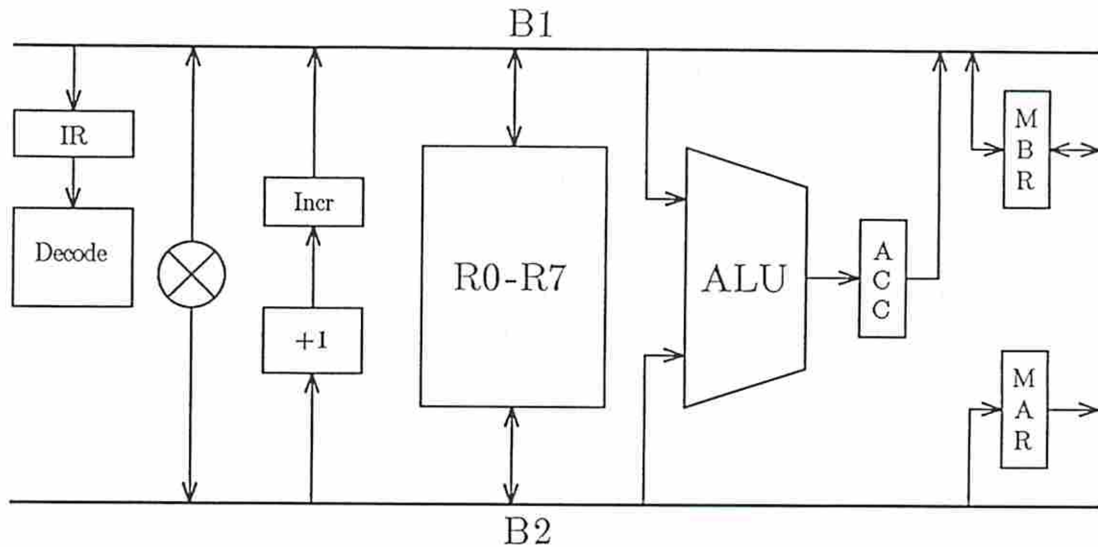


Figure 3-6: Logical structure of the simple computer.

4. boards of a number of types, e.g. printed, multiwire, and wire-wrap,

5. multichip packaging modules such as the IBM-pioneered thermal conduction modules,

6. packaged chips, e.g. DIPs and flat-packs,

7. "naked" chips as yet unpackaged,

8. regions on a chip, e.g. standard cells, macrocells, and gates, and

9. primitive cells, e.g. interlayer contacts.

Each of these variants has special requirements; the total number of requirements is too large and technology-dependent to list here. Nets have similar variations, depending on whether they represent round wires, power supply cables, shielded and twisted-pair lines, PC traces, or on-chip routing.

Blocks and nets have physical attributes such as size, weight, and cost. They can also be given labels that denote technology (e.g. TTL or CMOS),

implementation strategy (e.g. PLA, random logic), function (e.g. "contact pad" and "pullup"), and electrical parameters such as capacitance, inductance, source impedance, and resistance.

A block has explicitly denoted *pins*, which are its connection points. Blocks and nets obey the following rules.

1. Each block is either a primitive block or it is composed of blocks and nets. Hence the block A may be composed of the primitive block B and two nets M and N. Because B is a primitive it contains no further blocks or nets.

2. Block and net instantiations have locally unique names, positions and transformations. In the context of A, for example, M and N are locally unique.

3. A block or net instantiation made by means of a special subscripted instantiation represents a regular array of objects. The array instantiation includes position expressions that describe the detailed placement of each array element. For example, a block E might contain blocks F_i, $0 \leq i \leq 15$, at positions described by the coordinates $(100, 200i)$.

4. A block or net transformation consists of a rotation and/or a mirroring.

5. A net may only be connected to a block at a pin. That is, the pins of a block are its formal connection points, and other connections are syntactically invalid, and should be flagged by circuit-connectivity checkers.

Physical layout and description of electronic components is well handled by such formalisms as EDIF [Crawford 84]. A decision was taken, early in the development of the DDS, to concentrate on the behavior and logical structure subspaces, with later incorporation of constructs to cover the capabilities of EDIF as needed. Hence the DDS is incomplete in this respect.

3.6. Bindings

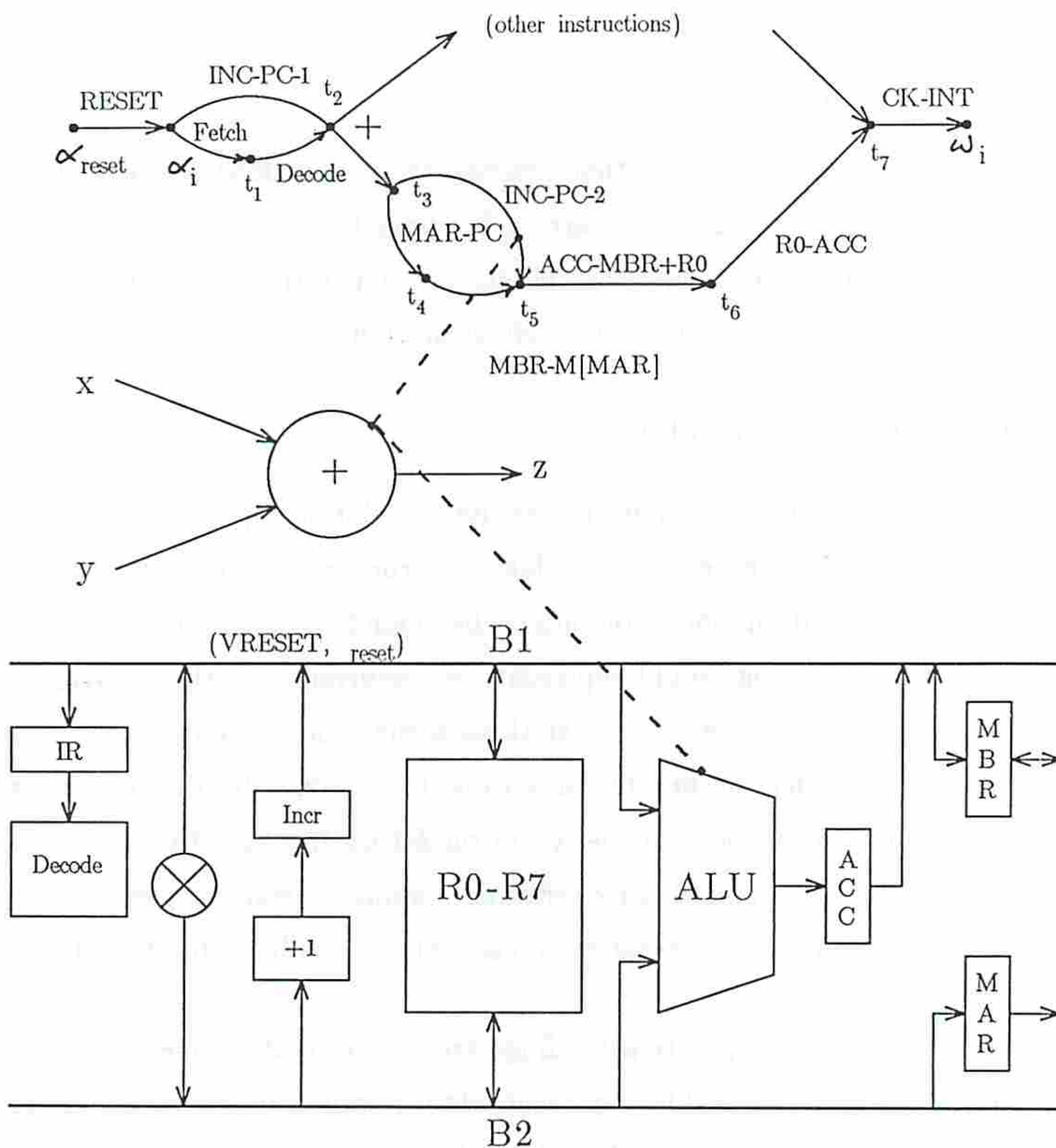
The explicit relations that describe the connections between elements of the four models are called *bindings*. Bindings fall into two classes: *operation* bindings and *implementation* bindings. Operation bindings are ternary relations while implementation bindings are binary relations.

3.6.1. Operation Bindings

An operation binding represents a relationship between a dataflow element, a time range, and a logical structure element. For example, a particular addition operation might be bound to a specific ALU and time range: that binding would represent the occurrence of the operation in the hardware at that time and under those sequencing circumstances. As another example, consider the binding of a value to a carrier *B1* during a range: this means that the value is to be found on *B1* during that time. There is no *a priori* restriction placed on operation binding element types, though some kinds of bindings (e.g. operation to carrier) seem to be of limited utility.

Examples of operation bindings are shown in the table of Fig. 3-7. The first entry in this table represents the relationship between a dataflow operation named *ADDN*, which is of kind *16-bit-2s-complement-addition*, a logical-structure module *ALU-1*, which is of kind *16-bit-ALU*, and a time range with the descriptive (to us) but semantically meaningless name "*ACC←MBR+R0*", which might be of kind *ALU-add-op*. The dotted line in the figure is a pictorial representation of the same binding.

The remainder of the operation bindings shown in Fig. 3-7 concern values: two express bindings to the buses *B1* and *B2*, and one to the module *ACC*, which represents a register.



Operator Binding Table

Node/Value	Module/Carrier	Range(Time Interval)
Node Add	Module ALU	ACC←MBR+R0
Value x	Carrier B1	ACC←MBR+R0
Value y	Carrier B2	ACC←MBR+R0
Value z	Module ACC	R0←ACC

Figure 3-7: The simple computer: behavior, structure, and bindings

It is worthy of note that it would not be possible to use the timing graph as shown to express correct program counter semantics[6]. This is the case because the ranges *INC-PC-1* and *INC-PC-2* are not described in sufficient detail. Consider the value bound to the register module *INCR* during this time: on the one hand it is undefined, i.e. before the incrementer has propagated through; on the other, it is the new value of the PC. If these two distinct values were both bound to the same resource *INCR* during the single interval *INC-PC*, a *value collision* indicative of erroneous scheduling could be detected. The same argument applies to the PC: during the first part of *INC-PC-1* it has the old value, and during the later part of the same range it has a different value. Hence in order to represent the correct semantics, the ranges *INC-PC* would have to be split into parts and the parts bound separately.

Note also that the single module *R0-R7*, which represents an entire register file, would suffer from a similar difficulty, i.e. that multiple values could be simultaneously bound to it without stating which register held which value. The solution to all such problems is to define the bindings at the proper level of hierarchy.

Another problem, alluded to in Section 3.5, is the problem of races. In the example shown here, the PC increment operation may conflict with the fetch operation. This is possible because the fetch and PC increment operations are concurrent; it is not clear whether the *MAR* is loaded from the *PC* before, after, or during the increment operation. This can be resolved by defining some new points and adding either constraining arcs or nominal time values that ensure a race-free operation cycle. If constraining arcs are added,

---

[6] In this example the architecture is like that of a PDP-11 in that one of *R0-R7* is used as a program counter; hence there is no explicit program counter in the figure.

a test for value collision will detect the problem; if nominal time values are added, then a test of earliest-beginning and latest-ending times will detect any possible overlap.

3.6.2. Implementation Bindings

An implementation binding represents a relationship between a logical structure element and a physical structure element. Such relationships do not vary with time; hence the bindings do not include time ranges. Examples of implementation bindings are the connection between the schematic block "CPU" and its card cage or cabinet, and the connection between a gate in a logic diagram and a region of the chip on which the gate has been implemented. Nets and carriers can be similarly bound to one another. Again, no restriction on the element types used in a binding is made in the DDS, although such restrictions can be created and used as syntactic checks on the validity of DDS structures.

3.6.3. Binding Hierarchical Objects

The bindings that connect the four models of a design description are made more complex by the hierarchical nature of the models.

Each component has a set of interspace bindings. The bindings are a direct part of the component definition. Because the models that represent the design in each of the four spaces are hierarchically defined, the elements of the binding tuples are paths into definition hierarchies. Hence an implementation binding (excluding time) might look like

```
CPU.Address-Calculation.Adder.Top-Four-Bits  <-->
                    Cage.Board-3.U-402.
```

This means that the top four bits of an adder that is found in the CPU's address calculation hardware are implemented by a chip named U-402,

which is on board 3 in the cage. U-402 presumably is an instance of some chip that is capable of performing four-bit additions, e.g. it has *kind* 74283.

3.6.4. Bindings and Syntactic Checks of Correctness

Syntactic checking is an important benefit of the explicit representation of bindings. For example, one can easily develop graph algorithms that traverse the four models to see whether certain constraints apply, e.g.:

1. that every operator is bound to at least some time range and logical structure (complexity is linear in the number of operators);

2. that every logical structure has a physical counterpart (linear);

3. that no two values are bound to a carrier, or operations to a module, at the same time (depending on the strength of assumptions about the timing graph, the complexity can be as great as cubic in the number of ranges [Parker 84b]); and

4. that the timing subgraph to which a value is bound (i.e. the set of ranges to which the value is bound) forms a connected graph, and it reaches from the value's creation to all of its uses along a conditional-independent transitive path. (That is, irrespective of conditional branches. If under some conditions of execution the path would not be continuous, a potential overwrite error is implied. The complexity of this test is linear.)

These are essentially syntax checks on the collection of models that represent a design. The relationship of this representation to that developed by Hafer [Hafer 81] is very close; essentially the bindings are the same as the zero-one variables of Hafer's model. However, Hafer's model hinges on mixed integer-linear programming which accomplishes both synthetic and correctness-checking tasks, whereas the graph algorithms of the *V* collision detector [Parker 84b] are strictly for correctness checking.

3.7. Looping and Subscripts

Looping traditionally involves the re-use of variable locations, but this cannot be done in a single-assignment dataflow environment. The way we indicate the correspondence between the successive operations and values of a loop is by the use of subscripts, as in the case of an operation f which is applied to the vector $\underline{x}$ to produce $\underline{y}$:

$$y_i = f_i(x_i),\ 1 \leq i \leq 10$$

With such notation, we can both analyze the inherent parallelism of the problem, and denote the assignment of distinct values and operations[7] to hardware which may or may not exploit the parallelism.

3.7.1. Unfixed loops and their effect on subscripting

A *fixed loop* is a loop that will be executed a fixed number of times. The indices of such loops are easily handled by assigning constant boundary values to subscripts as shown in Section 3.7. An *unfixed loop* is a loop which has an iteration count that is undeterminable at specification time; there are two such loops nested in the following example, which has been given in an Algol-like notation for purposes of exposition.

```
repeat
        programcount := 0;
        reset := false; {note that another process sets reset}
        repeat
                fetch( instruction[ programcount ]);
                programcount := programcount + 1;
        until reset;
forever;
```

To distinguish explicitly between the values and operators of successive traversals of an unfixed loop, we take two steps:

1. describe the initial and final conditions of the loop, and

---

[7] The applications of f above are distinct operations; one can easily imagine them being implemented in separate hardware units.

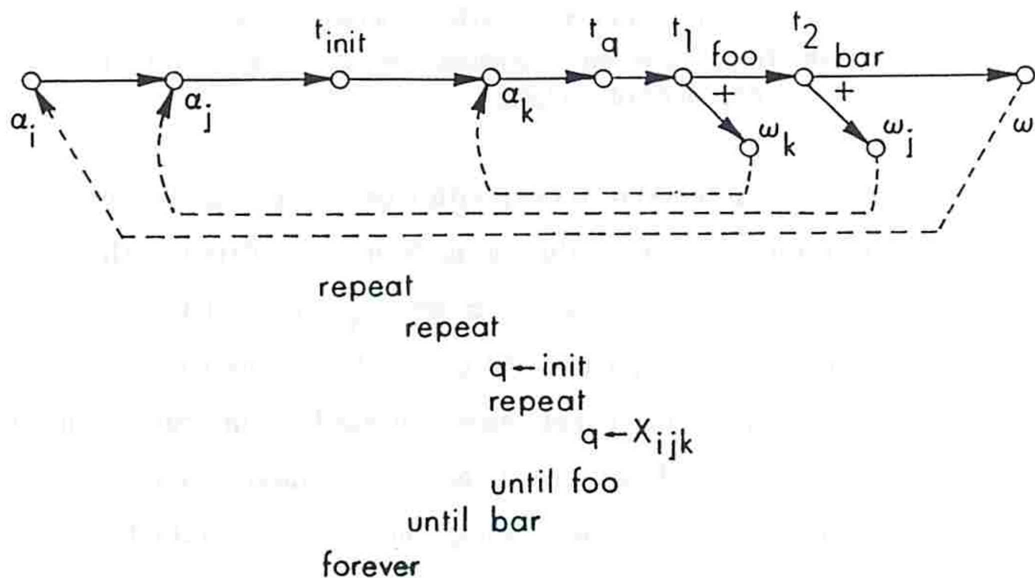
2. establish an equivalence relation between sets of values and operators from different traversals so that each and any can be treated as members of a class.

The formalism chosen to deal with this situation is a symbolic subscript, as in [McFarland 83]. Any value or node may be fitted with a subscripting value or expression; such values as x_i and y_{j-3}, as well as the node ADD_{k+2} can therefore be made explicitly distinguishable. Note that the values used in subscript expressions need not have physical counterparts in the target machine; not all loops have an explicit loop counter. Equivalence classes can be defined (e.g., all those items whose subscripts modulo four are equal to zero). Note that the use of equivalence classes can be applied to fixed loops as well; this is especially desirable in the case of large loop counts. The use of a symbol, as opposed to a range of numbers, allows the formalism to apply to unfixed loops as well as fixed loops.

All subscript values fall into a subrange of the positive integers; in the case of unfixed loops this subrange is finite but of unknown cardinality. Hence we can define a subscript value in this manner:

```
var i : 0 .. I;
```

where I is the name given to the final value of the subscript. Because the value of I is unknown at design time, the symbol I must be used. Note that in the case of a nested pair of unfixed loops, the innermost subscript i may well have many distinct values of I (final loop counts); these values are distinguished by a subscript on I, denoting the loop count of the outer loop in which the particular value of I happened to be the final loop count of the inner loop. Examples of this situation are shown in Fig. 3-8.



value	carrier	range	predicate
$\perp$	q	α_i-α_j	$i = 1$
$\perp$	q	α_j-t_{init}	$i = 1 \wedge j = 1$
init	q	t_{init}-α_k	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i$
init	q	α_k-t_q	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i \wedge k = 1$
X_{ijk}	q	t_q-t_1	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i \wedge 1 \leq k \leq \mathcal{K}_{\mathcal{J}_i}$
X_{ijk}	q	t_1-t_2	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i \wedge k = \mathcal{K}_{\mathcal{J}_i}$
X_{ijk}	q	t_1-ω_k	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i \wedge 1 \leq k < \mathcal{K}_{\mathcal{J}_i}$
X_{ijk-1}	q	α_k-t_q	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j \leq \mathcal{J}_i \wedge 1 < k \leq \mathcal{K}_j$
X_{ij-1k}	q	α_j-t_{init}	$1 \leq i \leq \mathcal{I} \wedge 1 < j \leq \mathcal{J}_i \wedge k = \mathcal{K}_{j-1}$
X_{ijk}	q	t_2-ω_j	$1 \leq i \leq \mathcal{I} \wedge 1 \leq j < \mathcal{J}_i \wedge k = \mathcal{K}_j$
X_{ijk}	q	t_2-ω_i	$1 \leq i \leq \mathcal{I} \wedge 1 = \mathcal{J}_i \wedge k = \mathcal{K}_{\mathcal{J}_i}$
X_{i-1jk}	q	α_i-α_j	$1 < i \leq \mathcal{I} \wedge j = \mathcal{J}_{i-1} \wedge k = \mathcal{K}_{\mathcal{J}_{i-1}}$

Figure 3-8: Subscripting: Effect of Unfixed Loops

3.7.2. Binding Predicates

On the right-hand column of Fig. 3-8's binding table are *predicates* on the bindings. These predicates are used to express the values of subscripts for which the bindings are valid. For example, the binding of the carrier q during the interval α_i-α_j is to a special value $\perp$ (undefined) during the first pass through the loop, as shown in the top row of the table. The bottommost row of the table shows the value bound to q during subsequent passes through the loop.

3.8. Timing Predicates

The predicates of Section 3.7 describe the conditions under which bindings are valid. There are two further kinds of predicates, that have to do with the conditions under which ranges will be traversed:

- *synchronous* predicates, and
- *asynchronous* predicates.

A synchronous predicate is attached to each of the out-arcs of an conditional fork point. Such a predicate indicates the conditions under which the out-arcs become active. For example, in the description

```
if a then foo else bar;
```

the branch *foo* would have the predicate *a* attached to it, indicating that it would only be traversed if *a* were true; similarly, *bar* would have the predicate *not-a*. The characteristic that distinguishes a synchronous predicate is that the time at which the predicate is tested and the branch decision is known. In the example above, the predicate `a` is not tested until the statement `if ...` is encountered.

The asynchronous predicate is different in that it is not known *a priori* when the predicate will become true, and the response to the predicate must

be immediate. It is needed to model events such as resets and timeouts. For example, the 555 chip, which is a timer, can be used to implement a timeout; it consists of an analog timer and an RS latch with asynchronous clear. Upon being triggered, the analog timer is started and the latch is set; when the analog timer times out, the latch is reset. The *clear* input of the latch is taken from a pin of the 555; at any time after triggering, the latch can be reset by activating that pin, regardless of the state of the analog timer; the act of clearing the latch discharges the timing capacitor. Conceivably such a circuit could be represented by

```
repeat
        if a then begin
                latch := set;
                while not (time or clear) do begin
                        sample(time);
                        sample(clear)
                end;
                latch := reset
        end
forever.
```

This representation represents "busy waiting", having the disadvantage that in every loop traversal the signals must be sampled. This is not necessarily a great disadvantage for timeouts. Now consider the case of a more complicated sequence, e.g.

```
repeat
        programcount := 0;
        interrupt := false;
        repeat
                fetch(instruction[programcount]);
                programcount := programcount + 1;
                execute
        until interrupt

forever.
```

This model is appropriate for an interrupt, but for a genuinely asynchronous signal like a system reset it is inadequate because *interrupt* is not tested until the end of the instruction cycle.

One could test for reset in between every pair of statements in the loop;

this is still inadequate, because some machines have block instructions. One could, in fact, push the reset test all the way down in the hierarchy, and insert a conditional (exclusion) point at the beginning of every primitive range: which will only work if all primitive ranges are as short as the shortest time interval of interest.

However, in theory a range is infinitely divisible into subranges and points. If we specify the meaning of an asynchronous predicate to be an *implicit* conditional fork at every point of every range that inherits that predicate, we have in effect created the kind of continuous monitoring we need. Each of these implicit forks has two out-arcs: one to the next implicit or explicit point in the normal-operation timing graph, and one to the first point of the reset-operation timing graph.

An asynchronous predicate consists of

1. a Boolean expression that denotes the monitored value (in the case above, the expression is just the value *reset*), and

2. a destination point to which control will pass if the expression ever becomes true.

In Fig. 3-5 the asynchronous predicate (*VRESET*, α_{reset}) is attached to the interval *RESET*. This predicate is inherited by succeeding ranges; its significance is that if some value *VRESET* becomes true, the instruction cycle will asynchronously end, and the next time interval to be "executed" will be that beginning with the point α_{reset}, i.e. the initialization interval of the computer. When the initialization interval is "executed" all of the bindings that formerly applied become invalid; i.e. a different set of values and nodes become bound following the reset.

Note that every range inheriting an asynchronous predicate is subject to that predicate, i.e. "control" may be passed from that range to the

destination point at any arbitrary subinterval of the range. Hence all branches of a cobegin or conditional can inherit the same predicate and be subject to the same arbitrary transfer of control. The result of this possibility is that behaviors following the destination point[8] should be defined with great care, because of the possibly undefined state of the system at such a point. This, however, is a general problem of asynchrony and not of the DDS *per se*.

This model of asynchrony is limited in its response to very low-level phenomena, and is primarily intended to be serviceable in the context of clocked logic. For example, a system reset appears to be an instantaneous transfer of control when viewed in a context of clocked logic. At the gate level, however, the reset (and for that matter the clock) is not instantaneous, because of propagation delays. The position taken by the DDS is that the behavior of clocked logic is defined under the assumption that the clock interval is long enough that delay-sensitive behavior is not exhibited; and that asynchronous behavior can be modeled as an instantaneous transfer of control between delay-insensitive (i.e. clocked or hazard-free) states, with a partial loss of deterministic state information. Hence, for example, the state of a machine reset halfway through a clock cycle is not defined immediately after the reset.

The semantics, inheritance and use of asynchronous predicates is an active research area [Granacki 86]. The original model of Knapp and Parker [Knapp 83b] states that an asynchronous predicate is inherited by chronological successors of the range to which the predicate is attached, and by their hierarchical subranges. That report is incomplete in the matter of *cancellation* of predicates, i.e. when a successor range does not inherit. The interpretation in that model is that an asynchronous predicate cancels all

---

[8] Strictly speaking, behaviors bound to the range following the destination point.

previous asynchronous predicates. Using such a model, the reset destination can be arbitrarily changed from one range to the next. A range that has no asynchronous responses at all can be represented by the use of Boolean *FALSE* as a condition.

3.8.1. Example: Traffic Light Controller

The following example, from Mead [Mead 79], is a specification for a traffic light controller. The state diagram is shown in Fig. 3-9.

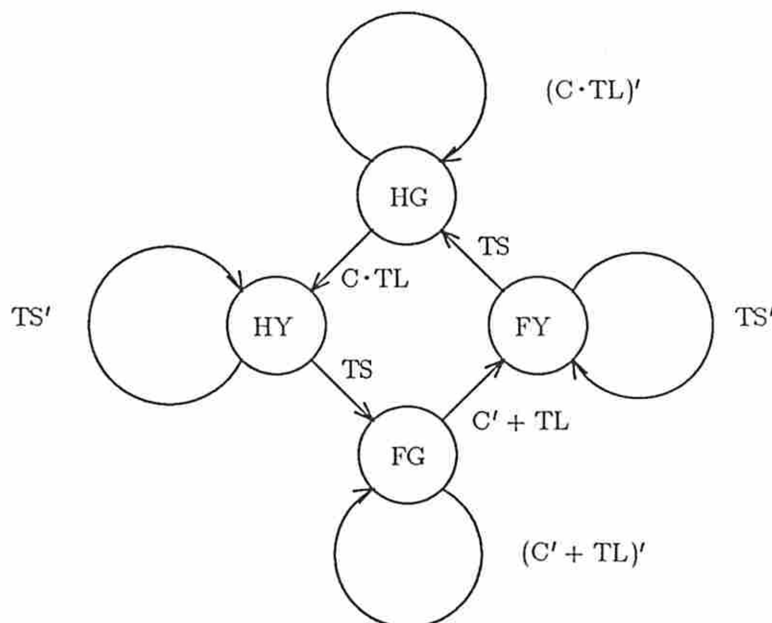


Figure 3-9: State diagram of a traffic light controller.

The controller has four states, which correspond to states of the traffic lights at an intersection of a farm road and a highway. *HG* is the state in which the highway light is green, *FG* the state in which the farm road light is green, and *HY* and *FY* are respectively the states in which the highway and farm road lights are yellow.

There is one input, *C*, representing the presence of cars on the farm

road. Two outputs, *FL* and *HL*, connect the controller to the lights. A timer is also present, which provides a short time-out *TS* for yellow lights, and a long time-out *TL* which is the minimum time for the highway green, and the maximum time for the farm road green. Both timeouts are reset by the same signal.

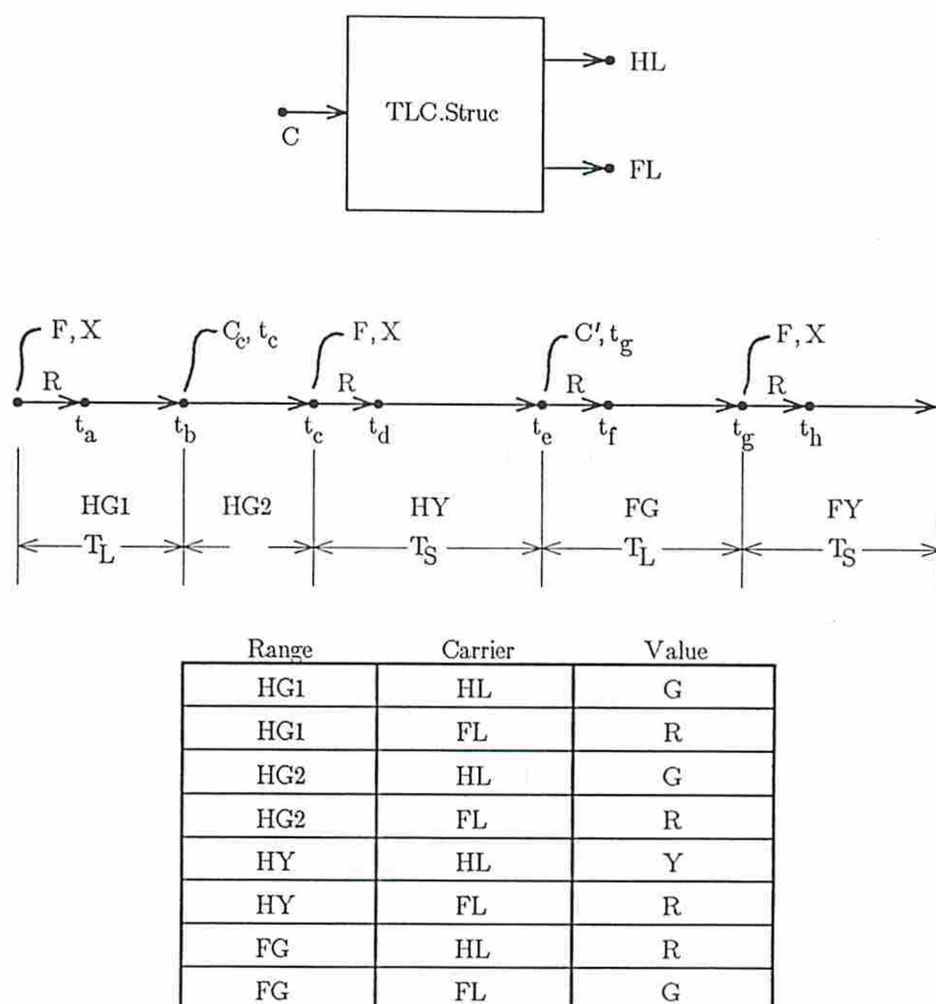
We can express this in the form of an Algol-like program:

```
repeat
        FL := red;        HL := green;
        reset(timer);
        while not (TL and C) wait;      {highway green}
        HL := yellow;
        reset(timer);
        while not TS wait;              {highway yellow}
        HL := red;        FL := green;
        reset(timer);
        while not (not C or TL) wait;   {farm green}
        FL := yellow;
        reset(timer);
        while not TS wait;              {farm yellow}
forever.
```

This behavior can be specified by the timing, dataflow, and logical structure shown in Fig. 3-10. The most important part of that behavior is the timing graph, which is isomorphic to the program code given above.

The timing graph begins with a point α, corresponding to the `repeat` of the program. It ends with ω, corresponding to the `forever`. As shown, the graph represents a nonterminating loop.

The first interval of the timing graph is the interval during which the timer is reset. This is modeled as a short interval. Following that is an interval such that the time α-t_b is equal to *TL*'s interval. At t_b, an asynchronous predicate is attached to the graph, indicating that should C become true, an immediate transition to t_c will occur. The length of the range t_b-t_c will otherwise be infinite, corresponding to the statement `while not`



Range	Carrier	Value
HG1	HL	G
HG1	FL	R
HG2	HL	G
HG2	FL	R
HY	HL	Y
HY	FL	R
FG	HL	R
FG	FL	G

Figure 3-10: DDS description of a traffic light controller.

`(TL and C) wait`. At t_c, the new asynchronous predicate (F, X) is attached to the graph, superseding the old predicate. This predicate means that under no circumstances will an asynchronous branch be taken, and hence the use of X for the don't-care destination. Note that while the semantics of the timer interval was hidden in the program code, the timing graph makes the semantics of time explicit, and allows a duration to be directly attached to the specification.

The logical structure of the specified system gives only the information

needed to specify the values attached to the input and output carriers, i.e. the carriers C, HL, and FL. The values, carriers, and time ranges are related by the bindings of the table. Hence during the range α-t_c, the value `green` is bound to HL, and so on.

3.9. Hierarchies

The four models of the DDS are hierarchical. The notation by which hierarchical decomposition is described is similar for all four. The way in which this is done will now be discussed.

There are two kinds of links inside a hierarchy: *connection* links and *composition* links. A connection link is one that denotes a connection between two objects at the same level of the hierarchy, for example a bus connecting a register and an ALU. A composition link is one that denotes the composition of an object, for example from the gates that make up the ALU to the higher-level description of the ALU. Connection and composition links have nonobvious interactions, which will now be discussed.

A composite object in the DDS is defined by a collection of references to objects. In the logical structure subspace, for example, the objects are modules and carriers. These modules and carriers may themselves be composite.

3.9.1. Net Lists

A carrier may be connected to many pins of many modules, and similar considerations apply to the other three models. Strictly speaking, therefore, the graph representations of the four models are bipartite graphs having two classes of vertices and a single class of edges. In the logical structure subspace, for example, the two classes of vertices are the modules and

carriers, and the edges are connections of carriers to module pins. A situation of this kind can be seen in Fig. 3-6, where the carriers *B1* and *B2* are both connected to multiple modules. We have ignored this rigorous interpretation elsewhere, because it easily leads to confusion. A discussion of net lists requires the precise interpretation, however. In this section the term *link* will be used to denote a second-class node of the bipartite graph representation; hence a dataflow value and a structural carrier are both links.

In the case of a link, for example a logical structure carrier, a reference has a *netlist*, which describes the modules and the pins to which the carrier is connected. For example the carrier *address-bus* might be connected to the pin *out* of a module *cpu* and the pin *addr-input* of a module *memory*. Each entry in the netlist represents a single connection, i.e. of the bus to a module.

Each net list entry corresponds to an edge in the bipartite graph, e.g. connecting a value to a node. The net list is attached to a link instance because it is the instance to which connections are made, and not the definition.

Netlists are complicated by the fact that a link may have a hierarchical structure, and it may be necessary to connect substructures individually. For example, the value instance x of Fig. 3-11 is of type *complex*, which is defined as two subvalues *real* and *imaginary*, each of which is of type *float*, which in turn is composed of a fraction *mantissa* and an integer *exponent*. The real and imaginary components of x could be connected to separate inputs of a node, or even to separate nodes. The netlist therefore contains a *path*, which is a path into the definition hierarchy of the value. For example, the high order bit of the mantissa of the real part of a complex number x might be denoted by *x.real.mantissa.bit-31*. If x were to be connected as a unit, then only a single netlist must be attached for x. However, if there are to be

different connections to substructures of x, it is necessary to split up its connections into several netlists. Such a situation is illustrated in Fig. 3-11, where x is split into *real* and *imaginary* parts, and the parts connected to two pins of a node g.

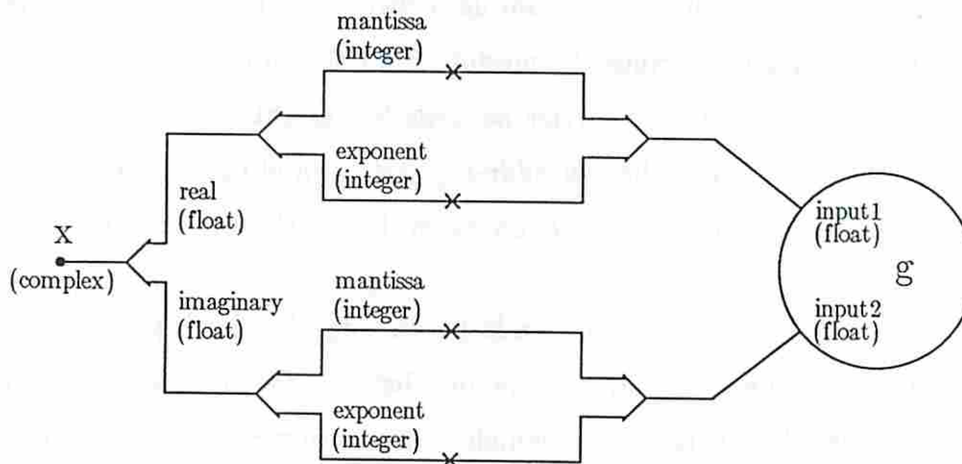


Figure 3-11: Example of a nontrivial structured connection.

The pin to which a link is connected may also be structured. For example, the pin *input1* of g is a pin of type *float*. Again, it may be desirable to connect the substructures of *input1*, e.g. *input-1.mantissa.bit-31*, as individuals; this is done through a *path* in the netlist item, analogous to the *path* of the previous paragraph. In Fig. 3-12 the internal representation of this situation is shown.

