Technical Report 402

Flexibility and Efficiency in a Computer Program for Designing Circuits

Drew Vincent Mcdermott

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

FLEXIBILITY AND EFFICIENCY
IN A COMPUTER PROGRAM FOR DESIGNING CIRCUITS

Drew Vincent McDermott

Massachusetts Institute of Technology

June 1977

Illustrations by Karen Prendergast

Revised version of a dissertation submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

This empty page was substituted for a blank page in the original document.

ABSTRACT

This report is concerned with the problem of achieving flexibility (additivity, modularity) and efficiency (performance, expertise) simultaneously in one AI program. It deals with the domain of elementary electronic circuit design. The proposed solution is to provide a deduction-driven problem solver with built-in control-structure concepts. This problem solver and its knowledge base in the application areas of design and electronics are described. The program embodying it is being used to explore the solution of some modest problems in circuit design. It is concluded that shallow reasoning about problem-solver plans is necessary for flexibility, and can be implemented with reasonable efficiency.

Acknowledgments

I thank Gerald Sussman, my advisor, for much good advice; Sussman and Allen Brown for help with electronics; Mitch Marcus and Charles Rich for ideas on control; Bob Moore and Arthur Nevins for predicate calculus; David Marr and Scott Fahlman for ideological consultation; my readers Marvin Minsky and Vaughn Pratt for useful comments; and Jon Doyle for careful reading and substantive suggestions in the later stages of this research. Marr, Patrick Winston, Guy Steele, and several others also made good suggestions to improve the organization of this work.

Finally, I thank Judi for, among other things, moral support; and myself for typing many drafts of what seemed like meaningless gibberish

Contents

I Introduction 7

I.A The Problem 7
I.B A Rule-Based Problem Solver 16
I.C Supplying Rules for Design 22
I.D Relation to Previous Work 33
I.D.1 Problem Solving and Reasoning 33
I.D.2 Electronics and Design 39

II Expressing Knowledge in NASL 43

II.A The Natural History of Actions 46
II.B Interpretation and Inference 56
II.B.1 The NASL Interpreter 57
II.B.1.1 Selecting a Task to Work On 57
II.B.1.2 Executing Tasks 59
II.B.1.2.1 Primitive and Problematic Tasks 60
II.B.1.2.2 Primary and Secondary Tasks 63
II.B.2 STP -- The Stupid Theorem Prover 64
II.C Choice and Rephrasing 71
II.C.1 The Choice Protocol 73
II.C.2 Rephrasing 76
II.D Dependencies Among Data and Tasks 78
II.E Handling Mistakes 82
II.F Programmer's Guide 84
II.F.1 Predicate-Calculus Techniques 86
II.F.2 NASL Programming Techniques 88

III Design of Hierarchical Systems 90

III.A The Representation of Knowledge about Devices 99
III.A.1 Hierarchies of Device Types 99
III.A.2 The Representation of Device Diagrams 101
III.B Design Actions and Plans 106
III.B.1 DESIGN 107
III.B.2 Making Things 112
III.B.3 Constraints 114
III.B.4 Changing Devices 118
III.C Composition of Partial Solutions 120
III.D Constraint Collapse 121
III.E Programmer's Guide 126

IV Electronics 128

IV.A Physics 128
IV.A.1 Connections and Constraints on Components 128
IV.A.2 Signals 132
IV.A.3 Multiple Models of Linear Systems 135
IV.B Electronic Design Knowledge 137
IV.B.1 Rephrasing Electronic Design Problems 137
IV.B.2 Reconciling Partial Solutions 138
IV.B.3 Changing Circuits 141
IV.B.4 Electronics Analysis Knowledge 145
IV.C Device Schemata 146
IV.D Programmer's Guide 148

V Results 150

V.A Using DESI 150
V.A.1 Loading and Running the Program 150
V.A.2 DESI Talks to You 151
V.A.3 You Talk to DESI 152
V.B Experimental Results 153
V.B.1 A Simple Amplifier 155
V.B.2 Converting a Square Wave into a Sine Wave 171
V.B.3 NOAH in NASL 176

VI Conclusions 178

VI.A Successes 180
VI.B Failures 186
VI.C Further Work 190

Appendix 1 -- NASL Syntax and Informal Semantics 194

Appendix 2 -- A Listing of DESI 202

Appendix 3 -- A Listing of ZORCH 216

Appendix 4 -- Details of STP for Theorem Provers 251

Bibliography 255

Figures

Figure I.1 Redescribing a Design Problem 9
Figure I.2 Two Circuits Suggested by Parts of the Problem 10
Figure I.3 A Cascade of the Two Partial Circuits 11
Figure I.4 Signal Conversion Problem 12
Figure I.5 Spectrum of Square Wave 13
Figure I.6 Spectrum of Sine Wave 13
Figure I.7 RC Filter 14
Figure I.8 A Circuit for Adding a Pole 15
Figure I.9 Structure of DESI 23
Figure I.10 A Rephrasing Subtask 29
Figure I.11 Rephrased Task Network 30
Figure I.12 Retrieved Circuit and Constraint Task Network 31
Figure II.1 A Task Net, or "Plan" 50
Figure II.2 Task Network with Subnets 52
Figure II.3 Logical Taxonomies of Tasks 53
Figure II.4 Life History of a Task 58
Figure II.5 Enablement Relations in Subnets 62
Figure III.1 Function-Structure Graph 92
Figure III.2 A Two-Stage Cascade 94
Figure III.3 An LC-coupled Amplifier 95
Figure III.4 A Hierarchy of Types of Device Types 100
Figure III.5 Devices in The Type Hierarchy 101
Figure III.6 Design Action Taxonomy 107
Figure III.7 Design Rephrasing Plan Schema 109
Figure III.8 Rephrased Design 110
Figure III.9 Quantity-Value Protection Plan Schema 117
Figure III.10 Radio Spectrum With Two Stations 122
Figure III.11 Relevant Constraints 123
Figure III.12 Constraint Network 124
Figure IV.1 Terminals and Nodes 129
Figure IV.2 Composite Device Terminals 130
Figure IV.3 Inserting a Device into a Node 130
Figure IV.4 Ports and Nests 131
Figure IV.5 Fourier Transform of an Offset Square Wave 134
Figure IV.6 Bias Plans 142
Figure IV.7 General BJT Coupling Plan 143
Figure IV.8 Common-Emitter Direct Voltage Coupling 144
Figure IV.9 General and Specialized Emitter-Coupled Pairs 147
Figure VI.1 Provinces of Artificial Intelligence 181
Figure VI.2 The Rule-Based Utopia 182

This empty page was substituted for a blank page in the original document.

I Introduction

I.A The Problem

This thesis reports on the exploration of a classic AI controversy regarding the representation and use of knowledge: the trade-off between flexibility (or modularity or additivity) on the one hand, and efficiency (or expertise or performance) on the other. It focuses on the knowledge required for designing elementary electronic circuits. The conclusions I have reached are that this kind of flexibility requires all important operations to be mediated by explicit rules driven by changes to an associative data base; and that the inevitable inefficiency of this organization can be controlled. The proposed approach has been tested by implementation of an extensible design program called DESI.

The theory of design that I have implemented is based on the idea of "functional reasoning." (Freeman and Newell, 1971) A problem is stated as a property which an electronic circuit is to have. The system searches its memory for circuits whose known functions fit the requirement. In the attempt, it constrains the connectivity and component values of the circuit until enough constraints have accumulated to find values which satisfy the original property. This simple theory must be complicated in various ways: A requirement may not be expressed in a form which triggers suggestions of familiar circuits; it may be necessary to transform the requirement until it does. More than one device type may be brought to mind in a situation; there must be criteria for deciding among them, or ways of synthesizing new circuits that perform the functions of all the ones retrieved. A suggested circuit may not work out; the theory must specify how plans are changed.

I require that the embodiment of this theory be "additive"; that is, to

the extent possible that new knowledge be expressed by new formulas rather than by changes to old. This is partly because of the ease with which such a system can absorb new information; and partly because a creative designer requires the ability to see the individual parts of its various, often conflicting, plans and goals. For this reason, the theory is embodied as a "***rule-based***" system.

By a "rule-based" system I mean a system which makes progress by pattern-driven operations on a data base. There are several paradigms for such systems; the classical ones are predicate-calculus theorem provers (Nevins, 1974a), production systems (Newell, 1973a), and AI languages with pattern-directed procedure invocation. (Hewitt, 1972, Bobrow and Raphael, 1974) In what follows, I will attempt a synthesis of good features of all of these. The result may be described as a system in which plans are assembled piece by piece. The rules used resemble predicate-calculus implications. They differ in these ways: they may be used to infer what tasks are required or what solutions are possible; they are less constrained in the kind of inference rule and "self-referential" deductions allowed; they specify how they are to be used; and they come in larger, better organized chunks than is traditional in predicate-calculus applications.

Before elaborating further on these requirements, let me bring these problems to life with two examples from elementary electronic design, illustrating as I go how DESI deals with them. The first example shows how choices must often be based on knowledge of current plans. The second example illustrates some of the other complexities I mentioned. (These informal scenarios are meant to give you a picture of the design problems DESI is meant to handle, not the structure of the program or its actual behavior. I will go over these problems again, once in this chapter to illustrate the

representation and use of knowledge by the system, and again in Chapter V to show the performance of the actual implementation.)

Redescribing Problems and Choosing Solutions

Imagine that DESI is given the problem, "Design an amplifier with an input resistance of 30 kohm and a voltage gain of 5." For now, let us assume this problem will be broken into the three subrequirements, "amplifier," "input resistance = 30 kohm," and "v-gain = 5." (This must be done with care, since these three characteristics are of rather different kinds.) This is only part of the problem, for these fragments of the original problem are too precise to be suggestive; DESI must further alter the description so that these numbers become "range descriptions" like, "high input impedance" and "moderate voltage gain." (See Fig. I.1.)

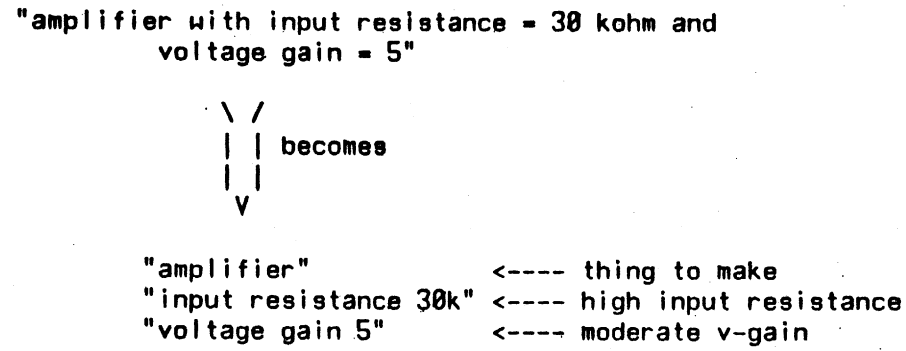

Figure I.1 Redescribing a Design Problem

Once the problem has been described in this form, its fragments trigger suggestions of possible solutions. For example, in the context of making an amplifier, "high input impedance" should suggest "common-collector amplifier," and "moderate voltage gain" should suggest "common-emitter amplifier." (Fig. I.2.)