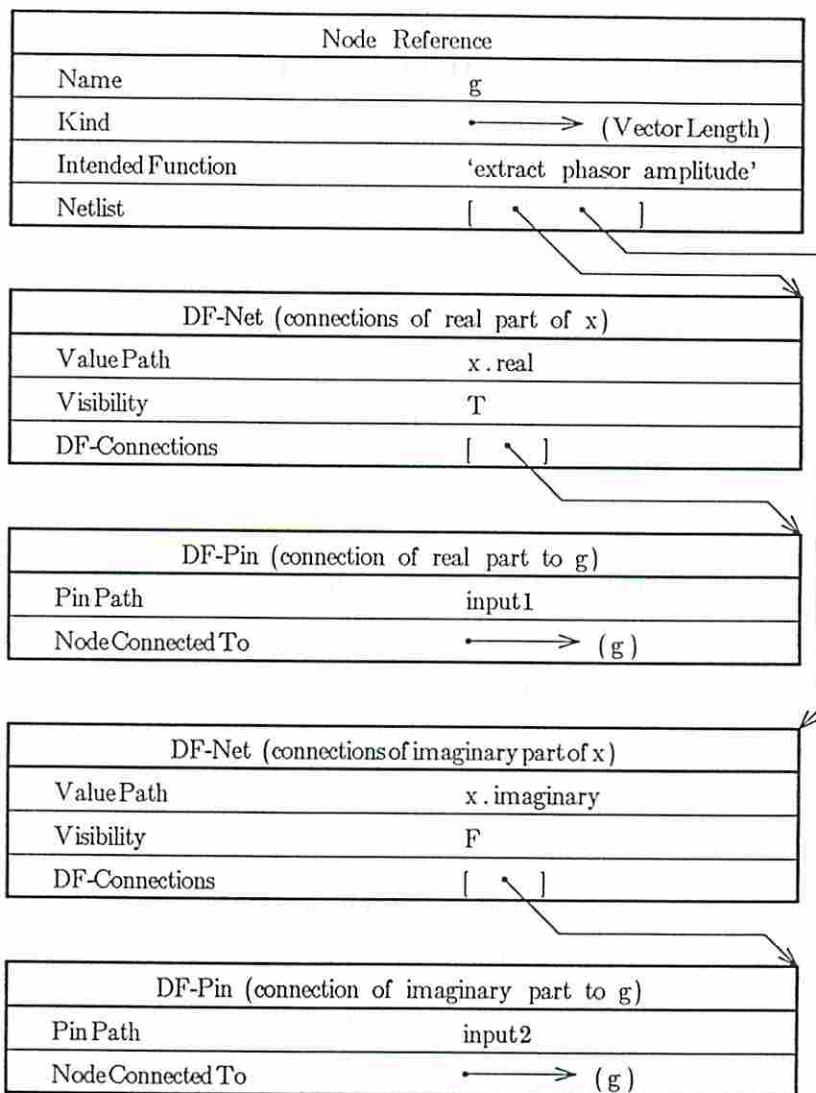


Figure 3-12: Nontrivial structured connection: internal view.

The remaining complication of netlists is the matter of *visibility*. A pin is actually just a *visible link*, i.e. an internal link that has been declared to be visible. Hence the pin *input1* of g is an internal value of g, of type *float*, which is declared to be visible in the definition of g. As another example, a 7404 TTL chip has fourteen nets that are available for connection in the form of physical pins. These are *visible* and can therefore be part of a well-formed connection. Thus visible pins provide a structured way to connect elements at different levels of hierarchy.

3.10. Discussion

3.10.1. Factors Influencing the Choice of Subspaces

The four subspaces were chosen as they were in order to maximize the freedom of the designer, and in order to maximize the correctness and ease with which the specification could be realized. The following three considerations are worthy of note.

First, there are many ways in which a design can be simulated. Each provides a way to check the behavior of a different aspect of the design. For example, the logical structure of the target can be combined with the behavior of the library components of which it is composed, in order to derive the behavior of the target. Once that has been done, the behavior of the target (expressed as a combined dataflow graph, timing graph and interface data) can be checked against the behavior of the specification, which could also be simulated in an effort to validate the specification.

Second, analytical methods of demonstrating partial design correctness can be employed. Such methods include the use of syntactic checks of all four models of a target, for example design rule checks on the physical layout. They can also be used across models, as in the V collision detector which is guaranteed to find all cases of value and operation collisions in a design. They can even be used between the specification, the target and the library, as in the case where the implemented behavior of the target is transformed in a stepwise fashion into an equivalent representation, in order to compare it with the specified behavior; or the behavior of library components is combined with the values and timing known to be present on their pins, in order to demonstrate that the behavior they implement is really the behavior given in the target description.

Third, many different implementations can follow from a single behavioral specification. These implementations may compete with or complement one another. For example, a set of preliminary designs may be constructed in order to explore the design space; these designs may not be at all detailed. On the other hand, a set of finished designs might represent an entire family of machines: one optimized for speed, one for ruggedness, another for low cost. All would share the same specification of behavior in terms of dataflow, but they would differ in the other subspaces.

3.10.2. Relations Between Specification, Target and Library

The four models of a specification are related to the corresponding models of an implementation in that the implementation should be "compatible" in some sense with the specification; that is, the dataflow behavior of the implementation should be a superset of the dataflow behavior of the specification, and all of the constraints of the specification should be met by the implementation. Proving the compatibility of the specification and the target is a large part of proving the overall correctness of the design. This is not necessarily straightforward; if the two dataflow graphs are syntactically very different, then proving input-output equivalence may be impossible as a practical matter [McFarland 81].

A rather different relationship exists between the elements of the design library and the target implementation. The target is composed of library components: these are regarded as physical instantiations of the library components in the target. Hence a target might have four 2901 CPU slices: each is a physical instance of the library component. Put another way, the library components of which a target is composed can be found by looking in its physical structure for blocks marked as being library instances, and seeking the corresponding definitions in the design library.

This is important because of the differences between the library description and the target description. These differences are greatest in the timing and dataflow subspaces. For example, the 2901 mentioned above might be used to perform an addition in one cycle, and a subtraction in the next. There is no close correspondence between the target behavior that the 2901 is used to implement, and the behavior of the 2901 as it is characterized on a test bench. At the best it can be said that the two behaviors are compatible in much the same sense that the target and the specification should be compatible.

The *unused behavior* of a library component is that part of the component's behavior that is recorded in the design library but which is never used: for example, the 2901 is capable of an XOR operation, but that behavior may never be required of it in a particular instance.

Certain correctness-checking algorithms were discussed in Section 3.6. These algorithms use the four models of the target design in conjunction with explicit relationships between the models. By maintaining a set of explicit relationships between data items of the target, the specification, and the design library, additional checking and synthesis possibilities arise. These possibilities are yet to be developed, and specific representational constructs to support them are not part of the DDS as it stands.

3.11. Capabilities of the DDS

The DDS captures a broad range of hardware description semantics. It borrows features from a number of representational formalisms to do this. Its representational capability has allowed its use in a number of ways:

1. a prototype translator PHRAN/SPAN has been constructed, which translates a specification written in a restricted subset of English into DDS constructs [Granacki 86],

2. a register-transfer level pipeline allocator called Sehwa [Park 85a] has been built, using a reduced version of the DDS dataflow subspace,

3. a register-transfer level allocator that uses critical path allocation, named MAHA, has been built on top of the same reduced version of the DDS's dataflow subspace [Mlinar 86],

4. a program has been written that translates from the value trace (VT) of Snow [Snow 78]into the reduced DDS [Mlinar 86], hence establishing a link between the DDS and ISP, which is the source for the VT,

5. a program called V that tests a combined timing graph and set of operation bindings for value and operation collisions in order n^3 time has been constructed [Parker 84b],

6. a program named Agis, which directly edits DDS entities using an interactive color-graphics approach, has been implemented [Parker 84b],

7. a prototype database PKM and data model of the DDS [Afsarmanesh 85b], based on the 3DIS database formalism of Afsarmanesh [Afsarmanesh 85a] have been constructed; this work is directed toward using the database system as a central repository for design data in the ADAM system, and

8. a design library interface called CATALOG [Brotoatmodjo 86] has been constructed, which uses a reduced DDS form to represent information taken from a proprietary standard-cell library and SSI/MSI TTL databooks.

Work is also in progress on defining and extending the formal properties of the dataflow and timing subspaces of the DDS [Granacki 86]. These capabilities were designed into the DDS from the beginning, by borrowing ideas and capabilities from existing formalisms and incorporating them into the framework of the DDS:

1. the dataflow subspace is similar to the dataflow graphs used in optimizing compilers,

2. the dataflow subspace is also similar to the value trace [Snow 78], with the exception of some constructs that fit into the timing and logical-structure subspaces,

3. the dataflow subspace, when bound to the timing subspace, can be abstracted to form the equivalent of tokenized dataflow graphs such as VAL [Dennis 75], which are used for used for dataflow programming,

4. the control and timing subspace by itself is a superset of the control flow graph used by Nagle [Nagle 80] for microcode synthesis,

5. the bindings between dataflow, timing, and structure of the DDS are generalizations of the zero-one variables of Hafer [Hafer 81],

6. it is believed that Petri nets can be mapped into the DDS control and timing subspace, and

7. the logical and physical structure subspaces were designed in such a way that integration of the full power of, e.g., EDIF [Crawford 84] into the DDS would be possible.

The DDS is not complete in the sense that it can provably represent any kind of digital hardware. It is not expected ever to become complete; rather it is expected to evolve to suit the needs of programmers. It now has the capability to capture most of the semantics of the VT and of a dataflow programming language, so it can support RTL synthesis. It is not clear that all of the implicit structural information provided by the VT can be translated directly into DDS constructs. It is also nearly certain that extraneous information (e.g. structures used in defining behavior) contained in a VT or other description cannot entirely be excluded from the DDS translation. It can support much of the semantics of a tokenized dataflow representation such as VAL, but it is not certain that all such semantics can be represented. The DDS can support interrupt and reset semantics to a limited degree, with a simple formalism of a kind not present in either the

VT or VAL. The asynchrony of the DDS is, however, similar to the LEAVE construct of ISP. Some of the interface description facilities of SLIDE [Parker 81] are not directly present; for example, the DDS does not directly provide a rising and falling edge detection capability. However, it does provide a way to get around such problems using combined dataflow and timing graphs. The structure-description capabilities of the DDS are complete in the sense that one can use the DDS to represent arbitrary circuit graphs, hence capturing part of the capability of schematic diagrams. Where schematics use special symbols and annotations to express non-topological information, the logical structure descriptions of the DDS rely on tags and strings attached to modules and carriers. Little has been done with the set of available tags, and strings are difficult to make machine readable; hence the DDS cannot be said to be complete in this sense. On the other hand, the DDS is easily extensible as far as adding tags and special attribute fields is concerned, so this incompleteness is not especially important. In the area of physical-structure representation, the DDS is a superset of EDIF, except in the area of geometrical modeling primitives and tags to denote the diverse possibilities of the physical domain: for example the kind of conduit a power cable is routed in.

No mechanisms have been provided for the control of different versions of a single design, the addition of test hardware and testing data, documentation, liveness analysis, simulation decks, reliability analysis, queueing-theoretic performance analysis and prediction, or hardware life cycle support. Of greater importance here is the lack of a more powerful abstraction capability than the hierarchical graphs upon which the DDS is built. The DDS is as yet incompletely validated, and may prove unsuitable at the lowest levels of abstraction, e.g. at the gate and transistor levels.

The Place of the DDS

The use of a single representation for detailed design information is one step towards implementing a UDAS, which must integrate many separate design tools into a coherent framework. By using a single form, it becomes at least syntactically possible to apply design tools in arbitrary sequences. This is, however, only the first step. We will now turn to control issues, i.e. the question of tool selection and use.

Chapter 4
Planning: Overview

In the previous chapter the problem of unifying the design representation formalism was addressed. Having a single representation paves the way for the use of arbitrary sequences of design activities. Managing such sequences is the theme of this chapter, which presents a software package that uses knowledge about planning, knowledge about the digital design process, and the design description itself to manage the design process.

The process of design is modeled as a process in which abstract models of operators are applied to abstract models of design states in a simulation or *planning* space, until a sequence of operators (a *plan*) that would lead to a complete design has been constructed. The hypothetical design represented by the terminal state is then *estimated*. Either the planning process is then repeated, or the plan is executed in an *execution* space. The implementation of a working prototype of the software is overviewed in this chapter, and treated more fully in Chapters 6 and 7. We will begin by briefly overviewing the knowledge representation.

4.1. Knowledge Representation Overview

The knowledge representations of the *Design Planning Engine* (DPE) are divided into three classes. First, there is knowledge about the design itself, which is partly contained in the DDS as described in Chapter 3, and partly in the form of high-level assertions about DDS data. Second, there is knowledge about generic hardware types and design operations; this knowledge is

contained in a network of *frames*. Third, there is knowledge about planning design activities, which is expressed in a set of *planning rules*. We will now discuss each of these in outline, following which we will discuss DPE's structure and function.

The DDS is presented in Chapter 3. The high-level representation of design entities used by DPE for planning supplements the DDS. The reason a high-level representation is used is that DDS representations are too large and detailed to be used as data sets for practical reasoning systems. The high-level representation is based on *assertions*. Each assertion is a statement about a design; for example, the assertion (`dfg $main $main-df`) states that the dataflow graph (`dfg`) of a component `$main` is named `$main-df`. This assertion essentially duplicates a node reference in the DDS. Other DDS-like assertions might make constraints on the design. Assertions that have no direct counterpart in the DDS are also possible, e.g. an assertion to the effect that a design is pipelined. Such information is implicit in the DDS description, but is not always easy to extract. A full description of the assertions used by DPE is given in Chapter 7.

The design process knowledge of DPE is organized into a semantic net of frames. Frames are divided into two classes. First, there are *hardware* frames that contain completeness-checking rules, estimators appropriate to the hardware, and pointers to relevant task frames. Second, there are *task* frames that describe design operators. These contain knowledge about the preconditions of the operator, the results of running the operator (i.e. its postconditions), rules for the invocation of program code, and estimators that can calculate the advisability of applying the operator. The frame network can be used for other things besides planning, including direct automatic execution, interactive execution, precondition checking, design process history generation, and advice generation. Also contained in the frame set is a list of

criterial quantities, which are scalar measures by which designs are to be judged. For example, for VLSI designs a criterial quantity is area.

The planning knowledge of DPE is organized in a set of *planning rules*. This set of rules is divided into several subsets, as described in Chapter 6. These rules are independent of VLSI design; all domain-dependent information is in the frame set. The planning rules determine the planning and execution behavior of DPE. For example, a single rule controls the choice of planning vs. immediate execution; another rule specifies depth-first search; and another specifies the selection of the operator with the highest advisability. Note that changing these rules would radically change the behavior of DPE, for example from a planning system to a system with immediate execution, or to an interactive advisor and design history recorder.

It is important to note that the mechanisms and strategies described here could be used in a direct execution environment, where the system chooses a task and executes it immediately. Likewise, the same framework could support an interactive system where the user made or assisted in making tool selections. The prototype software has been designed to demonstrate the most sophisticated mode of operation, that of automatic planning. A production-quality system could easily have a mixed planning and execution strategy, and allow user interaction, without changing the basic system structure described here.[9] However, the reader is reminded that many design tasks are computationally expensive; hence the potential efficiencies of the planning approach make it desirable even for moderately sized problems.

---

[9]One rule change, as noted above, would allow the system to execute tasks immediately rather than plan them.

4.2. The Design Planning Engine

The architecture of DPE is shown in Fig. 4-1. It is designed to oversee a suite of knowledge-based and hard-coded design activities. DPE constructs sequences of design activities which are not explicitly coded into the system, but are constructed in response to the needs of a particular set of specifications and constraints. This software allows us complete flexibility in the choice of tool orderings.

The major data structures of DPE are shown as circles. They are the planning knowledge and domain knowledge described above; the *tool set*, which is the executable code associated with task frames; the *plan*, which is alternatively a schedule for future tool invocations or a record of past invocations; and the DDS in which design data is stored. The two major software modules are the *planning engine* and the *execution engine*; they are responsible for applying and interpreting the knowledge and code on the left side of the figure to the specific problems on the right side of the figure.

DPE builds a *design plan* by concatenating members of a set of analysis and synthesis operators, including operators for clocking scheme synthesis, component selection, critical path finding, and area estimation. The planner uses knowledge-based heuristics in order to guide its choice of operators where more than one option is possible.

4.3. Definition of Planning

Planning [Sacerdoti 77, Stefik 80] is an activity whereby the operation of a system is simulated in order to economize search or avoid irrecoverable errors. There are two spaces that must be considered in a discussion of planning: a *planning* space and an *execution* space. The execution space is populated with *operators* (for example, design tasks such as hardware

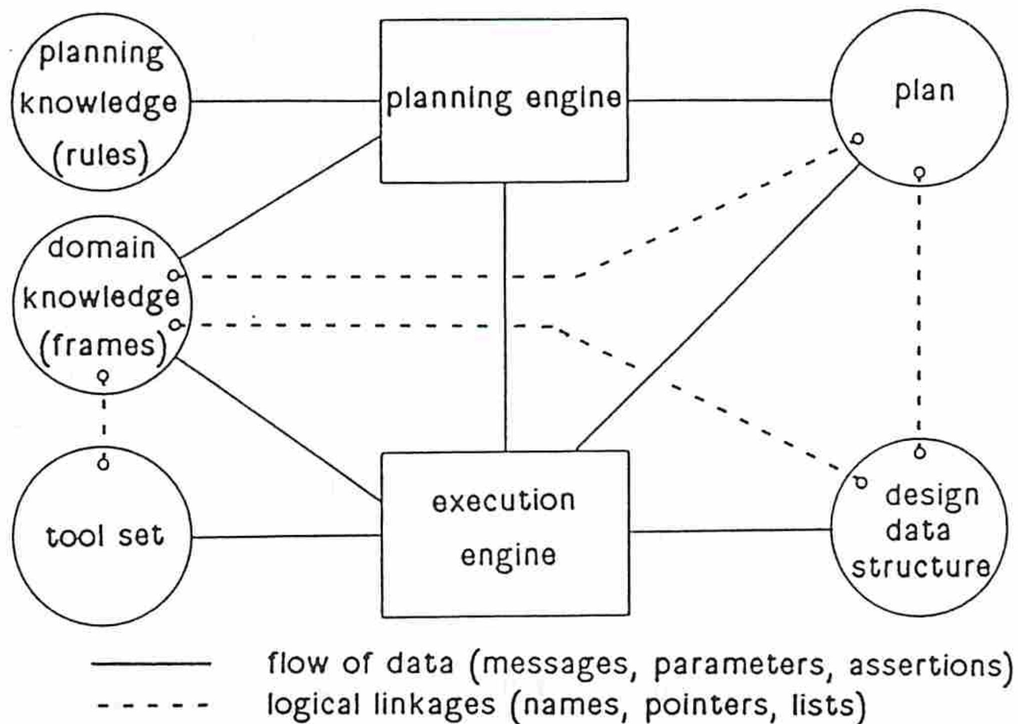


Figure 4-1: Architecture of the DPE program.

allocation, microprogram generation, and dataflow graph optimization), and *states* (graphs and symbols that describe design states, e.g. schematics, dataflow graphs, timing and control graphs, and physical layouts).

The planning space is populated with abstract models of execution space operators and states. States in the planning space are collections of assertions about designs. An example of an assertion is the LISP structure `(exist`

`PLA-Layout it)`, which states that a PLA layout of some circuit named `it` exists.

Abstract representations of operators are characterized by *preconditions* and *postconditions*; the preconditions express the conditions under which the operator is applicable, and the postconditions express or enforce the conditions that will pertain after the operation has been applied. Preconditions are represented by sets of assertions that must be matched for the operator to be considered applicable, and postconditions by either sets of assertions that are added and deleted by the application of the operator, or by rule sets that can be used to model more complex operator effects.

A collection of plans is represented by a *plan graph*, a directed tree whose nodes are abstract design states and whose arcs are abstract operators. The root node $\boldsymbol{R}$ represents the *initial state* of the design. A traversal from the root to some leaf $\boldsymbol{L}$ represents a single plan, i.e. a sequence of operators and states beginning at $\boldsymbol{R}$ and ending at $\boldsymbol{L}$.

For example, two ways to implement combinational logic are classical two-level Boolean minimization followed by standard-cell layout, and PLA minimization and synthesis. The precondition for both is are that logic equations exist; the postconditions of both are that logic equations exist, and that layout also exists. This is shown in Fig. 4-2.

4.4. Planner Operation

This section gives an overview of planner operation. The next section describes DPE's operation in more detail. In Chapter 6, the implementation details are given.

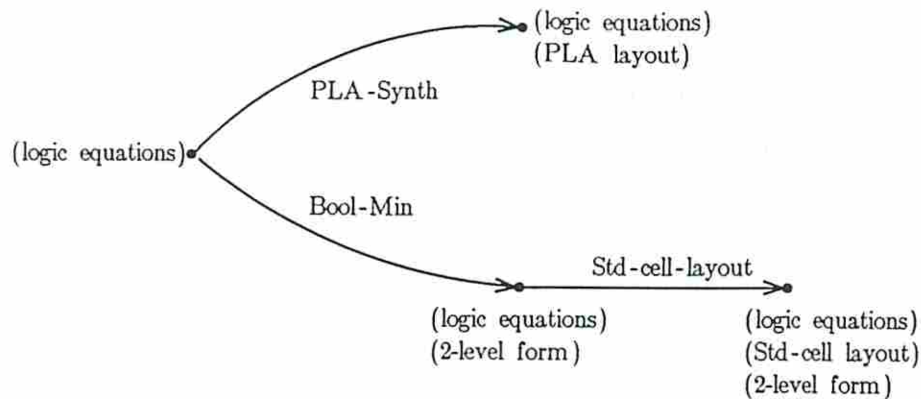


Figure 4-2: Illustration of planning.

4.4.1. Global Strategy

The global planning strategy implemented by the prototype DPE is specified by a set of domain-independent planning rules. As such it is very easily modified: new rule sets could result in radically different behavior on the part of DPE. The current set of rules expresses a certain overall strategy that will now be described.

The current rule set tries to force convergence toward an acceptable plan by means of tentative, computationally cheap explorations of the design space. These explorations take place almost entirely in the planning space; hence only abstract models of design alternatives are explored. These explorations are an attempt to eliminate blatantly bad alternatives from consideration before the more expensive execution space operators are applied. For example, if the overall task was to produce a fast digital filter, the use of a serial non-pipelined implementation, while possible, is obviously bad. The strategy tries to eliminate such alternatives with as little effort as possible.

4.4.2. Inputs to the Planner

Inputs to the planner are in two files. The first file contains the specification for the desired system in terms of a DDS construct; the second contains high-level information about the specification. Of the DDS information, only the dataflow graph is actually used by the prototype. The high-level specification contains, at a minimum, the name of the target, the name of its dataflow, and the constraints on the target's area, cycle time, power and time to design. Note that the high-level specification is a list of assertions, and that it is used as the root state of the design plan. This means, among other things, that arbitrary assertions can be made about the root state, e.g. that a module library has been selected, the name of the library, and average propagation delays associated with the library. This information is added during the planning process if it is not present in the specification state. In fact, if a complete design state were input, DPE would detect the fact and hence would not construct any plan at all.

4.4.3. Exploration

The planner starts by randomly choosing a dimension of measurement. In the current digital/VLSI frame set, the default dimensions are speed, power, and area. The chosen dimension is assumed by the planner to be of maximum importance while all the others are assumed to be of no importance. A plan is constructed on that basis. In effect, the planner is trying to construct a plan leading to the fastest (or lowest power, or smallest) design it is capable of constructing for the given specification, while ignoring all other constraints and objectives. When that plan is complete, an estimate of the criterial measures of the terminal design is calculated on the basis of planning space facts, i.e. the assertions contained in the terminal state of the plan. The planner makes no attempt to execute this plan; instead it merely

performs the planning space estimation.[10] The planner then chooses another dimension, for example area. It constructs a wholly new plan with area assumed to be of maximum importance and all the other dimensions assumed to be of no importance. The result is a new plan, whose final state's properties are estimated.

When the planner has built one of these single-objective plans for each of the dimensions of interest, it then attempts to calculate the relative importance of the various dimensions. The first two examples in Chapter 7 show such single-objective plans. This is done by comparing the estimated results to constraints given in the original specification. These importance measures have a strong effect on the way in which plans are constructed, through the heuristic that selects operators.

The planner then constructs a new plan, using the calculated importances. The properties of the final state of the new plan are then added to the collection of data points and the importance calculation process is repeated. The details of the interpolation algorithm, by which importances are calculated, are given in Section 4.5.

The planner repeats this plan-estimate-interpolate cycle until two identical plans have been constructed, at which point it executes the duplicated plan. This plan, when executed, produces an execution space design whose properties can be estimated much more accurately than those of planning space designs. However, the estimates take the same form and can be used in the same way to cyclically plan, execute, estimate, interpolate, and replan until an acceptable design is achieved.

---

[10]These estimates are somewhat crude, and are used to bound the design space produced by each design strategy, rather than to pinpoint a particular point in the design space. They should have certain properties, which we will discuss later.

In effect, the strategy implemented by the current rule set attempts to remove bad alternatives from consideration as quickly as possible, i.e. by using computationally cheap algorithms, before using more expensive algorithms. Consider the following hierarchy of ways to eliminate operators from consideration.

1. Only "relevant" operators are considered by the planner, because the knowledge base is indexed by the contents of plan states. For example, an assertion that a state represented a pipelined design would cause the knowledge base to be searched for operators "related to" pipelined designs.

2. Operators whose preconditions are not met are excluded from the plan by matching operator preconditions against plan state contents.

3. Operators whose preconditions are met, but which are estimated to be inferior, are excluded by the evaluation of an operator advisability heuristic. Advisability estimators, which are used to estimate relative advisabilities of operators, are described in detail in Chapters 6 and 7.

4. Exploratory plans are constructed but not necessarily executed. Their leaf states are estimated, using state estimators found in the relevant hardware description frame.

4.4.4. Planner Output

The planner's output is a set of plans. For most purposes, only the last plan produced need be considered; the others are purely exploratory. Each plan is a sequence of states, linked together by operators. The states are lists of assertions, like the root state but augmented with assertions generated during the planning process. Attached to the states of the plan are the importance numbers that directed the choice of operators in the sequence. Estimated properties of the terminal design states, and estimated design times (i.e. machine time to execute the plan) are also attached to the terminal

states. The operators that link the states of the plan represent invocations of design tools. Each consists primarily of a pointer to a task frame. The sequence can be executed by applying the code referenced in task frames to DDS entities. This is done by executing a set of rules attached to the task frame, or by directly invoking a LISP procedure.

An example of a plan constructed by DPE will now be given. The initial state passed to DPE consists of a DDS dataflow graph with 24 nodes and 47 values, and a high-level specification containing constraints on area, power, cycle time, and time to design, as well as the names of the target and its dataflow graph. This example is a direct translation of a test run's output into English. The plan DPE produces consists of the following steps:

1. *dataflow analysis*, in which the dataflow graph is analyzed and its critical path extracted,

2. *module library selection*, in which a set of primitive components is selected,

3. *module allocation*, in which dataflow operations are assigned to logical-structure modules and time slots,

4. *controller style selection*, in which a controller style (e.g. PLA, microprogrammed) is selected,

5. *structure graph construction*, in which the allocated modules are connected together by multiplexers and buses,

6. *control table construction*, in which control vectors are bound to time slots, and finally

7. *controller construction*, during which the controller is constructed in accordance with the decisions of steps 4 and 6.

Note that this plan was constructed from an unordered set of operators, with between five and nine applicable operators available at each step. The actual sequence of operators was determined at run time by DPE.

4.5. Operation: What DPE Does

This section details the operation of DPE. The root state ***R*** is DPE's starting point. ***R*** is assumed to be the planning-space representation of an incomplete design, which DPE must somehow complete. The input specifications form the root state, as described above.

DPE first determines whether ***R*** is so detailed as to be a complete design. If it is DPE simply exits; otherwise it attempts to construct a plan to complete the design.

The first step taken by DPE is the establishment of importance measures. This is initially done by random selection to bound the search space, and later done by interpolation. Let us assume that time is assumed to be of maximum importance, and all other measures are set to minimum importance.

The set of relevant operators is found in the knowledge base. This is done by using key phrases, e.g. "generic RTL hardware", found in the planning-space model of the root state.[11] These key phrases are used to search the knowledge base: all operators relevant to "generic RTL hardware" will be returned. The key is not taken from the DDS because determining the level of abstraction of a DDS entity is still a research issue. If more than one key phrase is present, the set of relevant operators will be larger.

The members of the set of relevant operators are then checked individually to see if their preconditions are met in the state to be transformed. Every operator whose precondition is met represents something that could be done in that state, but not necessarily something that should be

---

[11] DPE's current knowledge base is defined in terms of RTL constructs and operators.

done. For example, two candidates "partition dataflow" and "allocate hardware" might both be applicable to a design state that contained a dataflow graph, but it is not clear that either is the best choice in every situation where a dataflow graph is present.

The system therefore applies a heuristic choice function, also called an *operator estimator*, and described in Chapters 5, 6, and 7, to determine the "most advisable" operator. The function combines information from three sources:

1. the operator whose advisability is currently being ascertained,
2. the state of the design to which the operator would be applied, and
3. the measures of importance that pertain to the current plan as a whole.

The technique that generates advisability numbers out of information taken from arbitrary operators, design states, and importances is a key part of DPE. The technique uses an inference engine and a set of rules to generate an advisability number. The rules are associated with the operator being estimated; they are taken from the task frame and passed to the inference engine after precondition matching. The rules may deduce new assertions, or they may perform calculations of numeric quantities, or both. Only the rules that fire at estimation time participate in the estimation process; this means that the estimate can be tuned to a wide variety of situations by using a sufficiently sophisticated rule set. For example, if the implementation technology is known to be TTL, then a set of deductions about basic area, speed, and power consumption constants can be made. These deductions can then be used to make further calculations. Because a general rule interpreter is used to generate advisabilities, any algorithm can be built into an operator's advisability estimator.

A typical advisability estimation rule uses importance measures to generate a weighted sum. For example, suppose that the importance of speed is 0.4 and that of area is 0.9. For an operator that tended to give low-cost, low-speed implementations, the weights might be 5 for area, and 2 for speed. For another operator, that tended to give high-speed, high-cost implementations, the weights might be 2 for area and 5 for speed. Hence the advisability of the first operator would be $2 \cdot 0.4 + 5 \cdot 0.9$, or 5.3; while the advisability of the other operator would be 3.8. Thus in this situation the slower implementation would be favored. This topic is discussed in more detail in Chapters 6 and 7; note that the example given here is greatly simplified.

Once the advisabilities of all the operators whose preconditions are met have been determined, the operator with the greatest advisability is added to the plan. This results in the creation of a new state, which represents the design created by applying the operator to the old leaf state. The contents of the new state are defined by the postcondition of the operator.

The new leaf state is then tested for completeness. This is done by searching the domain knowledge base for a hardware description frame that contains a set of rules that define the completeness of the kind of hardware in question. If the state in question represents a complete design, DPE will either execute the plan or construct another plan starting at the root. If the plan has been repeated, it is highly probable that the system has either converged or that it is oscillating.

The current strategy for detecting convergence is to keep planning and replanning until two identical plans have been constructed. The interpretation of this situation is that it is probable that DPE is either unable to improve on

the duplicated plan, or it is oscillating between two or more plans. DPE does not distinguish between these cases; instead, it executes the duplicated plan.

Let us suppose that another plan is to be constructed. This begins with the adjustment of the importance measures, either by selection of a boundary for exploration, or by interpolation. DPE explores the boundaries of all of the criterial quantities (listed, as mentioned above, in the frame set) before it starts interpolating, and interpolates thereafter. A graphical representation of the process for two criterial quantities, area and time, is given in Fig. 4-3. The interpolation strategy is as follows.

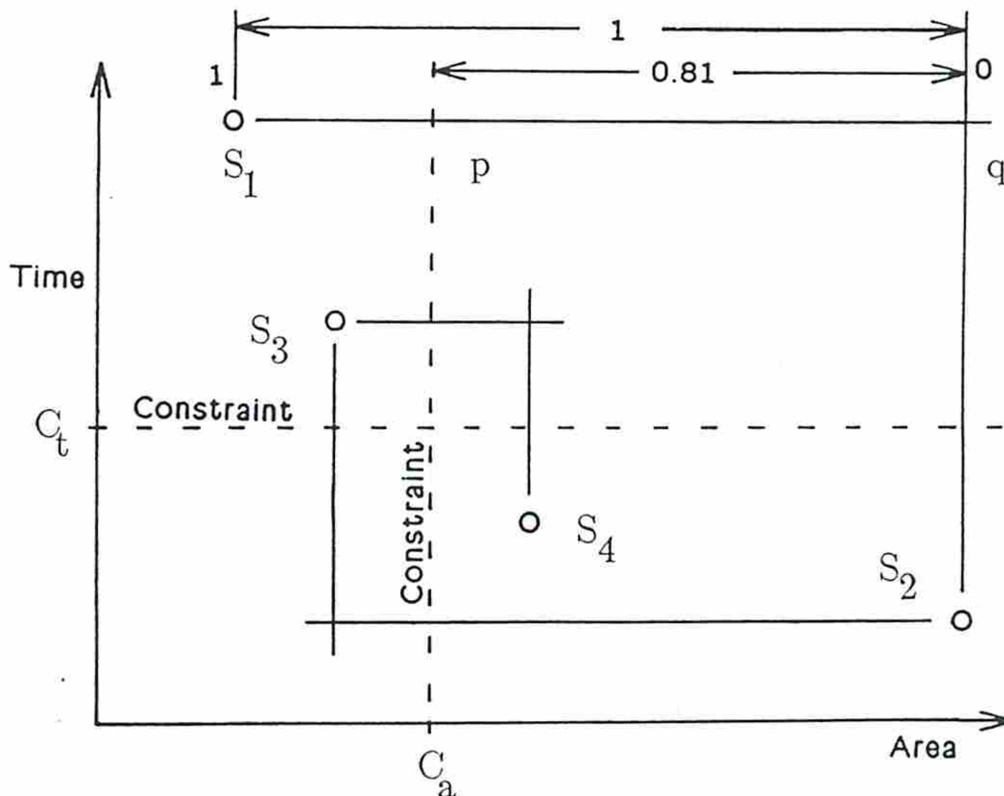


Figure 4-3: Interpolation technique used by DPE.

1. The criterial quantities of the leaf states are estimated. In the example, these properties are area and time. The result of this process is the plotting of points s_1 and s_2. The point s_1 corresponds to the leaf state for which the importance of area was 1, expressed by $I_a(s_1) = 1$, and all other properties were regarded

as unimportant, e.g. time, i.e. $I_t(s_1) = 0$. Point s_2 is the case where time had importance 1, i.e. maximum importance.

2. The point q corresponds to minimum importance of both measures. Construct the line s_1q. It intersects an area constraint (which was part of the high-level input state) C_a at point p. The new area importance I_a is

$$I_a = I_a(s_2) + (I_a(s_1) - I_a(q)) \frac{|pq|}{|s_1q|}$$

3. The new importance of time I_t is calculated by interpolating on C_t and s_2q in a similar way.

4. These importances are then used to construct a new plan, and when the new plan is complete, steps 1-4 are repeated.

Three things should be noted here. First, the example shown was of a two-dimensional interpolation, but in fact DPE handles multiple dimensions. These dimensions are all loaded into DPE as part of the frame set, except for the dimension of *design time*, which is presumed to be of concern regardless of the application domain.

Second, the points s_1 and s_2 are not necessarily the only candidates for interpolation. For example, suppose that the new importances I_a and I_t were used to construct a new plan, whose final state was s_3. The next interpolation would then be between s_2 and s_3, under the rationale that s_3 is closer to the constraint than s_1, and hence more likely to be informative if the design space is nonlinear; and also that s_2 and s_3 "straddle" the worst acceptable design, i.e. one is greater than the constraint, and the other is less. The determination of the "most reliable" pair of points is done by a heuristic. Basically the heuristic currently used (and expressed in the planner's rule set) is to select noninferior points, and from that set choose pairs of points that straddle constraints as closely as possible. The justification for this heuristic is an assumption that the design space is "smoother" for small parameter

changes than for large ones. For example, a design s_4 that is similar to a design s_3 is assumed to be a better predictor of s_3's attributes than a design s_2 which is radically different from both s_3 and s_4.

Third, the strategy of interpolation on constraining values will not work properly if one of the points is within the acceptable region; under those circumstances another strategy is needed. The strategy used by DPE in this situation is to tighten all importances uniformly, i.e. to try to force the next design produced to be closer to the origin. The assumption underlying this heuristic is that in most of the cases of interest, constraints will be difficult to satisfy, so that the strategy to be taken if constraints are met is of small importance when compared to the strategy to be used if constraints are not met.

Following the generation of a new set of importances, the planning process can be repeated from the root state. Ultimately the plans should converge, as suggested by the sequence s_1-s_2-s_3; there is no guarantee that they will converge on a single plan, however, or that the convergence will be toward an acceptable design. Experimental results demonstrate convergence; examples of test runs are given in Chapter 7.

Note that one important case in which DPE will fail is where the constraints are simply too restrictive. In this situation, DPE will either converge to a plan that probably will not meet constraints, or it will oscillate between two or more plans that probably will not meet constraints. DPE's strategy in such situations is the same, i.e. to execute the first repeated plan, under the rationale that it is the "best it can do". Because DPE is incomplete in that it does not execute plans, as discussed in Chapters 6 and 7, it stops at this point. In a complete system, this problem will still be present, because it will always be possible to set up overly restrictive constraints. The strategy

that should be taken in such situations is an open research issue. For example, one approach would be to input weighting factors, so that if a choice of unmet constraints existed, the system could make the choice. Another approach would be to query the user; yet another would be to ask the user to relax one or more constraints.

The actions taken during planning are discussed in Ch. 6, which covers the implementation of DPE and its internal functioning. DPE is based on a formal model of the design process. Chapter 5 describes this model. Chapter 7 presents example plans, traces of DPE's execution, and detailed examples of state, rule, and frame representations.

Chapter 5
Modeling the Design Process

In this chapter a formal treatment of the design process based on the first-order predicate calculus is presented. This model underlies the design planning engine. While the use of the model in this chapter focuses on the planning engine, the basic model can be used to model direct execution and user interaction as well. There are three levels of modeling.

1. The execution space is populated with designs and design algorithms. These are, respectively, models of concrete hardware implementations, and executables that transform such models.

2. The planning space is a model of design processes. The design processes involve programs operating on designs. The planning space therefore contains models of programs and models of designs.

3. The analysis of the planning and execution spaces, and the interactions that can take place between them, is based on mathematical models of the two spaces and of the relationship between them.

This chapter concentrates mainly on the second model, i.e. the planning space, which models design processes. When the behavior of DPE is to be discussed, however, it will be necessary to use the third model as well. This chapter will therefore introduce notation for both the second and the third models[12].

[12]The DDS can be regarded as a notation for the first model.

This chapter is organized as follows. First, we introduce notation, then we discuss operators and estimators. We use the model to prove some properties of DPE's technique, which leads to algorithms for bounding the worst-case complexity of design planning to meet single or multiple constraints. We then discuss some of the practical aspects of DPE in the light of the model.

5.1. Notation

Tables 5-1 and 5-2 summarize the notation used in this chapter. Table 5-1 gives notation for states of a design and operators that transform designs. It is divided into two columns. In the left column is the notation, and in the right column is the kind of item denoted.

Planning space states s, t are sets of assertions. The assertions describe (1) information about a partially complete or complete design, and (2) historical information about the design process. In DPE, for example, a planning space state might contain references to a schematic diagram, a dataflow graph, and a list of constraints. References are of the form (*attribute*, *variable*, *value*), for example (`dataflow-graph $main $mains-dataflow`). This example can be translated as meaning that some item `$main` has a `dataflow-graph` named `$mains-dataflow`. Other assertions are similar, and are discussed in more detail in Chapter 7. For the purposes of this chapter, these assertions can be regarded as predicates whose variables are bound.

An *execution space state* is the execution-space counterpart of a planning space state. We will denote execution space states by the letters S and T. In this chapter we will be almost entirely concerned with planning space states. Note that although historical information can be included in a

Notation	Item
$s, t, \ldots$	states (in the planning space)
$S, T, \ldots$	states (in the execution space)
$\boldsymbol{O}$	set of all operators
O_o	*o*th element of $\boldsymbol{O}$
$\boldsymbol{Pre}_o$	precondition of *o*th element of $\boldsymbol{O}$
$\boldsymbol{Add}_o$	add set of *o*th element of $\boldsymbol{O}$
$\boldsymbol{Del}_o$	delete set of *o*th element of $\boldsymbol{O}$
$\boldsymbol{D}$	set of all dimensions
D_d	*d*th element of $\boldsymbol{D}$
$\boldsymbol{C}$	set of all constraints
C_d	*d*th element of $\boldsymbol{C}$
$R_d(s)$	state estimator of *d*th dimension, with argument state s
$E_{o,d}(s)$	operator estimator for *d*th dimension and *o*th operator, with argument state s

Table 5-1: Notation for states and operators.

state, a single state does not necessarily fully or uniquely describe the sequence of operations by which it was constructed. This information is contained in a *plan*, defined below.

A *root state* is an initial state, e.g. the state of the design when it is passed to DPE for transformation into an implementation. A *leaf state* is a state which is complete with respect to some set of criteria. At the RT level, for example, completeness criteria include the presence of a set of logical

modules, an interconnect graph of carriers, a particular module called a controller, a timing graph forming an abstract microprogram, and so on.

An *operator* is a five-tuple (***F, E, Pre, Add, Del***). The elements of the tuple are as follows. ***F*** is the function code of the operator. That is, ***F*** is an executable that implements the operator in the execution space. For example, the operator `Sehwa` contains as its ***F*** the LISP program Sehwa. ***E*** is a set of *operator estimators*, defined below. ***Pre*** is the *precondition* of the operator. It is a sentence of the first-order predicate calculus, i.e. it is a set of predicates joined by the symbols of Table 5-2. ***Add*** and ***Del*** taken together are the *postcondition* of the operator, as expressed in the STRIPS notation [Fikes 71]. ***Add*** and ***Del*** are partial sentences of the first-order predicate calculus. Preconditions and postconditions are discussed below.

The set $\boldsymbol{O}$ is the set of all operators available to the system. Where it is desirable to differentiate between operators, we will use the subscript o, $1 \leq o \leq |O|$. The subscript o is used as an index into $\boldsymbol{O}$. Hence a generic member of $\boldsymbol{O}$ could be called $\boldsymbol{O}_o$, or two different members by $\boldsymbol{O}_{o_1}$ and $\boldsymbol{O}_{o_2}$. For example, the ***Add*** set of the operator $\boldsymbol{O}_o$ is denoted by $\boldsymbol{Add}_o$.

A *dimension* is a scalar representing some quantitative measure of a design's properties. Examples of dimensions are area, cycle time, and power dissipation.

The set $\boldsymbol{D}$ is the set of all dimensions. Individual dimensions are indexed by the subscript d, $1 \leq d \leq |D|$. $\boldsymbol{D}_d$ therefore represents a single dimension, e.g. area.

A *constraint* is a scalar representing the maximum value some dimension may take. For example, the area of a design may be bounded by

the constraint 200,000 mil^2. The set C is the set of all constraints. A member of C is indexed by d, to indicate the dimension of the constraint. Hence if D_1 is area, then C_1 is the constraint on area. Note that constraints are all expressed as upper bounds; where necessary, such properties as "speed" are inverted to give "time".

A *state estimator* $R_d(s)$ is a function that maps from a state s to a scalar dimension D_d. If D_1 is area, then $R_1(s)$ is an area estimator. A state estimator can potentially be applied to an execution space state S as well as to a planning space state s. State estimators are therefore polymorphic in that they can be applied to arguments of different types.

An *operator estimator* $E_{o,d}(s)$ is a function that maps from a state s to the dimension D_d. Note that it is indexed by operator as well as by dimension; that is, operator estimators are attached to operators. Recall that an operator is a tuple containing a set of estimators E; these estimators are operator estimators, and there are potentially $|D|$ of them. Operator estimators are also polymorphic.

5.1.1. Preconditions and Posconditions

Preconditions and postconditions of operators are expressed in the notation of the first-order predicate calculus. A summary of the relevant notation is given in Table 5-2. Note that the table is organized in levels of operator precedence; hence Boolean NOT takes precedence over all other operators, etc.

A *predicate* is a mapping of a set of values onto the set {true, false}. For example, if a design *A* has the dataflow graph *B*, the predicate *has-dataflow-graph(A,B)* is true.

$\sim$	NOT	highest precedence
$\wedge$	AND	second level of precedence
$\vee$	OR	
$\rightarrow$	Implication	third level of precedence
$\leftrightarrow$	If and only if	
$\equiv$	Equivalence	fourth level of precedence
$\exists$	Existential Quantifier	fifth level of precedence
$\forall$	Universal Quantifier	

Table 5-2: Notation for Predicate Calculus.

A state, for the purposes of this chapter, is modeled as a conjunction of assertions, i.e. a set of unquantified predicates in which there are no unbound variables, joined by the operator $\wedge$. For example, the assertions *target-component-name(D1)* and *has-microprogrammed-control(D1)* might be made of a state s.

Semantically, a planning state predicate can either represent a statement about a DDS component or class of components, or it can represent a statement about the history of the design process. The predicates that constitute a planning-space state are assumed to be **true** of the corresponding execution-space state.

5.1.2. Operators

The executable part F of an operator is a program. Exact and complete specifications of what such programs do and when they are applicable in terms of execution space constructs are potentially lengthy and complex; hence it is difficult to construct and verify such descriptions. The modeling

constructs ***Pre***, ***Add***, and ***Del*** are far simpler, being based on predicates over planning space states.

We attach the subscript *o* to preconditions and postconditions, so that the preconditions and postconditions of different operators can be distinguished.

The *precondition* $\boldsymbol{Pre}_o$ of $\boldsymbol{O}_o$ is a universally and existentially quantified sentence of the first-order predicate calculus. The interpretation of such a precondition is that the operator $\boldsymbol{O}_o$ is applicable in a state ***s*** iff the precondition $\boldsymbol{Pre}_o$ evaluates to true in ***s***.

Note that the precondition may contain variables. For example, the precondition of an operator may state that some controller *control* must be present. In that case, the actual controller may be named something like **FSM_4**. The *variable binding* of *control* to **FSM_4** is constructed through unification (a process described in Chapter 6), and used to construct assertions in ***Post*** that contain references to *control*, i.e. by substituting **FSM_4** for *control* wherever it occurs.

We will use the symbol ***s+*** to denote the new state resulting from the application of an operator $\boldsymbol{O}_o$ to a state ***s***. The state ***s+*** is constructed by copying the assertions of the state ***s***, and then making additions to and deletions from the state so created.

The *add set* $\boldsymbol{Add}_o$ of $\boldsymbol{O}_o$ is a sentence of the first-order calculus. When ***s+*** is being constructed, the variable bindings under which $\boldsymbol{Pre}_o$ was true are substituted into the predicates of $\boldsymbol{Add}_o$, and the resulting bound predicates are asserted in ***s+***.

The *delete set* is similar to the add set, with the difference being that the bound predicates of the delete set are asserted to be false in $s+$. This is done by simply removing the bound assertions from $s+$, the implicit assumption being that if an assertion is not present in a state, the assertion is false.

Any assertion of s that is not mentioned in either Add_o or Del_o of O_o is simply copied from s to $s+$; this is the "STRIPS assumption" [Fikes 71] under which all planning-space effects of an operator are explicitly stated in its description, and an operator is assumed, for the purposes of planning, to have no other effects.

We will denote the *application* of an operator by the symbol $\circ$. The application is performed by copying the assertions of a state s, and then adding and deleting assertions to form a new state $s+$, as described above. We will use the syntax $s \circ O_o \rightarrow s+$ to denote the application of O_o to the state s to produce the state $s+$. We can use the same symbol $\circ$ to denote application of an executable code module F to an execution-space state S to produce a new execution space state $S+$.

5.2. Monotonicity

We define a special class of *monotonic* estimators to be those estimators that preserve orderings but that are not necessarily accurate.

A *monotonic state estimator* is an estimator such that $R_d(s) > R_d(t) \leftrightarrow R_d(S) > R_d(T)$, where S and T are the execution-space counterparts of s and t respectively, and R_d is polymorphic across both planning and execution spaces. It is assumed that R_d is accurate when applied to an execution-space state, but that it may not be when applied to a planning-space state. The

assumption of monotonicity puts a bound on the inaccuracy of the planning-space estimation.

This is illustrated in Fig. 5-1, in which the solid line represents a monotonic estimator and the dotted line represents a nonmonotonic estimator, because its slope changes sign.

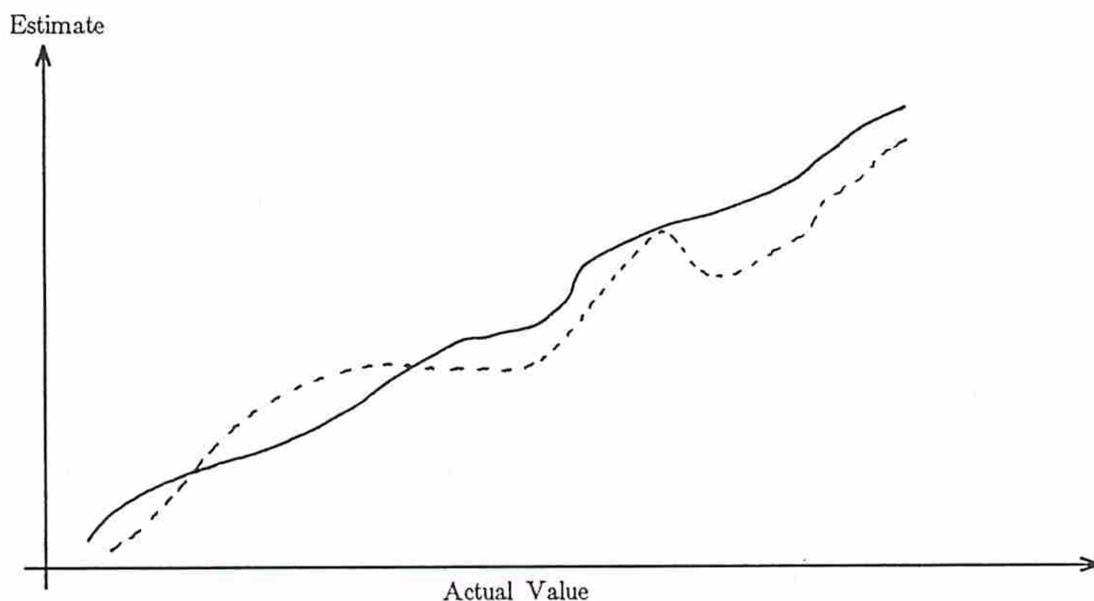


Figure 5-1: Monotonic and nonmonotonic estimators.

A *monotonic set* of operator estimators is a set of operator estimators $\{E_{o,d}(s)\}$ such that $E_{o_i,d}(s) > E_{o_j,d}(s) \leftrightarrow \forall\ s \in \mathrm{DESC}[O_{o_i}],\ t \in \mathrm{DESC}[O_{o_j}]$: $\{\ R_d(s) > R_d(t)\ \}$, where $\mathrm{DESC}[O_o]$ is the set of all leaf states descended from the state $s \circ O_o$, and R_d is a monotonic state estimator.

In English, a monotonic state estimator is one that reliably ranks states by some dimension. A monotonic set of operator estimators is one whose members reliably predict the *ranking* of outcomes of the application of the operators that correspond to the estimators. Note that the outcomes are defined in terms of *leaf* states; that is, an operator estimator is presumed to make commitments concerning the results of all operators between the

operator being estimated and the leaf state. This assumption is not as improbable as it might seem; high-level decisions must often be made under the assumption that the remainder of the design process will proceed normally. This assumption that "other things will be equal" allows intermediate decisions to be ranked. For example, suppose two dataflow nodes that produce identically equal result values are under consideration for merging. In order to make the decision to merge or not to merge, one must assume that either (1) a dataflow graph with fewer nodes will have a more efficient structural implementation, because of the saved node, or (2) the cost of saving and routing the value, if it is singly generated, outweighs the cost of the extra node. In either case, assumptions about the performances of the routing, module binding, and allocation processes have been made.

Recall the polymorphism of estimators, i.e. their ability to be applied in either the planning or the execution space. We can extend the idea of monotonicity to both planning and execution spaces by applying this polymorphism. That is, if we replace the planning-space states s and t with their execution-space counterparts S and T, we have defined monotonicity of state and operator estimators in the execution space. The following arguments therefore apply to both spaces.

Note that monotonic sets of operator estimators and monotonic state estimators are a theoretical construct; they may not be constructible in practice. They represent an ideal to be striven for. Two general observations can be used to buttress the assumption that monotonic estimation is possible. First, an estimator or estimator set with small nonmonotonicities can be treated as if it were monotonic to within some tolerance. Suppose, for example, a state estimator was guaranteed to rank any two designs monotonically as long as there was at least a 10% difference between them. Such an estimator would be useful as long as great precision was not the goal.

Second, no estimator of any kind can be expected to make reliable predictions if bad decisions are made after estimation. For example, an area estimator cannot be expected to perform well if subsequently used placement and routing packages do not predictably produce good results. Hence while it is certainly possible to render an estimate invalid, simply by assuming bad choices downstream of the estimate, to do so violates a "good faith" assumption made when estimation was undertaken.

Some interesting and useful properties can be inferred if we assume the existence of such estimators. We will now examine the consequences of this assumption.

5.2.1. Monotonic Operator Estimator Sets

Let us suppose that there is an aggregate population of $k = |O|$ operators, and that they can be formed into sequences of length n so that all of the sequences so formed terminate at correct designs. Note that this definition of correctness allows a correct design to violate any and all constraints in $\boldsymbol{D}$. We define an *acceptable* design as one for which all scalar constraints on dimensions in $\boldsymbol{D}$ are met. Hence, for example, a design whose outputs were not those specified would be incorrect, while a design that was too slow would be unacceptable. Let us further suppose that the set $\boldsymbol{D}$ of dimensions is of bounded cardinality, and that for the operators in $\boldsymbol{O}$ the operator estimators form monotonic sets.

If we further assume that no two estimates are ever equal, then the asymptotic time complexity to either meet scalar constraints or to prove that the operator set can construct no such design, is no worse than $O(n^2 k \log^2 k)$. Figure 5-2 will serve to illustrate and clarify the arguments that follow. In Fig. 5-2, the root state r represents the initial state of the design, i.e. the

specification state. Each arc represents an operator. Each leaf node represents a terminal state, i.e. a design which is considered to be complete at the current level of abstraction.

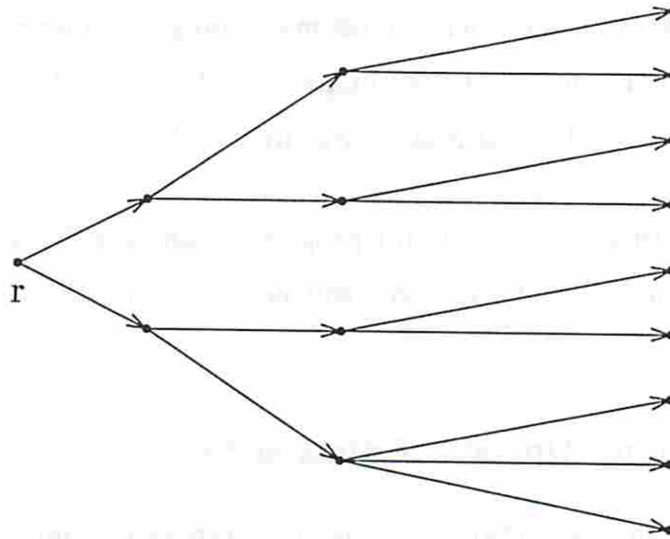


Figure 5-2: Illustration of the time complexity argument.

The proof of the foregoing is by construction. First an algorithm *Bound-Finder* (BF) will be given; this algorithm constructs the sequences that lead to the worst acceptable designs in each dimension. Then an algorithm *Blind-Path* (BP) will be given; it either produces a sequence that results in an acceptable design in all dimensions, or it proves that no such design exists. Note that in the implementation of DPE the simpler and faster interpolation heuristic is used instead of BF and BP. The interpolation heuristic is harder to analyze formally, but is is more robust in the presence of unsatisfiable constraints, and it requires less search. Thus, these algorithms are expected to provide a loose bound on the performance of DPE.

5.2.1.1. Boundary Finding Algorithm

The problem in the Boundary Finding (BF) algorithm is to find the *worst acceptable* state s_w for some dimension D_d, i.e. the state corresponding to an acceptable design such that there is no other acceptable design with a greater value of D_d, when all other members of C are ignored. The sequence of operators and states that leads to that design will be called the *boundary path* of D_d. Formally, if s_w is the worst acceptable state, and t is an arbitrary state, then

$$R_d(s_w) \leq C_d \wedge (R_d(t) > R_d(s_w) \rightarrow R_d(t) \geq C_d).$$

The BF algorithm uses binary search of the tree of possible design sequences to find the worst acceptable design in some dimension D_d.

Algorithm 5.1

1. Construct the maximal leaf state in D_d, i.e. the leaf state which has the highest possible value of D_d, by choosing the operator that maximizes D_d at each step. Call the entire sequence "UB" (upper bound). Mark the elements of the sequence (henceforth *path*) which led from r to the maximal leaf state with the tag UB. If the maximal leaf state meets C_d then return UB: there is no design that does not meet C_d.

2. Construct the minimal state in D_d, i.e. the state which has the smallest possible value of d, by choosing the operator that minimizes D_d in each step. Mark the path leading to the minimal state with the tag "LB". If the leaf of LB does not meet C_d then return nil: there is no state that meets C_d. Henceforth we will use the symbols UBL and LBL to denote the leaves of UB and LB respectively.

3. Construct the median path MED between UB and LB. This is done by following UB and LB from the root until they diverge, and choosing the arc M which most nearly falls in the center of

the ranking of operators between UB and LB at the state where they do diverge. Thereafter, the center-ranked operator at each node is chosen, until a leaf state is reached. If the choice is only between UB and LB, follow LB.

4. If MED and LB are the same, then stop. The leaf of MED is the state with the worst acceptable value of d, and MED is the boundary of $\boldsymbol{D}_d$, i.e. the path leading to $\boldsymbol{s}_w$ in $\boldsymbol{D}_d$. Return the path MED.

5. If MED meets C_d, mark MED to be the new LB. If MED does not meet C_d, mark MED to be the new UB. Go to step 3.

The total cost of the BF algorithm is found by multiplying the number of MED paths that the algorithm constructs by the cost of constructing a MED path.

Lemma 5.1: The cost of constructing one median MED at depth n is at worst O(nklog$k$).

Proof: The median MED is a chain of length n. Its construction consists of n steps. In each step, at most k operators must be ranked. This ranking consists of (1) applying the operator estimator for each of k operators, (2) sorting the estimates, and then (3) choosing the median operator. The estimation is assumed to take constant time. Hence the bottleneck here is the sort, which is of order $k$log$k$ [Aho 74]. Because this is done for each of n steps, the complexity is O(nklog$k$).

The following lemma tells how many medians must be constructed.

Lemma 5.2: At most $n$log$k$ medians MED will have to be constructed before LB=MED.

Proof: It is trivial to demonstrate that there are at most n^k leaves on

the tree searched by BF. The definition of monotonic operator estimator sets, together with step 1, ensures that BF cannot enter its main loop unless s_w is better than the initial UBL. That is because the initial leaf of UB has the largest D_d the system can construct; if any other constructible design were worse, monotonic estimation would cause that design to be the terminal of UB.

Similarly, the loop cannot be entered unless s_w is at least as bad as the initial LBL.

By the above argument, s_w must be between the initial UB and LB if the loop is ever traversed.

At every traversal of the loop, the path MED is either equal to UB, or to LB, or it lies between the two, in the sense that at the deepest common ancestor s_u of UBL and the leaf of MED, with out-arcs O_{o_1} and O_{o_2}, $O_{o_1} \in$ MED and $O_{o_2} \in$ UB, $E_{o_1,d}(s_u) < E_{o_2,d}(s_u)$, and similarly for the deepest common ancestor s_l of LBL and the leaf of MED.

At every traversal of step 3, a new MED is constructed, such that for every arc O_{o_h} at depth t on MED, O_{o_i} at t on UB, and O_{o_j} at t on LB, either the three paths coincide, or O_{o_h} coincides with O_{o_j}, or $E_{o_j,d}(s_l) < E_{o_h,d}(s_m) < E_{o_i,d}(s_u)$, where s_u is the state at depth t on UB, s_m the state at depth t on MED, and s_l the state at depth t on LB. Furthermore, because of the ranking and median-construction step, the path MED bisects the out-arcs between UB and LB where they separate, and bisects the k-ary subtree between UB and LB at every succeeding arc.[13]

---

[13]Strictly speaking this is only true in the worst case, where all k operators can be applied at every step of every plan.

Because of step 4, the loop will terminate if ever MED=LB. As long as this is not the case, step 5 ensures that the subtree between UB and LB will be halved with each loop traversal. If n and k are both finite, the loop must terminate. Furthermore, since the number of leaves in the tree is k^n, and since the number of leaves remaining between UBL and LBL is halved at each loop traversal, the maximum number of loop traversals is $\log(k^n) = n\log k$ if k is a power of 2, or it differs from $n\log k$ by a constant.

The condition under which the loop terminates is the test of step 4, i.e. MED=LB. Since UB is set equal to MED when MED fails to meet the constraint, UB and all paths above it do not meet the constraint. LB is set equal to MED only when the MED meets the constraint, so LB and all paths below it meet the constraint. Since a new MED cannot be equal to LB unless there is no other path between UB and LB, the leaf of MED is s_w at the termination of the loop.

Since the number of loop traversals is $O(n\log k)$, the time in the loop must be $O(n\log k)$ times the time in one pass. The time in one pass is $O(nk\log k)$, because the pass is dominated by the construction of MED, which has that cost by Lemma 5.1. Therefore the loop must terminate in $O(n^2k\log^2k)$ time.

5.2.1.2. The Blind Path Algorithm

Once the BF algorithm has been run for all of the $|D|$ dimensions, the blind path (BP) algorithm is guaranteed to find an acceptable design in $O(kn)$ time if one exists. We will make use of the name CN to denote the current node in the tree search process, and the name D^* to denote the set of dimensions whose boundary paths pass through CN. Note that for every CN not equal to r, D^* is potentially a different subset of D.

Algorithm 5.2

1. Set the current node (CN) equal to the root r. Let the initial D^* be the set of boundary paths through the root, i.e. $D^*=D$.

2. If CN is a leaf then stop. The path from the root to CN is the desired sequence.

3. If CN is not a leaf then it has out-arcs. Call them the set OUT(CN). Every member $O_o \in$ OUT(CN) must conform to one of the following conditions, where O_{o_d} is the arc on the boundary path of D_d having the same depth as O_o,[14], and $s_{d,t}$ is the state to which O_{o_d} is applied:

   a. $\exists\, D_d \in D^* \mid \{\, E_{o,d}(CN) > E_{o_d,d}(s_{d,t}) \,\}$. If this is the case then delete the arc corresponding to O_o and all of its descendants; it cannot lead to an acceptable solution (by Lemma 5.4, below.)

   b. $\forall\, D_d \in D^* \mid \{\, E_{o,d}(CN) < E_{o_d,d}(s_{d,t}) \,\}$. If this is the case, all descendants of O_o are acceptable (Lemma 5.3, below.) Choose a path to a leaf descended from O_o and exit the BP algorithm.

   c. $\exists\, D_d \in D^* \mid \{\, E_{o,d}(CN) = E_{o_d,d}(s_{d,t}) \,\}$. If this is the case, mark the arc corresponding to O_o **live.**

4. Once all $O_o \in$ OUT(CN) have been tested in this way, delete all arcs that are not **live.** There are at most $|D^*|$ of them, by Lemma 5.6 below. If no arcs remain, then trace backward, deleting nodes and arcs along the way, to the first node that has a **live** out-arc. If there is none, then there is no solution (Lemma 5.5): exit. If there is such a node, make it CN.

5. Choose one of the **live** out-arcs of CN. Call it LCA. Note that for some set $D+ \subseteq D^*$, $D_d \in D+$, LCA $= O_{o_d}$. Make the set $D+$ the D^* set of the node at the other end of LCA and go to 2.

---

[14] If the boundary path has no arc at the same depth as O_o, then use the last arc of the boundary.

This algorithm is guaranteed to find an acceptable solution if one exists. We will now prove that this takes at most $|D|$ passes involving at most O(n) arc traversals each.

The first lemma demonstrates that a CN having out-arcs O_o such that O_o is estimated to fall below the boundary paths for all dimensions, has no unacceptable descendants.

Lemma 5.3: $\exists\ O_o\ \forall\ D_d \in D^* \mid \{\ E_{o,d}(CN) < E_{o_d,d}(s_{d,t})\ \}$ then all descendants of O_o must be acceptable solutions.

Proof: (*by contradiction*). Assume that there is a dimension D_d and a leaf state s descended from O_o such that s violates C_d. That is, $\exists\ D_d,\ s \in$ DESC(O_o) $\mid \{\ R_d(s) > C_d\ \}$. If $E_{o,d}(CN) < E_{o_d,d}(s_{d,t})$ then by the definition of monotonicity, $\forall\ s \in$ DESC(O_o), $t \in$ DESC(O_{o_d}), $R_d(t) > R_d(s)$. Because of the construction of the bounding arc O_{o_d}, $R_d(t) \leq C_d$; hence there is a contradiction. Therefore Lemma 5.3 must hold.

The next lemma proves that no acceptable solution can descend from CN if each of its out-arcs lies above the boundary path in some dimension.

Lemma 5.4: If $\forall\ O_o \in$ OUT(CN) $\exists\ D_d \in D \mid \{\ E_{o,d}(\text{CN}) > E_{o_d,d}(s_{d,t})\ \}$, where $s_{d,t}$ is the state on the boundary of D_d at the same depth as CN, then there is no acceptable solution descended from CN.

Proof: (*by contradiction*). Assume that $\forall\ D_d\ \exists\ t \in$ DESC(O_o) $\mid \{\ R_d(t) \leq C_d\ \}$, i.e. that there is such a solution. By monotonicity, $E_{o,d}(CN) > E_{o_d,d}(s_{d,t}) \rightarrow\ R_d(t) > R_d(s_m)$, where s_m is the leaf of the boundary path; but $R_d(s_m) \leq C_d$, which implies a contradiction.

The next lemma gives a necessary condition for the existence of any

acceptable solution descended from CN. If CN is the root, then it is a condition on the entire design process.

Lemma 5.5: $\exists\ O_o \in \mathrm{OUT(CN)}\ \forall\ D_d \in D \mid \{E_{o,d}(CN) < E_{o_d,d}(s_{d,t}) \vee (O_o = O_{o_d} \wedge s = s_{d,t})\ \}$, where $s_{d,t}$ is the state on the boundary path at the same depth as CN, is a necessary condition for the existence of an acceptable solution descended from CN.

Proof: If we negate the expression above, we have the expression of Lemma 5.3, for which no solution can exist. That is, the contrapositive of this Lemma is Lemma 5.3.

The next lemma shows that the number of **live** out-arcs of CN is at most the cardinality of the D^* coming into CN, i.e. the number of boundary paths that pass through CN.

Lemma 5.6: At most $|D^*|$ out-arcs of CN can be **live.**

This is a consequence of the fact that (1) every arc has a unique estimate, hence no two arcs can bound for the same D_d; and (2) an arc can only be **live** if it has a bounding value for at least one member of D^*.

The next lemma shows that all of the descendants of a live arc are acceptable in dimensions that are not members of D^*.

Lemma 5.7: If a live arc O_o does not bound in $D_d \in D^*$, then no descendant of O_o can violate the constraint or bound in D_d.

This follows directly from the fact that $E_{o,d}(s)$ is assumed to be unequal to every other estimate at the same depth, and the definition of monotonicity.

The following lemma shows that the cardinality of successive D^* sets

can only shrink or stay the same; i.e. that the number of live paths spawned by an arc is bounded by the size of the D* at the arc.

Lemma 5.8: A **live** path, which is on the boundary for $D+ \subseteq D^*$ dimensions, can spawn at most $|D+|$ live paths. This is a consequence of the fact that live paths spawnable from $D+$ represent disjoint subsets of $D+$, of which there are at most $|D+|$, and no constraint not in $D+$ can spawn a live path, by lemma 5.7.

Finally, we can prove the time complexity result given above, that at most $|D|$ paths need be searched.

Theorem 5.9: At most $|D|$ paths can be **live** in the entire search tree.

Proof: (*by induction*). If $|D| = 1$, then the only possible live path is the path bounded in the sole member of D.

If $|D| = k$, then we assume that the number of paths is bounded by k.

If $|D| = k + 1$, then at most one new live path can be added by the addition of the new D_{k+1}, because its bounding path either follows some other bounding path, or it diverges from all others. If it follows some other, then no new live path is added; if it diverges, then it is the only member of D^* along that path, and it cannot spawn further live paths.

The result of Theorem 5.9 is that the BP algorithm can search the $|D|$ paths in $O(|D|n)$ time, which, because $|D|$ is bounded, is of lesser order than the time taken by the BF algorithm.

Note that only the assumption of non-equal estimates allows this conclusion to be reached. Presumably, if $E_{o_1,d}(s) = E_{o_2,d}(s)$, then the property of monotonicity implies that because neither estimate is less than the

other, at least one descendant of each is better than at least one descendant of the other; i.e. neither x nor y is clearly superior in d. Under such circumstances, if no path of the form described in Lemma 5.3 exists, there are potentially k^n live paths to be searched in the tree. Philosophically, the implication of this is that the estimators must be very precise indeed for the $O(|D|)$ bound to hold, so that with probability one no two estimates will ever turn out to be equal.

5.2.2. Monotonic State Estimators: Discussion

While the above analysis and the concept of monotonic operator estimation may seem far afield from the practical, it is interesting to examine current design practice in the light of the concept.

If we define n levels of design abstraction, such that a design at one level can be mapped into the next level down, i.e. from level i to level $i+1$, and such that the discrete differences between designs at level i can be used to monotonically predict differences at level $i+1$ for all i, then we can perform pure top-down design with at most $n^2 k \log^2 k$ detailing steps, where k is a bound on the number of discrete differences.

To give a more concrete example, the practice of design at the register-transfer level is the construction of abstract designs. The mapping of RTL constructs into simple gates is nearly monotonic for area, speed, and power: if we change an RTL design incrementally, we can usually tell in which direction the gate-level design will move, in terms of gate count, average gate power, and average gate propagation delays. As such, register-transfer design is a good predictor of gate-level properties.

When gate-level diagrams are mapped into SSI or into semi-custom MSI

and LSI implementation styles, then estimation of layout from gate-level is again nearly monotonic, and indeed can be accurate as well. Notice that the estimators for standard cells and gate arrays are, however, rather different than those for SSI.

However, when we map gate diagrams into full-custom silicon, the degree of monotonicity suffers. Adding a single gate may not affect the area of a behaviorally equivalent PLA at all, and could even reduce it. Doubling functional units may reduce area because of routing differences; adding extra buffer gates may speed up a design, again because of routing. In such situations, top-down design does not work so well, because it is difficult to evaluate the effect of changes at level k on properties at level $k+n$. In effect, the low-level design must be constructed, or another abstraction (such as a floor plan) used, which was not necessary where gates were mapped into SSI.

Chapter 6
DPE Implementation

6.1. Overview

In this chapter we will examine a software implementation of DPE (Design Planning Engine). We will also examine DPE's domain knowledge representation.

The high-level functioning of DPE has already been discussed in Chapter 4; this chapter covers the implementation in terms of its internal structure and the function of its subunits. Test cases are given in Chapter 7. This chapter and Chapter 7 are semi-independent. If the reader is more interested in the "black-box" behavior of DPE than in its internal details, it is possible to skip this chapter and go directly to Chapter 7. The reader is warned, however, that some of the behavior described in Chapter 7, in particular the interpretation of rules, may seem a little mysterious if this is done.

The overall architecture of DPE in its environment is shown in Fig. 6-1. On the left are three major static data structures: the *planning rule set*, the *frame set*, and the *tool set*. On the right are two dynamic data structures: the *plan* and the *DDS*. Separating them are the *planning engine* and the *execution engine*. The planning rules are a knowledge base used by DPE to construct and oversee the execution of plans. The frame set contains the VLSI design domain knowledge; by loading another frame set a completely different domain could be captured. The tool set is the collection of actual programs

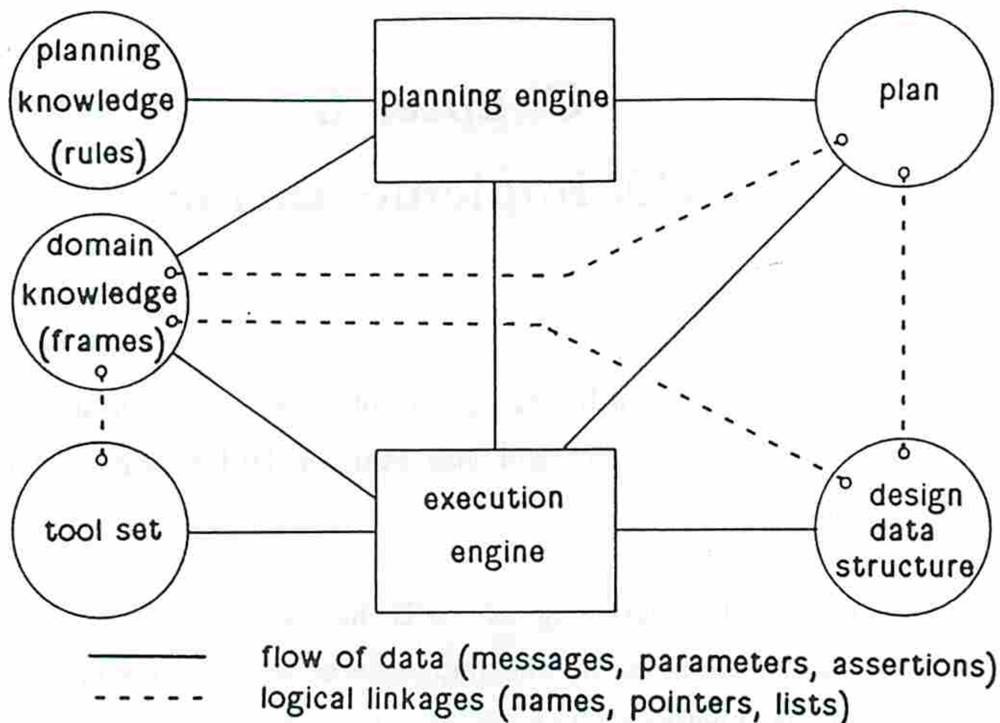


Figure 6-1: Architecture of DPE.

that DPE can invoke; some of these programs may be implemented as LISP function calls within DPE, and some may be arbitrary exectuables in the Unix file system. Hence the tool set is only partially contained within DPE. The planning rules, frame set, and tool set are static data structures; they do not change as a result of DPE's operation.

The dotted lines of Fig. 6-1 represent logical pointers. For example, the dotted line from the domain knowledge to the tool set represents the relationship between operator representations in the frame set and the actual tool program code.

The plan is a dynamic data structure; it is constructed by DPE. It consists of *states*, which represent states of the design process; and *operations*, which represent planned tool invocations. States and operations

form a directed acyclic graph, where states are nodes and operations are edges. Logical pointers from the frame set to the plan represent links between assertions about the design (e.g. "the design is at the RTL level"), and frames describing the meaning of the assertions (e.g. the frame "generic RTL designs"). Logical pointers also link operations (in the plan) with operators (in the frame set); for example, the plan arc "pipeline-allocate" corresponds to the operator description frame "pipeline-allocator", which in turn corresponds to the Unix executable "Sehwa", which is the pipeline allocator program.