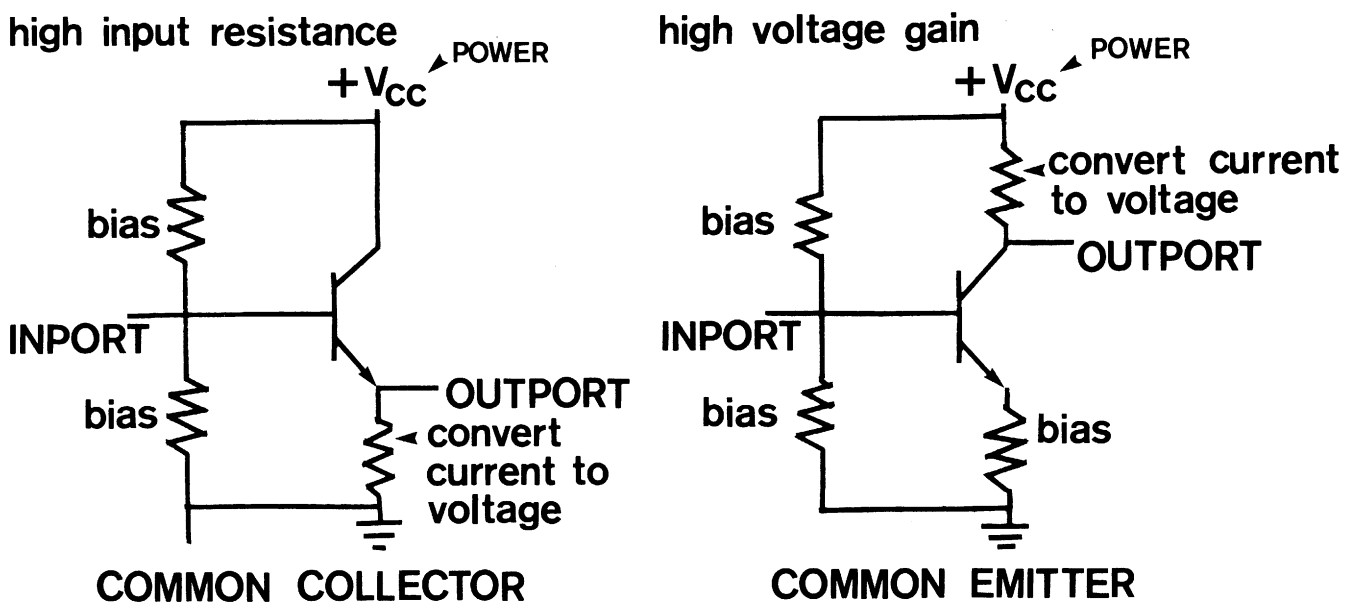

Figure I.2 Two Circuits Suggested by Parts of the Problem

If just one of these had been suggested, the problem would be easy; DESI could select a standard schema for the type that came to mind and make sure the given numerical constraints could indeed be satisfied. What must it do when two or more options occur? In some cases, all but one may be excluded on the basis of further considerations beyond the simple problem features that suggested the competing options; often, however, what is required is a synthesis which combines two suggestions to provide the functions of both. In this case, DESI must possess information to the effect that an option suggested because of what it does for an amplifier's input resistance may be "cascaded" with another option.

So far, I have drawn these circuits as if they always contained the parts shown. However, the notions of "common-collector" and "common-emitter"

amplifier each corresponds to a range from general to specialized circuits. When a common-emitter amplifier *simpliciter* is desired, the circuit of Fig. I.2 is selected almost without thought. But a practiced designer knows that the "abstract idea" of this circuit may be realized in other ways. To cascade the two circuits of Fig. I.2, DESI would not just "draw" the same two pictures and cram them together in some way. Instead, it chooses more general diagrams of these circuits, and reconsiders how they are to be biased and coupled. This will involve further choices. The result is the circuit of Fig. I.3.

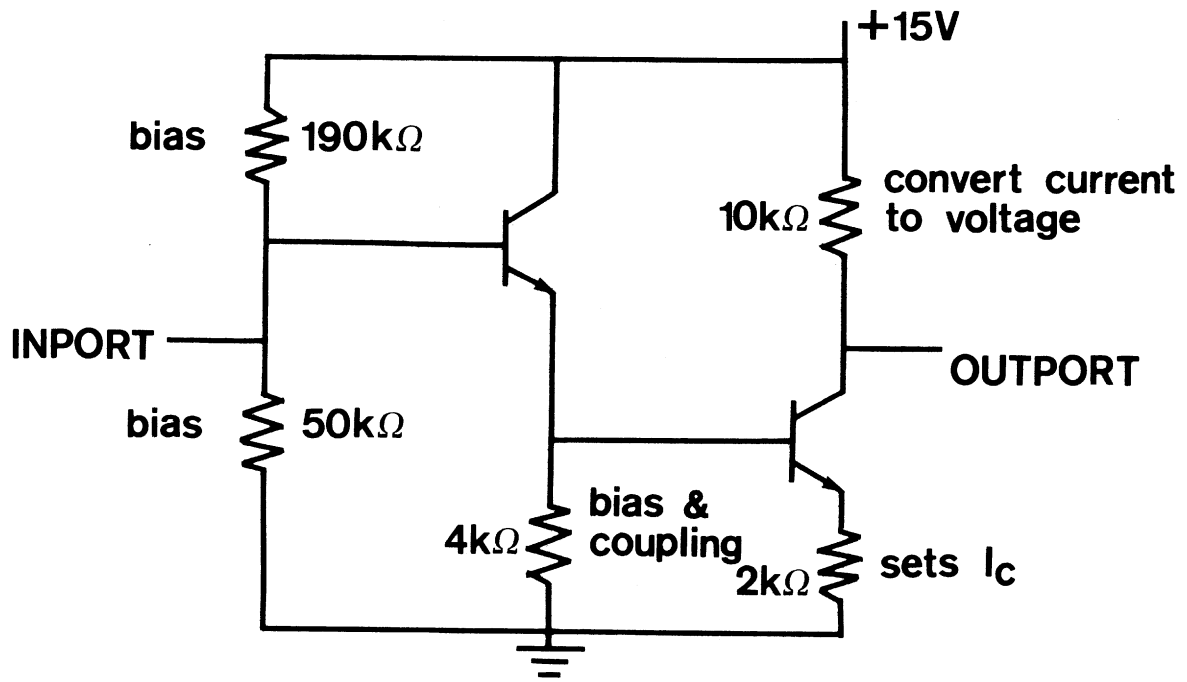

Figure I.3 A Cascade of the Two Partial Circuits

Finally, the numerical constraints set aside in favor of more revealing descriptors are taken up again to give the component values shown in the figure.

Transforming Problem and Solution

In this second example, I wish to show how much more complicated the phases of design can get. Imagine that the problem given is

Design a network which converts a 1kHz square wave of amplitude 1V into a sine wave with the same frequency and amplitude.

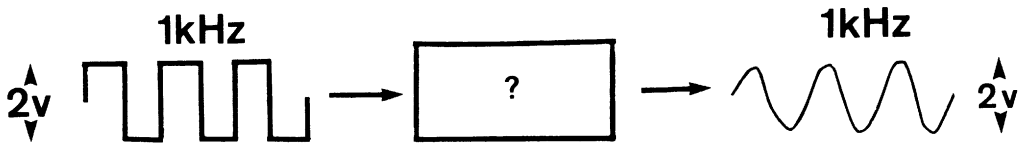

Figure I.4 Signal Conversion Problem

Even if you know nothing about electronics, it may be worth thinking about this problem for a minute before going on. (You don't have to, but the problem can be amusing, and illustrates an interesting and common phenomenon of problem solving.)

* * *

If you know some elementary mathematics, it probably occurred to you to take the Fourier series of each of these signals. In this "*frequency domain*," the problem becomes:

Design a network which converts a signal with spectrum

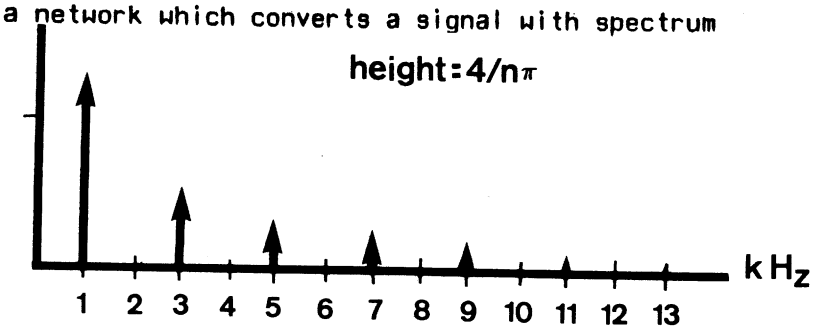

Figure I.5 Spectrum of Square Wave

into a signal with spectrum

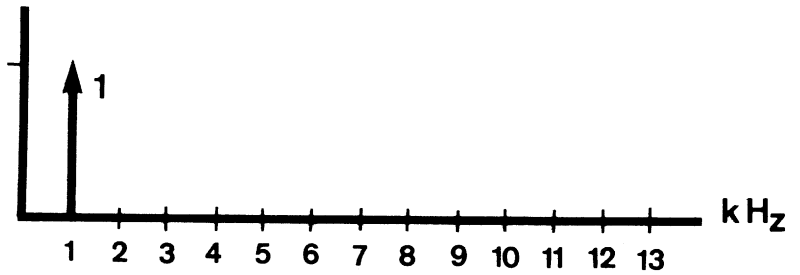

Figure I.6 Spectrum of Sine Wave

such that the amplitude of the spike at 1kHz is 1V.

If you know some electronics, it might then have occurred to you to try a *low-pass filter* circuit like

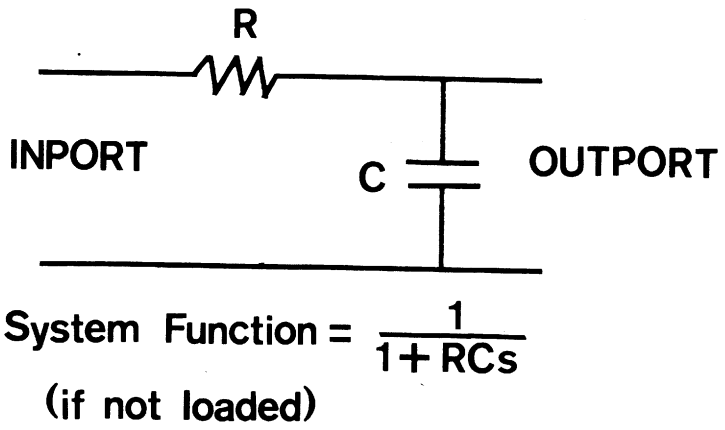

Figure I.7 RC Filter

as your answer, and then try as before to finish the problem off by assigning values to the primitive components (here, the capacitor and resistor) which satisfy the constraints we have discovered.

The interesting constraints on this circuit may be stated as follows,

"Make the amplitude of the output spike at 3 kHz less than 10% of the amplitude of the output spike at 1 kHz."

"Make RC be the reciprocal of the frequency 1 kHz (in rad/sec)."

"The ratio of the amplitudes of the outputs at two frequencies depends on the amplitudes of the inputs and the selectivity of a device."

"The selectivity of an RC filter is ..."

DESI can derive these constraints from the statement of the problem (starting with a lot of knowledge about RC filters and frequency-domain operations).

Unfortunately, these constraints cannot be solved simultaneously. The circuit given will make a square wave look "rounder," but the approximation to a sine wave will not be good enough. The constraint that deserves the blame in this case is that on the selectivity of an RC filter. How can this be improved? One way is by "adding a pole" to its system function with this circuit:

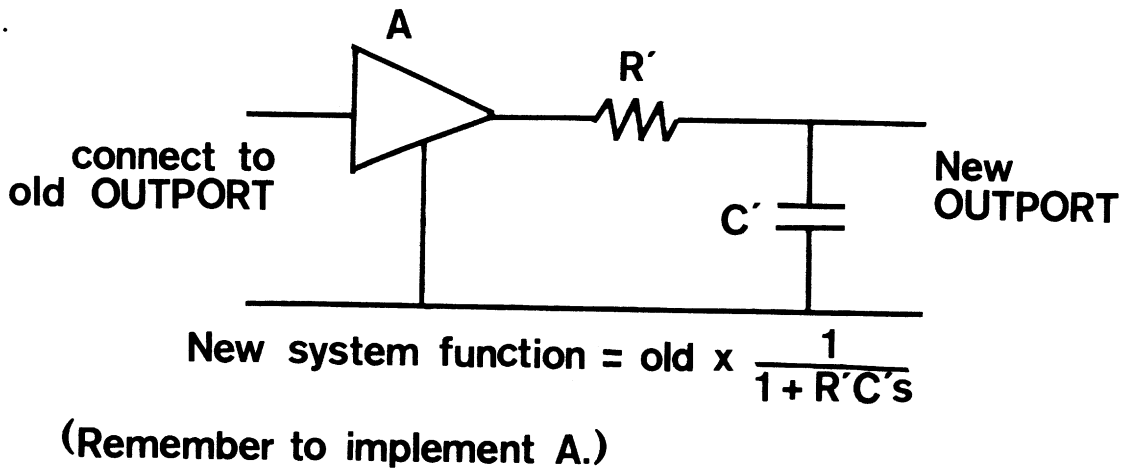

Figure I.8 A Circuit for Adding a Pole

This makes DESI's view of the solution much more promising. (I won't pursue this example any further, because the current implementation lacks the competence to go any further.)

Let me list the various types of information that I appealed to in my brief overview of how such problems may be solved. First, DESI needs knowledge about ***transforming the problem*** into a tractable form; this ranges from a relatively simple sorting out of multiple requirements, to a more difficult transformation from a time-domain to a frequency-domain description of a problem. Second, and quite important, there had to be a basis for ***choosing*** among more than one approach. Third, several ***constraints*** had to be satisfied in a ***consistent*** way. This required knowledge of the ***physics*** of electronic circuits. Fourth, we had to be able to ***change plans*** when our first try failed.

To make all these kinds of information usable, DESI has to be able to reason about its current plans and goals. Transforming a problem may be seen

as redescribing the topmost goal. Choice of a solution to one problem often depends on the other problems under consideration. The calculation of a design quantity to satisfy one constraint is pointless unless all the other constraints on that quantity are taken into account. And, of course, one cannot change plans without knowing what they are. An organization which makes using such knowledge practical has been the goal of this research.

I.B A Rule-Based Problem Solver

Here is my thesis: problems are solved by reducing them to subproblems. Some of these subproblems result in action, others in constraints on action. As the solution progresses, the way in which new subproblems are approached depends more and more on the state of other subproblem solutions, that is, on the requirements derived from the physics of the evolving solution and on the goal structures that have already been elaborated. It is impossible to know, as new facts are discovered, what subsequent subproblems will depend on them, so all such facts must be stored in the same communal data base and accessed whenever they become relevant to a later problem reduction.

This is accomplished by implementing DESI as a set of rules manipulated by a ***rule-based problem solver*** called NASL. A rule-based system (Shortliffe, 1976, Davis and King, 1975) is one in which knowledge is expressed as conceptually small units called ***rules***.

There are two sources of inefficiency in a system organized this way: the overhead paid for storing almost all knowledge in the same associative data base, and the nondeterminism inherent in the possibility that more than one rule may apply to a problem. The first kind of inefficiency is the price of flexibility, but it can be limited by proper organization. One important

principle of organization is allow rules to come in well-organized chunks. In DESI; these "packets" of rules (McDermott, 1975) are used to represent circuit diagrams, signal descriptions, partial plans for solving problems, and groups of rules for making choices.

The second source of inefficiency, nondeterminism, can be controlled by confining it to the information retrieval module. Above this lowest level, potential nondeterminism is shut off by applying "choice rules" in ambiguous situations.

In this section and the next, I will sketch the form and content of DESI's rules. This sketch will be filled in in Chapters II, III, and IV. Chapter V gives the results that have been obtained by implementing it.

In any rule-based system, each rule is associated with a ***pattern*** by which the system accesses it. The system also maintains a data structure, the "active processing site," that is intended to describe important aspects of the current situation. Rules are ***matched*** against this structure, and rules whose patterns match are applied in some way.

The potential advantages of a rule-based system are these: (1) the system can see what it is doing because important steps occur at standard times in a standard way; (2) the system can keep track of its deductions and/or actions in order to explain or undo them; (3) the system can be augmented simply by adding new rules.

Realizing these potential advantages has not been easy. There are three classic types of rule-based systems:

(1) ***Predicate-Calculus Theorem Provers*** -- Here the rules are axioms and the currently active processing site is that rule which is being treated as a goal. Applying a rule generates new rules or answers to the problem at hand. (Robinson, 1965, Nilsson, 1971, Nevins, 1974a, R. Moore, 1975)

(2) ***Production Systems*** -- Here the rules are condition-action pairs, or "productions," and the current site is a small list called the "Short-Term

Memory," or STM. Applying a rule consists of performing manipulations on the STM or doing simple input/output operations. (Rychener, 1976, Newell, 1973a)

(3) ***Artificial Intelligence Programming Languages*** -- These languages may be said to be rule based if pattern-directed procedure invocation is taken as rule application and if such procedures are taken as rules. The processing site is a flexible record-oriented data base, in particular, the records currently being added, deleted, or retrieved. These languages include PLANNER (Hewitt, 1972) and its descendants QA4 (Rulifson *et. al.*, 1972) and Conniver (Sussman and McDermott, 1972).

(More specific examples will be mentioned for comparison with NASL later.)

All of these systems have suffered from problems which have kept them from realizing their potential. The predicate-calculus systems are the least deterministic of the group. The control of application of predicate-calculus rules has not itself been rule-directed or directed by much knowledge of any kind. However, one of their strong points is that the proofs they generate play a natural role in keeping track of their actions or justifying them to a human user.

Production systems have very low-level rules. The system provides simple symbol-manipulation ability, but each programmer must provide his own control concepts, starting at the level of the subroutine call. This tends to defeat real extensibility, since two sets of rules probably have different calling conventions. (Production systems have been evolving toward greater richness.)

AI languages provide more direction and control over problem solving, but at the cost of making "rulishness" only a token aspect of procedures. The small patterned interface between a program and its callers is usually dwarfed by the body of the procedure. Other procedures know that this one is there, but they do not know what it is doing.

A general problem of all these systems is that they are insensitive to important aspects of their own operation. Production systems and pattern-directed procedures generally do not manipulate themselves. (That is, systems

built on them do not encourage or simplify this any more than the LISP interpreter in which they are generally embedded.)

To remedy these defects, I have implemented the NASL rule-based system to depend heavily on explicit representation of control. NASL's active processing site is a PLANNER-type data base of rules, but more is stored there than in the typical AI-language system. Besides a model of the current problem situation, the data base includes a representation of the "current plan." Rules are used, not to trigger actions directly, but to ***add tasks*** to this representation. When the rules are used in a forward-deductive way, they resemble productions, with an extra layer of "carefulness" between application of the rule and actual execution of the task. (Sussman, 1975) When the rules are used in a backward-deductive way, when the interpreter is attempting to find a way of carrying out a task, they augment the plan much the way a PLANNER-like language invokes procedures.

The difference between a plan and a procedure is that a plan may represent action at a more abstract level. In particular, the order of steps within and between subplans is itself rule-governed. Furthermore, not all tasks correspond to subroutine calls to bring something about or calculate something. Some tasks are intended to represent "parasitic" actions which influence the execution of other tasks, or which require occasional commitment of resources. Examples from circuit design are the actions, "Constrain RC to be $1/2\pi f$," "Make sure every requirement in the given design problem is accounted for," and "Take note of the high-gain requirement in making this amplifier."

These secondary tasks, or "***policies***," are particularly useful in choice situations, in which the problem solver tries to decide among more than one course of action. The existence of a policy often amounts to the existence of

certain rules for suggesting options or deciding among them.

Another important difference between AI-language procedures and NASL plans is that plans are completely deterministic. All search is done in the rule-appliers that try to retrieve, construct or choose among plans. If the resulting plan does not work, a "mistake correction" plan must be sought which is appropriate to the kind of mistake that occurred.

Inability to retrieve a solution plan via the simple deductive retrieval mechanism does not cause any kind of "failure." Instead, the system attempts to transform, or "rephrase," the problem until it is in a more familiar form. This requires that the rules and records of the system be manipulable by rephrasing tasks.

To explain its actions and correct its mistakes, the system must keep records of why it did what it did. These are of two kinds: stored chains of rule applications, and relations between tasks. The user may ask for an explanation of a task in terms of the tasks it was designed to accomplish. The system may edit these networks of relations when it runs into trouble.

I mentioned at the beginning of this section that rule-based systems are potentially "extensible," that is, able to accept new information additively, without major reorganization. We can distinguish short- from long-range extensibility. Over the long range, putting new information in is part of a simple but important kind of learning-- "taking advice." It is not the only part, because reorganizing descriptive structures and debugging or disambiguating what one is told are also crucial.

Therefore, it is easier to see the importance of extensibility over the short range. It is common in AI research these days to assume that knowledge is represented in large, well-organized chunks (usually called "frames"). (Minsky, 1974) Assuming this to be true, we still have the problem of

interactions of two simultaneously active chunks. This is just the extensibility problem in the small, since each chunk appears to the other to contain new information that may be relevant. Unless all frame interactions have been foreseen in advance (which is normal in most computer science, but not in AI), the information in each chunk must be expressed in terms the other can understand. (This is why programs appear to me to be such a poor frame representation, since a program is just a large chunk of lines of code, none of which means anything outside of its particular context.) At the very least, a system must notice potentially relevant interactions and ask for further information when it does.

In NASL, each rule resembles a Skolemized formula of predicate calculus. (Robinson, 1965) (In fact, few substantive restrictions on predicate calculus have been preserved in this notation.) The rules are manipulated by a PLANNER-like theorem prover. (Cf. R. Moore, 1975) However, the rules can come in large chunks called "packets." (McDermott, 1975) Packets can include other packets (since they are logically just large conjunctions of formulas). One use of this is the implementation of hierarchies like those of "semantic network" systems. (E.g., Bobrow and Winograd, 1976) Circuit diagrams, among other things, are stored as packets.

There are several "framish" concepts used in implementing NASL. For example, the active tasks of the current plan might correspond to the "important questions" terminals of a Minskian frame. (Minsky, 1974) However, I have felt free to diverge from the orthodox conventions of frame theory, and one must not assume that "plans" or "packets" correspond directly to frames.

I.C Supplying Rules for Design

To apply NASL to a problem domain, knowledge about that domain must be supplied in the form of rules. The major rule sets I have developed are for design in general and electronics in particular. When they are added to NASL, the system has the structure shown in Fig. I.9.