The DDS has been described in Chapter 3; it contains detailed representations of the design data. The DDS is modified by tools; it is a dynamic structure. Logical links between the DDS and the plan represent the correspondence between plan states and DDS entities. For example, a plan state might contain the assertion "the design is named `foo`"; then there is a DDS component named `foo`, which is the target design. Assertions about `foo`, e.g. "`foo` is pipelined", reflect facts about the component `foo`[15].

The planning and execution engines oversee the construction and modification of the dynamic data structures, under the direction of knowledge contained in the static data structures. These engines are far more complex, and far less easily partitioned, than the figure suggests.

For example, rule sets for estimation and terminal state detection are built up at run time from rules taken from both the planning knowledge base and the domain knowledge base. This is done by concatenation of rule lists, following which the inference engine is invoked with the concatenated rule set as its knowledge base. The advantage of this strategy is that some rules, for

---

[15]Note that the plan may reflect *hypothetical* facts about the DDS; if the plan is yet to be executed, the assertions may not be reflected in the DDS. If the plan has been executed, then the assertions should be true statements about the design.

example those for data setup and result reporting, are domain-independent; whereas other rules, for example those for area estimation of RTL constructs, are domain-dependent.

In order to describe the structure and function of the planning and execution processes, which in fact are collections of objects communicating by message passing, it is necessary to discuss the layers of interpretation that are implemented in DPE.

6.1.1. Layers of Interpretation in DPE

DPE is built in five main layers. Each layer can be thought of as a programming language. Each layer is implemented using the constructs of the next layer down. Such an architecture is based on the idea that it is easier to capture the essentials of a problem in a language tailored to the problem than in a more general-purpose language.

The layers in DPE are

1. frames, which represent the language of design entities,
2. actors (rule-based objects), whose rules are written in the language of design entities and in the language of constructing design plans,
3. rules, which are used to implement complex actor behaviors,
4. Flavors, which gives an underlying message passing layer, and
5. LISP, which underlies them all.

This chapter will now proceed to the Flavors message-passing programming language, because it will then be easy to understand the layers built on Flavors. Then the discussion will proceed to the ways in which production rules are used and interpreted. Following that, the frame system

in which design domain knowledge is represented will be described, after which the actors that oversee the construction and execution of plans will be described. Finally, the reasoning behind some key decisions will be given.

6.2. Flavors

The Flavors system is almost an exact copy of that described in [Winston 84]. Only a brief description will be given here. Flavors is an "object oriented" programming language with the following features:

1. it encapsulates behavioral abstractions in *flavors* whose behaviors are invoked by means of message passing,

2. it encapsulates data abstractions by allowing access to the object's internal data only through messages and private procedures,

3. both behaviors and data structures can be inherited, and

4. types and instances are both subsumed into flavors,

A system based on objects of this kind is easier to program and modify than other systems, primarily because of increased modularity and information hiding, but also because the inheritance and message passing paradigms are powerful techniques for keeping coding effort at a minimum.

The actors described in the previous sections are each *sui generis* flavors. DDS entities are also flavors, using an embedding of the DDS similar to that described in [Afsarmanesh 85a].

6.3. Rule System

This section will describe the technique used for the implementation of complex behaviors, i.e. the production rule system.

A production rule system consists of three major parts.

1. The first is a set of *rules*, each of which consists of a *condition*, and a *result* or *action* which follows upon the condition. These rules are often referred to as the *knowledge base*, a designation that will be avoided here because of possible confusion with the frame system. Instead, the designation *rule set* will be used.

2. Second is a set of problem-description information called the *data set*. This models the problem to be solved and the intermediate results of the solution process.

3. Third, there must be an *inference engine* that cyclically applies the rules of the rule set to the information represented in the data set.

Such systems are well documented (e.g. in [Winston 84] and [Barr 81].) Briefly, a *forward-chained* inference engine searches the rule set for a rule whose condition is met by the data set; the rule is then *fired* and the result is placed in the data set. This process is repeated until some termination condition is met. A *backward-chained* system is similar, but the results are assumed and the conditions that would produce those results are added to the data set, again cyclically, until an initial condition in the data set is found.

6.3.1. The Inference Engine

The inference engine is a forward-chained, non-backtracking, unifying type, with three rule priority schemes and several escape and debugging features. It differs from other inference engines primarily in its handling of logical negation and arithmetic. Its other features are reminiscent of the LOOPS rule interpretation engine [Bobrow 81], for example its use of object-oriented semantics, local context or task information, and in some of its rule firing control disciplines. It differs from, e.g., OPS5 [Forgy 81] in that it does not use compiled rule sets, it does not backtrack, and it does not use recency criteria for rule priority ordering.

The inference engine is supplied with a set of rules and a **Context**

variable. The **Context** variable is used to store temporary data, primarily the sets of value-variable bindings that are generated by unification. The rule set is an ordered list of rule IDs. Each rule has a **Flag**, an **If-Part**, and a **Then-Part**. These will be explained below.

The **context** variable contains a list of **Facts**, i.e. assertions, a **Firing-History** of rules that have been fired, and temporary storage which is used for variable unification.

6.3.1.1. Basic Cycle

The basic cycle of the inference engine is divided into two phases, the *test* phase and the *fire* phase. This cycle is repeated without limit. It can be exited by firing a rule with an exit clause in its consequence part, or by having no firable rule. The test phase consists of testing *firability* of each rule in the rule set in turn. There are two strategies for conflict resolution, corresponding to two inference procedures an actor can call:

1. *first-fit* tests each rule in sequence. When it encounters a firable rule it enters the fire phase immediately, i.e. it fires the first firable rule it encounters, and

2. *best-fit* tests all rules and selects the rule which is both firable and has the longest **If-Part**, measured in number of clauses.

A rule is firable if its **If-Part** will unify in the context of the facts in the data set (i.e. if some combination of facts will make the **IfPart** true), its **Flag** either has the value **fire-always** or if its **Flag** has the value **fire-once**, and it has not been fired since the last time the **Firing-History** of the **Context** was reset to **nil.** This use of **fire-once** rules was borrowed from LOOPS [Bobrow 81].

The **If-Part** of a rule is composed of one or more *clauses.* These

clauses are tested against the **Facts** of the **Context**, to see if they can be matched and any variable references unified. The unification procedure is semantically similar to that used in PROLOG [Clocksin 81].

The leading clauses of a rule's **If-Part** may be *negative* in two senses. All other clauses are *positive*. Only if all of the positive clauses can be unified, and the negative clauses cannot be, will the rule be considered firable. The matching criteria for positive clauses will now be discussed, following which a discussion of the two kinds of negation will be given.

6.3.1.2. Variables and Unification

An assertion in the fact set is a list of atoms, e.g. `(dataflow-graph ALU-1 DFG-1)` which has the meaning that a design called `ALU-1` has a dataflow graph named `DFG-1`. The **If-Part** clause `(dataflow-graph ALU-1 DFG-1)` will match this assertion trivially.

However, it is often desirable to include variables in a match clause; for example in the rule **if** `(dataflow-graph ?it ?itsdfg)` **then** `(optimize ?itsdfg)` which states that if the dataflow graph of some component **?it** exists and has name `?itsdfg`, then an attempt to optimize the graph should be made. The leading "?" on the variables `?it` and `?itsdfg` is used to denote that these atoms should match any atom; hence this rule's `If-Part` will match the assertion given above, with the variable replacement bindings `ALU-1` and `DFG-1` for `?it` and `?itsdfg` respectively. Furthermore, the binding of `?itsdfg` will be used in the assertion `(optimize ?itsdfg)`, giving the unified assertion `(optimize DFG-1)`.

This can be extended to cover arbitrary numbers of clauses and match variables; DPE performs *unification* on the match variables. That is, some consistent set of variable bindings must be found.

Most expert systems handle negation in ad-hoc ways. For example, MYCIN could fire a rule on the basis of a positive statement that a condition was untrue; e.g. "there is no evidence of infection". As another example, PROSPECTOR used a pair of probability-like numbers attached to each clause, which described the contribution the presence or absence of a match would make.

Negation is handled in DPE by means of two kinds of negative clauses. A `no`-clause is one that completely disables the firing of the rule if there is any match in the fact set. For example, the clause `(no (structure ?any ?any-other))` in a rule's **If-Part** will cause the rule to fail if there is any assertion in the data set matching it.

A `nor`-clause is used where a subset of variable unifications is to be rejected. Essentially, the `nor`-clause is unified in the same way that positive clauses are, but separately from the positive clauses. Then all of the variable bindings that unified in the `nor`-clause are removed from the binding sets of the positive clauses; if no bindings remain then the rule becomes unfirable. If there is more than one `nor`-clause in the rule, then this process is repeated for each independently. For example, the **IfPart**

```
(nor (structure ?it ?any))
(pipelined ?it) (design ?it)
```

would reject any design that already had a structure, and select designs that were pipelined. So if two pipelined designs A and B were present, and only A had a structure, then B would activate the rule and be operated on by it.

The rule-firing phase of DPE's inference engine is comparatively simple. All of the clauses of the rule's **Then-Part** are unified, then sequentially asserted into the **Fact** set of the **Context**, unless they are escape clauses, i.e. they begin with an escape keyword.

6.3.1.3. Escape Clauses

The following kinds of escape clauses are provided by DPE's inference engine.

1. The inference procedure can be reset and exited by escapes, which is used when an actor is finished with its task.

2. Communication between actors is provided by escapes that cause Flavors messages to be passed, and by escapes that cause assertions to be made in other fact sets. This is useful because rule sets can easily query and inform other rule sets.

3. Assertions can be removed from the fact set. This permits rules to deny the truth of statements in their own and other fact sets.

4. Numeric computations can be performed by the use of a simple postfix interpreter, which permits the system to perform calculations, e.g. for the purpose of constructing estimates.

5. Tracing is implemented by a number of reporting and record-keeping escapes, which is extremely helpful in debugging.

6. The inference engine can be stopped and an interactive debugging loop can be entered, which is also very helpful in debugging.

6.3.2. Uses of Rules

The inference procedure and the unification pattern matcher are used in a number of ways. Briefly, the rule system is used for executing

1. complex actor behaviors implemented with rule sets,

2. rule-based postconditions (i.e. postconditions that are computed using a set of rules),

3. rule-based estimators, and

4. rule-based operator application.

The unification matcher is used for matching preconditions and constructing

non-rule-based postconditions as well. These uses of the unification pattern matcher and the rule interpretation system as a whole will be taken up in the following sections.

6.4. Frames

There are two basic frame types in the current version of the system. The first is the *hardware* frame; the second is the *task* frame. A hardware frame describes a particular hardware structure in terms of the tasks that pertain to it. Task frames describe operators that can be put into a plan.

6.4.1. Hardware Frames

A hardware frame is used to represent knowledge about a particular general class of hardware structure, e.g. register-transfer-level designs. A hardware frame consists of two major fields: an **Estimators** field and a **To-Build** field. The **Estimators** field contains information relevant to the estimation of the properties of the kind of hardware. This is expressed as a collection of rule sets, the estimators being of the form discussed above.

The **To-Build** field consists of a set of pointers to **Task** frames: it represents the collection of operators known to be relevant to the process of designing hardware of the given type. For example, the task frames referenced by the hardware frame `Generic-RTL` include those for dataflow analysis, control style selection, operator and register allocation, and preliminary microprogram synthesis. The tasks are organized as an unordered set any or all of which may be used; in another frame, the tasks could be organized as a sequence (i.e. something similar to a script, the difference being in setup information that is calculated on the fly by execution rules).

6.4.2. Task Frames

Task frames are more complex than hardware frames. They contain information about operators, much of which has been described above, although not under a single heading. That information is

1. the preconditions of the task,

2. the postconditions of the task,

3. the estimator rules for the operator,

4. the execution rules for the operator, and

5. the subtask structure of the operator.

These will now be described as the task frame model collects them into a cohesive whole.

The preconditions of the task are those conditions that must pertain in order for the operator to be semantically appropriate. That is, when the operator is considered for chaining, some set of conditions on data availability, format, and semantics must apply, or the operator is semantically inappropriate and there is no point to considering it further. This set of preconditions is expressed by a list of assertions, which may contain match variables. These assertions are matched against the modeled state of the design, i.e. the representation of the design in the planning space. If all of the assertions are matched in the fact set of a modeled design state the operator is considered to be semantically and syntactically applicable to the state.

The postconditions of the task consist either of the combined add set and delete set of the operator, or of the operator-effect description rules discussed above. In the case of an operator whose postcondition is described by an add set and a delete set, the assertions of the delete set are unified (i.e.

any match variable references are replaced by actual strings as matched during the testing of applicability) and the assertions then deleted, if they are present, from the next model state; then the add set is unified and added to the next model state.

In the case of rule-based postconditions, the match variable bindings constructed during the precondition match process are stored in a special **static-bindings** location and the inference engine invoked using the rule set and a copy of the old design state model.

Each task frame has a set of operator estimator rules[16] attached to it. These rules take the design state and the current goals of the **planner** into consideration, to produce a single number "advisability". This is done using rules similar to the example estimator rule given above.

The execution-control (application) information for the operator is also expressed in the form of a set of rules. These rules are used for setup, invocation, and reformatting the results of operator execution. In some cases, where the entire execution-space effect of the operator can be expressed in rules, there may be no real "invocation"; the execution rules themselves in conjunction with the inference engine constitute the execution space operator.

Finally, the subtasks of the operator are expressed as a list of task names preceded by a keyword. In the current implementation, the keyword can take on one of two values, signifying either a production system or a procedure. In a system with hierarchical planning, which was the original intention of this field, the subtasks field would describe either a procedure, a

---

[16] It is to be hoped that these rules form monotonic sets of operator estimators. It is difficult to prove, however. Some empirical evidence is given with the experimental results of Chapter 7.

rule collection, or a collection of further task frames, organized in any of a number of ways.

6.4.3. Rule-Based Domain Knowledge

The following kinds of rule-based and unification-based domain knowledge are embedded in the frame system.

6.4.3.1. Preconditions

Operator preconditions are expressed in the same way as rule **If-Parts**, i.e. by sets of clauses that must be unified. The variable bindings generated during unification are saved for later use by either rule-based or assertion-based postconditions. For example, the operator MAHA has the following preconditions.

```
(design-focus-of-attention ?it)
(dataflow-graph ?it ?its-dataflow)
(module-library ?modlib)
```

This precondition means that in order for MAHA to run, a component `?it` must exist, `?it` must be the current focus of attention, and there must be a named module library present. Note that because the atom `?it` appears in two clauses, it is unified; i.e. only the dataflow graph of the design focus component will satisfy the second clause.

6.4.3.2. Postconditions

The current implementation allows the effect of an operator to be modeled in either of two ways. First, an *assertion-based* postcondition is expressed as one or two lists of assertions. The first of these, the *add set* represents assertions that are added to the data set by the operator's simulated invocation. The second, the *delete set*, represents assertions that will be deleted from the data set. Both added and deleted assertions are first unified using the match variables of the precondition.

The difficulty of such a representation is that it is all-or-nothing: there is no idea of conditional or data-dependent effects. As such, this model of the function of a complex operator is weak.

Rule-based postconditions are represented through the use of a production system. In such a case, the add and delete sets are supplanted by a collection of rules. This is semi-transparent to the system; when operator effects are calculated, either an add/delete set is used or a set of rules is passed to an inference engine, depending on what is present in the operator description.

Because the rules that describe the effect of an operator in the model world can be made highly data-dependent, and intermediate inference, message passing, and invocation of LISP functions can be done, the model of an operator's effect under the production system model is more general.

For example, the program MAHA has a rule-based postcondition that includes the following two (edited) rules.

```
(IF (number designTime ?D)              ; old design time
    (dataFlowGraph ?it ?itsDataFlow)
    (number ?itsDataFlow nodes ?n)) ; how many nodes?
        (THEN (setr designTime ?n 5 * ?D +))
           ; remember it's postfix!
(IF (importance area 1)                 ; maximum value
    (designFocusAttn ?it))
        (THEN (wordSerial ?it)          ; no parallelism
              (moduleCount ?it 1)       ; an ALU
              (architecture ?it bus))
```

The first rule states that the estimated time it will take to execute the plan will be the time it takes to execute all operators previous to MAHA (i.e. `?D`), plus five times the number of nodes in the dataflow graph of the design focus of attention, and the second rule states that if the importance of area is 1.0, then the implementation will be bus-oriented, it will have a serial

architecture, and it will have a single functional unit aside from registers. Other rules are used to predict the results of running MAHA under other circumstances. Note that MAHA's execution may not result in exactly these results; it may not produce a bused architecture, or the number of operators may be greater than one. The postcondition represents a "best guess without execution".

6.4.3.3. Estimators

Rule-based estimators address a problem that is fundamental to the UDAS: that of highly discrete and inhomogeneous design spaces. In this domain, the facts that are present may be important to proper estimation, but there is little assurance that the facts will be present in any particular set or combination. Hence a nonprocedural technique is convenient. DPE uses the rule inference engine and sets of rules specialized to the particular entity to be estimated. The inference engine was augmented with escape clauses that allow the evaluation of arithmetic expressions by a postfix interpreter.

This technique uses a special form of consequent clause to perform arithmetic. An example of such a rule, taken from DPE's knowledge base and edited for readability, is the area-estimation rule given below.

```
(fire-once-rule
          ((structure-based-estimate)
           (design-focus-of-attention ?it)
           (mux-count ?it ?n)
           (number average-mux-size ?x)
           (number area ?A))
          (setr area ?A ?n ?x * +)).
```

This rule was taken directly from the rule set (abbreviated names were expanded to their full length: the original rule has much shorter strings). It illustrates the power and flexibility of the rule-based approach to estimation; arbitrary facts can be built into the rule's **If-Part** and incorporated in the arithmetic and logical calculations of the **Then-Part**. The rule is a

fire-once-rule, meaning that it is only allowed to fire once during the course of an estimation; such a stricture is needed in arithmetic calculations of this kind. The condition part of the rule is composed of five clauses, all of which must be met for the rule to be applicable. In English, these five clauses have the following meanings.

1. The estimation currently being calculated must be on the basis of logical structure, i.e. a schematic diagram.

2. Some schematic module, whose name is to be bound to the variable `?it`, is known to be the focus of attention, i.e. it is the thing whose area is to be estimated.

3. `?it` has some number `?n` of muxes inside it.

4. A previously calculated number `?x` represents the average mux size for `?it`. This number is a guess, not an exact calculation: an exact calculation would negate the value of this rule. Another rule came up with the guess.

5. A previously calculated number `?A` represents the combined area of `?it` before the contribution of muxes is added in.

The action part of the rule consists of one clause, beginning with the arithmetic escape keyword `setr`. This keyword invokes an arithmetic interpreter, whose first argument is the name of a variable, and whose other arguments form a postfix expression whose value will be computed. The postfix interpreter will assert the fact `(number area` *area*`)` into the current data set, where the meaning of the entire assertion is that there is a `number` attached to a variable `area`, and *area* is the number. This syntax allows numeric attributes to be treated in the same way as other attributes.

One of the main advantages of the rule-based approach is its behavior when data is incomplete. Under such circumstances, the more specialized (and hence more accurate) rules cannot fire; however, the estimator can still

function because less specialized rules that do not require so much information can still be fired.

6.4.3.4. Application Rules

The last major category of rules is that of *execution* rules. These rules are organized in sets that are attached to operators in the same manner as operator estimation rules. Execution rules are used when the execution of a task is necessary. There are normally three stages to this process.

First, the data and control parameters for the operator are set up in the correct form. For example, a register-transfer level clocking scheme synthesis tool does not need physical information about elements in the design library: only an abstract cost and a time delay associated with each element. Such abstractions may be supplied in the form of a procedure, a message, or an operating system call. In the case of external operators, i.e. stand-alone program code running under the operating system, some mechanism for data access, either a shared file or a pipe, must also be provided.

The control parameters are also set up by the rule set. For example, a program for nonpipelined RTL synthesis may need an overall time constraint, an area constraint, and a flag telling whether to push area or time to an optimum given that both hard constraints can be met.

There is more than one way to invoke an operator. It can be implemented directly in the form of rules; it can be a chain, lattice, or graph of procedural tools; it could even be a recursive invocation of the UDAS's top level. Procedural tools can be invoked by means of messages, rule interpreter escapes, function calls, operating system **fork** and **shell** calls, or combinations of the above.

There may also be little or no procedural code associated with an operator. In such cases the rules normally associated with setup perform the operation themselves. We will now describe the *actor* level of computation.

6.5. Actors

The topmost implementation structure of DPE is that of the *actors* [Hewitt 73]. An actor is an independent software module which encapsulates behavior (i.e. code) and private data. Actors communicate with one another by *message passing*. In this thesis, the term *actor* is used to denote a "smart object", i.e. an object with rule-based behavior, as opposed to passive objects and objects with only procedural behaviors. Actors can also have procedural behaviors where the behavior is logically related to the actor's normal function, but is too simple to warrant the power and expense of using a rule-based inference engine.

Each actor can be thought of as an "expert" on some aspect of a data-management or design strategy problem. For example, a single actor, the design data manager, encapsulates information about and access procedures for DDS information. Another actor, the **planner**, is responsible for the overall strategic direction of the UDAS. The major actors of DPE will now be discussed, following which a description of their collective operation as a whole will be given. The major actors are shown in Fig. 6-2, which does not include minor or dynamic actors.

6.5.1. Major Actors

There are three main actors. These are the **planner**, the frame set expert (or **framex** for short), and the plan manager (**manager** for short). The function of the **planner** is to co-ordinate the building and execution of plans in accordance with the specifications and goals specified by the user; the

function of the **framex** is to act as an information-hiding layer between the **planner** and the frame system; and the function of the **manager** is to act as an information-hiding layer between the **planner** and the details of the plan.

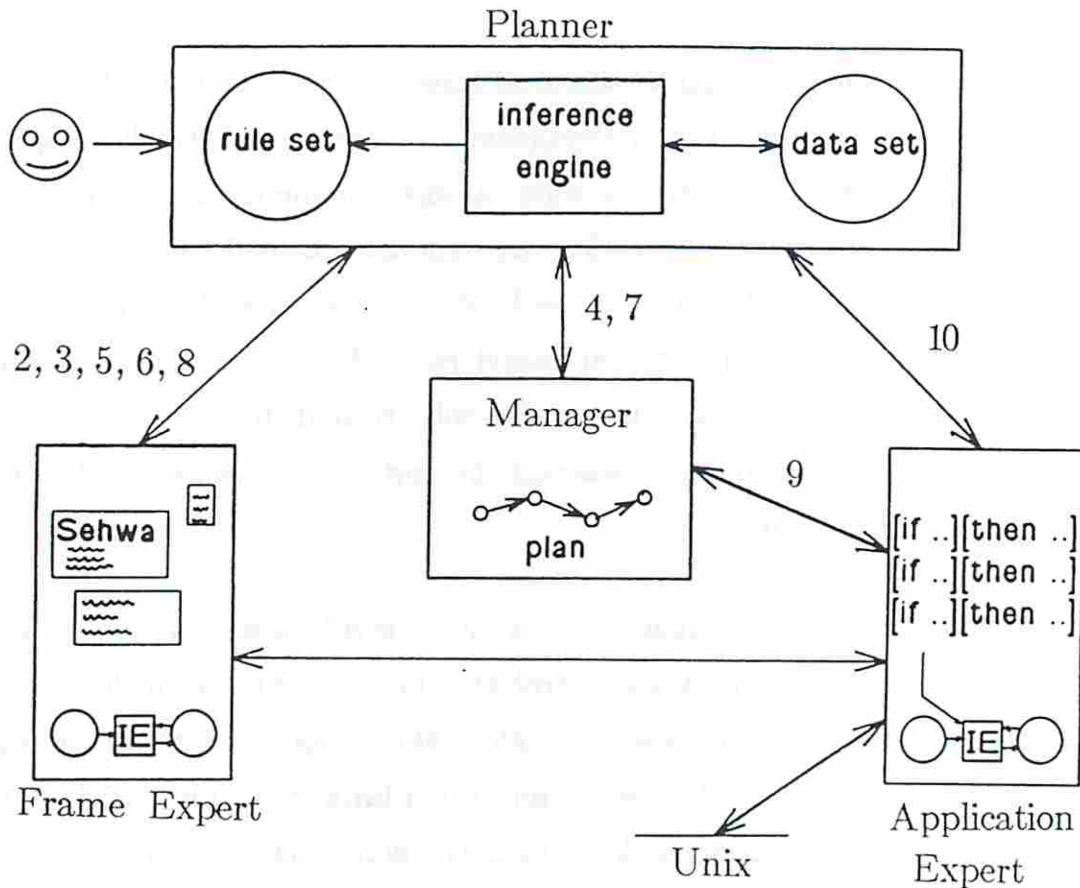


Figure 6-2: Major actors of DPE.

The **planner** and the **framex** each have sub-actors with special rules for the particular tasks of plan extension and plan expansion; these are the planner extension actor (**extender**), the plan expansion actor (**expander**), the frame set extension actor (**framext**), and the frame set expansion actor (**framexp**). Each of these uses rule-based inference. Each has its own rule set.

Other rule-based actors include the *done expert* or **donex**, which uses a set of problem-specific rules to detect a complete plan; the **guesser** which constructs initial importances to start the interpolation process as described in Chapter 4, the **evaluator** which oversees the construction of estimates; and a dynamic collection of nameless **estimator** actors which are constructed and invoked by the evaluator to perform the actual estimation.

An actor that is constructed on the fly at design time, and discarded after it has performed the required inference, is called a *dynamic* actor. Such actors vary in complexity. They use rules constructed by concatenating lists of rules drawn from each of the relevant sources.

The various actors will now be discussed individually, following which their action as a coordinated system will be described. The description of each actor is divided into two parts. In the first part, the inputs and outputs of the actor are given; in the second, the internal operation of the actor is given. The cardinalities of the actor rule sets are also given, as an indication of the relative complexity of their tasks. However, the actor rules themselves are not given, because they are complex and detailed, and would not contribute anything except a mass of implementation detail at a very low level. Domain knowledge base rules are, however, given in Chapter 7 and in Appendix II.

6.5.1.1. The Planner

The most important of all the actors is the **planner**. This actor is responsible for the overall strategic operation of DPE. This area of responsibility is further divided into three subareas.

The first subarea is that of *extension/expansion*: i.e. the decision whether to extend a plan or expand it. In the former case, a new operator is added to the plan; in the latter, operators that make up the plan are

executed. This involves 41 rules altogether. Its input is the message **run**, which is passed to it from the user. It then drives the remainder of the system by sending messages to other actors and drawing inferences. The inferences are based on the current set of plans, current goals, and on replies passed back to the planner from the other actors. The internal operation of this actor is described in detail below.

The second subarea is that of *extension*: the selection of an operator to add to the plan, and its actual addition. The **extender**, which implements this function, has eight rules, and is implemented by the **extender**. It is invoked by a **extend** message from the **planner**. The message also contains the name of the leaf state to be extended. The result of extension is a message to the **manager** to the effect that a set of operators should be added to the plan at the chosen leaf state, and a message to the **planner** naming the new leaf state with the highest advisability.

The third subarea is that of *expansion*: the selection and execution of an operator already in the plan, which is carried out by the **expander**. This function has 16 rules. It is passed the message **expand** and the name of a planned operator. It replies with a message reporting either success or failure; these correspond to the normal or abnormal termination of the operator program code, respectively. We will now discuss these functions in more detail.

The extension/expansion decision is made by the planner on the basis of the completeness of the plan (which is determined by another actor, the **donex**, described below,) and on the basis of the projected worth of executing a complete plan. The strategy it currently implements is described in Ch. 4.

Included in the extension/expansion decision is the decision of what

state to extend from or what step of which plan to expand. In the case of extension, there is a single *prime* state found by operator estimation of a set of leaf states and their immediately preceding operators. The prime state is usually the one which has the highest advisability, but in the case of a tie an arbitrary choice between the top contenders is made.

In the case of expansion, choosing the plan to be expanded is a side effect of the expansion trigger. That trigger is the detection of a duplicate plan, as described in Ch. 4. When such a situation occurs the newer duplicate plan is executed and the old duplicate is deleted from the plan tree. The steps of a plan are executed in strict plan order.

The **extender** is a specialization of the **planner**, adapted for the extension task by the use of a set of rules for extension. It is passed the name of the state to be extended, and it then runs the inference procedure to discover what operator to extend the plan with; it then informs another actor, the **manager** (described below) of the identity of the operator and the state. The **manager** then performs the actual concatenation of the operator onto the plan.

The basis for operator selection is twofold. First, the identities of all the operators that are relevant to the current situation are requested from the frame expert. Then the applicability of each operator is determined by comparing its preconditions to the design state in question. Each operator's action is then modeled by using its postconditions to generate a new leaf state. Finally, the prospective value (advisability) of the applicable operators is estimated and reported to the **planner**, along with the names of the new plan arcs corresponding to the operator applications.

The **expander** is invoked when a plan is to be executed. It is passed the

name of a planned operation application (i.e. an arc.) It then performs the following actions.

1. The application rules for the operator associated with the plan arc are gotten by querying the frame expert.

2. The rules are then used as a basis for constructing a dynamic actor by associating them with a generic execution-expert actor. The rule set is augmented with a set of rules common to all execution experts.

3. A message is sent to the new expert, which causes it to run the inference procedure on its rule set. At some stage this rule set will normally cause the actual program code to be invoked, as described below.

6.5.1.2. The Frame Expert

The frame expert, or **framex** for short, is responsible for managing the frame system. The overall function of the **framex** is to answer queries about names and fields of frames, and to search for relevant knowledge in the frame system. It does this in response to queries from the other actors of the system.

The **framex**'s responsibilities are divided into an organization that parallels that of the **planner**, consisting of three subareas of responsibility. Each is a rule-based actor.

The **framex** proper has LISP-based behaviors for loading and organizing the frame set when DPE is started up. It does this automatically in response to an `initialize` message. It is also responsible for finding state estimators that are appropriate to the kind of hardware described in a state, when requested to do so by a `find-estimators` message. Its reply to this message is a set of rule names. It finds estimators by searching for keywords in the frame set. The keywords are gotten from the design state to be estimated. The **framex** then causes the construction of a new actor if the

estimator is rule-based, or it invokes the LISP interpreter if the estimator is expressed as a lambda body.[17] The **framex** proper has eight rules to perform this function.

Second, the frame set extension expert (**framext**) was constructed for the handling of queries specifically dealing with plan extension. It responds to several messages, having to do with requests for names of relevant operators, requests for operator pre- and post- conditions, and requests for operator estimator rule sets. The **framext**'s main function is finding relevant operators and returning them as candidates when asked by the **extender**. This is done by keyword search of the frame set, on the basis of keywords found in the current design state. In the current implementation the keyword used is "level", which can take on values such as "RTL". This is used to find all RTL operators in the frame set. These functions are implemented by eight rules.

Third, the frame set expansion expert (**framexp**) is specifically concerned with expansion (i.e. execution) of plans. It responds to messages asking for the application rules, code type (e.g., LISP, rule-based, or foreign), and code of an operator. For example, an operator might have application rules that set up data and parameters for code that was written in C; the code type would be "foreign", and the response to the `get-code` message would be the ID of a program to be forked and executed.

---

[17] These estimators are not to be confused with those estimators that are regarded as first-class (i.e. plannable) operators.

6.5.1.3. The Manager

The **manager** is responsible for the maintenance and access of the plan data structure. This includes operations arcs, also called *tasks*, and states, as well as links to the execution space models of states (i.e. the DDS) and tasks (frame system operators.) Hence while the **framex** is responsible for the static domain knowledge contained in the frame system, the **manager** is responsible for the dynamic knowledge about particular design problems contained in the plan and DDS structures. The **manager** responds to messages asking for state contents, updating state contents, adding operators to a plan, propagating operator postconditions, and queries concerning DDS entities.

The **manager** generally performs LISP-based functions of plan data structure maintenance, access, object counting, and so on. One of its functions, however, is rule-based. This function comes into play when an operator is added to the plan. In some situations (described in Section 6.4.) this requires the firing of operator-specific postcondition generation rules, which is done by the **manager**, which "adopts" the rules and then invokes the inference procedure.

The **manager** is also responsible for the application of mapping functions that transform DDS structures into execution data; for example, the construction of specialized data formats for the pipeline-synthesis program.

6.5.1.4. Miscellaneous Actors

The folowing actors are more specialized and of less global importance than the major actors described above.

An actor called **donex** has eight static[18] rules and a set of situation-dependent rules taken from the frame set. Its area of responsibility is that of testing the completeness of a planned design state; it is called by the **planner** before every extension, with the state to be extended as its argument. The eight static rules are used to get the relevant hardware completeness rules from the **framex** by examining the design state in question and passing keywords to the **framex**. When the **framex** finds the frame(s) that describe the hardware in question, a collection of rules is returned to the **donex**, which then constructs a dummy expert. The new expert has the specific function of testing the design state for completeness. The **donex** then reports the result to the **planner** and disposes of the dummy expert.

The class of **estimator** actors, which includes otherwise nameless actors constructed by concatenating rule lists as described in Section 6.1, has 27 static rules, divided into six groups. The static estimator rules perform setup and reforming tasks for dynamic actors that are constructed from rules embedded in the frame set. Two other rule groups are the rules for the **evaluator**'s state and operator estimator setup functions respectively. The evaluator constructs the required kind of estimator by concatenating rules from the other four static groups with problem-specific rules taken from the frame system, in response to a `get-estimator` message.

The **guesser** is an actor that sets preliminary importance numbers in order to start the planning/interpolation/replanning cycle. It has ten rules, which oversee the loading of the dimensions of interest from the frame set, and the making of initial hypotheses concerning those dimensions.

---

[18] A *static* rule is one that is the same for all of the actors in a class, and taken from the domain-independent rule set; as opposed to *dynamic* rules, which are taken from the frame set.

6.5.2. Operations at the Actor Level

The following discussion is heavily predicated on the current set of domain-independent rules; if these rules were to be changed the behavior of the system could become radically different. Note also that, in the following discussion, the flow of control and the message traffic of the system have been abstracted considerably in the interest of clarity and brevity. Each message described here has an attached sequence number; refer to Fig. 6-2, in which the numbers are attached to arcs to indicate the source and destination actors.

The system is started by the receipt of a **run** message (1) from the user. This message goes to the **planner**, which then asks the **donex** (not shown in Fig. 6-2) to test the root state for completeness. Presuming that the root state is incomplete in some respect, the next thing that happens is that the **planner** decides to extend the root state. The decision to extend cannot be carried out immediately, however, because no criteria for judging the value of operators or the completeness of states yet exists. The **framex** is therefore asked by the planner (2) to find the criteria of interest (e.g. area and time) and the completeness rules for the kind of hardware specified in the root state. The **guesser** (not shown) is then asked by the planner to create hypothesized importances for the criteria of interest. Now the **planner** calls the **extender**, which will add a task to the plan.

The task is added by the **extender** in the following way. First the **extender** asks the **framex** for the set of applicable tasks (3). The **framex** then asks the **framext** for the set of relevant operators; these are gotten by matching design state information against frame set fields. The set of relevant operators is then reduced by comparing the design state against the operator preconditions. All of the applicable operators (i.e. those whose

preconditions are met) are then concatenated onto the prime state as tasks by the **manager** (4) and new leaf states generated by applying the operator postconditions.

The new prime state is then calculated by the **extender**, which applies the operator estimators of the tasks, and the **planner** to which the extender reports the estimation results. Each of these estimators is constructed as a dynamic actor using rules taken from the static rule set and rules taken from the relevant operator frames. Each returns a scalar called "advisability", which is calculated on the basis of the current importances and design state information. The advisability numbers are then compared to one another to determine the best operator; the ranking is passed back to the **planner**, which then selects a prime state and repeats the cycle on the new prime state.

When the **donex** finally reports that a prime state is complete, a state estimator is constructed using rules taken from the relevant frames (5, 6). The form and operation of such estimators is described below. The estimator returns values for all of the dimensions of interest; these values will be used in the interpolation process. The test for duplicated plans is also performed at this time.

When the **planner** decides to expand a plan, the name of the first task is passed to the **expander**. The **expander** then asks the **manager** (7) for the name of the operator that the task is an instance of; it then asks the **framexp** (8) for the subtask composition of the operator. If the operator is a primitive the **expander** constructs an application expert to apply the operator to the state. The application expert does this by creating a new dynamic expert using rules found in the operator frame, and then asking the new expert to execute its rule set. The rules of the new expert can ask the **manager** (9) to format DDS and planning space data, can change that data,

and can call arbitrary LISP or Flavors functions, including operating system calls which are used to link to non-LISP and external program code.

The operator application rules also report the success or failure of the operator application (10), and any new planning-space facts that should be present as a result of the application. If the application succeeded, the **expander** will in turn report success to the **planner**, which then makes the expand/extend decision once again. When the final step of the plan to be executed has been successfully exited, the **planner** stops altogether.

6.6. Discussion

In the following sections, some of the reasons for and advantages of the use of particular implementation strategies will be discussed. Three major issues will be mentioned; those of the layered-interpreters model, the nonprocedural estimation technique, and the choice of LISP and LISP-based Flavors as a programming language.

6.6.1. Layered Interpretation

As stated in the first part of this chapter, DPE is implemented using a set of four layers: the LISP, Flavors, actor, and frame layers. There are a number of reasons to use a layered architecture.