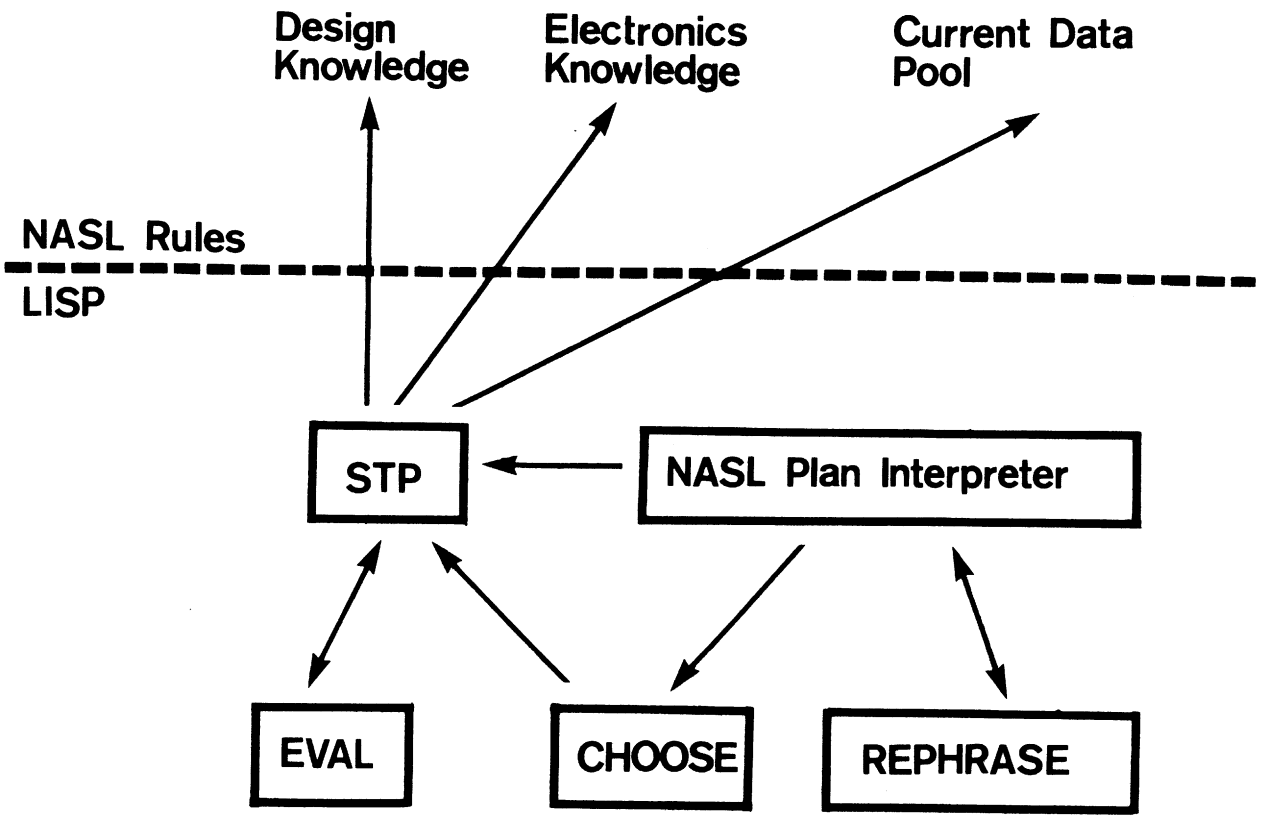

Figure I.9 Structure of DESI

The program, loosely known as DESI, has the following hierarchical organization (from the bottom up):

STP -- A PLANNER-like theorem prover
EVAL, CHOOSE, REPHRASE -- non-standard inference mechanisms
NASL -- The interpreter for plans
DESI (proper) -- A set of rules for designing hierarchical systems
ZORCH -- Rules embodying electronics knowledge

The rules themselves are highly structured. Some of them specify physical relations among things like nodes and signals. Higher-level rules are used to influence problem choice and transformation. Almost all of them have some

procedural component, in that they refer to the current task network. For example, even the simplest statement of Ohm's law, that the current through a resistor equals its voltage over its resistance, is stated as a constraint on the values of these physical quantities.

Most of the "raw information" in the system is stored in packets defining known circuits, such as common-emitter amplifiers, voltage dividers, etc. When a circuit of a given type is created, its packet will be instantiated. Each such packet contains information of these kinds:

(1) Component definitions like

```
[COMPONENTS ?##VOLTAGE-DIVIDER
            <(R1 ?##VOLTAGE-DIVIDER) (R2 ?##VOLTAGE-DIVIDER)>]
```

(NASL formulas are always enclosed in [brackets]. See Appendix 1 for details of this and subsequent NASL notation. The prefix "?" indicates a predicate-calculus variable; in a packet, "?##X" refers to an object that will be bound later to a particular instance of the concept the packet describes. (McDermott, 1975). Angle brackets enclose tuples.)

(2) Connection specifications like

```
[NODES ?##VOLTAGE-DIVIDER
       <(TOPNODE ?##VOLTAGE-DIVIDER)
        (MIDNODE ?##VOLTAGE-DIVIDER)
        (BOTNODE ?##VOLTAGE-DIVIDER)>]
[NODE-TERMINALS (MIDNODE ?##VOLTAGE-DIVIDER)
                <(#2 (R1 ?##VOLTAGE-DIVIDER))
                 (#1 (R2 ?##VOLTAGE-DIVIDER))>]
```

(3) Constraints and other "frozen tasks" which will be awakened when an instance of the packet is created. These are used to associate with a device a description of its purposes and further requirements. The commentary appearing around the diagrams in Figs. I.2, I.7, and I.8 is represented as a set of such tasks.

These circuits come in hierarchies of various kinds. The components of a circuit may themselves be circuits. Circuits may be arranged in classes (such as "amplifier") which share properties. One circuit may be derived from another by assuming special conditions; for example, the specific and general common-emitter circuits I pointed out in Sect. I.A are of this type. All of

these hierarchies may be represented by the device of allowing packets to include other packets.

A solution to a design problem is represented as a structure of instantiated circuits, with primitive-component values selected. A top-level design problem is of the form, "Find a circuit structure with property...."

It is the information required to go from property to circuit that is of most interest. This falls into several classes: (1) knowledge about transforming problems, (2) rules for making choices, (3) plans for altering and improving circuits, and (4) knowledge about physical constraints on quantities. Each of these categories is represented by rules in DESI and ZORCH. DESI is a small set of design rules that are intended to be independent of any one physical domain; it provides a vocabulary and task structure within which ZORCH's rules can operate.

For example, DESI provides a standard framework for rephrasing design problems. The idea is to transform an unfamiliar design problem into the making of a familiar kind of circuit obeying physical constraints, using more suggestive hints like "make it linear" or "notice the high gain." (Cf. Sect. I.A.) ZORCH provides rules to do this decomposition and then to use the hints and constraints to zero in on a diagram.

The DESI rephrase plan contains tasks to

"Explode" the given property into "shards."

Classify shards as to whether they suggest a familiar device type, a constraint, or a "design features" like "linearity."

Gather the suggestions into a new task network.

In this process, rules like this (from ZORCH) become important:

```
[-/> A (D-SHARD ?+P [λ (_?+V) (= (V-GAIN (/?/? _?+V)) _?+G)])
       (-/> G (/> (DEN ?+G) 50)
              (D-FEATURE ?+P [RANGER V-GAIN HIGH]))]
```

This says that a voltage gain greater than 50 should be noticed as "high." The symbol "-/>" signifies implication; the letter after it identifies the way in which the rule is to be used. (See Chapter II. The "A" means when the left side is recorded, record the right; the "G" means call the theorem prover to find answers to the left side, and record the right side for each substitution returned. The actual rules in Appendices 2 and 3 are more indirect and give more information.)

Once the task network has been transformed, other ZORCH rules come into play. These rules are of these kinds: (1) definitions of fundamental wiring operations; (2) physical laws like Ohm's which constrain numerical quantities; (3) plans and pieces of plan for biasing, coupling, and performing other standard operations on circuits; (4) rules for choosing among sub-types of inclusive circuit categories suggested by the rephrase rules.

Fundamental wiring operations are defined using the built-in relation /:MOD-MANIP (for "model manipulation"). For example, connecting terminals to form nodes may be defined by this rule

```
[/:MOD-MANIP ?TASK (CONNECT ?T1 ?T2) <>
    <'(EXISTS (N) (AND (DEV-TYPE ?N NODE)
                       (NODE-TERMINALS ?N <?T1 ?T2>))   )>].
```

This defines an "addlist" (Fikes and Nilsson, 1971) for this action. (Its "deletelist" is empty.)

Physical laws are defined by rules like this:

```
[-/> A (DEV-TYPE ?X RESISTOR)
       (EXISTS (T)
          (/:TASK ?T <>
             (λ () (CONSTRAIN <'(V ?X) '(I ?X) '(R ?X)>
                              (λ (V I R) (= ?V (* ?I ?R))   )))
              <>)   )]
```

which commits the designer to the given constraint. CONSTRAIN is a kind of policy action defined in DESI; again, DESI provides the vocabulary for ZORCH.

Choice rules are used for differential diagnosis or synthesis of partial solutions. For example, in choosing an amplifier, the rule

```
[-/> A (/:OPTION ?C ?A1 [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+AMP>
                            (MAKE COMMON-COLLECTOR)])
       (-/> A (/:OPTION ?C ?A2
                  [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+AMP>
                     (MAKE COMMON-EMITTER)])
              (/:RULE-TOGETHER <?A1 ?A2>
                 [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+AMP>
                    (MAKE (CASCADE COMMON-COLLECTOR
                                   COMMON-EMITTER))])))]
```

This rule says that a common-collector amplifier and common-emitter amplifier may be cascaded to accomplish the purposes of both. (The actual rule comes closer to saying this.) The conclusion [/:RULE-TOGETHER...] is used by the choice protocol of Fig. I.9. It is used to specify composition of separately unsatisfactory choices; /:RULE-IN and /:RULE-OUT are used to narrow the list of options.

Many aspects of the operation of the system cannot be brought out by this kind of summary. In the next three chapters, I will describe its knowledge more systematically. The major omission so far has been a good description of the task network and its evolution. Without this description, much of the content of the rules is lost.

To compensate, let me sketch DESI's behavior on the second problem I used as an example in Sect. I.A, indicating points of interest as I go along. The problem is presented as the problem of designing a circuit to convert signals

satisfying an input predicate to those satisfying an output predicate. (See Chapter IV.) This is written

```
[DESIGN
   (λ (CKT)
     (CONVERT ?CKT
        (λ (IN)
          (AND (PERIODIC (TFUN ?IN) 10.0E-3)
               (FORALL (T)
                  (AND (-/> C (/< ?T 0)
                              (= ((ONE-PERIOD (TFUN ?IN)) ?T)
                                 1))
                       (-/> C (NOT (/< ?T 0))
                              (= ((ONE-PERIOD (TFUN ?IN)) ?T)
                                 -1)))   ))   )
        (λ (IN OUT)
          (= (TFUN ?OUT) (λ (T) (SIN (* 2000 PI ?T))   ))   ))   )]
```

The input predicate is just a time-domain definition of "1kHz square wave." The output predicate defines the time function (TFUN) of the output to be a sine wave. (C after "-/>" means: to prove the right-hand side, detach the left as a subgoal.)

The design problem is used to start a ***task network*** or ***plan***. The goal of the problem solver is to accomplish every extant task. In the course of doing this, ***subtasks*** of various kinds may be generated, which must be gotten rid of.

In the case at hand, the complicated design problem does not match any stored subplan directly. The resulting failure of the theorem prover causes the NASL interpreter to set itself the task of "***rephrasing***" the design task.

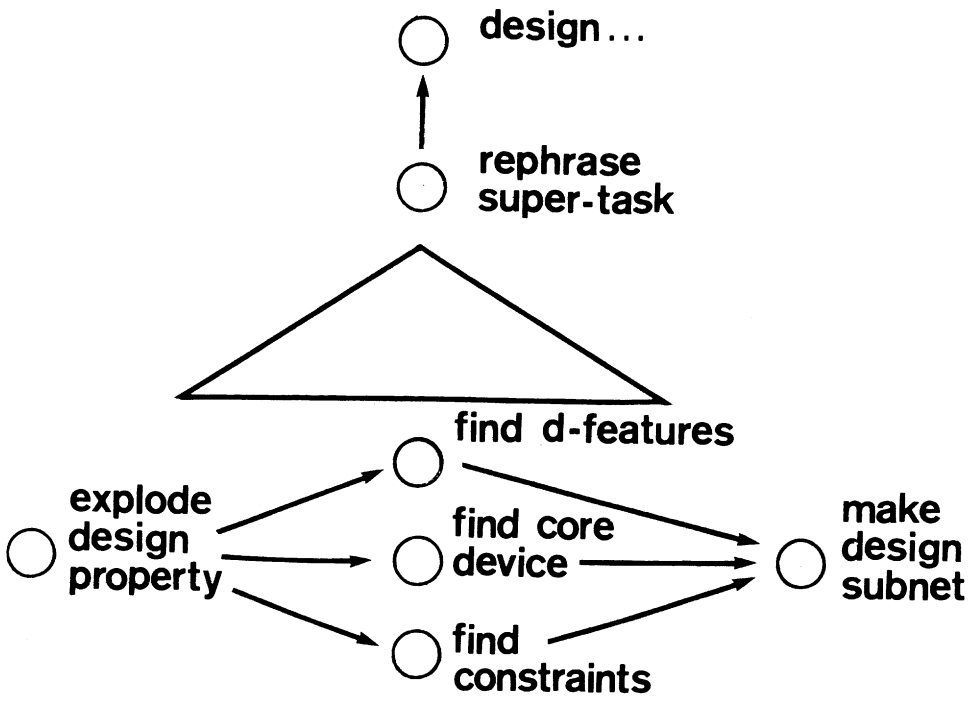

Figure I.10 A Rephrasing Subtask

This rephrasing process notices the conversion problem in the description of the desired circuit, and spends some time trying to calculate and compare the frequency spectra of the input and output signals. This process results in the re-description of the problem as a low-pass filtering problem. (The complex details of this example are described in Chapters IV and V.)

This operation of rephrasing the original problem is carried out by the NASL interpreter, operating at a different logical level. In particular, its behavior is rule-governed in the same way. The only difference is that problems at the lower level become objects of manipulation at the higher level. The result of this manipulation ultimately appears as a new problem network at the original level.

The signal descriptions I showed earlier are subject to rules from ZORCH which suggest looking at them in the frequency domain (Figs. I.5 and I.6), and looking for a simple transformation between them. The transformation

discovered, namely [LOW-PASS 1000], generates the design shards

```
[λ (CKT) (DEV-TYPE ?CKT LOW-PASS-FILTER)]
[λ (CKT) (= (LOW-CUTOFF-FREQ ?CKT) 1000)]
```

which in turn suggest a basic device type (low-pass filter), constrained to reduce its output at all frequencies above 1 kHz to negligible values.

The task net has now been "elaborated" to the structure of Fig. I.11.

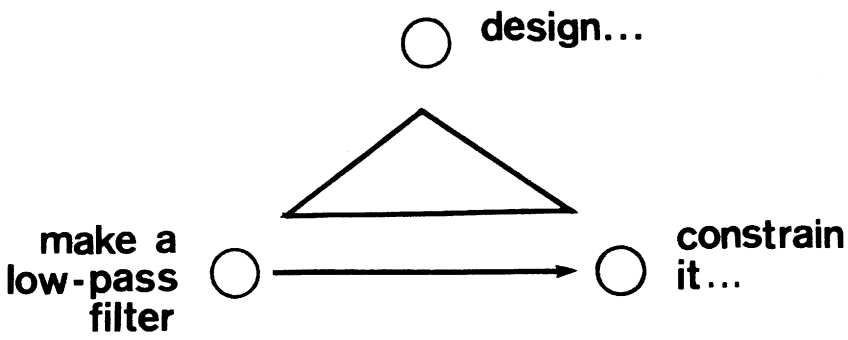

Figure I.11 Rephrased Task Network

The problem has become "make a low-pass filter, and constrain it to fit the exact desired characteristics." The first subproblem in this structure is much simpler than the original, and results in the retrieval of a useful plan, in the form of a "*device schema*" for an RC filter. (I will defer the possibility of more than one schema being brought to mind until later.)

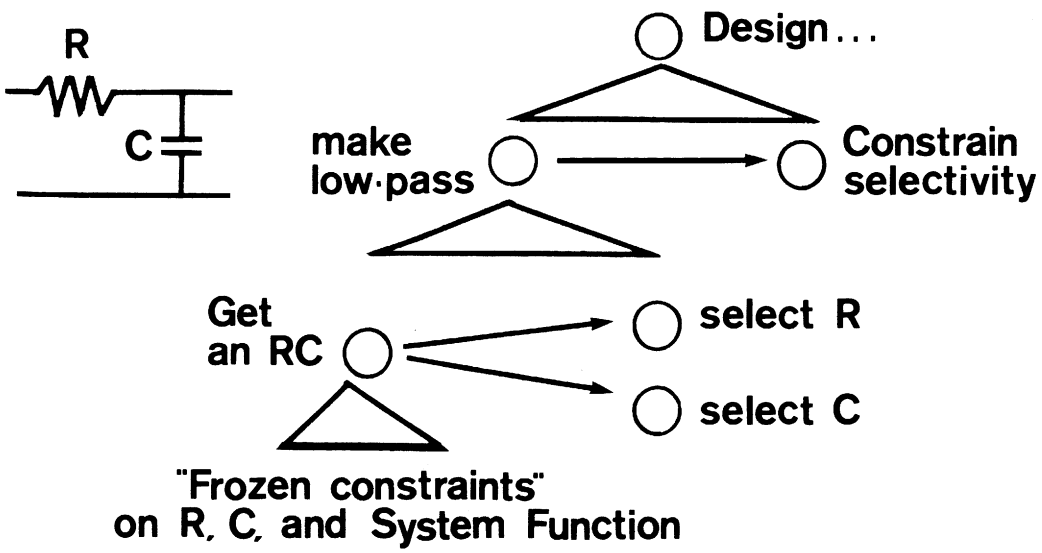

Figure I.12 Retrieved Circuit and Constraint Task Network

The problem now (see Fig. I.12) is to satisfy the constraints given. Some of these came with the problem statement, but many more tag along with the schema for RC filter (which includes facts about filters in general). A useful feature of the NASL language is that we can express the purposes of devices in the same language the system uses to express its own: as tasks. To use an RC filter is to insist that its resistor and capacitor have values compatible with its desired system function. Such tasks are called "***frozen policies.***"

Such already established tasks are not the only useful kind that ride along in device schemata. There are also "expansion obligations" which remain to be done. An example of this technique is the definition of "active transistor," as a "raw" transistor plus the commitment to bias it in whatever context it appears. In the case of the filter, the only expansion obligations are to select values for the primitive components. These tasks (see Fig. I.12) are carried out by interactions with the new and frozen constraints.

(In the current implementation, most algebraic symbol manipulation is carried out by the human user.) Values are to be found which satisfy all the constraints. When they are found, the fact that they satisfy the constraints is to be "*protected.*" (Sussman, 1975) This can be a complex task in itself. (Chapter III.)

As we saw before, there are no values which satisfy all the constraints. Even for engineering purposes, an RC filter cannot quite do the job. In this case, a failure occurs, which causes the insertion into the network of a correction task. This task may have to edit the task network as well as the current circuit model in order to solve the design problem adequately. (The current implementation stops before this point.)

I have glossed over the problem of choice in this example. It is more obviously relevant in the case of the first example of Sect. I.A, designing a buffered amplifier. In this case, rephrasing is relatively simple, being a matter of unpacking a lambda expression such as

```
[λ (X) (AND (DEV-TYPE ?X AMPLIFIER)
            (= (V-GAIN ?X) 5)
            (= (INPUT-R ?X) 30000))].
```

However, these fragments suggest more than one kind of amplifier, as we saw. (Fig. I.2) In other words, the system has converted a problem with no apparent solutions into a problem with two apparent solutions. This embarrassment of riches is handled by invoking the choice mechanism, a simple "protocol" for calling the theorem prover. In this mode, a series of staccato deductions are made which attempt to rule out alternatives, vote in favor of alternatives, or synthesize new ones. (See Chapter 2.) The relevant rule is the one that says, "if you are trying to choose among different ways of making an amplifier, and option 1 was suggested because of its input resistance, and option 2 for some other reason, replace these options with [CASCADE |option 1|

[option 2]]." (A simplified version of this rule appears above.)

Other choices that occur in these examples are handled similarly. There is a rule that the general common-emitter circuit is the starting point for implementing a common-emitter coupled to something else. In the second example problem, if the system is ever told about LC filters, we will also have to give it rules like

"Use an LC filter if the power involved is high."

"Other things equal, don't use an inductor circuit if you can help it."

For DESI's actual behavior on these problems, see Chapter V.

I.D Relation to Previous Work

I.D.1 Problem Solving and Reasoning

The problems I am attacking in this research are not new. The problems of generality vs. expertise were originally studied by Allen Newell and his coworkers around 1960. (Newell, 1962) Their efforts produced a "General Problem Solver" which we have been trying to debug ever since. (Ernst and Newell, 1969) GPS was a "means-ends analysis" problem solver which applied state transformation operators to bring it to its goal. McCarthy (1959) gave us the term "advice taker" to describe a program which can take new information and use it to do better. The creation of such a program is still my long-term goal and, in a sense, that of most other researchers.

While this research was in progress, a tide of "rule-based" AI programs has risen which NASL seems to be a part of. Its ancestors are the systems I described in Sect. I.B. More recent, special-purpose rule-based systems have sought to overcome their limitations. Shortliffe's (1976) MYCIN is a limited but elegant medical-diagnosis system which uses a backward-chaining deductive

system. Sussman and Stallman's (1975) EL does electronic circuit analysis by forward deduction. Both of these systems keep a record of deductions. EL uses these records to rethink deductions based on unworkable assumptions. (Stallman and Sussman, 1976) Both systems use them to explain their deductions to a human user.

The NASL system differs from these in that its control language is aimed at a higher level of abstraction. Its rules, expressed in a predicate calculus, specify conclusions rather than actions. Action is achieved by having certain conclusions be interpreted as "required tasks" by the action interpreter. The notion of "task" is intended to be very inclusive.

MYCIN and NASL can both be given new rules, which, if they are not buggy, interact with the rest of the system in efficient ways. However, MYCIN's rule language is deliberately restricted to the domain of fault diagnosis in poorly understood systems. (Davis *et. al.*, 1975) (MYCIN is superior to NASL in having a more developed procedure for graceful assimilation of new rules. (Davis, 1976)) EL's rules have stylized LISP code bodies. They can do anything, in principle, but the system functions most elegantly when organized around the satisfaction of numerical constraints. MYCIN does almost entirely backward chaining during deduction; EL, forward chaining.

The most important conceptual problem I have found in working on NASL is the requirement that the control structures of a problem solver ought to be simple enough to be inspectable, but contain enough higher-level concepts to be useful when inspected. The MYCIN group express this as a demand for "stylized programming" (Davis *et. al.*, 1975). They have achieved impressive results in two areas. First, by use of "meta-rules," their diagnostic program can guide its own flow of control. This is something like my "choice protocol" in which NASL uses choice rules to decide how to proceed. Second,

their knowledge-acquisition program knows enough about the "stylization" to participate in writing and debugging new rules. I shall make a more detailed comparison of these two capabilities with actual and potential abilities of my program in Chapter VI.

The main limitation of MYCIN's style of rule-based programming is that it is wholly oriented toward making tests and letting them "cast votes" for a result. There is no concept of planning or even acting. Davis *et. al.* (1975) acknowledge that for a domain with a more precise theory than medical diagnosis, a different control structure is called for. I hope DESI is an example.

Stallman and Sussman's (1976) EL is implemented using a language called ARSE which is embedded in LISP. The primitives in ARSE implement a system of forward deduction and "guessing," aimed toward finding a consistent assignment of variable values in a constraint network. ARSE has been used experimentally on other tasks involving constraints (Mason, 1976, Doyle, 1976), and so has modest pretensions to generality. ARSE's control structure is formally close to NASL's (and helped inspire it). ARSE maintains "demon queues" generated by ongoing deductions. However, EL lacks the need or power to inspect these queues efficiently. NASL embeds the control structures in an associative data base, and generalizes the notion of queue to a task network.

In this respect, the closest control structure to NASL is Sacerdoti's (1975) NOAH. This is a brilliant program for planning a mechanical assembly and advising a person carrying it out. The planning part constructs a hierarchical network of ever more detailed plans. These plans are not programs; in particular, they do not have to be totally ordered. As parts of the plan are expanded, their interactions with each other are noted and corrected for by enforcing orderings.