First, a layered architecture is makes program construction comparatively easy. In an application domain which is in a state of flux, as that of unified design automation systems currently is, many program-design decisions are based on incomplete and exploratory ideas. That is, we do not yet know even what all the issues are, let alone their resolutions. The freedom to experiment without a large penalty in lost programming effort is therefore less a luxury than a prudent safety precaution.

Programming errors can also be made less important in such a system, because the users of an interpreter can often work around its defects. Suppose a feature F of the interpreter is known to have bugs. As long as other development can proceed using substitutes for F, the programming effort of debugging the interpreter can be separated from the effort of developing new tools using the language it implements. A related consideration, that of optimization, is also made considerably more straightforward, because optimization can be delayed until it is known which constructs will repay the effort of doing the optimization.

Another reason to use nested interpreters for the UDAS application is that the VLSI design problem is constantly being changed by the addition of new software and hardware techniques.

Furthermore, there is as yet no firmly established science of design. In the absence of such a science, it is very helpful to be able to capture different methods, e.g. various mixtures of top-down and bottom-up design. This is very different from the problem of adapting specific techniques to the prevailing technology for circuits; presumably CMOS and ECL circuits can both be designed in either top-down and bottom-up methodologies. A key factor in programming UDASs is that these meta-design tradeoffs are far less well understood than tradeoffs in the target technology.

Hence a UDAS that is "hard-coded" in, e.g. assembly language is more likely to become obsolete, because the effort of updating its methods to respond to a shift in design practice or methodology may very easily be a significant fraction of its original cost. As the language of the topmost layer of interpretation is specialized to fit the kinds of tasks being programmed, this cost decreases. However, the cost of bridging the gap between the underlying machine and the high-level language rises, and the high-level

language is more likely to change when the target technology undergoes a shift. A layered architecture allows the difference between successive layers to be narrowed; hence those parts of the system that are truly made obsolete by technology shifts can be more narrowly targeted for updating.

6.6.2. Nonprocedural Estimation

The use of nonprocedural estimators is a key part of DPE's implementation. Recall that DPE is intended to deal with a complex, highly discrete design space. The implication is that many possibly relevant facts must be taken into account if they are present, but their absence must not cause estimator failure.

For example, suppose that the area of a pipelined function is to be estimated. Clearly the estimation must take into account more information than the mere fact that the design to be estimated is pipelined; one candidate fact is the internal interconnect topology, another is the underlying technology, another is the clocking and data-transfer discipline. These factors define a complex, discrete, nonlinear space. If an estimator must be constructed for each discrete region of the property space, the complexity of the problem will quickly run away with the programmer. Using a nonprocedural estimator, the discrete regions can be described in terms of one or more rules apiece.

Another reason facts may not be present is that they may not be produced in "time"; e.g. where they could be used in operator estimation, but the operator being estimated precedes the operator that would produce them in the plan.

For example, suppose an operator `control-style-select` is to be

estimated. Clearly this can be done regardless of any underlying facts; after all, one can just declare by fiat that a horizontal architecture will be used. Therefore it must be possible to calculate an advisability for control style selection in the presence of no facts whatsoever. However, the number of states, and the number of separate control lines, will have a strong influence on the size of the controller. Hence if area is important it is better to make the decision after more information becomes available. Hence the advisability of selecting a style of controller should be low when there are no facts, but higher when more information is present, and very high when the only thing remaining to be done is the construction of a controller.

Nonprocedural estimators are useful in this situation for two main reasons. First, because the estimator is embedded in a production system, it is possible to infer facts such as the fact `structure-based-estimate` in the example of Section 6.4. If, for example, no structure existed, or if the structure that did exist was incomplete, an estimate that was based on this rule would be meaningless. Furthermore, the number `average-mux-size` can be computed using any number of rules: it need not be an ad-hoc quantity, but can be made to depend on any number of other variables, such as the speed or area goals of the **planner** (which is currently the case), and an average or computed fan-in.

Second, if information is missing the estimator can continue to function. Suppose the mux count of `?it` is not known; in that case the rule shown cannot fire, and deduction of area will rest upon other, less problem-specific rules.

6.6.3. Choice of Programming Language

There are a number of possibilities for a base language for the implementation of a system like DPE. Some of them are:

1. The Xerox LOOPS language [Bobrow 81],
2. The Maryland Flavors package [Allen 83],
3. A packaged inference engine such as OPS5 [Forgy 81],
4. An object-oriented language such as Smalltalk [Goldberg 83] or Flavors [Winston 84], or
5. A logic programming language such as Prolog [Clocksin81]
6. A general-purpose language such as LISP [Wilensky 84].

A combination of theoretical and practical reasons influenced the eventual selection of LISP as an implementation language.

The criteria for language selection were as follows.

1. It should be possible to implement the desired actor/rule/procedure based programming primitives without much effort.
2. It should be well supported by a healthy user community.
3. It should be supported by available computing hardware.
4. It should be easy to interface foreign programs.

LOOPS is a very strong candidate, which implements all of the requisite functions, but it is not supported by Berkeley 4.2 Unix, and hence would be difficult to interface to the programs which are. Moreover, 4.2 BSD and its near relatives Sun 1.4 and 2.0 Unix are the best development systems available at USC.

Maryland Flavors and YAPS, which are available as a single package, are another strong candidate which unfortunately became available only after development of DPE had begun. A reimplementation of DPE might well use YAPS. However, as with all systems using compiled rule sets, building rule sets on the fly can only be done by the use of clumsy and inelegant hacks. This is not necessarily a make-or-break problem, because rule sets are not arbitrarily mixed but are rather carefully concatenated by DPE; but questions about the construction of discrimination nets and copies of data sets, with the attendant overhead, might still necessitate rethinking of important aspects of the system.

OPS5 and similar systems were not seriously considered because they are not convenient for the imperative and message-passing paradigms. Moreover, it is difficult in OPS5 to define rule sets on the fly, because rules are compiled; it is difficult to interface foreign programs; and switching actor contexts and rule sets requires cumbersome and inelegant expedients.

Smalltalk and Flavors both implement message-passing protocols. However, the only Flavors known by the author to be supported by 4.2 BSD is the Maryland package; and Smalltalk is not well supported by Berkeley Unix.

Performance requirements quickly remove languages such as Prolog from consideration. While they provide a good mechanism for unification-based programming they are less than effective for the imperative and object-oriented styles of programming, and there is no well supported way to invoke foreign programs.

LISP was the language eventually chosen, in its Franz [Wilensky 84] dialect, because it is well supported, its behavior is well known, it has a

large user community, the ways in which the message-passing, unification, and production systems can be implemented are documented in many textbooks, it is easily interfaced to any other Unix-supported program, it provides a very comfortable development environment, and other ADAM program development efforts are proceeding in Franz LISP at USC.

6.6.4. Completeness and Correctness

As with all nontrivial software, the design planning engine is not easy to verify. At best, a statement of the function and structure of the major subunits of the program (in this case, the knowledge bases and the interpreters of knowledge bases that form the layers of DPE) can be asserted to be correct independently of any flaws in the layers below, and empirical evidence adduced to bolster the argument that the function and structure at the high levels "does the right thing" under testing. In Chapter 7, which follows, such empirical evidence of correct operation is presented.

Chapter 7
Test Cases

7.1. Overview

In this chapter we will discuss the results of running DPE. In Section 7.2 we will begin by presenting the output from a typical test run, which will then be explained. Following that, in Section 7.3, the behavior of the system's interpolation strategy and the designs that result will be discussed for different sets of input constraints. In Section 7.4 we will discuss DPE's behavior when one or more operators is unusable due to input data semantics. Following that, we will discuss the ranking functions by which operators are selected for chaining, and the influence of importance (or "tightness") numbers on the way operators are chosen. In Section 7.6, we present two example design states, and give a detailed explanation of the design state representation formalism. Then we will examine two operator frames in detail.

We will use the actual notation of DPE for examples. This has been done in an effort to provide insight into the actual workings of DPE at the frame-language level. It should also serve as an introduction to the frame language examples of Appendix II. The frame language is build around lists, i.e. sets of one to five LISP atoms enclosed in parentheses. The atoms used have specific meanings, which are not always completely obvious, because of abbreviation and specialized usage. The examples will be accompanied by English translations.

Four special symbols should be noted: the semicolon, the dollar sign, the question mark, and the percent sign. The semicolon is the LISP comment character; all text following a semicolon on a line is ignored by LISP. The dollar sign, which may be the first character in an atom, e.g. the atom `$main`, means that the atom is a global variable. In particular, such globals represent names of DDS entities, which for convenience have been made globally accessible. However, not all globals or DDS entities are so marked; where a guaranteed freedom from name conflicts exists, DDS entities are given more ordinary names. The question mark, which also may lead an atom, means that the atom is a match variable subject to unification, as discussed in the previous chapter. The percent sign also may lead an atom, and it denotes a variable name corresponding to a construct created by DPE (as in the case of a DDS entity), which must be given a unique name.

7.2. General Capability

This section describes a complete test run of DPE. The test run was chosen in order to demonstrate some of the general aspects of DPE, and to give a framework for the details of the following sections.

In this as in all of the test runs, the system was started by invoking the LISP interpreter. The DPE source code and knowledge base are read in by the interpreter, which then loads the specification of the desired system. The specification is expressed in two files; one contains a DDS representation (in the current DPE only a dataflow graph is expected), and the other is a high-level description of the desired system, including constraints and goals, discussed in detail in Section 7.6. The high-level specification for the first example expresses the following requirements:

1. the desired system is a DDS component, known by the name `$main`, having a dataflow graph named `$main-df`,

2. the level of abstraction at which `$main` is to be designed is register-transfer (RTL), and

3. implementations of `$main` must meet the following constraints: area must be 1,000,000 units or less; power dissipation must be 1000 units or less; cycle time must be 400 units or less; and it may not take more than 30,000 units of time to design, as estimated in the postconditions of the scheduled operators.

Note that while the units of measurement are arbitrary, they have been chosen to be approximately equivalent to common measures of area, power, cycle time, and program run time; i.e. square lambda, milliwatts, nanoseconds, and seconds respectively.

The dataflow graph representing the behavior to be implemented consists of 24 nodes and 47 values. It is the same in all of the test cases. Its nodes correspond to RTL operations, and its values to RTL data values. Changing this dataflow graph would not substantially change the behavior of DPE, because DPE is not yet integrated with a powerful dataflow graph analysis program. The estimates obtained from the graph are admittedly crude, but they are sufficient for the kind of planning-space exploration DPE currently does.

DPE starts a run by issuing the prompt `0>`. Note that the prompt consists of a number with an angle-bracket concatenated onto it. The number is the "logical inference count" (LIC), i.e. the number of rules that have been fired so far. When the system is started, the LIC is zero, as shown. The run shown below is started by the user typing a number in response to the planner's prompt. The number the user types in to start DPE is the number of rules to be fired before DPE will stop. Note also that many of the trace statements (e.g. `"4: pinging area"`) are preceded by an LIC, which is helpful in debugging. When DPE stops, the user can restart or debug the

system.[19] The numbers to the right of each plan step are advisabilities; their significance in operator selection, and the operators that compete with the operators shown, are discussed in Section 7.5. This plan "constructs" a datapath and controller.

```
0> 10000
4: pinging area
78: analyze-df 100
176: setlib 5
321: maha 6
492: sel-ctrl 13
670: build-stg 10
862: build-ctbl 10
1072: build-ctrl 50
1151: estimated properties of state p^s37
Context p^s37
        (estimated dtime $main 1085)
        (estimated area $main 228000)
        (estimated ctime $main 1430)
        (estimated power $main 148)
```

Shown here is an initial pass of planning. This is a calibration plan, corresponding to one of the initial guesses described in the previous chapter. The quantity to be calibrated is area, as denoted by the trace message `4: pinging area`, which means that the system is trying to estimate the smallest possible implementation of the given dataflow graph, regardless of speed and other constraints. The plan so constructed is the sequence of operators {`analyze-df, setlib, maha`...} until at LIC 1072 the plan is complete, i.e. the conditions for plan completion in RTL have been met. By LIC 1151 the properties of the terminal state `p^s37`[20] have been estimated to be area 228,000; speed 1430; and so on. DPE now has a terminal state representing the smallest design implementing the given dataflow, and the sequence of operations (the plan) by which that design can be constructed.

That sequence consists of seven operators: `analyze-df` (analyze

---

[19] The user can also interrupt and restart the system at any point.

[20] An abbreviation for plan state **37**.

dataflow graph); `setlib` (choose a module library); `maha` (run the MAHA RTL allocator, which binds dataflow nodes to logical structure modules and time ranges in order to produce a minimum area design); `sel-ctrl` (select a controller style); `build-stg` (construct a structure graph, i.e. a schematic); `build-ctbl` (build a control table, e.g. a PLA program or microprogram); and `build-ctrl` (construct an actual controller.) Each of these operators adds some data to the hypothetical design state as the plan is built.

The statements *lic: operator number* mean that of all applicable operators, the one with the highest "advisability" is *operator*, and the advisability is *number*. The method of calculating these numbers is discussed in Section 7.7.

7.2.1. Search for the Fastest Design

Following the construction of a plan minimizing area, DPE explores the time dimension, to find the fastest design it can construct.

```
1174: pinging ctime
1251: analyze-df 100
1349: setlib 5
1494: sehwa-s 9
1665: sel-ctrl 13
1843: build-stg 10
2035: build-ctbl 10
2245: build-ctrl 50
2321: estimated properties of state p^s82
Context p^s82
        (estimated dtime $main 69515)
        (estimated area $main 8241000)
        (estimated ctime $main 24)
        (estimated power $main 3915)
```

This plan is essentially the same as the last one, with the main difference being that the operator `sehwa-s` is called (LIC 1494) instead of the operator `maha` (LIC 321). The effect of `sehwa-s` is to create a pipelined implementation where `maha` created a serial implementation. The difference in end state properties is large because of this and other, hidden differences that

is discussed in Sections 7.6 and 7.7. The cycle time is reduced, and the area is increased.

7.2.2. Interpolation

DPE's effort to characterize the minimum possible power will not be reported here; it is analogous to that for area and speed. Note that the minimum design time is not estimated by constructing a plan with minimum design time as its goal. That is, no effort is made to explicitly estimate the minimum possible design time. Instead, the design times estimated for other plans are used as input for the interpolation procedure. Design time is treated in this way because DPE's operator selection estimators (advisability estimators) do not behave well in the complete absence of other criteria (e.g. area, speed). Whether this is a fundamental problem or not is an open issue.

The results of an interpolation operation are shown below. Interpolation is performed by the **guesser** actor after all of the criterial quantities (excluding design time) have been explored.

```
(tight area 0.913229915500063)
(tight power 0.7738253251924608)
(tight ctime 0.732574679943101)
(tight dtime 0.5774514102002046)
```

In this case, new *tightnesses*, which are the implementation counterpart of importances, are interpolated from the estimated properties of the terminal states shown above. These numbers can be interpreted as meaning that the area of the desired design lies .91 of the way between the area of smallest design and the area of the fastest, and similarly for speed, design time, and power. If the designs constructible by DPE were uniformly distributed in the reachable design space, and the combination of prospective and retrospective estimators is monotonic, the next pass can be expected to result in a plan moderately close to the "best" plan DPE could construct. A graphical representation of the interpolation operation is shown in Fig. 7-1.

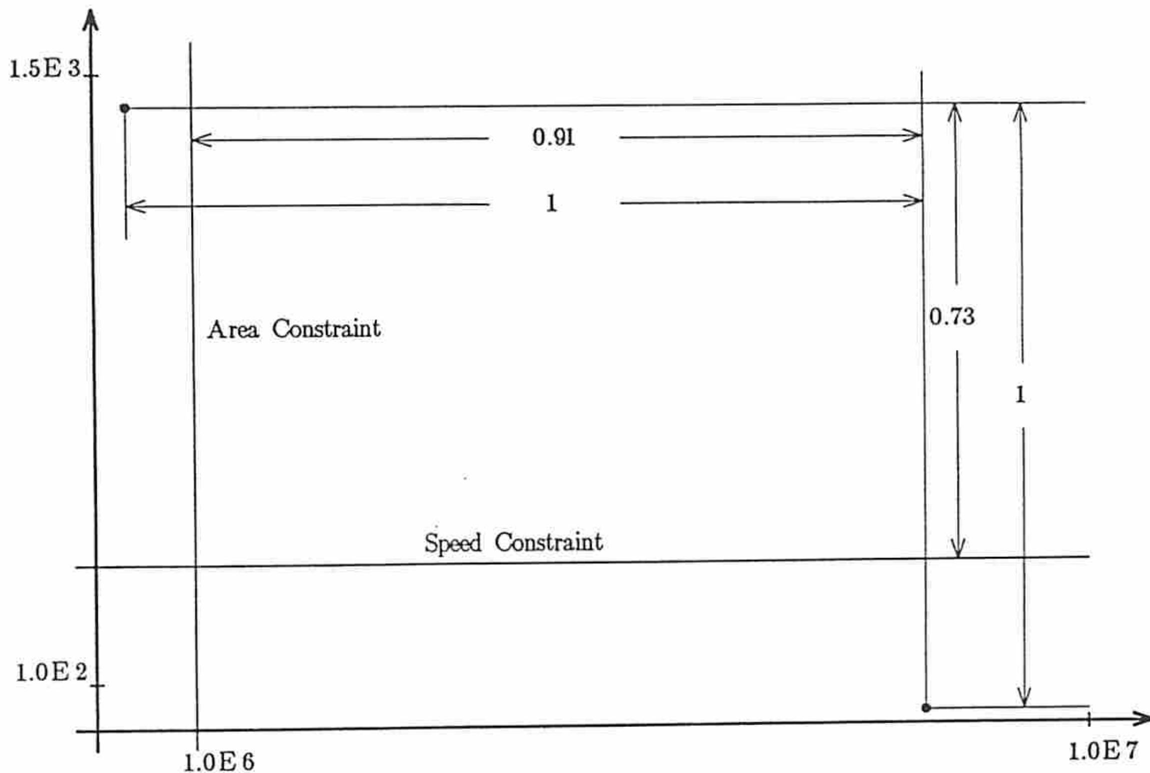


Figure 7-1: Interpolation.

```
3595: analyze-df 100
3693: setlib 5
3842: maha 14
4019: sel-ctrl 13
4203: build-stg 10
4401: build-ctbl 10
4617: build-ctrl 50
4697: estimated properties of state p^s172
Context p^s172
        (estimated dtime $main 903)
        (estimated area $main 228000)
        (estimated ctime $main 1430)
        (estimated power $main 142)
```

The plan constructed after the first interpolation is shown above. That plan is the same as the second one shown, except that the comparatively high importance of area and power resulted in the use of the `maha` operator, which constructs smaller and slower designs than `sehwa-s`. These two operators are

discussed in more detail at the end of this section. Note that the estimated results are far from the constraints in all dimensions. Note also that there is only a small difference between the results of this plan and the first one shown; that is because of thresholding effects in the interaction of importances and operator estimators. New importances are then calculated using the same interpolation strategy as before.

```
(tight area 0.913229915500063)
(tight power 0.7738253251924608)
(tight ctime 0.9284836981924651)
(tight dtime 0.333450131142205)
```

Note that as far as area and power importances are concerned, the experience of the latest plan is ignored; the new number is identical to the old one. This is because the same points are being interpolated. The other numbers are different, however, because the leaf state was far from the constraints on speed and design time, and the new point was distinct from the first in those dimensions. These importances generate a new plan.

```
4795: analyze-df 100
4893: setlib 5
5041: maha 12
5221: sel-ctrl 13
5408: build-stg 10
5609: build-ctbl 10
5828: build-ctrl 50
5908: estimated properties of state p^s217
Context p^s217
        (estimated dtime $main 903)
        (estimated area $main 2052000)
        (estimated ctime $main 382)
        (estimated power $main 907)
5917: stopping at decision to execute plan
5917> q
   1013.6 sec, 5917 rules 225 ops 5.837 lips 0.221 pops
```

The new plan, shown above, uses the same operator sequence as the last, but because of the increased influence of speed, the effects of each operator are modeled differently, resulting in a much greater area and power dissipation, but reduced cycle time.

The new plan is superficially identical to the previous plan, but operator effect modeling differences (possibly in more than one operator, an issue that is discussed below) have resulted in a hypothetical design that is closer to constraints than the last. Because this plan uses the same set of operators as the last, DPE then decides to execute the plan, to find out what the results would be in the execution space. At this point the example ends; DPE could not execute the plan, because it had no execution-space operators to invoke. A rule in the planner's knowledge base was therefore modified. The function of this rule is to mediate the transition from planning to execution, but a "stop" instruction was placed in its **Then-Part**, causing DPE to break and enter debugging mode at that point. The user chose to exit DPE, which resulted in the print out of DPE's performance data for that run, in CPU seconds, rules fired, plan steps chained, and respective rates for rule firing and operator chaining.

7.2.3. MAHA and Sehwa

In the plans of this and other sections, the main difference between plans is the datapath allocator used in the third (and in some cases the fourth) step of the plan. There are three possibilities, which will now be discussed.

First, the datapath allocator MAHA [Mlinar 86] is used primarily at the low-cost end of the spectrum. MAHA constructs nonpipelined datapaths. It does this by mapping out and allocating the critical path first, and then allocating noncritical portions of the design. It partitions the design into *time steps*, and then attempts to share hardware resources across different time steps. Hence the least expensive designs can be constructed using MAHA, simply by forcing a single time step per dataflow operation, and sharing a single ALU between all operations. It is also capable of partitioning the design

into fewer steps, with concomitant increases of performance and cost. The model reflects some of this flexibility by means of rule-based postconditions that state different degrees of serial execution for different importances. MAHA is capable of allocating datapaths where nested loops are executed on parallel execution branches in a straightforward way, as long as the inner loops are fixed (Section 3.7). If the inner loops are unfixed, MAHA can still be used, but the dataflow graph must be transformed.

The allocator Sehwa [Park 85a], [Park 85c] is a pipelined allocator. That is, it constructs pipelined datapaths, which are faster and more expensive than the datapaths constructed by MAHA. It is modeled as two separate operators in DPE's knowledge base: `Sehwa-f` and `Sehwa-s`. That is because Sehwa has two basic search modes: exhaustive and non-exhaustive. Exhaustive search is potentially very time-consuming, so the `Sehwa-s` operator, which represents Sehwa called with exhaustive search enabled, is modeled as having very long execution times and better results than `Sehwa-f`. In most of the plans given in this chapter, the operator `Sehwa-f` is preferred because of the design time constraint and the size of the problem. Note that if the problem were smaller or the design time constraint looser, `Sehwa-s` would be preferred more often. Both are preferred over MAHA when the problem calls for a fast and expensive design. Sehwa cannot be used where nested loops are present, instead the target dataflow must be transformed by unwinding the inner loops.

We will now return to the example run, in order to demonstrate the interpolation and replanning features of DPE.

7.3. Interpolation and Convergence

When the calibration runs are all finished, DPE begins to interpolate, as shown above. Succeeding interpolations are then constructed in an effort to converge on the best possible plan.

The following examples show the trajectory of the system as it attempts to converge in the example given above. The reader will recall that state 3, the exploration of minimum power, was not shown. In order to show the complete trajectory, we will also show the initial (calibration) states. The table has five major headings, representing respectively the *state* at the end of each plan, and the *area*, *cycle time*, *power* dissipation, and *design time* to construct that state. The four quantity columns are each subdivided into two, with importance numbers on the left and the estimated results on the right. The exact sequences of operators are not shown, because they differ in only one operator (the allocation step) and in the modeled effects of operators; the interested reader is referred to Sections 7.4 and 7.5, where more plans are given, for examples.

State	Area		Ctime		Power		Dtime	
(Plan)	Imp.	Est.	Imp.	Est.	Imp.	Est.	Imp.	Est.
1	1	2.3e5	0	1.4e3	0	1.5e2	0	1.1e3
2	0	8.2e6	1	2.4e1	0	3.9e3	0	7.0e4
3	0	3.1e5	0	2.9e3	1	7.0e1	0	5.2e2
4	.91	2.3e5	.73	1.4e3	.77	1.4e2	.58	9.0e2
5	.91	2.1e6	.93	3.8e2	.77	9.1e2	.33	9.0e2
Constraint	1e6		4e2		1e3		3e4	

The example above illustrates the following points. First, the system overshot its mark in going from the fourth to the fifth plan.

Second, by accident the estimated values of the fifth plan are close to the constraints; but the accuracy of the estimators used in these examples is low. The agreement between the area and speed estimates and the constraints is therefore a "lucky hit."

Third, the system settled at this point, i.e. plans generated after this were duplicates of plans 4 and 5, with only small variations in estimated properties. In other words, DPE could not arrive at a better plan.

The constraints lie in an "empty zone", a part of the design space that the planner cannot reach using the knowledge base it has. As a result, in such situations the planner would oscillate between the alternatives for some number of cycles, finally settling to the "best approximation". This behavior is not particularly interesting, so the rule that initiates execution, which is normally triggered by detection of a duplicated plan, was disabled by inserting a breakpoint, which is where the example ends.

The plan that DPE would have executed rule would have been plan 5, the last plan. This plan is not essentially different from plan 4; the operator sequence is the same. The difference in estimated properties is due to the sensitivity of the operator-effect models to changes in importances (e.g., the *model* of MAHA states that it will shift from a full serial to a more parallel implementation as speed importance increases.) This does not force such a decision on the actual operator MAHA, which is free to use either scheme as its algorithms dictate.

7.3.1. Effect of Varying Constraints

We will now turn to another example, in which the same dataflow graph was given to the system, but with different constraints, toward the low-speed, low-area end of the tradeoff curve. The following trajectory is the result.

State (Plan)	Area Imp.	Area Est.	Ctime Imp.	Ctime Est.	Power Imp.	Power Est.	Dtime Imp.	Dtime Est.
1	1	2.3e5	0	1.4e3	0	1.5e2	0	1.1e3
2	0	8.2e6	1	2.4e1	0	3.9e3	0	7.0e4
3	0	3.1e5	0	2.9e3	1	7.0e1	0	5.2e2
4	.95	2.3e5	.52	1.4e3	.88	1.4e2	.58	9.7e2
5	.95	2.3e5	.77	1.4e3	.88	1.4e2	.33	9.7e2
Constraint	7e5		1.0e3		1.5e3		3.0e4	

In this case, the planner was unable to meet the constraints because the required design lies in an "empty zone".

In the next example, the constraints have been moved up toward a higher-speed, higher-cost design. The **maha** allocator is still used in the last two plans, but because of the effect of importances on the operator postcondition, a substantial difference is evident.

State (Plan)	Area Imp.	Area Est.	Ctime Imp.	Ctime Est.	Power Imp.	Power Est.	Dtime Imp.	Dtime Est.
1	1	2.3e5	0	1.4e3	0	1.5e2	0	1.1e3
2	0	8.2e6	1	2.4e1	0	3.9e3	0	7.0e4
3	0	3.1e5	0	2.9e3	1	7.0e1	0	5.2e2
4	.79	2.1e6	.84	3.8e2	.72	9.1e2	.58	7.6e2
5	.79	2.1e6	.90	3.8e2	.65	9.1e2	.33	7.7e2
Constraint	3e6		2.5e2		1.2e3		3.0e4	

In the next example, shown below, the constraints have been set in such a way that **sehwa-f** and **maha** are almost equally matched. The result is a longer run, and an illustration of the weakness of DPE's execution heuristic, which is discussed below. In this table the plan numbers have been augmented with the tags **s**, **ss**, and **m**, to indicate whether the fast pipelined allocator **sehwa-f**, the exhaustive (optimal) pipelined allocator **sehwa-s**, or the nonpipelined allocator **maha** was used.

State (Plan)	Area Imp.	Area Est.	Ctime Imp.	Ctime Est.	Power Imp.	Power Est.	Dtime Imp.	Dtime Est.
1(m)	1	2.3e5	0	1.4e3	0	1.5e2	0	1.1e3
2(ss)	0	8.2e6	1	2.4e1	0	3.9e3	0	7.0e4
3(m)	0	3.1e5	0	2.9e3	1	7.0e1	0	5.2e2
4(s)	.60	4.2e6	.86	5.0e1	.69	1.9e3	.58	3.4e3
5(m)	.67	2.3e5	.75	1.4e3	.80	1.4e2	.35	6.8e2
6(m)	.67	2.1e6	.85	3.8e2	.80	9.1e2	.35	6.8e2
7(s)	.62	4.2e6	.85	5.0e1	.76	1.9e3	.35	3.4e3
8(s)	.62	4.2e6	.85	5.0e1	.79	1.9e3	.21	3.4e3
Constraint		3.5e6		2.0e2		1.3e3		3.0e4

In the fourth plan, `sehwa-f` was selected (the details of this plan will be discussed in Section 7.5.) In the fifth, with area more important and speed less so, the allocator `maha` was used. This resulted in a bad undershoot, so area and speed were interpolated again, with the result that `maha` was used again in the sixth plan. Using the ordinary execution heuristic of two duplicated plans after the beginning of interpolation, DPE fired the execution rule at this point, which caused the break and stopped DPE. However, the behavior of DPE at this juncture seemed interesting, so the breakpoint and execution rule were overriden, to produce plans seven and eight, in both of which `sehwa-f` was used. After plan eight there was no substantial change in behavior. Hence the two-duplicate-plan heuristic did not give the same results as waiting for the system to settle. On the other hand, it is still unclear what the result of using the real MAHA and the real Sehwa would be in this situation; hence the need for execution and then further interpolation using execution results.

The behavior of DPE under faster and higher-cost constraints is comparatively uninteresting, being analogous to the first run of this section, but with the pipelined allocators substituted for `maha`.

7.4. Unusable Operators

The previous section dealt with problems where three allocators (`maha`, `sehwa-s`, and `sehwa-f`) were usable, but the constraints caused one or another to be chosen. In this section we will examine a case where a dataflow graph unacceptable to the allocators is given, with the result that the graph must be rebuilt before allocation is possible. Rebuilding is simulated by the addition of an operator called `dftran` (dataflow transform) to the plan, which has as its postcondition the assertion of a modified dataflow graph. The operator `dftran` is needed only in such cases, so it was not present in any previous plan. Furthermore, it is modeled as being incompletely effective; it removes the condition that renders the graph completely unusable, but it does not remove it entirely, so that neither pipelined allocator can run. This occurs because the MAHA allocator can accept dataflow graphs in which loops are nested, but the Sehwa and exhaustive-Sehwa allocators cannot. None of the available allocators can successfully operate on an outer loop inside which another loop is nested concurrently with other operations; the `dftran` operation transforms the graph so that it is acceptable to MAHA but not Sehwa.

The simulation of such a situation in DPE demonstrates two related capabilities. First, if an operator is needed to enable another operator, it can be automatically added to the plan; and second, if an operator is known to be inapplicable it is not used.

First we will show the first two plans constructed. These plans are exploratory; they represent the smallest and fastest designs the planner is able to construct under the circumstances.

```
0> 4: pinging area
76: analyze-df 100
177: dftran 20
274: setlib 5
374: maha 6
488: sel-ctrl 13
609: build-stg 10
744: build-ctbl 10
897: build-ctrl 50
966: estimated properties of state p^s33
Context p^s33
        (estimated dtime $main 1109)
        (estimated area $main 228000)
        (estimated ctime $main 1430)
        (estimated power $main 148)
```

So far there is no difference between the first stages of this run and the last. However, because the pipelined allocators are unusable, the fastest design DPE can construct is much different (see, e.g., the example of Section 7.2.1.)

```
1064: analyze-df 100
1165: dftran 20
1262: setlib 5
1364: maha 3
1480: sel-ctrl 13
1603: build-stg 10
1740: build-ctbl 10
1895: build-ctrl 50
1956: estimated properties of state p^s72
Context p^s72
        (estimated dtime $main 539)
        (estimated area $main 9000000)
        (estimated ctime $main 198)
        (estimated power $main 2592)
```

The result of the fact that the pipelined allocator cannot be used is that the fastest possible design is one-eighth as fast as it would otherwise be. Note also that the operator `dftran` follows immediately after the dataflow analysis step `analyze-df`.

The trajectory of the set of plans constructed under these circumstances is as follows.

State (Plan)	Area Imp.	Area Est.	Ctime Imp.	Ctime Est.	Power Imp.	Power Est.	Dtime Imp.	Dtime Est.
1	1	2.3e5	0	1.4e3	0	1.5e2	0	1.1e3
2	0	9.0e6	1	2.0e2	0	2.6e3	0	5.4e2
3	0	3.1e5	0	2.9e3	1	7.0e1	0	5.4e2
4	.58	9.0e6	1.0	2.0e2	.50	2.5e3	.90	6.6e2
5	.82	4.2e6	1.0	3.8e1	.73	1.8e3	.82	8.1e2
Constraint	4e6		2e2		1.4e3		3e5	

An interesting effect, illustrated here, is that of operator masking. In this set of plans only the nonpipelined allocator operator was used, even when the goal was to try to construct a very fast design. In previous problems, the high end was reached by using the pipelined allocator, which for fast designs was preferred to the nonpipelined allocator. The pipelined allocator in effect masked out the potential high-end performance of the nonpipelined allocator. Note that where the form of the input made the pipelined allocators unusable, `maha` actually produced a slightly inferior result (compare with Section 7.3); but the unusability of the pipelined allocator did not cause the system to fail altogether.

7.5. Choice of Chainable Operators

In this section we will discuss the choice of operators to be chained. Recall that at each step of the plan, a number of operators may have satisfied preconditions, but under depth-first search, only one can be added to the plan (i.e. chained). The choice of which operator to chain is controlled by the heuristic choice function, which is expressed by the computation and comparison of advisability numbers.

We will now examine a test run in which reporting of advisability numbers has been enabled. These advisabilities are used for ranking operators; the operator with the greatest advisability in each step is shown with an asterisk to the right of its advisability. Note that an advisability of zero or even a negative advisability does not mean that the operator in question is unusable; the operator with the greatest advisability is chosen, even if the advisability is negative.

Note that the importances that generate this plan correspond to a probe of the minimum-area end of the design space.

```
                 Area importance           1.00
                 Cycle time importance     0.00
                 Power importance          0.00
                 Design time importance    0.00
                            Operator          Advisability
step 1                      setlib                 5
                            analyze-df           100        *
                            dftran                20
                            sel-ctrl               0
step 2                      setlib                 5        *
                            analyze-df          -100
                            dftran              -100
                            sel-ctrl               0
step 3                      maha                   6        *
                            sehwa-f                2
                            sehwa-s                3
                            analyze-df          -100
                            dftran              -100
                            sel-ctrl               3
step 4                      maha                 -94
                            sehwa-f              -98
                            sehwa-s              -97
                            analyze-df          -100
                            dftran              -100
                            build-stg              0
                            sel-ctrl              13        *
step 5                      (7 chainable operators)
step 6                      (8 chainable operators)
step 7                      (9 chainable operators)
```

In the test above, the first four steps of the plan are shown. The importances are given above the table. The "advisabilities" are computed by means of rule-based estimators, which were described in Section 6.4; detailed examples of advisability calculation rules are given in Section 7.7.

Note that the most interesting step is step 3, in which three operators have roughly comparable advisabilities. In other steps, what should be done is comparatively obvious. For example, the advisability estimator for `analyze-df` states that if the analysis has not been carried out, then the advisability is 100; otherwise it is -100, as illustrated in steps one and two. These large numbers override every other consideration in the first and subsequent steps, with the effect that `analyze-df` is always the first step in the plan. Note that these numbers would have to be adjusted if a competing dataflow analyzer were to be installed in the system. An alternative technique for accomplishing the same thing is to attach an additional clause to the operator precondition, to the effect that the analysis must not have been previously performed. There is a philosophical difficulty with this approach, however. The meaning of a precondition is that the precondition is met if the operator is possible in the situation given; and certainly dataflow analysis is possible at any time there is a dataflow graph. Hence the underlying semantics of all operator preconditions would be corrupted by adding such exceptions, and while such a trick would work in the short term, it might hinder the future establishment of more powerful reasoning mechanisms.

The next set of examples demonstrate the effects of changing importance numbers on the advisabilities generated in step 3 of the plan. All of the examples of this section were taken from the same test run, so together they form a sequence wherein the trajectory of the system is seen through the interaction of the importance and advisability numbers.

In the first plan, cycle time is being calibrated; hence the cycle time importance is set to one, while all other importances are zero.

```
                 Area importance          0.00
                 Cycle time importance    1.00
                 Power importance         0.00
                 Design time importance   0.00
                            Operator         Advisability
        step 3              maha                  3
                            sehwa-f               6
                            sehwa-s               9        *
                            analyze-df         -100
                            dftran             -100
                            sel-ctrl              3
```

The effect of the minimum-cycle time probe is to heavily bias DPE in favor of the optimal pipeline allocator **sehwa-s**, although the fast pipeline allocator **sehwa-f** also gets a high number. **Maha**, on the other hand, gets a lower rating. Note also that the control style selector **sel-ctrl** is comparable to **maha**. This does not necessarily mean anything but that control style selection is considered to have a strong effect on speed at this stage of the design process.

In the next case, the interpolation process has begun. The importances are as given.

```
                 Area importance          0.598
                 Power importance         0.694
                 Cycle time importance    0.861
                 Design time importance   0.577
        step 3              maha                 12
                            sehwa-f              13        *
                            sehwa-s              11
                            analyze-df         -100
                            dftran             -100
                            sel-ctrl              3
```

In this case, the operator **sehwa-f** is favored because of the high importance of speed, and the comparatively low importances of area and power. The predominance of both over **sehwa-s** is due to the influence of design time.