The main difference between this part of NOAH and NASL is that NOAH is a plan compiler, while NASL is an interpreter; that is, it expands and executes pieces of plan as it goes. This is necessary because NOAH has a simple STRIPS-like (Fikes and Nilsson, 1971) assumption about actions which NASL doesn't share. In particular, NASL is not as sanguine as NOAH about expanding a future action, because it has a limited model of the world at that point. It does not attempt to summarize the effects of all tasks as state changes, so it cannot have a domain-independent algorithm for checking interactions between steps. In particular, actions like "Design..." and "Constrain...," whose effects depend on how they are carried out and/or what else is being executed, do not fit into Sacerdoti's framework. This makes room for more sophisticated knowledge about action, but it is a pity that I cannot use Sacerdoti's simple and (within their limits) powerful algorithms.

I have also profited from reading papers by Nilsson (1973) and Philip Hayes (1975) on interleaving planning and execution. Several researchers (Schank and Abelson, 1975, Abelson, 1975, Rieger, 1976, Charniak, 1975) have done research on classification of plans analogous to my taxonomies of Chapter II. Usually, however, they have been more concerned with analyzing narratives than with actually solving problems, which has led to different criteria for classification. Perhaps some synthesis of these approaches will be possible.

A class of systems with which NASL shares certain properties are the "utility" AI systems which have appeared recently. These are systems which provide data and control representations for users, who are expected to use these facilities for problem-specific programs. Examples are the AI languages (Bobrow and Raphael, 1973), Bobrow and Winograd's (1976) "Knowledge Representation Language," and Srinivasan's (1976a,b) "Meta Description System." The AI languages provide assertion-based data bases like NASL's.

(NASL and STP are descended in this direction from the AI language Conniver. (Sussman and McDermott, 1972)) The other two systems are more semantic-network oriented. (Woods, 1975) This is in many ways merely a formal difference. Other differences between these research efforts depend on healthy differences of focus. For example, the KRL group is more concerned with recognition problems than I have been.

A bigger philosophical difference is that NASL is an attempt to provide a plan description language rather than a programming language. The distinction may be wholly metaphysical; however, I believe that several features of NASL plans, especially the notion of "policy," if implemented properly, belong to plan description rather than programming. A concrete distinction between NASL and the traditional "AI utility" (Hewitt, 1972) is that NASL, far from requiring a program to specify a piece of knowledge, requires a body of knowledge to specify a program. I believe that Srinivasan's MDS results from a similar orientation, but he is more concerned with general puzzle solving than capturing the knowledge in a rich domain.

In any case, the older, less pretentious AI languages are the only members of this list of systems (NASL included) which are mature enough for their flaws and strengths to be visible. Which features of the newer systems will endure remains to be seen.

Unlike most of the problem solvers mentioned, NASL uses a theorem prover to do sophisticated deductive information retrieval. This use of theorem provers has been suggested by many people. (Travis *et. al.*, 1972, Darlington, 1969, Moore, 1975) My theorem prover, STP, resembles most closely that of Nevins (1974a), with features from the work of Bledsoe (1975) and Ernst (1971, 1973). Those familiar with the theorem-proving literature will enjoy Appendix 4, which describes its features.

Other people have studied somewhat different uses of theorem provers in problem solving. (E.g., Fikes and Nilsson, 1971) In the past couple of years, one group of people has been urging the use of a predicate-calculus theorem prover as the *only* interpreter of a problem solver. (Kowalski, 1973, 1974, Warren, 1974, Hayes, 1973b, Tarnlund, 1975) I think this view is misguided. Generally one does not go very far with this approach before he starts adding corruptive features, such as ordering the axioms, putting in dummy predicates to control search, allowing rules to refer to formulas, etc. (Warren, 1974) My conclusion was that it is better to admit defeat from the start, so I put control features in as concepts manipulable by the calculus and defined by the interpreter, and tried to preserve some of the purity of the theorem prover itself. I shall have more to say about the success of this attempt in the conclusion.

I should mention that the concepts of action and decision have concerned philosophers for hundreds of years. Recently, a whole branch of analytic philosophy has sprung up around them. (Langford, 1971, Brand, 1970, Danto, 1965, Goldman, 1970) Many of the workers in this field have interesting things to say about the logic of action. For example, the computer scientist's notion of a "primitive" is reflected (somewhat dimly) in statements like, "... those actions, ... performed by M, which he cannot be said to have caused to happen ... I shall designate as *basic actions*." (Danto, 1965) Unfortunately, these philosophers are much too reluctant to imagine that the mind behaves like a device with a real structure; all of their definitions are in terms of phenomenological rather than technological categories. For example, Goldman (1970) gives the following exegesis of the notion of "basic action": A basic act is an act *A* such that "if *S* wanted to exemplify *A* (at *t*), he would exemplify *A* (at *t*)." He then must spend no

little effort explaining away paralytics. I think that in the long run philosophers exposed to AI ideas are most likely to arrive at useful concepts in this field by explaining "want" and "act" in terms of hypothesized internal constructs.

I.D.2 Electronics and Design

The usual problem domain for a researcher with my pretensions is some class of puzzles (Ernst and Newell, 1969) or "narrative understanding." (McDermott, 1974a, Charniak, 1972). I have chosen elementary electronic circuit design instead, for these reasons:

(1) It is not as broad and sloppy a domain as "story understanding." One can reach "critical mass" with a data base much faster. There are clearer criteria for success. Electronics involves, I hope, as few mental competences as possible in an interesting domain.

(2) On the other hand, there is room for a variety of kinds of knowledge. The domain cannot be, and doesn't have to be, represented fully by a state space and a set of operators. Puzzles are both too easy and too hard at once; they are probably a misleading example of problems that succumb to human thinking.

(3) The subject matter is already formalized to some degree, so that I can focus on formalizing the *control* knowledge that is necessary.

(4) Electronics is simpler than other engineering domains in that it requires less knowledge of space, time, and motion. Expertise in these areas presumably draws on innate abilities we have difficulty bringing to light.

(5) My research has had the benefit of being part of an ongoing MIT AI laboratory project in automating electronics reasoning. Concepts used by Sussman and Stallman (1975), Stallman and Sussman (1976), and A. Brown (1975), have been taken over, sometimes with some modification, into DESI's knowledge base. (This is especially true of the parts concerned with signal description and electronic analysis.)

(6) My wretchedness as an electrical engineer should make it easy to construct a program as good as its creator.

The main drawbacks to electronics are

(1) It is somewhat inaccessible to the average AI researcher or

psychologist. People lose interest in documents regarding something they know little about. (Who knows what DENDRAL (Buchanan *et. al.*, 1969) really does?) I have tried to keep large sections of this text independent of electronics. Only Chapter IV and Appendix 3 rely on it.

(2) Electronics knowledge as presented in introductory texts leans on spatial representations to some degree, even if not as much as other branches of engineering. Frequency-domain manipulations and pole-zero plots are examples of this. I have tried to preserve the structure of this knowledge in formal expressions (see Chapter IV), but I am aware that humans probably use more "wired-in" modes of spatial reasoning, whatever that may turn out to mean. I doubt that one could choose a better domain than electronics for avoiding this problem, however.

My knowledge of electronics is mainly derived from books. (Senturia and Wedlock, 1975, Hayt and Neudeck, 1976, Watson, 1970) This is reflected in the fact that the problems DESI has been exposed to are "problem set" problems, not the sort that a technician would encounter in daily practice.

There is a large literature on the theory of design, artificial intelligence and design, and "computer-aided design." So far, however, the intersection of these fields is almost empty. Books about the design process (Alexander, 1964, Asimow, 1962, Buhl, 1962, Glegg, 1973) consist mainly of advice for avoiding overlooking things in pondering problems and working out solutions. About proposing solutions to start with, most of these authors say things like this:

> "What enables us to draw from the warehouse of our experience just the right set of elements, and to put them into just the right combination so that they have a sense of fitting the situation, we do not know, since no definite solution exists." (Asimow, 1962, p. 45.)

This author is certainly correct that we do not know; programs like DESI are only tiny steps toward a theory of creativity. Of course, as a working hypothesis, we take issue with the claim that no solution exists.

Design has attracted artificial-intelligence researchers, particularly at Carnegie-Mellon University, for some time. Broadly speaking, areas like automatic programming, and, indeed, all problem solving, fall under the

heading of "design." However, the theory of design narrowly construed has been explored by workers like Grason (1970), who studied resolution of constraints on floor plans; Eastman (1968), who did a formal pychological study of performance on the task of redesigning a bathroom; Haney (1968), who studied the automatic design of computer instruction sets; and Latombe (1976), whose rule-based design system is an interesting alternative to mine. I have found especially useful the theory paper by Freeman and Newell (1969) on a general theory of design, from which I have borrowed heavily. (See Chapter III.)

One might expect the field of "computer-aided design" (CAD) (Vlietstra and Wielinga, 1973, Kuo and Magnuson, 1969, Furman, 1970, Rosenbrock, 1974) to have produced many expert programs for a general AI program to compete with. This is not yet the case; "CAD" usually has little to do with the automation of the actual design process, but concerns itself with graphics packages, analysis programs, and other interactive aids to it. For example, one author distinguishes "three modes of [computer] operation:

> (i) ***Analysis.*** An engineering situation is specified in full mathematical detail by the designer, and the computer draws certain further mathematical consequences....
> (ii) ***Synthesis.*** The designer specifies in detail the properties which his system must have, to the point where there is only one possible solution. The computer finds this solution. An example is optimal control.
> (iii) ***Design.*** This is the creative act of a designer, guided by calculations on the computer and interacting with them in a sequential manner to produce a satisfactory solution." (Rosenbrock, 1974, p. 29)

The electronics synthesis tasks to which computers have been put include very low-level operations such as printed-circuit layout (e.g., Fletcher, 1974) and filter design (e.g., Chohan and Fidler, 1974). The approaches taken by most people in this field are usually very "mathematical," and concentrate on techniques for discrete or continuous optimization. For example, one approach

to circuit design in the literature (Director, 1974) consists of putting in as many components as one deems plausible and letting the program find the optimal component values for the task given. Many of these components will assume null values and "vanish"; thus this approach starts with a big circuit and finds the subset that does the job!

CAD is only beginning to become aware of non-numerical techniques. (But see Powers, 1973.) DESI relies almost entirely on non-numerical techniques, and is very poor at constraint resolution and component-value optimization. A practical system would have to combine the two approaches.

It is impossible to survey this field in detail here. It includes its own journal, ***Computer Aided Design***, and supports periodic conferences.

II Expressing Knowledge in NASL

The heart of DESI is the NASL interpreter, and the STP theorem prover which it drives. The theorem prover gives the system a general and flexible notation; the interpreter imposes an innate interpretation on some of the expressions of this notation. In this Chapter, I will give a discursive introduction and overview of the interpreter, describing STP and other inferential protocols as they come up. (See Fig. I.9.)

The NASL interpreter is a problem solver of the "problem reduction" type. (Nilsson, 1971) That is, it solves problems by reducing them to simpler subproblems. The differences between this structure and a programming language are: that a problem solver must decide upon the order in which to attack subproblems; that a problem solver often has subproblems of the form "reduce problem so-and-so," where a programming language has only the subroutine call; and that a problem solver must occasionally choose between alternative approaches.

The designer of a problem solver must confront the problem of search. For problem-reduction problem solvers, this classically takes the form of a search through an AND/OR graph of possible approaches. (Nilsson, 1971, Fahlman, 1973, McDermott, 1974a) Whether the search strategy is depth-first or more clever, it depends upon being able to save and restore states of the problem solution and hence of the "world model"; this has recently tended to be implemented using a "context" mechanism. (Sussman and McDermott, 1972)

I believe that searching will always be a part of Artificial Intelligence technique. However, it seems to me that search among alternative sequences of subproblems and world models is a mistake. My principal reason for this belief is the observation that in the normal course of human problem solving,

a rather different faculty is used more heavily, namely, the ability to correct one's errors. The difference between these alternatives is this: if a "state of the world" is thought of as an internal data structure, completely known and under control, it is just as easy or easier to return to an earlier state to try something else as it is to generate a new one. But if states of the world are really states of the whole world, about which one's information is limited and his control slight, quite the opposite is the case.

So the question, for electronic circuit design, is whether the unfolding circuit model is to be thought of as an internal data structure or as a diagram on a piece of paper. A little reflection on this choice has drawn me to the second alternative. Since useful plans will ultimately have to be applied to the real world, whose surprises will always cause mistakes and revision, the problem of correcting errors rather than "popping them off the context tree" will have to be faced eventually. There is no point in perfecting a plan down to the last detail if circumstances will wreck it. This is probably why people worry so little about producing optimal plans.

If search isn't through states of the world model, but it is necessary, what is it that is searched? I think it is ***knowledge about courses of action.*** People can correct states of the world created by the wrong plan, but they don't normally do this as a way of stumbling on the right plan. Instead, they use knowledge like

> "Under circumstances ..., plan ... will work."
> "If ..., don't do ...."

Consider the difference between human and machine playing of chess. I will assume the reader is familiar with the usual program organization around the idea of minimax tree search. (Slagle, 1971) A human, by contrast, ***learns*** to play. His initial plan is simply to make a legal move, wait for his

opponent's reply, and repeat this until the opponent wins. As time goes by, and he sees and hears more and more about the game, where does he put what he learns? According to the theory I am presenting, it becomes part of the advice surrounding the "make a move" step. This advice is usually in terms of board patterns, phases of the game, etc. Eventually, more sophisticated advice in terms of anticipating possible opponent replies is assimilated. If the deductive system for manipulating this advice is adequate, simple tree searches will appear as a trace of its manipulations. But this will never be assimilated to the overall planning level. The planning level does not become nondeterministic. Instead, what begin to appear there as the player becomes more confident of his powers are "game plans," long-term strategies which influence his choice of moves.

This sort of search through knowledge about alternative courses of action is worth spending a lot of effort on. It has three loci in the NASL system. The principal one is the theorem prover STP. (Sect. II.B.2) This is supplemented by "*choice information.*" (Sect. II.C.1) If all else fails, the system calls itself recursively to "*rephrase*" an action. (Sect. II.C.2) I have worked hard to make these devices sophisticated. I have given less thought to the problem of undoing mistakes (Sect II.E), and none to the question of learning search knowledge.

Because deductions about courses of action are so central to the theory, NASL must be a language for describing problems, plans, and physics. The categories it uses for descriptions, and the inference algorithm it can call upon to manipulate them, determine its abilities and limitations. The limitations are in some ways as important as the abilities. The fewer ways there are to express something, the more likely it is that the formulas related to it will be noticed during rule application, and the more flexible

and extensible the system will be. Conversely, to the degree that each user is forced to make up his own control conventions, the less likely it will be that information from one user will ever affect the system's approach to problems posed by another.

NASL is not a typical programming language, since the user can intermix fragments of plans and axioms governing the physics of problem domains with fully developed programs. On the other hand, it bears a stronger family resemblance to programming languages than to anything else, so I have included a "programmer's guide" at the end of this chapter for those interested in programming in NASL.

II.A The Natural History of Actions

The fundamental concept implemented by the NASL interpreter is the concept of *task.* A task is an activity to which the interpreter is committed. The basic drive of the interpreter is to accomplish all the pending tasks. Examples of tasks from several domains are,

"Put Block A on Block B."
"Wait here for five minutes."
"Put the male chicks in this box, the females in that one."
"Win the war, and keep the peasants happy."
"Think of a fallible Irishman."
"Keep your promises."
"Convince yourself that all equilateral triangles are isosceles."

In electronic design, tasks range from wiring two objects together, to designing a hi-fi system, to finding a resistance that satisfies a constraint.

The reason for the broad definition is my goal that as much as possible of what the interpreter is doing should be explicit, so that reasoning about it can be shallow. For the same reason, it will be important that control information be expressed in a notation compatible with everything else. So I

represent tasks as NASL formulas of the form

```
. [/:TASK |name| < -input pvars- >
                 (λ ( -vars- ) |action|   )
                 < -output pvars- >]
```

Unfortunately, I must pause here to describe the notation, both object and meta. NASL formulas are always enclosed by [brackets]. When I am describing a formula, I enclose syntactic variables in brackets like this: "|...|" or like this: "-...-". The second kind indicates that a ***sequence*** of syntactic constructs is wanted. So, for example, an instance of a task might be of the form

```
[/:TASK (COUPLER PLAN#71) <'(BUFFER#72) '(AMP#73)>
        (λ (STAGE1 STAGE2) (COUPLE ?STAGE2 ?STAGE1)   )
        <'(CKT#74)>]
```

This describes a task, named [COUPLER PLAN#71], which requires taking the two circuits [(AMP#73)] and [(BUFFER#72)] and COUPLing them to make something which will be called [(CKT#74)]. (Notice that the NASL notation permits tuples of objects delimited by <angle brackets>, and λ-expressions to express functions and predicates.) /:TASK is a predicate of four arguments. It begins with the prefix "/:" which indicates that it is a built-in predicate used by the interpreter in some way. A complete catalogue of built-in predicates and functions appears in Appendix 1.

The word "pvars," for "plan variables," refers to terms, such as [(BUFFER#72)] and [(CKT#74)], which are set equal to calculated quantities in the course of executing tasks. They acquire values by appearing in "rewrite rules" of the form

```
[=/> '|term| |value|]
```

(Cf. (Bledsoe and Tyson, 1975), where they are called "reduction rules.") In my example, if [=/> '(BUFFER#72) DEV#75] and [=/> '(AMP#73) DEV#76] appear in the data base before execution of the task [COUPLER PLAN#71], DEV#76 and

DEV#75 might be coupled to produce DEV#77; then the interpreter would add [=/> '(CKT#74) DEV#77] to the data base. (For an explanation of the single quotes, see Appendix 1 or Sect. II.B.2.) In defining actions like COUPLE, I will indicate their outputs with the symbol "==>" thus:

[COUPLE |ckt 1| |ckt 2|] ==> <|new ckt|>

to show what new value formulas they leave in the data base.

Anyone familiar with the AI languages (Bobrow and Raphael, 1974, Hewitt, 1972) will recognize the concept of "present in the data base." In NASL, there is always a current "*data pool*" for formulas to reside in. Formulas found there are supposed to be true; those absent have unknown truth values. (See Sect. II.B. The phrase "data pool" is meant to supersede the misleading word "context." (McDermott and Sussman, 1973, Rulifson *et. al.*, 1972))

This notion of task already embodies a complexity not found in the action languages of Sacerdoti (1975) and others (Sussman, 1975), namely, that tasks will not be fully specified until their input pvars are known, and that tasks can compute values to be used by subsequent actions. It will be seen that this broadens the scope of the action system considerably, while making future actions harder to analyze. It seems essential for automating design.

With just this much machinery, plus a simple forward deduction scheme, we have a notation similar to that of a production system (Newell, 1973a, Rychener, 1976). For example, we might have a deductive rule that says

```
[(DEV-TYPE ?A COMMON-EMITTER)
 ⊃ ∃B(/:TASK ?B <> (λ () (BIAS ?A)) <>)],
```

meaning, "Every common-emitter amplifier must be biased." (I have introduced standard logical notation for implication and quantifiers. (Suppes, 1957) Variables are prefixed by "?"; free variables are supposed to be universally quantified. The internal notation for implication will be explained below.)

This rule is analogous to a production in having a condition on the left and an action on the right, but it differs in certain crucial ways.

First, we are *not* limited to condition-action pairs. The more basic case is "condition-condition." This enables us to treat deductive information retrieval as a process distinguishable from plan execution. It can be optimized separately, using techniques specific to the kind of search that arises during deduction. (Moore, 1975, Fahlman, 1975b) More important, since the system knows when it is doing deduction, and when action, it can keep more revealing records for use in choosing courses of action, explaining what it did, and recovery from errors. (In the future, such records could be used for careful assimilation of new, possibly unreliable, information. Cf. (Davis, 1976, McDermott, 1974a). By contrast, since the meaning of a condition-action pair depends entirely on the meanings (some deductive, some not) given to symbols by the behavior of the rest of the system as a whole, it is impossible to say whether a new rule of this kind is "correct" without extra commentary.)

Second, deducing /:TASKS before executing them gives an extra layer of "carefulness" to the system. (Cf. (Sussman, 1975), where the term "careful mode" is introduced.) A task is always noted in the current data pool before being executed. Here it can trigger other tasks or be available for other deductions. Furthermore, the system can note a task some time in advance of when it actually decides to do it. For example, a task can appear before its pvars' values are known. More generally, a formula of the form

[/:SUCCESSOR |task name 1| |task name 2|]

must mean that task 2 is to be postponed until task 1 has been "begun" (in a sense explained below). In this way, a network of tasks linked by /:SUCCESSOR relations and variable flows is created (which the interpreter will munch "from left to right"; cf. Fig. II.1).

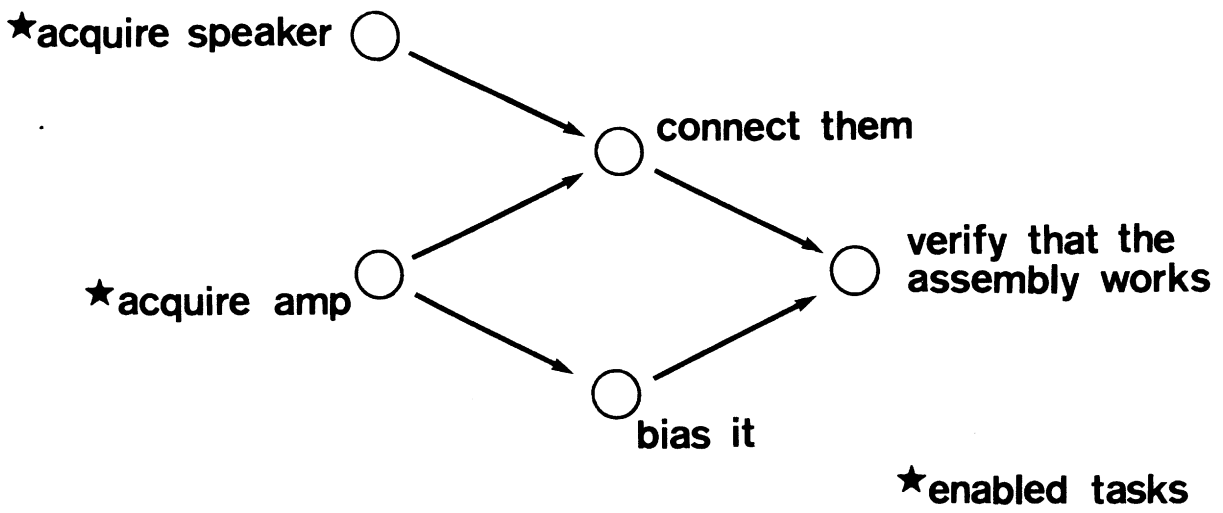

Figure II.1 A Task Net, or "Plan"

Finally, a typical production-system action is always a primitive that can be carried out immediately, while some NASL tasks require must be broken down into subtasks in order to be executed. This requirement is what makes NASL a problem solver. In other words, a task can be as much a part of the problem as of the solution; it looks like part of the solution to its superiors, and part of the problem to its subtasks.

Thus, a task (or action) is either ***primitive*** or ***problematic***. An action may be primitive in two ways. It can have a LISP program for carrying it out, or it can have a set of ***model manipulation*** statements that hold true of it. These statements are the same as STRIPS's add- and delete-lists. (Fikes and Nilsson, 1971) They are sufficient to represent completely only the simplest of actions, but they make these actions easy to reason about. (Cf. Sacerdoti, 1975).