In the next case, the importance of speed and design time have been lowered and those of area and power raised.

```
                  Area importance            0.668
                  Power importance           0.804
                  Cycle time importance      0.755
                  Design time importance     0.345
step 3                       maha                      12      *
                             sehwa-f                   10
                             sehwa-s                   10
                             analyze-df              -100
                             dftran                  -100
                             sel-ctrl                   3
```

The increased importance of area and power results in a lowered advisibility for **sehwa-f** and **sehwa-s**. In the case of a tie, DPE chooses the first of the tied operators, on the rationale that the difference (in advisability) cannot be resolved in the planning space.

In the next plan, the importance of speed is increased, while area, design time, and power importances stay the same. The combined effect of these changes is to incrementally advance the two pipelined allocators.

```
                  Area importance            0.668
                  Power importance           0.804
                  Cycle time importance      0.848
                  Design time importance     0.345
step 3                       maha                      12      *
                             sehwa-f                   11
                             sehwa-s                   11
                             analyze-df              -100
                             dftran                  -100
                             sel-ctrl                   3
```

At this point the planner would normally stop, because a duplicated plan (i.e. the one using **maha**) has been detected. This means that either an oscillatory or a converging state has been reached, so there would normally be little purpose served by continuing to plan. In this case, however, the user forced it to continue.

In the next example, the importance of area and power have again decreased, in response to the relatively good area and power performance of maha. The importance of cycle time, however, has increased slightly, and that of design time remains the same.

```
                Area importance             0.620
                Power importance            0.762
                Cycle time importance       0.854
                Design time importance      0.345
        step 3             maha                         10
                           sehwa-f                      11        *
                           sehwa-s                      10
                           analyze-df                 -100
                           dftran                     -100
                           sel-ctrl                      3
```

In the next case, the increased importance of power results in decreased advisability for both sehwa-f and sehwa-s; however, this is offset in the case of sehwa-s by the lower importance of design time and sehwa-s's better results (recall that sehwa-s does exhaustive search.) Maha, on the other hand, suffers because a large part of its advisability in the previous cases was due to the influence of design time. The result is a tie between the pipelined allocators, of which the first would be chosen.

```
                Area importance             0.620
                Power importance            0.788
                Cycle time importance       0.854
                Design time importance      0.345
        step 3             maha                          9
                           sehwa-f                      10        *
                           sehwa-s                      10        *
                           analyze-df                 -100
                           dftran                     -100
                           sel-ctrl                      3
```

At this point the planner was halted, because no further interesting behavior could be expected.

7.6. Design States

The planner adds operators to the plan by matching preconditions to assertions in the current design state. Once the operator is appended, a new design state is created, which is a modified copy of the old one, the modifications being the result of applying the operator's postcondition. A *design state* represents a complete or incomplete design, with constraint and (possibly) historical information added. In the planning space, such design states are highly abstract, as opposed to the highly detailed design states of the DDS.

The structure and content of planning-space design states will now be described. A planning space design state (henceforth, *state* for short) consists of a collection of assertions. These assertions for the most part describe attributes of the design, but some also describe the names and locations of DDS entities. The state description given below is the specification used in the example of Section 7.2 above, i.e. the root state of the plans illustrated in that section.

```
; symbolic specification part file
(dfoa $main)                        ; task is to design $main
(compt $main)                       ; $main is a component
(dfg $main $main-df)                ; dataflow is called $main-df
(constraint $main area 1000000)     ; area constraint
(constraint $main power 1000)       ; power constraint
(constraint $main ctime 400)        ; cycle time constraint
(constraint $main dtime 30000)      ; design time constraint
(level $main rtl)                   ; level of abstraction desired
(number dtime 0)                    ; initialization
```

This specification consists of nine lists. Each list is an assertion. Note that a specification or design state can be composed of any number of assertions. All text following semicolons is ignored by the software; semicolon is the LISP comment character.

This specification describes a component (abbreviated `compt` in line 3) named `$main`, which is the design focus of attention (`dfoa`). This is interpreted to mean that the thing to be worked on is `$main`, as opposed to other components, dataflow graphs, or other DDS entities that might be mentioned in a state.

The dataflow graph (`dfg`) of `$main` is called `$main-df`, as expressed in the third assertion. Note that this is the name of a DDS dataflow graph. Hence the DDS component `$main` must have a dataflow model named `$main-df`. In that sense there is redundancy between state models in the planning space and in the execution space.

The next four lines describe the constraints that apply to `$main`. These assertions state that the area of `$main` should be less than or equal to 1,000,000 units. The units are presumed to be approximately equivalent to square Lambda, in order to increase understandability, but are otherwise arbitrary. In order to use different units, some translation rule would have to be added to the design knowledge base, as it constructs area estimates measured in equivalent units. The other constraints are similar, being expressed in units approximately equivalent to milliwatts, nanoseconds, and seconds respectively.

Following the constraints is the assertion `(level $main rtl)`, which states that the level of abstraction at which `$main` is to be designed is the register-transfer level. This assertion acts as a key by which a set of frames is selected. The current frame set has only RTL operators and descriptions, but if it had others their presence would be transparent because of this assertion.

The next assertion states that a number dtime, representing the current hypothetical time spent in designing the dfoa, is zero. This number is updated as the design time effects of different operators are modeled. This assertion initializes the design time calculation.

7.6.1. Terminal State Model

The design state of the previous example is quite simple. We will now turn to the terminal states of the example of Section 7.2. This state represents the end of a plan, so it is an implicit record of all of the transformations applied to the specification. It has been broken up into manageable chunks interspersed with explanatory text.

```
(estimated power $main 907)
(estimated ctime $main 382)
(estimated area $main 2052000)
(estimated dtime $main 903)
(number dtime 903)
```

The assertion (estimated power $main 907) means that the planner has estimated the power dissipation of the design focus of attention (dfoa) $main to be 907 units. Similar assertions express the estimated cycle time, area, and design time of the terminal state. These estimates are wholly derived from the remainder of the state information, by means of the rule-based state estimators described in Chapter 6. Note that though the numbers given may have many digits, the precision of the current estimation process is poor; hence the extra digits are meaningless. One enhancement to the system as it now stands would be the addition of more precise estimators.

The assertions following the estimated properties concern the controller hardware added during the later phases of the plan.

```
(subunit $main ctrlr31)
(hdwe ctrlr31 ctrl)
(implemented ctrlr31)
(c-tbl $main ctbl8)
(c-styl ctrlr31 pla)
```

These state, in order, that there is a `subunit` of `$main` named `ctrlr36`; that it is of hardware (`hdwe`) type control (`ctrl`); i.e. it is a controller of some kind. The next assertions state that it has been implemented; that it has a control table (i.e. an abstract microprogram or timing graph) named `ctbl8`; and that it is implemented using a PLA.

The assertion that it is a PLA style controller is worthy of note. This assertion is one of three possibilities, all contained in the model of controller style selection, as expressed in the controller style selection operator's postcondition. That model uses a rule set to express its postcondition. The rules determine the style of controller that will be present in the hypothetical design. Such assertions are dependent on the importances and facts of the design state, and are hypotheses about the effect of the design tools in the given situation.

```
(stg $main stg17)
(op-c $main 9)
(arch $main mux)
(semi-serial $main)
(nmrb $main nmrb62)
```

Following the assertion of the controller's attributes, the assertion `(stg`

The assertion following the structure graph is an estimated operator count (`op-c`). Such estimates are derived in much the same way that the PLA controller style is, i.e. by means of rule-based postconditions. In the case of hardware counts, the postcondition rules base the estimate on the dataflow graph node and value counts, the critical path length (if the design is parallel or pipelined), and the basic architecture (e.g., whether it is pipelined or serial, mux- or bus-oriented) constructed by the other operators.

The basic architecture in this case is semi-serial and mux- rather than bus-oriented. This is asserted by the lists `(semi-serial $main)` and `(arch $main mux)`. These assertions are automatic results of the application of the MAHA allocator, which constructs such designs. However, the architecture is not presumed to be maximally parallel. This is expressed by the assertion `(semi-serial $main)`, which states that `$main` uses some operator sharing.

The last effect of the allocator model is the assertion `(nmrb $main nmrb62)`, which states that a set of operation bindings (node-module-range or **nmr** bindings in the planner's parlance) is part of `$main`, and the set of bindings is named `nmrb62`. These bindings relate behavior to structure and are produced during RT synthesis.

```
(number av-mux-ct 5)
(number av-reg-ct 5)
(number av-mod-ct 20)
(number av-mux-sz 8000)
(number av-reg-sz 10000)
(number av-mod-sz 40000)
(number power-factor 4)
(mod-l $smalllib)
```

The next six assertions give estimated hardware costs and speeds. For example, `(number av-mux-ct 5)` states that the average multiplexer a propagation delay is 5 units. Similarly, `(number av-mux-sz 8000)` states

that the average size of a multiplexer input is 8000 units. DPE uses such information, in conjunction with the statement that `$main` has a multiplexer-oriented architecture, in estimation of the design's area, speed, and power. The assertion following that is a multiplier for power calculation; it is set differently for different module library options. These assertions are all postconditions of the module library selection operator, which also made the assertion `(mod-l $smalllib)`, the statement that the module library to be used (`mod-l`) is named `$smalllib` (for **small lib**rary), i.e. the area-efficient module set. Provided that this decision is not retracted at execution time, such estimates can be considered to be highly accurate, because they are characterizations of existing components. Averages were used within a library in an effort to simplify the models of area and speed.

```
(unco-plel $main-df)
(no-plel-loop $main-df)
(cpath $main-df nodes 9)
(number $main-df nodes 24)
(number $main-df values 47)
(tight dtime 0.333450131142205)
(tight ctime 0.9284836981924651)
(tight power 0.7738253251924608)
(tight area 0.913229915500063)
```

The next five assertions are predicates on the dataflow graph `$main-df`. The first of these, `(unco-plel $main-df)`, is a precondition for the MAHA allocator, having the meaning that the dataflow graph has no nested unfixed loops. The second, `(no-plel-loop $main-df)`, is an analogous precondition for the Sehwa pipelined allocator (note that as mentioned above, these are not the results of an actual analysis of the dataflow graph; rather they are the result of a triggering condition inserted in the root state in order to simulate detection during analysis.)

Another dataflow analysis result is (cpath $main-df nodes 9), which states that the critical path of $main-df has nine nodes. Again, this is not the result of actual analysis, but is a heuristically computed part of the postcondition of the dataflow analysis model, which is not attached to any real code. The assertions (number $main-df nodes 24) and (number $main-df values 47), on the other hand, are results of counting the nodes and values in the actual graph.

The analysis of the dataflow graph (numbers of nodes and values) is presumed to be exact, and to have been carried out during planning. Without such an analysis the basis for planning would be too meager to expect good results. This is an example of a case where planning and execution should be mixed.

Following the dataflow analysis "results" are the importance (or "tightness") numbers for design time, cycle time, power, and area respectively. These are the output of the interpolation process, and represent the conditions under which the state was generated. The effect of the relatively high importances of cycle time and area can be seen most clearly in the assertions dealing with the architecture and the module library. Again, the number of digits in these numbers does not reflect anything but the computational precision of the LISP interpreter.

```
(level $main rtl)
(constraint $main dtime 30000)
(constraint $main ctime 400)
(constraint $main power 1000)
(constraint $main area 1000000)
(dfg $main $main-df)
(compt $main)
(dfoa $main)
```

The remainder of the state model is the original set of specification assertions, which have been carried along from the root state because no postcondition ever deleted them.

7.7. Modeling Operators with Frames

Frames are used in DPE for two purposes: modeling operators and modeling generic hardware classes. An operator model frame (also called a *task frame*) in DPE's knowledge base is a machine-readable representation of an operator's preconditions, postconditions, invocation apparatus, and advisability estimation apparatus.

We will now examine the modeling of two operators in detail, so that the origin of previous example plans and state models will become clear. The first operator we will examine is the operator `analyze-df`, because it illustrates both the strengths and weaknesses of DPE. We will then turn to the operator `maha`, based on the MAHA [Mlinar 86] allocator program.

7.7.1. Task Frame Analyze-DF

This section covers the frame describing the operator `analyze-df`, the dataflow graph analysis tool. The dataflow graph analysis tool is a program that counts nodes and arcs in a dataflow graph, and extracts the critical path as well, passing its size to the planner. It is presented in fragments in order that commentary can be interspersed.

```
(task analyze-df          ; the name of the frame
  (level (rtl))           ; level of abstraction at which it works
  (invar ((dfoa ?it) (dfg ?it ?itsdfg))) ; precondition is the dfg
```

The `task` field gives the name and type of the frame. The name in this case is `analyze-df`, and the type is `task`, i.e. the frame represents an

operator model. Other frames can refer to this one as being a member of the set of **task** frames, or by its name.

The **level** field contains the single atom **rtl**, meaning that the level of abstraction for which this operator is appropriate is the register transfer level. The keyword **rtl** can be used to index other, related frames, or to construct the set of frames containing **analyze-df**.

The **invar** field contains assertions in the form of lists that must be matched in the design state for this operator to be considered applicable (i.e. the precondition, also called the **invar**iant, in the notation of [Fikes 71].) The assertions of the invariant lists must each match some assertion in the design state, and furthermore the match variable **?it** must have a *consistent binding*, i.e. **?it** unifies in the design state (see Section 6.3). The meaning of the lists is that there must be a known design focus of attention (**dfoa**), and that the **dfoa** must have a known dataflow graph. Hence the assertions **(dfoa $main)** and **(dfg $main $main-df)** in the state examples given above will match the invariant, thus enabling this operator.

```
(Add-Rules (
  (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
       (number ?itsdfg nodes ?n))
      (setr dtime ?n 5 * ?D +)) ; rule to adjust dtime
  (F1 ((nor (number ?itsdfg values ?any)) (dfoa ?it)
       (dfg ?it ?itsdfg))
      (ask counter count values ?itsdfg)) ; count the values
  (F1 ((nor (number ?itsdfg nodes ?any)) (dfoa ?it)
       (dfg ?it ?itsdfg))
      (ask counter count nodes ?itsdfg))  ; count the nodes
  (F1 ((nor (cpath ?itsdfg nodes ?any)) (dfoa ?it)
       (dfg ?it ?itsdfg))                 ; counter fakes this
      (ask counter cpath nodes ?itsdfg))  ; count critical path
```

The **Add-Rules** field contains the postcondition of the operator, expressed in the form of rules. Each of the rules shown here begins with the key **F1**, meaning that it may fire only once in a given context. Following that

is the **If-Part**, which may begin with a **no-** or **nor**-clause; following the **If-Part** is the **Then-Part**, as described in Chapter 6.

The first rule of the `Add-Rules` field updates design time (`dtime`), while the second and third count the nodes and values of the dataflow graph respectively. The fourth simulates counting the critical path; the actor `counter` actually just divides the node count by a constant.

```
(F1 ((nor (nested-unfixed ?itsdfg)) (dfoa ?it)
      (dfg ?it ?itsdfg))       ; fake detection of no nested
     (no-plel-loop ?itsdfg)) ; loops. This Rule Must Go.
(F1 ((no (maha-unsuitable)) (dfoa ?it)
      (dfg ?it ?itsdfg))       ; fake det. of unfixed, nested
     (unco-plel ?itsdfg))     ; loops. This Rule Must Go.
          )) ; end of the postcondition of analyze-df.
```

The two rules given above are the means by which detection of unsuitable dataflow graphs is simulated. If no assertion to the effect that there is no nested unfixed loop is present, the first rule fires, and states that there is such a loop: and similarly for the second.

Execution of such rule-based postconditions causes the construction of complex and data-dependent assertions for the result design state. This set of rules begins with rules that count the number of nodes and values in the dataflow graph, by means of a Flavors call `ask` to a low-level actor `counter`. The `counter` sends a message to another low-level actor, the `DDS-manager`, and gets in reply the internal form representation of the DDS dataflow graph `$main-DF`. The `counter` is also asked for the number of nodes on the critical path `cpath`. Note that while the `counter` actually counts nodes and values in the input dataflow, it does not actually extract a critical path and count those nodes; instead it returns a fixed fraction of the original node count. This is an area where the program is incomplete. (A critical path finder and synthesis

programs do exist as unintegrated parts of the system.) The next two rules also represent "stubs" where actual program code (to detect pathological dataflow graphs) has yet to be constructed and integrated into the system.

```
(Application-Rules (
      (F1 ((dfoa  ?it) (dfg ?it ?itsdfg)) ; essentially always
          (and (success)                  ; assume success
               (results ((number ?itsdfg nodes 24)
                         (number ?itsdfg values 32)))))))
```

The `Application-Rules` field in this case contains a dummy rule inserted for testing purposes, and to prevent DPE from failing when no such rule is found during execution testing. The rule given prevents DPE from assuming failure during execution testing, which is still at an incomplete stage. The rule also tests the ability of the planner to correctly deal with execution results that do not agree with planned results; the assertions resulting from the application rule firing do not agree with the results returned by the counter during testing.

```
(Estimators (all (
  (F1 ((dfoa ?it) (dfg ?it ?itsdfg))
      (setr advis 100)) ; if it exists and has a dfg
  (F1 ((dfoa ?it) (dfg ?it ?itsdfg)
       (number ?itsdfg nodes ?M)
       (number ?itsdfg values ?V))
              ; if the dfg has already been
              ; analyzed, overwrite the old advis.
      (setr advis -100)))))
```

The `Estimators` field contains the operator estimator for `analyze-df`. Note that the estimator produces "advisability" numbers rather than estimates of area, speed, power etc. These numbers, taken with the importance numbers, are closely related to more direct estimates, which are more difficult to generate on the basis of partial designs. The same arguments on monotonicity can be applied here as well, although in a less tractable form. The implementation of DPE uses this technique rather than the more difficult operator estimators of Chapter 5.

The **Estimators** field can contain more than one subfield, but in this case contains only the subfield **all**, which means that the rules will always be applied at planning time. The rules shown here are both intended to estimate the quantity **advis**, which is the **advis**ability of using this operator in a given situation. The first rule states that if the dataflow graph of the **dfoa** is known then the advisability should be set to the comparatively high value of 100. Note that it is triggered by only two clauses, i.e. the assertion that the **dfoa** is known, and that the **dfoa** has a known dataflow graph. The second states (in effect) that if the analysis has been performed already, the advisability should be set to the very low value of -100. Note that this rule detects that the analysis has been performed by its third and fourth clauses, which match results of analysis (i.e. the node and value counts, which are derived in no other way). This usually prevents the analysis from being performed twice on the same graph, which would be a waste of time. Note that it does not prohibit reanalysis; it just assigns it a very low advisability so that almost anything else will be done in preference.

```
(subtasks (none)) ; this task uses application rules
(hardware (generic-rtl))) ; end of task frame analyze-df
```

The **subtasks** field is intended to be used to point to hierarchical descendants of the current task, for hierarchical planning, as in [Sacerdoti 73]. It also can be used to point to procedural program code, providing a redundant mechanism where application rules might not be desired. Because neither execution nor hierarchical planning is actually done, this field is really a "stub" that permits testing.

The **hardware** field contains a set of pointers to hardware frames. These frames are taken to be those that describe kinds of hardware for which dataflow analysis of this kind is relevant.

7.7.2. Maha Task Frame

We will now turn to the frame **maha**, which is the model of the MAHA datapath allocator. This frame has been edited by the addition of extensive comments, and by the deletion of the **Application-Rules**, **subtasks**, and **hardware** fields, which are substantially the same as those in the previous example.

```
(task maha              ; MAHA datapath allocation task.
  (invar (              ; all of the following must exist
    (dfoa ?it)          ; a particular hardware unit
    (dfg ?it ?itsdfg)   ; its dataflow graph
    (unco-plel ?itsdfg) ; the dfg may not have conds in parallel
    (mod-l ?mod)        ; a known module library
         ))             ; end of invar field of maha
```

The **task** field give the name of the frame, which is **maha**, and denotes that it is an operator model. The other frame type currently used by DPE is **hdwe**, denoting a hardware description frame, as described in Chapter 6. DPE is capable of reading and accessing a **style** frame as well, but the planner rules do not use or mention such frames.

The **invar** field, whose function is to express preconditions, contains four lists. As in the previous example, there must be a **dfoa**, and the **dfoa** must have a known dataflow graph. The dataflow graph must further be asserted to have no unfixed loops nested within parallel branches (**unco-plel**), and there must be a known module library (**mod-l**). If there is no set of variables {**?it**, **?itsdfg**, **?mod**} that unifies these lists, the operator **maha** is considered inapplicable.

```
(Add-Rules (                    ; postcondition rules
  (F1 ((dfoa ?it))
      (ask manager make-new (nmrb ?it %nmrb)))
        ; add a NMR binding structure
  (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
       (number ?itsdfg nodes ?n))
      (setr dtime ?n 5 * ?D +)) ; update design time
  (F1 ((no (tmp decided-timing)) (tight area ?A)
       (tight power ?P) (tight ctime ?C))
      (and      ; do all of the following
          (setr fast-advis ?C 100 *)
          (setr serial-advis ?A 100 *)
          (setr ser-advis ?P 100 *)))
```

The `Add-Rules` field contains a total of eleven rules. The first rule simply adds a set of operation (node-module-range or NMR) bindings to the hypothetical design state. The second updates the design time estimate, by means of the call to `setr`; its effect here is $D := D + 5n$, where D is design time, and n is the number of nodes in the dataflow graph, as indicated by the clauses `(number dtime ?D)` and `(number ?itsdfg nodes ?n)`. When unification is performed, the variables `?D` and `?n` will be replaced by numbers, which are then passed to `setr` for postfix evaluation. The third rule derives a set of scaled importances that will be used to estimate the results of the allocation, i.e. in the rule below.

```
(F1 ((number fast-advis ?F) (number serial-advis ?S)
     (number ser-advis ?T))          ; predict results
    (and    ; assert all of the following
       (not (number fast-advis ?F)) ; remove trash
       (not (number serial-advis ?S))
       (not (number ser-advis ?T))
       (setr parad 1.5 ?T ?S + / ?F >=) ; setr is postfix!
       (setr serad ?F ?T > ?F ?S > or)))
```

The fourth rule performs comparisons between the scaled importance numbers, e.g. the Boolean **parad** (**par**allel **ad**visability), which is set true if the sum of area and power importances is less than 1.5 times the speed

(ctime) importance, i.e. $parad := A + P < 1.5S$. **serad** is defined as $serad := A > S$ OR $P > S$, where A is area importance, and P and S are respectively power and speed importance. Note that this choice of threshold points was essentially arbitrary.

```
(F1 ((boolean serad t) (dfoa ?it)) ; if serad is true
    (and (serial ?it)              ; make it serial
         (not (boolean serad t))
         (op-c ?it 1)              ; op count 1
         (arch ?it bus)            ; bus architecture
         (tmp calc-regs)           ; need to calc. regs
         (tmp decided-timing)))
(F1 ((no (tmp decided-timing))  (dfoa ?it)
     (boolean parad t)
     (dfg ?it ?itsdfg))
    (and (parallel ?it) ; make it parallel
         (not (boolean parad t))
         (arch ?it mux) ; mux architecture
         (tmp decided-timing)))  ; flag variable
```

The rules given above use the Booleans **serad** and **parad** as the basis for asserting that the design will be either serial, semi-serial, or parallel, and whether it will be bus-oriented or multiplexer-oriented. Hence the postcondition rules are a way to estimate the effect of applying MAHA in a wide range of circumstances. These rules were constructed primarily for purposes of testing and debugging, and to demonstrate the power of rule-based postconditions. By adding an appropriate set of rules, an expert system could be built into the postcondition in this way.

Of the following rules, the first two clean up the state model by deleting temporary assertions. After that is a rule that estimates operator count in the case where the design is parallel. The rule after that is another cleanup rule, which deletes an unnecessary assertion.

```
(F1 ((boolean serad ?either))
    (not (boolean serad ?either)))
(F1 ((boolean parad ?either))
    (not (boolean parad ?either))) ; garbage
(F1 ((parallel ?it) (dfoa ?it)
     (cpath ?itsdfg nodes ?n) (dfg ?it ?itsdfg)
     (number ?itsdfg nodes ?m))
    (setr opcount ?n 2 ?m / +)) ; estimate operator count

(F1 ((number opcount ?n) (dfoa ?it))
    (and
        (op-c ?it ?n)                ; restate operator count
        (not (number opcount ?n)))) ; remove garbage
```

The last rule of the postcondition is a default timing result estimator. If no other timing hypothesis exists, this rule will make the assumption that the timing is semi-serial, i.e. that it is neither fully serial nor fully parallel; it will estimate the operator count on the basis of the critical path cardinality.

```
(F1 ((no (tmp decided-timing)) ; if no timing known
     (cpath ?itsdfg nodes ?n)  ; make default assumption.
     (dfg ?it ?itsdfg) (dfoa ?it))
    (and (semi-serial ?it)  ; assert it's a compromise
         (arch ?it mux)            ; mux-based
         (op-c ?it ?n)
         (tmp decided-timing)))))  ; end of add rules
```

The next field of the frame is the `Estimators` field. This is the operator estimator for `maha`.

```
(Estimators (all (              ; estimation rules
  (F1 ((dfoa ?it) (nmrb ?it ?nmrb); allocation done?
       (number advis ?a))
      (setr advis 100 ?a -))  ; it's a waste if so
  (F1 ((tight dtime ?D) (number advis ?a)
       (dfoa ?it) (nmrb ?it ?nmb)); if design time is tight
      (setr advis ?D 20 * ?a -)) ; even more waste
  (F1 ((tight dtime ?D) (number advis ?a))
              ; if design time is tight
      (setr advis ?D 8 * ?a +)) ; then this is a good choice
  (F1 ((tight ctime ?C) (number advis ?a))
      (setr advis ?C 3 * ?a +))
              ; it is less good if cycle time is a problem
  (F1 ((tight area ?A) (number advis ?a))
      (setr advis ?A 6 * ?a +))
              ; it is good if area is a problem
  (F1 ((tight power ?P) (number advis ?a))
      (setr advis ?P 5 * ?a +)))))
              ; or if power is a problem.
```

The `Estimators` field contains rules that are always used, as indicated by the `all` keyword. All of these rules are for the calculation of advisability. The first states that if the allocation has already been done, as indicated by the presence of NMR bindings (`nmrb`), then the application is a waste of time and hence inadvisable. The second adds a further penalty for re-application, proportional to the importance of design time (`dtime`). The third and following rules calculate a weighted sum, where the weighting coefficients are built into the rules, and the numbers by which they are multiplied are the importances of cycle time, area, design time, and power. Note that the rules could easily be changed to implement another advisability function.

The third rule multiplies a positive weight by the importance of design time. This reflects a perception that MAHA is a fast program, capable of giving results quickly. The rule following that states the weight to be assigned to cycle time, followed by area and power weightings. The numbers and formulae expressed in these rules are in some sense "expert knowledge"; they are not arbitrary, but are chosen to reflect the judgements of the user.

The remainder of the frame set is presented in Appendix II, to which the interested reader is referred. The frames in that appendix are syntactically similar to the two presented here, so this chapter serves as a guide to the frames presented in the appendix.

Chapter 8
Conclusions

This thesis has described a model of an approach to the digital VLSI design process. The model is based on the idea of representing not only designs, but the effects of operations on designs, i.e. the transformations that send preliminary specifications to final implementations. But what does the model buy us? How does it help us to design systems? And what insights can we draw from the model and the nature of the model? This chapter attempts to address, if not completely answer, these questions.

To begin with, this model was intended to serve as a guide in the making of implementation decisions during the process of constructing a unified design automation system (UDAS). As such, it is necessary for the model to have some predictive power: it should give a clear understanding of choices that would otherwise be made on the basis of guesswork, tradition, or ad-hoc reasoning. It should provide some insight into the design process: why things happen the way they do, what the underlying reasons for common cliches and traditions are, and what questions we can ask to extend our understanding. It should also serve as a kind of mental shorthand: a way of thinking about things that saves effort by sharpening important distinctions and allowing us to ignore the merely incidental and the temporarily irrelevant.

Let us begin with a summary of the problem the model is intended to help us with.

8.1. The UDAS Problem: Summary

In the introduction to this thesis a particular kind of program, the *unified design automation system* (or UDAS), was discussed. Such a system would be able to map from an initial specification to a final implementation. It could include tools from many sources and could be used to implement many different kinds of target designs. It should allow a smooth tradeoff between design cost and design quality; and it should be possible to build and maintain the UDAS itself at a reasonable cost.

8.1.1. The Fractured Design Space

A major source of difficulty in the construction of a UDAS is the discrete nature of the design space. This comes about because of differences in implementation technologies, design styles, numbers and types of submodules, and differences between the tools and methods used to construct designs. The various choices interact in ways that are not always easily predictable, and the number of potentially useful combinations is large even for small designs. Even the importance and effect of single variables is not constant from one design problem to the next.

8.1.2. Tool Grain Size

Another problem is that of the size of tools. Large tools, i.e. those that cover a large part of the design process, tend to be comparatively clumsy and inflexible; small tools can be invoked in more complex patterns, but someone or something must make the decisions that chain them, and humans are apt to run out of patience and to make mistakes.

8.1.3. Tool Ordering

The order in which tools should be invoked is variable and problem-dependent. This is primarily a consequence of the discrete nature of the design space, but it can also be a result of partitioning, i.e. discreteness in the design itself. Tool ordering is made even more difficult to predict by the "mixed-bag" approach to the tool set; different tools may have very different postconditions, resulting in radical differences between chains. The problem of tool ordering is exacerbated even further by that of the tool grain size; where small tools are used, the number of possible tool orderings explodes.

8.1.4. Interaction

Humans, it has been noted above, tend to become impatient and to make mistakes when many small decisions have to be made. Hence there is a strong motivation to shield the human designer from the decision-making process. On the other hand, too much shielding results in a loss of design quality. It is as inadvisable just to draw the boundary at an arbitrary level of abstraction as it is to draw it around some structure of the design; and yet human intervention should be kept structured so that the system has at least a chance of understanding the situation.

8.1.5. Verification

When an error has been introduced, either by a program or by a human, it is advisable to find it and correct it quickly. This should be done by programs wherever possible. However, an error in one situation is not an error in another; the checking programs must be well-suited to the problems at hand, and they must be invoked at appropriate times. If errors are detected, some remedial action should be taken, but again this is highly situation-dependent.

8.2. The Model

The model of the design process described in Chapter 5 is based on the idea that a design progresses in steps, from an initial specification to an eventual implementation. Each step is an operation on the design, and is carried out by a tool. The tools then have to be characterized in terms of preconditions and postconditions; this leads to a STRIPS-like model which is both programmable and conducive to analysis.

8.2.1. Syntactic and Semantic Compatibility

The model assumes no *a priori* indication of the ordering of tools. Hence tool ordering is a function of particular design states and objectives. Thus the output of every tool is potentially the input of every other tool; and this in turn implies that tools should use a common representation for design information. The DDS provides this "syntactic" compatibility between tools.

The DDS has a number of important properties as a central organizing model for design data, which augment its capabilities as a syntactically compatible interfacing medium. These include

- the uniformity of specifications, targets, and library elements,
- the ability to represent both data-transformation and timing behavior separately from any structure,
- the possibility of embedding the representation in a data base management system (DBMS),
- the ability to separate the design data from the code that implements design transformations, and
- semantic redundancy through which verification and validation techniques can be applied.

Just because the form of the data passed from one tool to another can be assured to be compatible, however, does not mean that the meaning of the data is compatible. Again, we do not know what the details of the design will be until we see them; this means that checking for the appropriateness of operators should be done at design time, using preconditions attached to the operators being considered. This gives us a "semantic" measure of compatibility.

In addition to preconditions, we model the results of operator application by creating explicit but abstract models of design states. This allows us to construct *chains* of operators by matching preconditions against modeled design states. These design state models can be used for estimation and further chaining, and they can often be constructed at a far lower cost than the actual execution-space states.

By building a more perspicuous model of operators into the simulation, we can ask questions about execution time as well. This can be important in a domain like that of digital design, where many programs represent heuristic approximate solutions to very hard problems. By predicting run times, it becomes possible to trade off design time and quality.

The maintenance of an abstract representation of the target design also gives us a "reasonable" notation, i.e. one which an expert system can use as its data set without paying a large execution-time penalty. Another benefit of having both a "reasonable" notation and an explicit model of operator action is that high-level facts created by operators do not get buried in a morass of detail, and intent information does not get lost altogether.

8.3. DPE: an Updatable Program

The DPE program was presented in chapters 4 and 6. This program performs operator chaining to construct plans. It forms its own high-level objectives (importances) to respond to the idiosyncrasies of the particular specifications it is given; it iteratively constructs plans in an effort to converge to an acceptable plan.

DPE includes several features that are intended to keep programming costs down, both for front-end system construction and for system expansion and update. This is in response to the dynamic nature of the VLSI environment, which makes tools and techniques obsolete at a rapid rate.

8.3.1. Layered Interpreters

The use of nested layers of interpretation allows programming tasks to be carried out at a high level where appropriate. These layers are the following:

1. the rule level, which captures high-level planning strategy and estimation algorithms,

2. the frame level, which captures models of operators and hardware structures,

3. the Flavors (object) level, which encapsulates data structures and operations on data structures, "areas of responsibility" to be cleanly separated, and

4. the LISP level, which is the lowest level of interpretation.

8.3.2. Rule-based Objects

DPE includes rule-based objects such as the planner, the manager, and the frame expert. Rule-based interpretation of complex data structures, which is the function of these actors, means that the representations can be changed with a minimum of programming effort.

8.3.3. Rule-based Estimators

Rule-based estimators are preferable to estimators based on nested case statements for two reasons. First, it is conceptually simple to deal with discrete differences in style, structure, and intent and second, rule-based estimators are comparatively robust.

8.3.4. Negative Feedback

The use of a negative-feedback strategy where operator estimators are adjusted to search the design space means that the absolute accuracy of operator estimation only affects run time, not the quality of the ultimate results. Because the negative-feedback strategy can be extended to execution (or alternatively, the planning-space state estimators can be regarded as prospective estimators of execution-space quantities), this consideration can be applied to state estimators as well, although DPE does not do this.

8.3.5. Frame Representations

Frame representations for operators and hardware structures are comparatively easy to change. It is also easy to add new ones as new tools and hardware-implementation styles are added to the system. Frame representations are also comparatively easy to understand, so they can be constructed and debugged easily.

The applicability and effect models of operators are easily refined. This, with the use of rule-based prospective estimators, means that the system's "judgement" can be tuned with little effort.

8.4. Simulated Design and Level of Abstraction

The incomplete information available to the system at each intermediate step of the design process leads us to the conclusion that the search heuristics are not going to find the best designs, or necessarily even good ones, very quickly. Another way of saying this is that each new design problem has its own mapping into the space of implementations, and its own idiosyncratic "shape" in that space; that in the absence of more general rules, the shape will have to be determined by exploration.

Insofar as major differences in shape are a result of high-level decisions about the design, an abstract model of design states and operators can be used to predict the relative merits of particular sets of high-level decisions. That is, we can simulate the effects of design activities. This gives us a cheap way to explore the design space. We can phrase our simulation either in terms of high-level structural models or in terms of high-level design activities, or both.

We can see this in the use of traditional "levels of abstraction"; for example the use of register-transfer level allocation and optimization algorithms. In that case the abstraction mapping is from layout space to an RTL space; but we can use the RTL diagram to predict the quality of design alternatives without actually constructing them in the layout space. In this case the mapping between design activities on the layout level and the RTL level is obscure, but the mapping between designs on the two levels is much clearer.

The interesting property of monotonicity also has an interpretation in this example. As long as we map RTL into SSI and MSI packages, the cost estimates we can derive from RTL are quite close to being monotonic; however, when we map from RTL into monolithic LSI, the cost estimates have to be much more elaborate to take into account the effects of routing and placement.

8.4.1. Estimation, Hypothesization, and Search

The whole question of estimation becomes crucial in this model of the design process: it is the only thing that stands between us and a combinatorial explosion. This comes about because of poorly understood or cyclic dependencies between stages of the design process: before a decision at one stage can be made, the effects of future decisions must be made the subject of hypotheses.

Where these hypotheses are discrete-valued, there is a clean way to regard the planning process itself as a way to estimate the value of a sequence of design activities, because the planning process is a way to predict future states.

We can also regard the execution of a design activity sequence as the construction of hypotheses for estimation: the design that results is an abstract representation (e.g. a RTL diagram) of a physical chip. When the RTL diagram is used as a basis for the evaluation of criterial quantities such as area and speed, or when it is compared to other RTL designs, the properties of the chip it represents are being estimated.

Hence we come to regard the design process as the recursive making of increasingly sharper hypotheses, based on representations of successively more

detailed models of the chip being designed. In the DPE program there is provision for the following levels of hypothesis generation:

- importance numbers, which place no discrete-valued constraints on future activities,
- prospective estimates of operator value, which make implicit hypotheses about future activities, and
- modeling of operator effects, which introduce discrete-valued commitments.

These layers of hypothesization and estimation can be contrasted with more traditional "levels of abstraction" of VLSI design, e.g. the decomposition of circuits successively into behavior, RTL structure, logic, transistor and layout levels. On the one hand, the layers in DPE form a procedurally oriented model of the design process; on the other, the traditional level set forms a more data-oriented model. Both of these models are useful, and each can tell us interesting things about building unified design automation systems.

8.4.2. Summary

In summary, the results of this thesis research are as follows.

First, the design data structure (DDS) provides a common representational formalism for design data. This allows collections of operators to be applied to a single design in orderings determined at run time, without the expense (and other difficulties) of individual input filters and back annotators.

Second, the design planning engine (DPE) is capable of constructing arbitrary chains of operators (plans). This differs from the work of Stefik [Stefik 80] in that Stefik's emphasis was on the propagation of constraints

within a plan, the plan itself being the object under design; and from that of Tong [Tong 85], Steinberg [Steinberg 84], and Kelly [Kelly 82] in that they do not explicitly model operators capable of transforming designs, but rather rely on propagation of constraints between subunits of the design to narrow the search space.

In order to construct DPE, it was necessary to construct a representation for design operators, in terms of preconditions, postconditions, advisability estimators, and application rules. This representation assures semantic compatibility between operators and the design states to which they are applied; gives a way to estimate the results of application; gives a heuristic for choosing between applicable operators; and allows complex and data-dependent calling sequences to be attached to operators. The representation can be used both for planning systems such as DPE, and for systems that execute operators at chaining time, whether interactive or automatic. It is similar to that of STRIPS [Fikes 71], but STRIPS does not include either heuristic choice or operator application information; and furthermore, STRIPS has static (i.e. assertion-based) postconditions only, whereas DPE also has the far more powerful rule-based postconditions.

The representation for domain operators is declarative, which greatly eases updating and tuning, both of the knowledge base of domain operators, and of the programs that access the knowledge base.

A formal model of the design process, based on operator pre- and postconditions, has been defined. Under the assumption of monotonicity, this model can be used to bound the complexity of a design search process in which arbitrary sequences of operators are considered. This model is predictive in that it can be used to predict properties of collections of operators and estimators, and prescriptive in that it can be used as a

conceptual foundation for building software systems. It can be contrasted with work such as that of Koomen [Koomen 79] and Rammig [Rammig 85] in that those models are descriptive, but do not provide positive guidance in the construction of systems, being based on characterization of information flow rather than on applicability and effect of explicit individual operators. The model could be applied not only to a system which plans, but also one which directly executes at chaining time, either interactively or automatically.

With respect to the construction of a unified design automation system, the following accomplishments can be listed:

1. a representation for domain knowledge covering both operators and generic hardware structures has been constructed,

2. the domain knowledge representation allows the integration of arbitrary LISP functions, and has a partial mechanism for the integration of non-LISP program code,

3. a way of representing and testing the applicability of operators, i.e. of whether they can be expected to work in specific contexts, forms part of the domain knowledge representation,

4. a representation for design data has been constructed at the DDS level,

5. a representation for design data has been initiated at the planning-space level, and partially linked to the DDS-level representation, and

6. an inference engine capable of handling procedure calls, and interfacing to arbitrary program code, has been implemented; this inference engine also supports multiple co-operating expert systems, and can build a "dynamic" expert system from multiple rule sources.

All of the above can be used in systems that plan before executing, as DPE does, and can also be used in systems that execute at chaining time. Strictly in the planning domain, DPE embodies the following accomplishments:

1. a prototype planner, in the form of DPE's actor architecture, has been constructed,

2. the planner can be converted to a chaining-time execution scheme by changing one rule,

3. a theory of monotonic estimation and planning has been developed, which allows us to bound search complexity,

4. sets of prototype estimators for operator selection, state property estimation, and state content estimation (i.e. postcondition rules), have been constructed and validated for monotonicity, and

5. a successive-approximation technique (interpolation) has been implemented that can be used to set up operator selection criteria, and which at this early stage seems to be relatively insensitive to nonmonotonicities.

References

[Abadir 86] M. Abadir.
A Knowledge Based System for Designing Testable VLSI Circuits.
PhD thesis, University of Southern California, 1986.

[Ackland 85] B. Ackland, A. Dickenson, R. Ensor, J. Gabbe, P. Kollaritsch, T. London, C. Poirer, P. Subrahmanyan, and H. Watanabe.
CADRE: A System of Co-operating VLSI Design Experts.
In *Proceedings of the International Conference on Computer Design*, pages 99-104. IEEE, 1985.

[Afsarmanesh 85a] H. Afsarmanesh.
The 3 Dimensional Information Space (3DIS), An Extensible Browsing-Oriented Framework for Database Systems.
PhD thesis, Department of Computer Science, University of Southern California, 1985.

[Afsarmanesh 85b] H. Afsarmanesh, D. Knapp, D. McLeod, and A. Parker.
An Extensible Object-Oriented Approach to Databases for CAD/VLSI.
In *Proceedings of the 11th VLDB Conference.* VLDB Endowment, 1985.

[Aho 74] A. Aho, J. Hopcroft, and J. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[Allen 83] E. Allen.
YAPS: Yet Another Production System.
Technical Report TR-1146, Maryland Artificial Intelligence Group, 1983.

[Bachant 84] J. Bachant and J. McDermott.
R1 Revisited: Four Years in the Trenches.
AI Magazine :21-32, Fall, 1984.

[Barbacci 79a] M. Barbacci and A. Nagle.
The Symbolic Manipulation of Computer Descriptions ISPS Simulator.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1979.

[Barbacci 79b] M. Barbacci, G. Barnes, R. Cattell, and D. Siewiorek.
The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1979.

[Barr 81] A. Barr and E. Feigenbaum.
The Handbook of Artificial Intelligence.
William Kaufmann, 1981.

[Batali 83] J. Batali.
Dependency Maintenance in the Design Process.
In *Proceedings of the International Conference on Computer Design*, pages 459-462. IEEE, 1983.

[Birmingham 84] W. Birmingham and D. Siewiorek.
MICON: A Knowledge Based Single Board Computer Designer.
In *Proceedings of the 21st Design Automation Conference*, pages 565-571. IEEE, 1984.

[Blackman 85] T. Blackman, J. Fox, and C. Rosebrugh.
The SILC Silicon Compiler: Language and Features.
In *Proceedings of the 22nd Design Automation Conference*, pages 232-237. IEEE, 1985.

[Bobrow 81] D. Bobrow and M. Stefik.
The LOOPS Manual.
Technical Report KB-VLSI-81-13, Xerox PARC, 1981.

[Brachman 85] R. Brachman.
I Lied About the Trees.
The AI Magazine 6(3):80-93, Fall, 1985.

[Breuer 85] M. Breuer and X. Zhu.
A Knowledge Based System for Selecting a Test Methodology for a PLA.
In *Proceedings of the 22nd Design Automation Conference.* IEEE, 1985.

[Brewer 86] F. Brewer and D. Gajski.
An Expert System Paradigm for Design.
In *Proceedings of the 23rd Design Automation Conference,* pages 62-68. IEEE, 1986.

[Brotoatmodjo 86]
E. Brotoatmodjo and J. Pizarro.
Catalog and Librarian: Rapid Access to Information Contained in Cell Libraries.
August, 1986.

[Brown 82] H. Brown and M. Stefik.
Palladio: An Expert Assistant for Integrated Circuit Design.
1982.
Memo KB-VLSI-82-17 (working paper), Xerox PARC.

[Brown 83] H. Brown, C. Tong, and G. Foyster.
Palladio: An Exploratory Environment for Circuit Design.
IEEE Computer 16(12):41-56, December, 1983.

[Brug 85] A. Van de Brug, J. Bachant, and J. McDermott.
Doing R1 With Style.
In *Proceedings of the Conference on Artificial Intelligence Applications,* pages 244-249. IEEE, 1985.

[Bushnell 83] M. Bushnell, D. Geiger, J. Kim, D. LaPotin, S. Nassif, J. Nestor, J. Rajan, A. Strojwas, and H. Walker.
DIF: the CMU-DA Intermediate Form.
Technical Report CMUCAD-83-11, Carnegie-Mellon University, 1983.

[Bushnell 86] M. Bushnell and S. Director.
VLSI CAD Tool Integration Using the ULYSSES Environment.
In *Proceedings of the 23rd Design Automation Conference,* pages 55-61. IEEE, 1986.

[Camposano 84] R. Camposano, A. Kunzmann, and W. Rosenstiel.
Automatic Data Path Synthesis from DSL Specifications.
In *Proceedings of the International Conference on Computer Design*, pages 630-635. IEEE, 1984.

[Camposano 85] R. Camposano.
Synthesis Techniques for Digital Systems Design.
In *Proceedings of the 22nd Design Automation Conference*, pages 475-481. IEEE, 1985.

[Clark 85] G. Clark and R. Zippel.
Schema- An Architecture for Knowledge Based CAD.
In *Proceedings of International Conference on Computer Aided Design*, pages 50-52. IEEE, 1985.

[Crawford 84] J. Crawford.
An Electronic Design Interchange Format.
In *Proceedings of the 21st Design Automation Conference*, pages 683-685. ACM IEEE, June, 1984.

[Davis 82] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, and S. Polit.
Diagnosis Based on Description of Structure and Function.
In *Proceedings of the 1982 National Conference on Artificial Intelligence*, pages 137-142. AAAI, 1982.

[DeKleer 77] J. de Kleer.
Multiple Representations of Knowledge in a Mechanics Problem-Solver.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 299-304. AAAI, 1977.

[DeKleer 78] J. de Kleer and G. Sussman.
Propagation of Constraints Applied to Circuit Synthesis.
Technical Report 485, MIT AI Laboratory, 1978.

[Dennis 75] J. Dennis and D. Misunas.
A Preliminary Data Flow Architecture for a Basic Data Flow Processor.
In *Second Symposium on Computer Architecture*, pages 126-132. 1975.

[Director 81] S. Director, A. Parker, D. Siewiorek, and D. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
IEEE Transactions on Circuits and Systems CAS-28:634-645, July, 1981.

[Durfee 85] E. Durfee, V. Lesser, D. Corkill.
Coherent Cooperation Among Communicating Problem Solvers.
In *Proceedings of the 1985 Distributed Artificial Intelligence Workshop*, pages 231-276. AAAI, 1985.

[Ernst 69] G. Ernst and A. Newell.
GPS: A Case Study in Generality and Problem Solving.
Academic Press, 1969.

[Estrin 78] G. Estrin.
A methodology for design of digital systems - supported by SARA at the age of one.
In *Proceedings of National Computer Conference*, pages 313-324. NCC, 1978.

[Estrin 86] G. Estrin, R. Fenchal, R. Rezouk, and M. Vernon.
SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems.
IEEE Transactions on Software Engineering SE-12(2):293-311, 1986.

[Fikes 71] R. Fikes and N. Nilsson.
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.
In *Proceedings of the 1971 International Joint Conference on Artificial Intelligence*, pages 608-620. AAAI, 1971.

[Floyd 67] R. Floyd.
Assigning Meaning to Programs.
In *Proceedings of the 19th Symposium on Applied Mathematics*, pages 19-32. American Mathematical Society, 1967.

[Forgy 81] C. Forgy.
OPS5 User's Manual.
Technical Report CMU-CS-78-116, Carnegie-Mellon University Computer Science Department, 1981.

[Foster 84] J. Foster.
A Unified CAD System for Electronic Design.
In *Proceedings of the 21st Design Automation Conference.* IEEE, 1984.

[Fox 83] J. Fox.
Optimization of the MacPitts Silicon Compiler for Telecommunications Circuitry.
1983.
MIT VLSI memo 83-132.

[Goldberg 83] A. Goldberg and D. Robson.
SMALLTALK-80:The Language and its Implementation.
Addison-Wesley, 1983.

[Granacki 85] J. Granacki, D. Knapp, and A. Parker.
The ADAM Advanced Design AutoMation System: Overview, Planner, and Natural Language Interface.
In *Proceedings of the 22nd Design Automation Conference,* pages 727-730. 1985.

[Granacki 86] J. Granacki.
1986
PhD thesis, in progress.

[Green 69] C. Green.
Application of Theorem Proving to Problem Solving.
In *Proceedings of the 1969 International Joint Conference on Artificial Intelligence,* pages 219-239. AAAI, 1969.

[Hafer 81] L. Hafer and A. Parker.
Automated Synthesis of Digital Hardware.
IEEE Transactions on Computers C-31(2):93-109, February, 1981.

[Hayes-Roth 83] F. Hayes-Roth, D. Waterman, and D. Lenat.
Building Expert Systems.
Addison-Wesley, 1983.

[Hewitt 69] C. Hewitt.
PLANNER: A Language for Proving Theorems in Robots.
In *Proceedings of the 1969 International Joint Conference on Artificial Intelligence,* pages 295-301. AAAI, 1969.

[Hewitt 73] C. Hewitt, P. Bishop, and R. Steiger.
A Universal Modular ACTOR Formalism for Artificial Intelligence.
In *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, pages 235-245. AAAI, 1973.

[Hitchcock 83] C. Hitchcock III and D. Thomas.
A Method of Automatic Datapath Synthesis.
In *Proceedings of the 20th Design Automation Conference*, pages 484-489. ACM/IEEE, 1983.

[Hsu 84] A. Hsu, L. Hsu, and P. Ulrich.
A Design Environment that Integrates Tools, Database, and User Interface.
In *Proceedings of the International Conference on Computer Design*, pages 733-741. IEEE, 1984.

[Hudlicka 84] E. Hudlicka and V. Lesser.
Meta-Level Control Through Fault Detection and Diagnosis.
In *Proceedings of the 1984 National Conference on Artificial Intelligence*, pages 153-161. AAAI, 1984.

[James 85] J. James, D. Frederick, P. Bonissore, and J. Taylor.
A Retrospective View of CACE-III: Considerations in Coordinating Symbolic and Numeric Computation in a Rule-Based Expert System.
In *Proceedings of the Conference on Artificial Intelligence Applications*, pages 532-538. IEEE, 1985.

[Johannsen 79] D. Johannsen.
Bristle Blocks: A Silicon Compiler.
In *Caltech Conference on VLSI*, pages 303-310. Caltech, 1979.

[Joobbani 85] R. Joobbani and D. Siewiorek.
WEAVER: A Knowledge Based Routing Expert.
In *Proceedings of the 22nd Design Automation Conference*, pages 266-272. IEEE, 1985.

[Kelly 82] V. Kelly and L. Steinberg.
The Critter System: Analyzing Digital Circuits by Propagating Behaviors and Specifications.
In *Proceedings of the 1982 National Conference on Artificial Intelligence*, pages 284-289. AAAI, 1982.

[Kim 83] J. Kim and J. McDermott.
TALIB: An IC Layout Design Assistant.
In *Proceedings of the 1983 National Conference on Artificial Intelligence*, pages 197-201. AAAI, 1983.

[Knapp 83a] D. Knapp and A. Parker.
A Data Structure for VLSI Synthesis and Verification.
Technical Report DISC 83-6, USC Digital Integrated Systems Center, 1983.

[Knapp 83b] D. Knapp, J. Granacki, and A. Parker.
An Expert Synthesis System.
In *Proceedings of the International Conference on Computer Aided Design*, pages 164-165. ACM-IEEE, 1983.

[Knapp 85] D. Knapp and A. Parker.
A Unified Representation for Design Information.
In *Proceedings of the Conference on Hardware Description Languages*, pages 337-353. North-Holland, 1985.

[Knapp 86] D. Knapp and A. Parker.
A Design Utility Manager: the ADAM Planning Engine.
In *Proceedings of the 23rd Design Automation Conference*, pages 48-54. IEEE, 1986.

[Koomen 79] C. Koomen.
Information Processing in System Design.
In *Proceedings of the 1979 International Conference on Cybernetics and Society*. IEEE, 1979.

[Kowalski 83a] T. Kowalski and D. Thomas.
The VLSI Design Automation Assistant: Prototype System.
In *Proceedings of the 20th Design Automation Conference*, pages 479-483. IEEE, 1983.

[Kowalski 83b] T. Kowalski and D. Thomas.
The VLSI Design Automation Assistant: Learning to Walk.
In *1983 IEEE International Symposium on Circuits and Systems*, pages 186-190. IEEE, 1983.

[Kowalski 84] T. Kowalski.
The VLSI Design Automation Assistant: A Knowledge-Based Expert System.
PhD thesis, Carnegie-Mellon University, April, 1984.

[Kowalski 85] T. Kowalski and D. Thomas.
The VLSI Design Automation Assistant: What's in a Knowledge Base.
In *Proceedings of the 22nd Design Automation Conference*, pages 252-258. IEEE, 1985.

[Kurdahi 86] F. Kurdahi and A. Parker.
Wiring Space Estimation of Standard Cell Designs.
In *Proceedings of the 23rd Design Automation Conference*, pages 467-473. 1986.

[LaPotin 85] D. LaPotin and S. Director.
Mason: A Global Floor-Planning Tool.
In *Proceedings of International Conference on Computer Aided Design*, pages 143-145. IEEE, 1985.

[Leive 81] G. Leive and D. Thomas.
A Technology Relative Logic Synthesis and Module Selection System.
In *Proceedings of the 18th Design Automation Conference*, pages 479-485. IEEE, 1981.

[Lesser 83] V. Lesser and D. Corkill.
The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks.
The AI Magazine :15-33, Fall, 1983.

[Manna 81] Z. Manna and R. Waldinger.
A Deductive Approach to Program Synthesis.
Readings in Artificial Intelligence.
Tioga Press, 1981, pages 141-172.

[Marwedel 79] P. Marwedel.
The MIMOLA Design System: Detailed Description of the Software System.
In *Proceedings of the 16th Design Automation Conference*, pages 59-63. IEEE, 1979.

[Marwedel 84] P. Marwedel.
The Mimola Design System: Tools for the Design of Digital Processors.
In *Proceedings of the 21st Design Automation Conference*, pages 587-593. IEEE, 1984.

[McDermott 82] J. McDermott.
R1: A Rule-Based Configurer of Computer Systems.
Artificial Intelligence 19(1):39-88, September, 1982.

[McFarland 81] M. McFarland.
Mathematical Models for Verification in a Design Automation System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.

[McFarland 83] M. McFarland and A. Parker.
An Abstract Model of Behavior for Hardware Description.
IEEE Transactions on Computers C-32(7):621-637, July, 1983.

[Mead 79] C. Mead and L. Conway.
Introduction to VLSI Systems.
Addison Wesley, Reading, Mass., 1979.

[Miller 60] G. Miller, E. Galanter, and K. Pribram.
Plans and the Structure of Behavior.
Holt, Rinehart and Winston, 1960.

[Minsky 81] M. Minsky.
A Framework for Representing Knowledge.
Mind Design.
MIT Press, 1981, pages 95-128.

[Mitchell 83] T. Mitchell, L. Steinberg, S. Kedar-Cabelli, V. Kelly, J. Shulman, and T. Weinrich.
An Intelligent Aid for Circuit Redesign.
In *Proceedings of the 1983 National Conference on Artificial Intelligence*, pages 274-278. AAAI, 1983.

[Mlinar 86] M. Mlinar and A. Parker.
VTRAN: A VT to PNF Translator.
Technical Report CRI-86-35, USC Computer Research Institute, September, 1986.

[Munson 71] J. Munson.
Robot Planning, Execution, and Monitoring.
In *Proceedings of the 1971 International Joint Conference on Artificial Intelligence*, pages 338-349. AAAI, 1971.

[Nagle 80] A. Nagle.
Automatic Design of Sequencers for The Control of Digital Hardware.
PhD thesis, Carnegie-Mellon University, October, 1980.

[Park 85a] N. Park.
Synthesis of High-Speed Digital Hardware.
PhD thesis, University of Southern California, 1985.

[Park 85b] N. Park and A. Parker.
SEHWA: a Program for Synthesis of Pipelines.
In *Proceedings of the 22nd Design Automation Conference*, pages 454-460. IEEE, 1985.

[Park 85c] N. Park.
Synthesis of Optimal Clocking Schemes.
In *Proceedings of the 22nd Design Automation Conference*, pages 489-495. IEEE, 1985.

[Parker 81] A. Parker and J. Wallace.
SLIDE: A Hardware Description Language.
IEEE Transactions On Computers C-30(6), June, 1981.

[Parker 84] A. Parker, N. Park, and D. Knapp.
Simulation Effectiveness and Design Verification.
Technical Report DISC/84-2, Department of EE-Systems, University of Southern California, 1984.

[Parker 86] A. Parker, J. Pizarro, and M. Mlinar.
MAHA: A Program for Datapath Synthesis.
In *Proceedings of the 23rd Design Automation Conference*, pages 461-466. IEEE, 1986.

[Quillian 69] M. Quillian.
The Teachable Language Comprehender: A Simulation Program and Theory of Language.
Communications of the ACM 12(8):459-476, August, 1969.

[Rammig 85] F. Rammig.
A Multilevel Cybernetic Model of the Design Process.
In *Methodologies for Computer System Design*, pages 87-103. IFIP, North-Holland, 1985.

[Reddy 73] D. Reddy, L. Erman, R. Fennell, and R. Neely.
The Hearsay Speech Understanding System: An Example of the Recognition Process.
In *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, pages 185-193. AAAI, 1973.

[Rosenschein 83] J. Rosenschein and V. Singh.
The Utility of Meta-Level Effort.
Technical Report Hpp-83-20, Stanford Heuristic Programming Project, 1983.

[Rosenstiel 85] W. Rosenstiel and R. Camposano.
Synthesizing Circuits from Behavioural Level Specifications.
In *Proceedings of the Conference on Hardware Description Languages*, pages 391-403. North-Holland, 1985.

[Sacerdoti 73] E. Sacerdoti.
Planning in a Hierarchy of Abstraction Spaces.
In *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, pages 412-422. AAAI, 1973.

[Sacerdoti 75] E. Sacerdoti.
The Nonlinear Nature of Plans.
In *Proceedings of the 1975 International Joint Conference on Artificial Intelligence*, pages 206-218. AAAI, 1975.

[Sacerdoti 77] E. Sacerdoti.
A Structure for Plans and Behavior.
Elsevier Scientific Publishing, 1977.

[Schank 81] R. Schank and G. Abelson.
Scripts, Plans, Goals and Understanding.
Lawrence Erlbaum, 1981.

[Shortliffe 76] E. Shortliffe.
Computer-Based Medical Consultations: Mycin.
American Elsevier, 1976.

[Snow 78] E. Snow and D. Siewiorek.
A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs.
In *Proceedings of the 15th Design Automation Conference*, pages 220-226. IEEE, 1978.

[Steele 78] G. Steele and G. Sussman.
Constraints.
Technical Report 502, MIT AI Lab, 1978.

[Stefik 80] M. Stefik.
Planning With Constraints.
PhD thesis, Stanford University, 1980.

[Stefik 81] M. Stefik, D. Bobrow, A. Bell, H. Brown, L. Conway, and C. Tong.
The Partitioning of Concerns in Digital System Design.
Technical Report VLSI-81-3, Xerox PARC, 1981.

[Steinberg 84] L. Steinberg and T. Mitchell.
A Knowldege Based Approach to CAD: The Redesign System.
In *Proceedings of the 21st Design Automation Conference*, pages 412-418. IEEE, 1984.

[Tong 85] C. Tong.
A Framework for Organizing and Evaluating Knowledge-Based Models of the Design Process.
Technical Report 21, Rutgers University, 1985.
AI/VLSI working paper series.

[Utt 85] W. Utt.
Directed Search With Feedback.
In *Proceedings of the Conference on Artificial Intelligence Applications*, pages 647-652. IEEE, 1985.

[Walker 85] R. Walker and D. Thomas.
A Model of Design Representation and Synthesis.
In *Proceedings of the 22nd Design Automation Conference*, pages 453-459. IEEE, 1985.

[Waxman 86] R. Waxman.
The VHSIC Hardware Description Language: A Glimpse of the Future.
IEEE Design & Test 3(2), April, 1986.

[Wilensky 83] R. Wilensky.
Planning and Understanding.
Addison-Wesley, 1983.

[Wilensky 84] R. Wilensky.
Lispcraft.
W. W. Norton, 1984.

[Winston 84] P. Winston.
Artificial Intelligence.
Addison-Wesley, 1984.

[Woods 75] W. Woods.
What's In a Link: Foundations of Semantic Networks.
Representation and Understanding.
Academic Press, 1975, pages 35-82.

[Zimmermann 79]
G. Zimmermann.
The MIMOLA Design System: A Computer Aided Digital Processor Design Method.
In *Proceedings of the 16th Design Automation Conference*, pages 53-58. IEEE, 1979.

[Zippel 83] R. Zippel.
An Expert System for VLSI Design.
In *1983 IEEE International Symposium on Circuits and Systems*, pages 191-193. IEEE, 1983.

Appendix A
DDS Details

In this appendix the generalization hierarchies for the DDS are given. The hierarchies are presented in the form of diagrams in which object types are represented as boxes, type-subtype relationships as directed arcs, and entity-attribute relationships as undirected arcs. Type-subtype relationships are denoted by arcs directed from the subtype to the type; e.g. in Fig. I-1, the type **Node** is a subtype of the type **Model.** As such, it inherits all of the attributes of **Model:** a **Name,** a **Designer,** and a **Complete-Bit.** It has as well a set of its own attributes; its **Function,** a set of **Node References,** and so on.

The type-subtype hierarchy for the dataflow subspace was discussed in Chapter 3. This appendix is a detailed and comprehensive reference. There are minor differences between the dataflow hierarchy and the other hierarchies. Some of the differences are discussed here, where the meaning of the figures is not obvious.

A.1. Notes on the Hierarchies

In Chapter 3 we discussed the dataflow subspace in detail. The other subspaces are not exactly the same. In this section we will enumerate some of the differences.

1. The **Function** field of a **Model** is an explanatory text string.
2. **References** fields, e.g. **Node** and **Value References** in the dataflow subspace, are lists of references to other defintions. In the

case of the dataflow subspace, **Node References** are to items of type **Node Referral**; a **Value Reference** is to an item of type **Value Referral. Referrals** and **References** are discussed below.

3. A **Range** has a tag field for **Causality**. That is, a range can express a causative relationship, or a measurement, or a constraint. The tag expresses the semantics that pertains to a given **Range**.

4. A **Range** also has a set of **Durations** representing its length in time. These are predicates in the form of numeric expressions, e.g. $t = 52$ **nanoseconds.**

5. The **Static Storage Tag** of a **Module** represents the possibility that it might contain flip-flops, i.e. static storage elements. This field is a Boolean.

6. Physical models **Blocks** have the following unique attributes.

   a. **Shape**, which describes a bounding polygon.

   b. **Boxes**, a list of (layer, rectangle) geometric primitives.

   c. **Contacts**, a list of interlayer connection points.

   d. **Power Requirements**, a list of power supply voltage and current requirements (e.g., for TTL +5 and 0 volts).

   e. **Package Description**, describes packaging. This field is intended to represent many possible variants, e.g. rack-mount, DIP, and PCB.

7. **Model** and **Link Referrals** have **Nil** as subtypes. This denotes the possibility that a referral may be empty. Hence a reference to a dataflow **Node** may be a reference to **Nil**, meaning either (1) that the definition containing the reference is incomplete, or (2) that the definition is of a primitive.

8. **Intended Function** fields are explanatory strings.

9. **Kind** fields are pointers to definitions, e.g. to **Models.**

10. **Block References** have the following special attributes.

a. **Mirroring** represents a coordinate transformation wherein either x or y, or both, coordinates are negated.

b. **Rotation** represents a coordinate transformation wherein the **Block** is rotated about the z axis.

11. **Links** have **Reference** fields, whidh contain lists of **Referrals**, which in turn are either **Nil** or they are **References** proper, in a manner analogous to the reference hierarchy of **Models**.

12. **Points**, which are the **Links** of the timing subspace, are an enumerated type, as described in Chapter 3.

13. **Carriers**, which are logical-structure **Links**, have a **Persistence Storage Tag**, which describes the possibility that the **Carrier** is capable of storing data dynamically.

14. **Nets**, the physical structure **Link** type, have the following special attributes:

    a. a **Path**, which is the physical path followed by the **Net**, i.e. a list of coordinate pairs,

    b. a set of **Boxes**, which are (layer, rectangle) pairs,

    c. a set of **Contacts**,

    d. a **Resistance**, and

    e. a **Capacitance**.

    Note that this set of special attributes is minimal. See Crawford [Crawford 84] for a list of special attributes with which physical models can be augmented.

15. **Link** references are complicated by the use of sub-fields, as shown in Figs. 3-11 and 3-12. Hence a **Value Reference** can be one of two types: a **Single Value**, which is a value as referenced in a **Node** definition, and a **Sub-Value**, which is a value as referenced in a **Value** definition. Note, e.g., that a **Sub-Value** has no **Netlist**, which expresses the connections of a **Single Value**.

16. **Points** do not need the equivalent of **Sub-Values** because

Points are of enumerated type; that is, there are no **Point** definitions. There is therefore no necessity to have **Netlists** in the timing and sequencing subspace; just sets of **Sink** and **Source Ranges** for each **Point**.

17. **Netlists** are sets of net connections; e.g. **DF-Nets** in the dataflow subspace. **DF-Nets** have the following attributes:

    a. a **Path**, which is a path into the **Single-Value's** definition hierarchy, and which denotes the **Sub-Value** being connected;

    b. a **Visibility**, which is a Boolean expressing whether the **Sub-Value** described in the **Path** is a connection point of the **Node** in which the **Single-Value** is used; and

    c. a **Pin List**, which is a set of **DF-Pins** to which the **Sub-Value** named in the **Path** is connected.

18. A **DF-Pin** has the following attributes:

    a. the name of a **Node**, which is the **Node** to which the **DF-Pin** belongs, and

    b. a **Path** into the **Node's** definition hierarchy, leading to the **Single-Value** or **Sub-Value** to which the connection is made. See Figs. 3-11 and 3-12 for an example of these concepts. The logical and physical structure subspaces are similar.

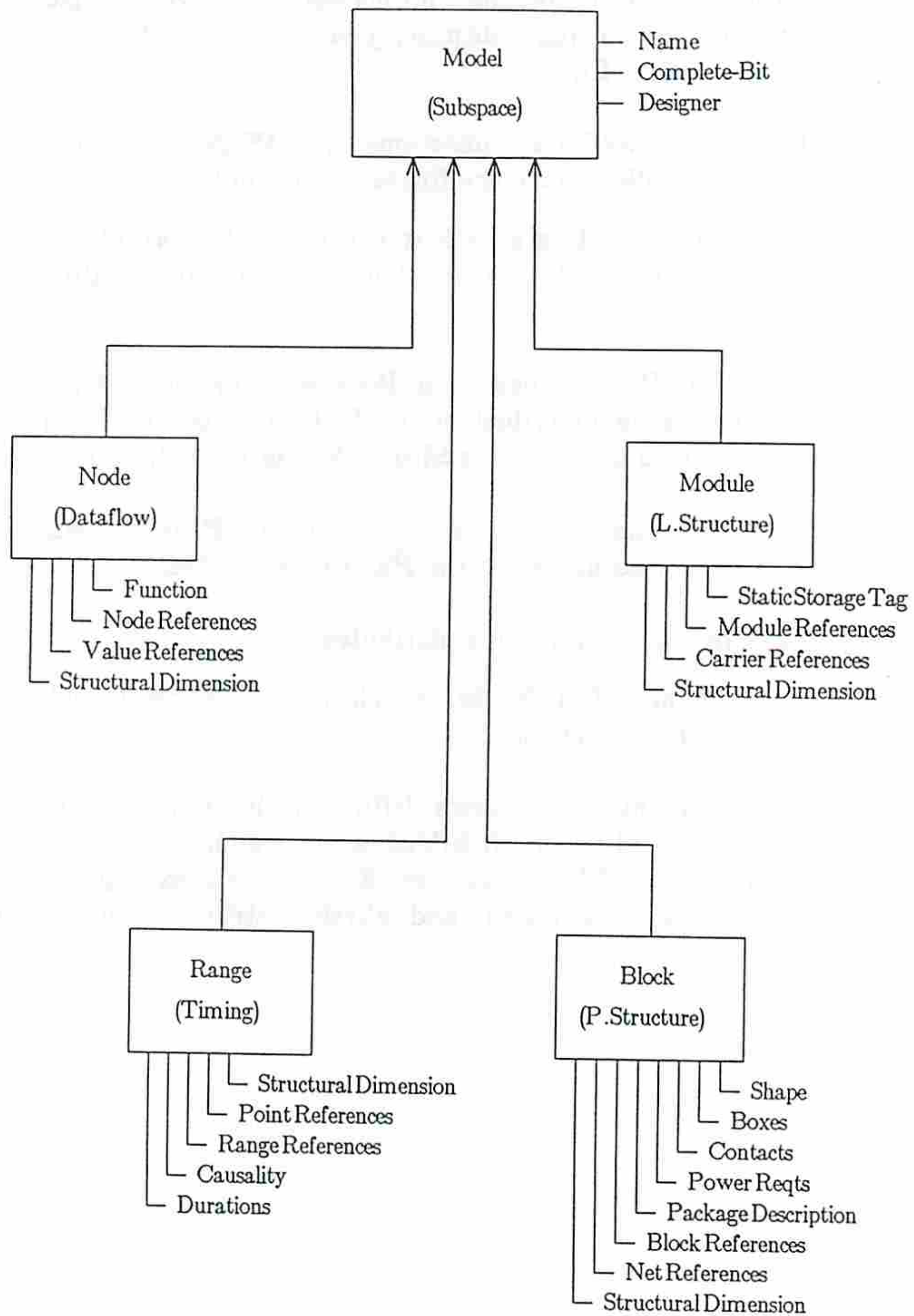


Figure A-1: Model Hierarchy

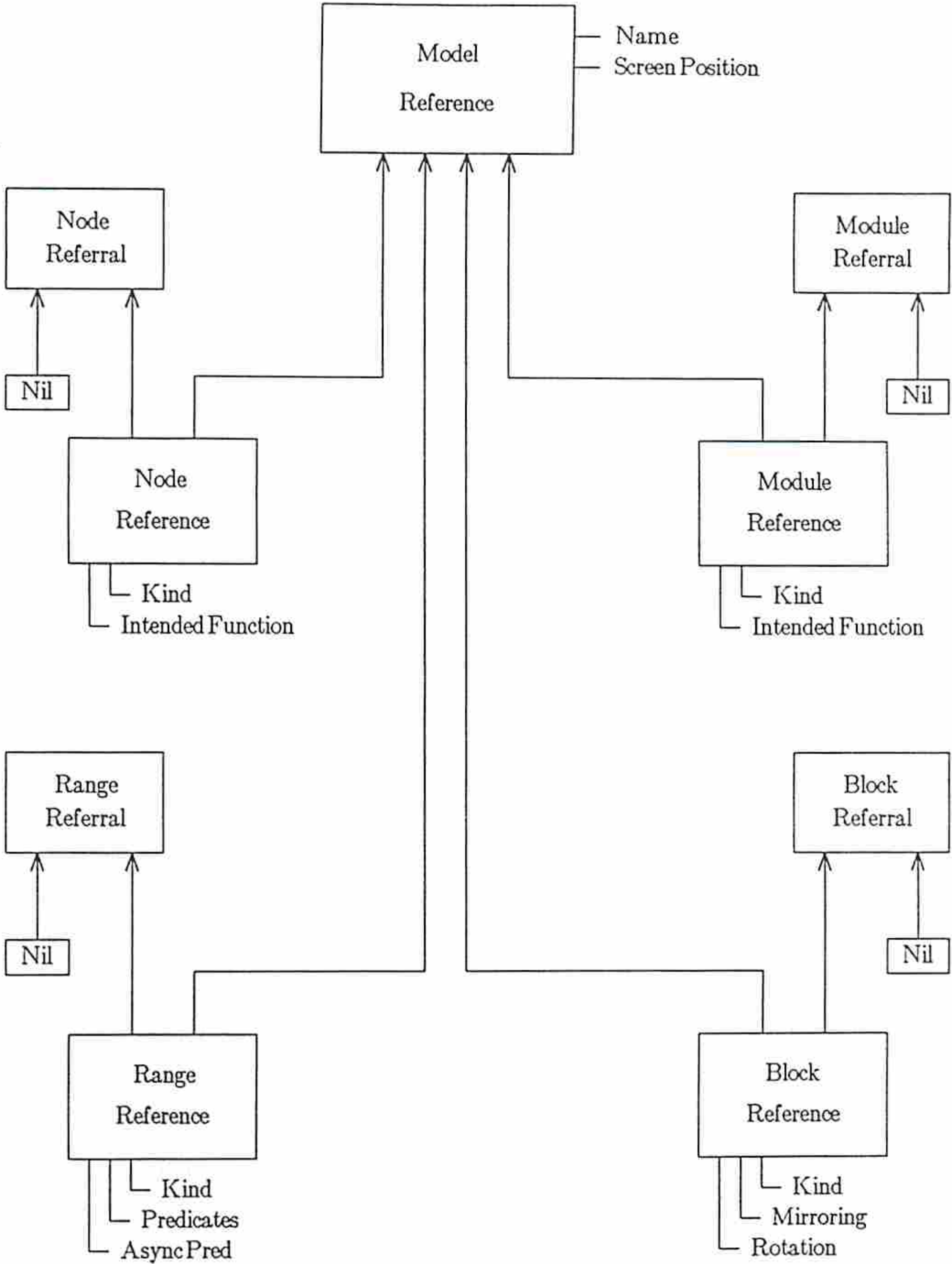


Figure A-2: Model Reference Hierarchy

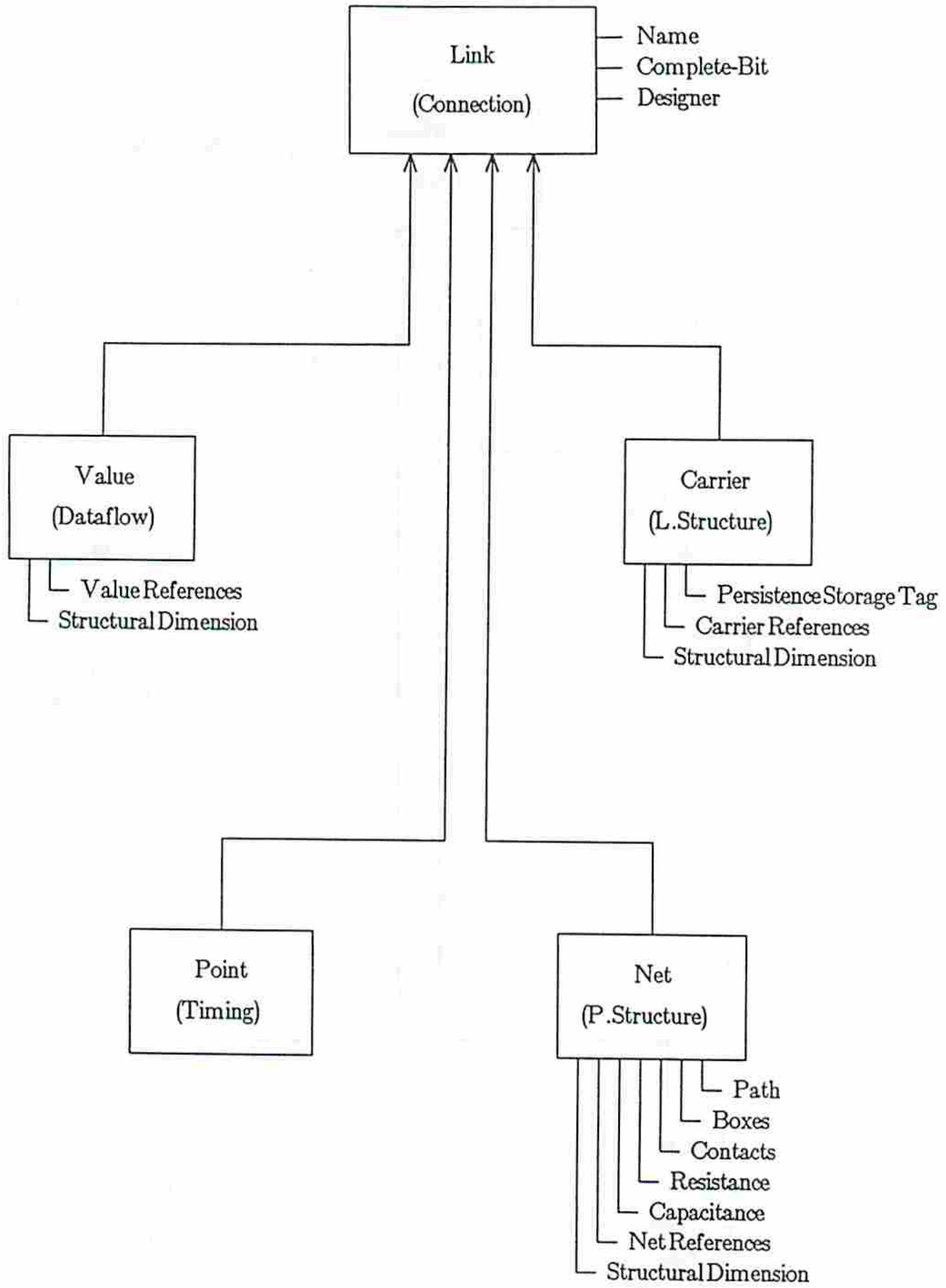


Figure A-3: Link Hierarchy

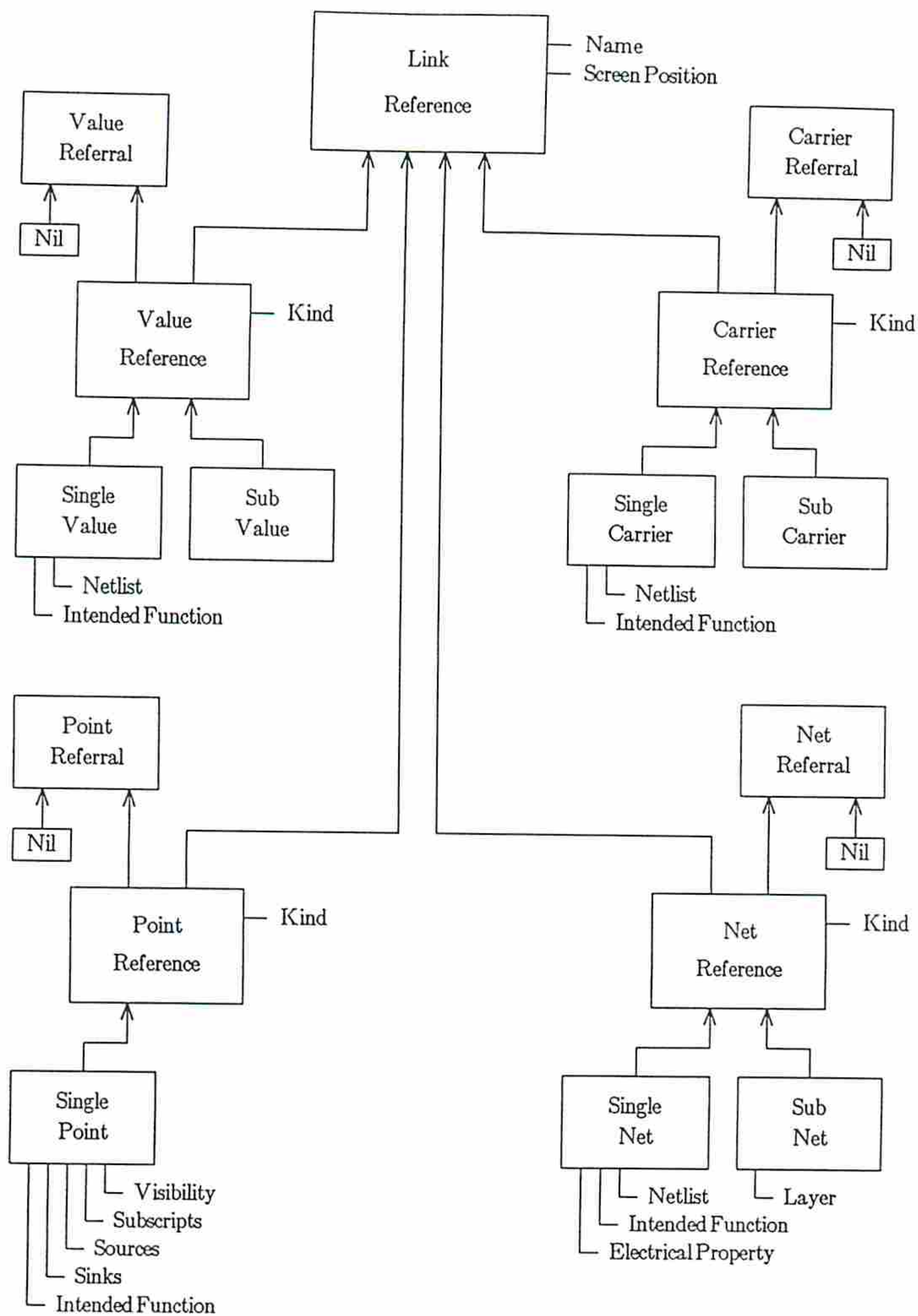


Figure A-4: Link Reference Hierarchy

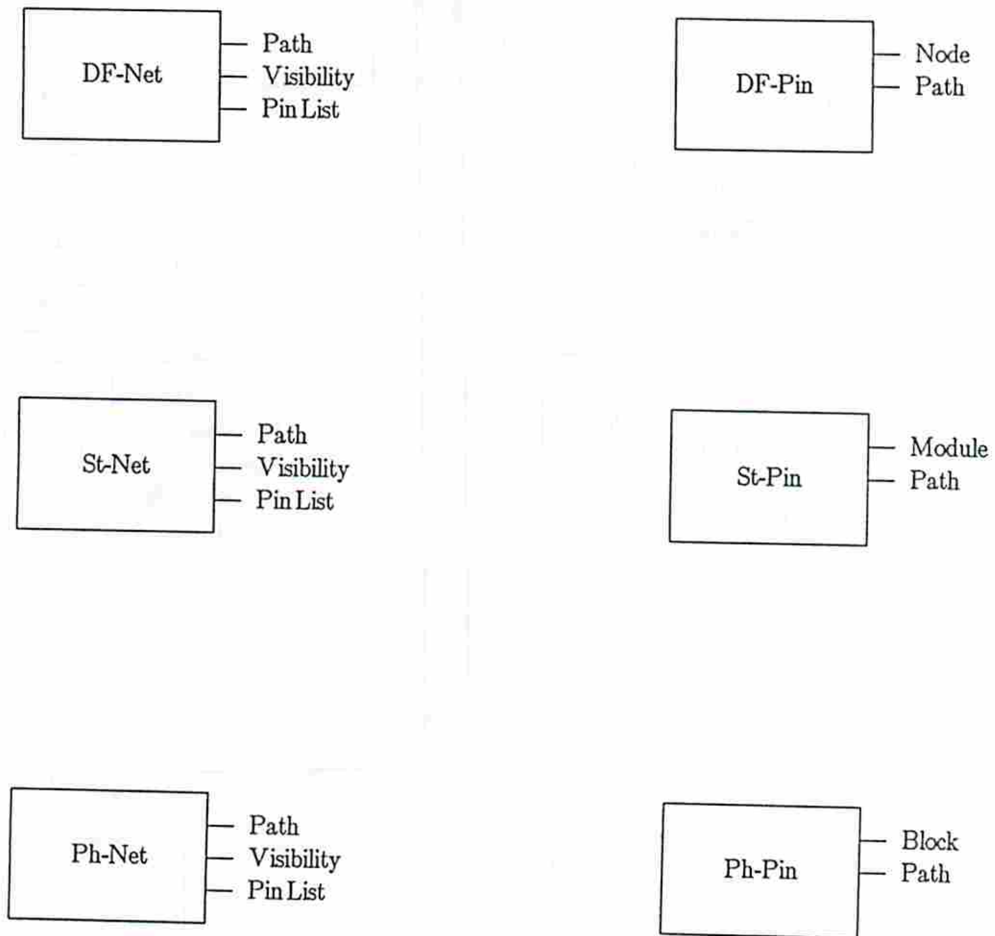


Figure A-5: Netlist Types

Appendix B
DPE's Frame Set

In this appendix the frame set of DPE is presented. This frame set is the current frame set at the time of writing; it was constructed more for debugging and validating DPE than as a production knowledge base. It is therefore considered to be more useful as a set of concrete examples and as an expository tool, than as a domain knowledge base corresponding to real-world design problems. The interested reader is referred to Chapter 7 for detailed explanations of the various fields of the frames; these frames have also been augmented by the addition of comments. Otherwise they are exactly as taken from DPE's knowledge base.

```
(criteria area ctime power) ; criterial quantities for VLSI
(task dftran    ; dataflow graph reform task.
; This task is a bit fakey. It creates one
; of two assertions in the data set, either
; (unco-plel ?itsdfg) or (no-plel-loop ?itsdfg)
; which are preconditions of maha and sehwa respectively.
; It must be "executed" at planning time
; in order to do this. See the
; add-rules to see how this is done.
  (invar (              ; all of the following must exist
    (dfoa ?it)          ; a particular hardware unit
    (dfg ?it ?itsdfg)           ; its dataflow graph
        ))
  (Add-Rules (
; it is sometimes difficult to express
; the nonexistence of a thing without
; a special rule to do so for you.
    (F1 ((dfoa ?it) (unco-plel ?itsdfg) (dfg ?it ?itsdfg))
        (exists unco-plel-dfg ?it))
; this rule fakes the execution of a
; dataflow loop modifier.
; when the modifier is available replace
; this rule with the
; invocation of the code.
    (F1 ((no (exists unco-plel-dfg ?it))
```

```
                (dfoa ?it) (dfg ?it ?itsdfg))
         (and
            (ask manager make-new (new-dfg ?it %dfg))
        ; replace with call to flattener when available
            (redundant-dfg ?it ?itsdfg)
            (not (dfg ?it ?itsdfg))
            (unco-plel %dfg)
         ))
    (F1 ((new-dfg ?it ?ndfg) (redundant-dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (number ?ndfg nodes ?n)) ;  may not strictly be true
    (F1 ((new-dfg ?it ?ndfg) (redundant-dfg ?it ?itsdfg)
         (number ?itsdfg values ?n))
        (number ?ndfg values ?n))
    (F1 ((new-dfg ?it ?ndfg) (redundant-dfg ?it ?itsdfg)
         (cpath ?itsdfg nodes ?n))
        (and
            (tmp cpath ?ndfg nodes ?n)
            (setr cpath-nodes 2 ?n *) ; should reanalyze
            (not (cpath ?itsdfg nodes ?n))))
    (F1 ((tmp cpath ?ndfg nodes ?n) (number cpath-nodes ?m))
        (and (not (tmp cpath ?ndfg nodes ?n))
             (not (number cpath-nodes ?m))
             (cpath ?ndfg nodes ?m)))
; fakery. should be a call to the dd mgr
    (F1 ((new-dfg ?it ?ndfg) (redundant-dfg ?it ?itsdfg))
        (and (dfg ?it ?ndfg) (not (new-dfg ?it ?ndfg))))
    (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (setr dtime ?n 1 * ?D +))
            ))
  (Estimators (all (
    (F1 ((dfoa ?it) (unco-plel ?itsdfg) (dfg ?it ?itsdfg))
         (exists unco-plel-dfg ?it))
    (F1 ((dfoa ?it) (exists unco-plel-dfg ?it)
         (number advis ?a))
         (setr advis 100 ?a -)) ; ie there is no point
    (F1 ((nor (exists unco-plel-dfg ?it))
         (dfoa ?it) (dfg ?it ?itsdfg) (number advis ?a))
         (setr advis 20 ?a +))
                 )))
  (Application-Rules (
   (F1 ((dfoa ?it)) (and (toggle stop) (stop)))))
        ; this rule is a HACK
  (subtasks (lisp dftran-lisp)) ; need not exist
  (hardware (generic-rtl)))
```

```
(task sehwa-f   ; fast pipelined timer/allocator
  (invar (      ; all of the following must exist
    (dfoa ?it)  ; a particular hardware unit
    (dfg ?it ?itsdfg)   ; its dataflow graph
    (no-plel-loop ?itsdfg); there can be no parallel loop
    (mod-l ?mod)        ; a module library
        ))
  (Add-Rules (
    (F1 ((dfoa ?it))    ; these will all be added to the design
        (ask manager make-new (nmrb ?it %nmrb)))
    (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (setr dtime 2 ?n ^ 5 * ?D +))
    (F1 ((no (tmp decided-timing))
         (tight area ?A) (tight power ?P) (tight ctime ?C))
        (and
            (setr fast-advis ?C 100 *)
            (setr serial-advis ?A 100 *)
            (setr ser-advis ?P 100 *)))
    (F1 ((number fast-advis ?F) (number serial-advis ?S)
          (number ser-advis ?T))
        (and
           (not (number fast-advis ?F))
           (not (number serial-advis ?S))
           (not (number ser-advis ?T))
           (setr parad 2 ?T ?S + / ?F >=)
           (setr serad ?F ?T > ?F ?S > or)))
    (F1 ((boolean serad t) (dfoa ?it) (dfg ?it ?itsdfg))
        (and (tmp decided-timing)
             (semi-serial ?it) (op-c ?it 3) (arch ?it mux)))
    (F1 ((boolean parad t) (dfoa ?it) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (and (tmp decided-timing)
             (ppl ?it) (op-c ?it ?n) (r-c ?it ?n)
             (arch ?it mux)))
    (F1 ((boolean serad ?either))
        (not (boolean serad ?either)))
    (F1 ((boolean parad ?either))
        (not (boolean parad ?either)))
    (F1 ((semi-serial ?it) (dfoa ?it)
         (cpath ?itsdfg nodes ?n) (dfg ?it ?itsdfg)
         (number ?itsdfg values ?m))
        (and
            (setr muxcount 6 ?n 2 * +)
            (r-c ?it ?m)))
    (F1 ((number muxcount ?n) (dfoa ?it))
         (and
                (m-c ?it ?n) (not (number muxcount ?n))))
    (F1 ((tmp calc-regs) (bw cp ?itsdfg ?n))
        (and (not (tmp calc-regs)) (r-c ?it ?n)))))
   (Estimators (all (
    (F1 ((dfoa ?it) (nmrb ?it ?nmrb) (number advis ?a))
```

```
        (setr advis 100 ?a -)) ; ie there is no point
   (F1 ((tight dtime ?D) (number advis ?a)
        (dfoa ?it) (nmrb ?it ?nmrb))
       (setr advis ?D 30 * ?a -))
   (F1 ((tight dtime ?D) (number advis ?a))
        (setr advis ?D 7 * ?a +))
   (F1 ((tight ctime ?C) (number advis ?a))
        (setr advis ?C 6 * ?a +))
   (F1 ((tight dtime ?D) (tight ctime ?C) (number advis ?a))
        (setr advis ?D 2 * ?C 2 * + ?a +))
   (F1 ((tight area ?A) (number advis ?a))
       (setr advis ?A 2 * ?a  +))
   (F1 ((tight power ?P) (number advis ?a))
       (setr advis ?P 2 * ?a +)))))
 (Application-Rules (
(F1 ((dfoa ?it))                    ;  design named it
     (ask ddm make-pnf ?it)) ;  build flat DFG
(F1 ((dfoa ?it) (pnf ?it ?itspnf)) ;  run the program
    (ask manager execute sehwa-f ?itspnf))
(F1 ((dfoa ?it) (success) (res-struc ?it ?rsl))
     (ask manager re-form sehwa-f ?rsl)) ;  it worked
        ;  now build results back into the data
        ; structures
 (F1 ((failure)) (reply failure)))) ;  if it failed

 (subtasks (lisp sehwa-fast-lisp)) ; how it is done
 (hardware (generic-rtl)))
```

```
(task sehwa-s
; slow but good module allocator/timing generator
  (invar (      ; all of the following must exist
    (dfoa ?it)  ; a particular hardware unit
    (dfg ?it ?itsdfg); its dataflow graph
    (no-plel-loop ?itsdfg); there can be no parallel loop
    (unco-plel ?itsdfg) ; may not have unfixd loops
                        ; in parallel
    (mod-l ?mod)        ; a module library
        ))
  (Add-Rules (
; these will all be added to the design
    (F1 ((dfoa ?it))
        (ask manager make-new (nmrb ?it %nmrb)))
    (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (setr dtime 3 ?n ^ 5 * ?D +)) ; expensive.
    (F1 ((no (tmp decided-timing))
         (tight area ?A) (tight power ?P) (tight ctime ?C))
        (and
            (setr fast-advis ?C 100 *)
            (setr serial-advis ?A 100 *)
            (setr ser-advis ?P 100 *)))
    (F1 ((number fast-advis ?F) (number serial-advis ?S)
          (number ser-advis ?T))
        (and
           (not (number fast-advis ?F))
           (not (number serial-advis ?S))
           (not (number ser-advis ?T))
           (setr parad 2 ?T ?S + / ?F >=)
           (setr serad ?F ?T > ?F ?S > or)))
    (F1 ((boolean serad t) (dfoa ?it) (dfg ?it ?itsdfg))
        (and (tmp decided-timing)
             (semi-serial ?it) (op-c ?it 3) (arch ?it mux)))
    (F1 ((boolean parad t) (dfoa ?it) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (and (tmp decided-timing)
             (ppl ?it) (op-c ?it ?n) (r-c ?it ?n)
             (arch ?it mux)))
    (F1 ((ppl ?it) (dfoa ?it) (op-c ?it ?n) (r-c ?it ?m))
        (and
            (not (op-c ?it ?n))
            (not (r-c ?it ?n))
            (setr opcount ?n .9 *)
            (setr r-c ?m .8 *)))
    (F1 ((ppl ?it) (dfoa ?it) (number opcount ?n)
         (number r-c ?m))
        (and
            (not (number opcount ?n))
            (not (number r-c ?m))
            (r-c ?it ?m)
            (op-c ?it ?n)))
```

```
  (F1 ((boolean serad ?either))
      (not (boolean serad ?either)))
  (F1 ((boolean parad ?either))
      (not (boolean parad ?either)))
  (F1 ((semi-serial ?it) (dfoa ?it)
       (cpath ?itsdfg nodes ?n) (dfg ?it ?itsdfg)
       (number ?itsdfg values ?m))
      (and
          (setr muxcount 5 ?n 2 * +)
          (r-c ?it ?m)))
  (F1 ((number muxcount ?n) (dfoa ?it))
       (and
               (m-c ?it ?n) (not (number muxcount ?n))))
          ))
(Estimators (all (
  (F1 ((dfoa ?it) (nmrb ?it ?nmrb) (number advis ?a))
       (setr advis 100 ?a -)) ; ie there is no point
  (F1 ((tight dtime ?D) (number advis ?a))
      (setr advis ?D 2 * ?a +))
  (F1 ((tight ctime ?C) (number advis ?a))
      (setr advis ?C 9 * ?a +))
  (F1 ((tight area ?A) (number advis ?a))
      (setr advis ?A 3 * ?a +))
  (F1 ((tight power ?P) (number advis ?a))
      (setr advis ?P 3 * ?a +)))))
(Application-Rules (
 (F1 ((dfoa ?it)) (and ; test rule
     (results ((compromise-timing ?it)
               (ctrl-table ?it $ctltbl)
               (number operators 2)
               (number muxes 6)
               (number latches 8)
               (struc-graph ?it $newsg)
               (timng-graph ?it $newtg)
               (datfl-timng-struc-binds ?it $mof)))
      (success)))))
(subtasks (lisp sehwa-s-lisp)) ; how it is done
(hardware (generic-rtl generic-gate)))
```

```
(task setlib              ; choose module library
  (invar ((no (mod-l ?any)) (dfoa ?it)))
  (Add-Rules (
    (F1 ((number dtime ?D))
        (setr dtime 5 ?D +))
    (F1 ((tight area ?A) (tight power ?P) (tight ctime ?C))
        (and
            (setr fast-advis ?C 10 * ?P 2 * ?A 2 * + +)
            (setr small-advis ?C 3 * ?P 6 * ?A 10 * + +)
            (setr lowp-advis ?C 2 * ?P 10 * ?A 6 * + +)))
    (F1 ((number fast-advis ?F) (number small-advis ?S)
         (number lowp-advis ?L))
        (and
            (setr faster ?S ?F >= ?L ?F >= and)
            (setr slower ?F ?S > ?L ?S >= and)
            (setr lowper ?S ?L > ?F ?L > and)))
    (F1 ((boolean faster t))
        (and (mod-l $fastlib)
            (number av-mod-sz 100000) (number av-reg-sz 23000)
            (number av-mux-sz 10000) (number av-mod-ct 10)
            (number av-reg-ct 2) (number av-mux-ct 1)))
    (F1 ((boolean slower t))
        (and (mod-l $smalllib)
            (number av-mod-sz 40000) (number av-reg-sz 10000)
            (number av-mux-sz 8000) (number av-mod-ct 20)
            (number av-reg-ct 5) (number av-mux-ct 5)))
    (F1 ((boolean lowper t))
        (and (mod-l $lowplib)
            (number av-mod-sz 60000) (number av-reg-sz 15000)
            (number av-mux-sz 7000) (number av-mod-ct 40)
            (number av-reg-ct 10) (number av-mux-ct 7)))
; now clean up
    (F1 ((number fast-advis ?f) (number small-advis ?s)
         (number lowp-advis ?l))
        (and
            (not (number lowp-advis ?l))
            (not (number fast-advis ?f))
            (not (number small-advis ?s))))
    (F1 ((boolean faster ?b1) (boolean slower ?b2)
         (boolean lowper ?b3))
        (and
            (not (boolean faster ?b1))
            (not (boolean lowper ?b3))
            (not (boolean slower ?b2))))))
  (Estimators (all (
        (F1 ((dfoa ?it) (number advis  ?a))
            (setr advis ?a 5 +)))))
```

```
(Application-Rules (
      ; must be fixed to look at actual tightnesses
      (F1 ((dfoa ?it) (pinging area)) ; this works...
          (mod-l compact))
      (F1 ((dfoa ?it) (pinging power))
          (mod-l cmos-lp))
      (F1 ((dfoa ?it) (pinging ctime))
          (mod-l fast))
      (F1 ((dfoa ?it)) (success))))
(subtasks (lisp setlib-lisp)) ; how it is done
(hardware (generic-rtl)))
```

```
(task build-ctrl; build a controller
  (invar (
 ; all of the following must exist for this to apply
    (dfoa ?it) ; a particular hardware unit
    (c-tbl ?it ?itsctbl)
    (subunit ?it ?itsctrl)
    (c-styl ?itsctrl ?any)))
  (Add-Rules (
        (F1 ((dfoa ?it) (subunit ?it ?itsctrl))
            (implemented ?itsctrl))
        (F1 ((number dtime ?D) (dfg ?it ?itd)
            (number ?itd nodes ?N) (tight area ?A))
            (setr dtime .5 ?A 1 - + ?N / ?D +))))
  (Estimators (all (
   (F1 ((dfoa ?it)) (setr advis 50)))))
    ; if it can be done it should be
  (Application-Rules (
   (F1 ((dfoa ?it) (hdwe ?itsctrl control)) (and
       (results ((controller ?it ?itsctrl)))
       (success)))))
  (subtasks (lisp build-ctrl-lisp)) ; how it is done
  (hardware (generic-rtl)))
```

```
(task build-ctbl; build a control table
  (invar (
    ; all of the following must exist for this to apply
    (dfoa ?it) ; a particular hardware unit
    (stg ?it ?itsstg); the complete structure graph
    (subunit ?it ?itsctrl)
    (c-styl ?itsctrl ?any)))
  (Add-Rules (
        (F1 ((dfoa ?it))
            (ask manager make-new (c-tbl ?it %ctbl)))
        (F1 ((number dtime ?D) (dfg ?it ?itd)
             (number ?itd nodes ?N) (tight area ?A))
            (setr dtime .25 ?A 1 - + ?N 5 * / ?D +))))
  (Estimators (all (
   (F1 ((no (c-tbl ?it ?any)) (dfoa ?it))
       (setr advis 10))
   (F1 ((dfoa ?it) (c-tbl ?it ?any)) (setr advis -100)))))
  (Application-Rules (
   (F1 ((dfoa ?it) (hdwe ?itsctrl control)) (and
       (results ((controller ?it ?itsctrl)))
       (success)))))
  (subtasks (lisp build-c-tbl-lisp)) ; how it is done
  (hardware (generic-rtl)))
```

```
(task build-stg; build a structure graph
  (invar (
    ; all of the following must exist for this to apply
    (dfoa ?it) ; a particular hardware unit
    (nmrb ?it ?itsnmrb)))
  (Add-Rules (
        (F1 ((dfoa ?it))
            (ask manager make-new (stg ?it %stg)))
        (F1 ((number dtime ?D) (dfg ?it ?itd)
             (number ?itd nodes ?N) (tight area ?A))
            (setr dtime .25 ?A 1 - + ?N 2 * / ?D +))))
  (Estimators (all (
    (F1 ((dfoa ?it) (subunit ?it ?ctrl)
         (c-styl ?ctrl ?fixed)
         (arch ?it ?known) (number advis ?a))
        (setr advis ?a 10 +))
    (F1 ((dfoa ?it) (stg ?it ?itsstg))
        (setr advis -100)))))
  (Application-Rules (
   (F1 ((dfoa ?it) (hdwe ?itsctrl control)) (and
       (results ((controller ?it ?itsctrl)))
       (success)))))
  (subtasks (lisp build-stg-lisp)) ; how it is done
  (hardware (generic-rtl)))
```

```
(task sel-ctrl
  (invar ((dfoa ?it) (dfg ?it ?itsdfg)))
  (Add-Rules (
; this is a little complicated because
; we want to simulate a choice
; from among a couple of alternatives
    (F1 ((dfoa ?it))
         (ask manager make-new (new-subunit ?it %ctrlr)))
    (F1 ((new-subunit ?it ?name))
        (and
            (subunit ?it ?name)
            (hdwe ?name ctrl)
            (not (new-subunit ?it ?name))))
    (F1 ((dfoa ?it) (dfg ?it ?itsd) (number ?itsd nodes ?N))
        (and (setr nodes-gt-40 40 ?N >)
             (setr nodes-lt-20 20 ?N <)))
    (F1 ((boolean nodes-lt-20 t) (dfoa ?it)
         (subunit ?it ?ctrl) (hdwe ?ctrl ctrl))
        (ask manager make-new (c-styl ?ctrl random)))
    (F1 ((boolean nodes-gt-40 t) (dfoa ?it)
         (subunit ?it ?ctrl) (hdwe ?ctrl ctrl))
        (ask manager make-new (c-styl ?ctrl mcpgm)))
    (F1 ((boolean nodes-lt-20 f) (boolean nodes-gt-40 f)
          (dfoa ?it) (subunit ?it ?ctrl) (hdwe ?ctrl ctrl))
        (ask manager make-new (c-styl ?ctrl pla)))
    (F1 ((boolean nodes-lt-20 ?v1) (boolean nodes-gt-40 ?v2))
        (and (not (boolean nodes-lt-20 ?v1))
             (not (boolean nodes-gt-40 ?v2))))
    (F1 ((number dtime ?D) (dfg ?it ?itsdfg)
         (number ?itsdfg nodes ?n))
        (setr dtime ?n 5 * ?D +))))
  (Estimators (all (
    (F1 ((dfoa ?it) (nmrb ?it ?nmrb) (number advis  ?a))
        (setr advis 10 ?a +))
    (F1 ((dfoa ?it) (mod-1 ?mod) (number advis ?a))
        (setr advis 3 ?a +))
    (F1 ((dfoa ?it) (c-styl ?ctrl ?any))
        (setr advis -10)))))
  (Application-Rules (
   (F1 ((dfoa ?it)) (and
                          (results ((hdwe ctrlr2 control)
                                    (c-styl ctrlr2 pla)))
                          (success)))))
  (subtasks (lisp sel-ctrl-lisp)) ; how it is done
  (hardware (generic-rtl)))
```

```
(hdwe generic-rtl ;hardware description frame
  (level (rtl))
  (to-build (random (maha sehwa-f sehwa-s setlib
        analyze-df dftran
        build-stg build-ctbl sel-ctrl build-ctrl)))
  (Done-Rules (
     (F1 ((dfoa ?it) (subunit ?it ?ctrl) (hdwe ?ctrl ctrl)
          (implemented ?ctrl) (stg ?it ?itsstg))
         (and (implemented ?it) (done ?it)))))
  (Estimators  (all (
; general hypotheses and facts
    (F1 ((dfoa ?it))
        (and (number area 0) (number power 0) (number ctime 0)))
    (F1 ((nor (serial ?it) (semi-serial ?it)
         (ppl ?it) (parallel ?it)) (dfoa ?it))
        (and
          (semi-serial ?it)      ; the default is vanilla
          (echo assuming it is semi-serial
                because no timing discipline)))
    (F1 ((dfoa ?it) (stg ?it ?itstg))
        (ask counter count modules ?itstg))
    (F1 ((dfoa ?it) (number operators ?n) (stg ?it ?itstg))
        (st-based))
    (F1 ((dfoa ?it) (op-c ?it ?N)) (st-based))
    (F1 ((no (st-based))) (df-based))
```

```
; area (square lambda)
    (F1 ((df-based) (ppl ?it) (number ?itsdfg nodes ?n)
         (number ?itsdfg values ?m) (number av-mod-sz ?x)
         (number av-reg-sz ?y))
        (setr area ?n ?x * ?m ?y * +))
    (F1 ((df-based) (parallel ?it) (number ?itsdfg nodes ?n)
         (number ?itsdfg values ?m) (number av-mod-sz ?x)
         (number av-reg-sz ?y))
        (setr area ?n ?x * 3 ?m / ?y * +))
    (F1 ((df-based) (serial ?it) (number av-mod-sz ?x)
         (number av-reg-sz ?n) (number ?itsdfg values ?p))
                (setr area  ?x 3 * ?n 5 ?p / * +))
    (F1 ((df-based) (semi-serial ?it) (number ?itsdfg nodes ?n)
         (number ?itsdfg values ?m) (number av-reg-sz ?y)
         (number av-mod-sz ?x))
        (setr area 5 ?m / ?y * 2.5 ?x * +))
    (F1 ((nor (op-c ?it ?n)) (dfoa ?it) (st-based)
         (number operators ?M))
        (op-c ?it ?M))
    (F1 ((nor (r-c ?it ?n)) (dfoa ?it) (st-based)
         (number latches ?M))
        (r-c ?it ?M))
    (F1 ((nor (m-c ?it ?n)) (dfoa ?it) (st-based)
         (number muxes ?M))
        (m-c ?it ?M))
    (F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
         (op-c ?it ?N) (serial ?it))
        (setr muxes 3 ?N *))
    (F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
         (op-c ?it ?N) (serial ?it))
        (setr latches 5 ?N *))
    (F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
         (number ?itsdfg ?nodes ?N)
         (serial ?it) (dfg ?it ?itsdfg))
        (setr muxes .3 ?N * 2 +))
    (F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
         (number ?itsdfg ?nodes ?N)
         (serial ?it) (dfg ?it ?itsdfg))
        (setr latches .2 ?N * 3 +))
    (F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
         (op-c ?it ?N) (ppl ?it))
        (m-c ?it ?N))
    (F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
         (op-c ?it ?N) (ppl ?it))
        (r-c ?it ?N))
    (F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
         (number ?itsdfg ?nodes ?N)
         (ppl ?it) (dfg ?it ?itsdfg))
        (m-c ?it ?N))
    (F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
         (number ?itsdfg ?values ?N)
         (ppl ?it) (dfg ?it ?itsdfg))
```

```
    (r-c ?it ?N))
(F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
     (op-c ?it ?N) (semi-serial ?it))
    (setr muxes  1.5 ?N *))
(F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
     (op-c ?it ?N) (semi-serial ?it))
    (setr latches  2.5 ?N *))
(F1 ((no (m-c ?it ?n)) (dfoa ?it) (st-based)
     (number ?itsdfg ?nodes ?N)
     (semi-serial ?it) (dfg ?it ?itsdfg))
    (setr muxes .7 ?N *))
(F1 ((no (r-c ?it ?n)) (dfoa ?it) (st-based)
     (number ?itsdfg ?values ?N)
     (semi-serial ?it) (dfg ?it ?itsdfg))
    (setr latches .3 ?N * 2 +))
```

```
     (F1 ((st-based) (dfoa ?it) (op-c ?it ?n)
          (number av-mod-sz ?x))
         (setr area ?n ?x *))
     (F1 ((st-based) (dfoa ?it) (r-c ?it ?n)
          (number av-reg-sz ?x) (number area ?A))
         (setr area ?A ?n ?x * +))
     (F1 ((st-based) (dfoa ?it) (m-c ?it ?n)
          (number av-mux-sz ?x) (number area ?A))
         (setr area ?A ?n ?x * +))
    (F1 ((number area ?a) (arch ?it mux) (dfoa ?it))
         (setr area 3 ?a *))
    (F1 ((number area ?a) (arch ?it bus) (dfoa ?it))
         (setr area 2 ?a *))
                 ; guess routing area

; ctime estimation (nsec)
   (F1 ((dfoa ?it) (ppl ?it) (number av-mod-ct ?x)
        (number av-reg-ct ?y))
       (setr ctime ?x ?y + 2 *)); Because I say so.
   (F1 ((dfoa ?it) (serial ?it) (number av-mod-ct ?x)
        (number av-reg-ct ?n)
        (number ?itsdfg values ?p)
        (number ?itsdfg nodes ?m))
       (setr ctime  ?n ?p * ?m ?x * + 2 *))
   (F1 ((dfoa ?it) (semi-serial ?it)
        (number ?itsdfg nodes ?n)
        (number ?itsdfg values ?m) (number av-reg-ct ?y)
        (number av-mod-ct ?x))
       (setr ctime 2 ?y ?x + 2 * ?m ?y * ?n ?x * + + /))
                    ; I like HP calculators too
   (F1 ((dfoa ?it) (parallel ?it) (cpath ?itsdfg nodes ?n)
        (number av-mod-ct ?x))
       (setr ctime ?n ?x 1.1 * *)); should use CP algorithm
; this could be extended for
; structure-based ctime estimation, and the
; difference between throughput and latency established.
; estimation of power (milliwatts). This
; is not a very good rule set for this... it is
; mostly just for testing 4D interpolation.
   (F1 ((number power 0) (st-based) (dfoa ?it) (op-c ?it ?N))
        (setr power 40 ?N *))
   (F1 ((st-based) (dfoa ?it) (r-c ?it ?n)
        (number power ?p))
       (setr power ?p 8 ?n * +))
   (F1 ((st-based) (dfoa ?it) (m-c ?it ?n)
        (number power ?p))
       (setr power ?p 4 ?n * +))
   (F1 ((number power 0) (ppl ?it) (dfoa ?it)
        (dfg ?it ?itsdfg)
        (number ?itsdfg nodes ?N))
       (setr power 40 ?N *))     ; crude but effective
   (F1 ((number power 0) (serial ?it) (dfoa ?it)
```

```
        (dfg ?it ?itsdfg) (number ?itsdfg nodes ?N))
        (setr power 8 ?N * 40 +))
   (F1 ((number power 0) (semi-serial ?it) (dfoa ?it)
        (dfg ?it ?itsdfg) (number ?itsdfg nodes ?N))
        (setr power 40 3 * ?N 8 * + ))
   (F1 ((number power 0) (parallel ?it) (dfoa ?it)
        (dfg ?it ?itsdfg)
        (cpath ?itsdfg nodes ?N) (number ?itsdfg nodes ?M))
        (setr power 40 ?N * 28 ?M * +))
   (F1 ((tight ctime ?C) (number power ?P))
        (setr power 1 ?C + 2 * ?P *))
        ; up to four times the power
   (F1 ((tight area ?A) (number power ?P))
        (setr power .3 ?A * 1 - ?P *))
        ; one-third decrease for area
; rules for reportage
   (F1 ((no (number dtime 0))) (estimated dtime))
   (F1 ((no (number area 0))) (estimated area))
   (F1 ((no (number ctime 0))) (estimated ctime))
   (F1 ((no (number power 0))) (estimated power)))))
```