A ***problematic*** task must be "reduced" to one or more ***subtasks***. This relation between tasks is expressed by formulas of the following sort:

[/:SUBTASK |subtask name| |supertask name|]

A task can be the subtask of more than one supertask.

Task reduction can occur in more than one way. The deductive system can infer a complete set of subtasks of a task in the course of forward deduction. However, this fails to give enough direction or power to the problem-reduction process. As described in Section II.B, the interpreter must have the concept of one action being a way of carrying out another, expressed like this:

[/:TO-DO |task name| |action| < -output pvars- > |method|].

This is intended to mean that the method is an effective, feasible, and permitted way of accomplishing the task consisting of the action. Such statements can be used in the creation of subtasks.

The "method" to which a task is reduced may consist of a single action, or it may be a "macro action" which stands for an entire subnet of new tasks. This requires the notion of a ***plan schema***, an abstract object, ***instances*** of which may be thought of as hanging as little subnets off nodes in the task network. (Fig. II.2) The manipulation of instances of these schemata is described in Sect. II.B.1.2.1.

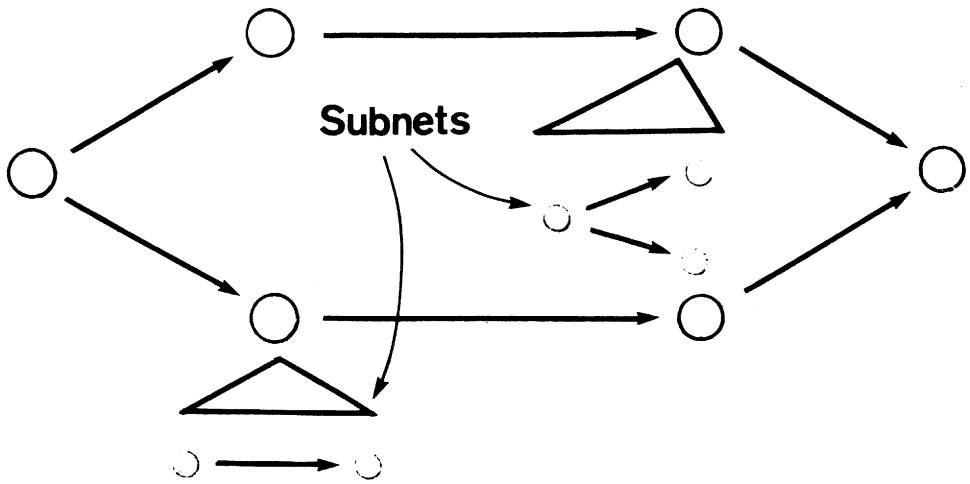

Figure II.2 Task Network with Subnets

Thus tasks may be classified according to whether their actions are primitive or problematic. These classifications form one of the taxonomies shown in Fig. II.3.

Problematicity

Primitive
    Model manipulator
    Macro
    Primitive policy
Problematic

Monasticism

Inferential
Worldly

Parasitism

Primary
Secondary

Figure II.3 Logical Taxonomies of Tasks

The other two taxonomic classifications are independent of this one. One classifies tasks according to "monasticism." Every task is either ***inferential***, in which case it consists of inferring formulas from other formulas supposed to be true; or "***worldly***," when it or some of its subtasks perform model manipulations. This classification is expressed by means of the predicate /:INFERENTIAL.

The last taxonomy is the important classification by "parasitism." A task is either ***primary***, meaning that it has steps to be pursued in order; or ***secondary***, meaning that its "execution" amounts to influencing or monitoring the execution of primary tasks. Ongoing secondary tasks are somewhat grandiosely called "***policies***." How they are handled is described in Sect. II.B. Some representative classes of policies, expressed in English, are

"Wait until ... is true."
"Notice if formula ... is removed."
"Take into account desired feature ... of the device you are designing."
"Constrain quantities ... to satisfy ..."

Policies, like primary tasks, may be primitive or problematic, worldly or inferential.

A policy may have a *scope*, which is the primary task whose execution (or whose subtasks' execution) it is intended to influence. As you might expect, this is indicated thus:

[/:SCOPE |secondary task name| |primary task name|]

Policies do not outlive their scopes. In drawing task networks, I will put a little cloud around a task to indicate that it is the scope of one or more policies; the policies will be tied to these clouds with a line. (Cf. Figs. III.7 and III.8.)

There is no mystery to the notion of policy. All computer programs embody policies; the particular data-base and interrupt mechanisms I use to implement them are commonplace in AI applications. The novelty is that the notion has been made explicit, and, in a modest sense, put into the logical calculus. This prevents two problems with the usual use of the implementation mechanisms. First, typical AI-language "demons" (Charniak, 1972) fire off in the middle of primitive data-base operations and get complete control of operations. Without conventions, it is difficult for other processes to know what the intentions of these little monsters are.

Second, policies are to be used to express things like "loop invariants" and "program assertions" (Floyd, 1967), which are usually extraneous to actual program text and only indirectly related to individual program steps. But a problem solver has need of the notion of a "partially-reduced" problem, some of whose subtasks have not been fully reduced to primitives. This is difficult to capture without the concept of a policy. For example, consider a program to count the prime numbers in a table. The text of the program contains instructions to initialize a counting variable and increment it just

after a prime number has been discovered. The purpose of this variable may be expressed by an invariant of the form "x is the number of primes in the part of the table already looked at." What I am trying to capture is the notion of an early, unfinished version of the program, in which the pieces of text do not yet exist, and the invariant is all there is.

A plan is, in a sense, this kind of unfinished program, with the difference that it gets executed without ever getting completely written. Comments on a plan are not there to explain an existing text or to help prove that it works; they are there to explain an ongoing course of action, and they must be executable. Their individual steps may indeed involve initializing and incrementing counters; these will become subtasks of the policy.

I will conclude this section by listing some limitations of this plan calculus. These fall into two categories: bad limitations and good limitations.

The bad limitations are those due to the fact that I knew the plan language was going to be used for designing and I didn't have the time to implement unnecessary features. So I didn't put in features such as other agents' plans, or notions like "prerequisite of an action." These and other inadequacies are described at slightly greater length in Chapter VI.

The good limitations are those arising from these goals:

(1) Deductions about plans ought to be simple and shallow.
(2) New knowledge must be expressed in a notation compatible with old.

By deductions about plans, I mean deductions about current plans, not "proofs of plan properties." (Cf. Sect. VI.C) It ought to be easy to deduce what you are doing. Otherwise, the executions of subplans cannot interact, and the notion of policy will be meaningless. The second requirement is related to our desire for flexibility. New knowledge is worthless unless it is expressed

in a familiar language. There should be just one obvious way to express any given piece of control information. (Keep this in mind as I expand on the set of control concepts in the following sections.)

An example of a good limitation is that no loops and conditionals are allowed in the language. That is, all iteration and testing is done in the deducer. There are no gotos in the system. There are instead much higher-level concepts like "choosing." It remains to be seen whether I have been successful in inventing transparent but powerful control ideas. (I should mention that recursion is not forbidden in the system; a plan-schema instance can have subtasks derived from an instance of the same schema. It probably should be forbidden, in this general form; I use it sparingly.)

II.B Interpretation and Inference

One thing to do with the predicates I introduced in the last section is to put them in axioms and prove things with them. For example, many of the electronics and design facts in Appendices 2 and 3 have conclusions of the form [/:TASK ...], meaning, "I should be doing ...." Clearly, a system which just proved things of this sort without acting on them would be a perfect catatonic. Its deductions would occur in a void. Their full meaning depends on there being an "action system" which interprets the result of such deductions as commands to act. I will call formulas like this which directly influence action ***pragmatic formulas***; the characteristic functions of these formulas are ***pragmatic functions*** or, more specifically, ***pragmatic predicates.*** I have already observed the convention that the names of such functions and predicates always start with "/:" to emphasize that ***their meanings depend mostly on the action system.*** In this section I will introduce more of them.

(A complete catalog appears in Appendix 1.) All predicates not directly influencing action mean, in some sense, only what the axioms they appear in say they mean.

In this section I describe the operation of the interpreter and the theorem prover it uses, called STP.

II.B.1 The NASL Interpreter

The outer loop of the interpreter is to

```
Pick a task to work on;
If it is primitive,
    Execute it or elaborate it;
    Otherwise, find a way to work on it ("reduce" it);
Repeat until there are no more tasks
```

The first step of the interpreter cycle, "picking a task," is done by a system of forward deduction of /:SUCCESSOR relations. The axioms that support these deductions are the user's responsibility. The system chooses at random from the tasks that it is logically permitted to do next.

Much of the work is in the second arm of the conditional. The existence of this step is what makes NASL a problem-reduction problem solver instead of a programming-language interpreter. Reducing a task involves a call to the theorem prover STP and some more powerful mechanisms. (Sect. II.C)

II.B.1.1 Selecting a Task to Work On

The NASL interpreter interleaves planning and execution of plans. (Cf. (Nilsson, 1973).) Different tasks are in different states, which change as time passes. The current state of a task is composed of its ***task-status***, its ***enablement status***, and, for problematic tasks, whether it is ***reduced***. (Fig.

II.4) When a task is created, its state is PENDING and BLOCKED. When a PENDING task is ready to be executed, it becomes ENABLED. While it is being worked on, it is ACTIVE. When the interpreter is through with it, it is FINISHED. The status of a task is expressed in a formula of the form

[=/> '(/:TASK-STATUS |task name|) |status|]

where status is one of the three states I gave.

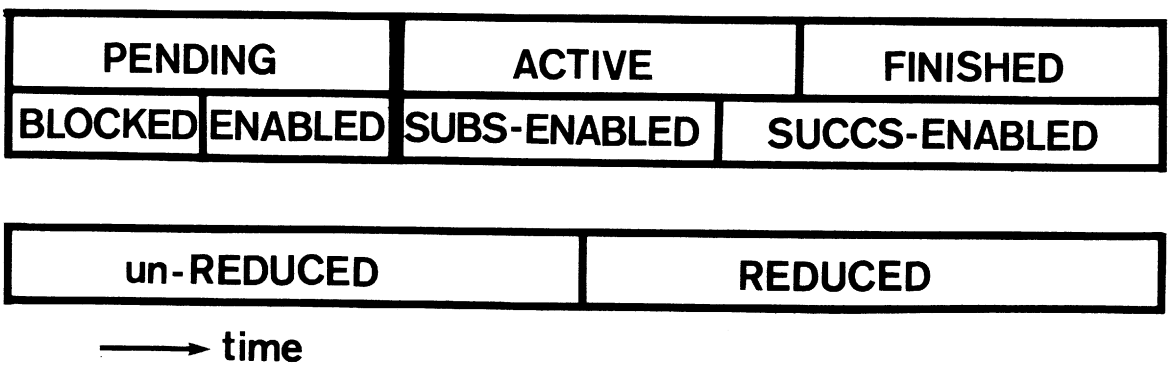

Figure II.4 Life History of a Task

Meanwhile, as a task evolves, its enablement status changes to "gate" its subtasks and successor tasks. Recall from Sect. II.A that the order of execution of tasks is constrained by /:SUCCESSOR relations. In addition, subtasks of a task may be deduced before the task itself becomes active; the subtasks must be postponed. So there are three facts that must be true before a task can be enabled: all its super-tasks must be ACTIVE; all of its input pvars must be known; and all of its predecessors must have enablement status "successors enabled." (Fig. II.4) When a task is FINISHED, its successors are always enabled, but the system must be flexible enough to allow execution of successors to begin before this. For this reason, I introduce the independent concept of enablement status,

[=/> '(/:ENAB-STATUS |task name|) |status|],

where status is BLOCKED, ENABLED, SUBS-ENABLED, or SUCCS-ENABLED. These flags

are synchronized with the ordinary task-status as shown in Fig. II.4. As a task becomes active, the system checks its subtasks, and enables all those with no other impediments; similarly, when the task enters SUCCS-ENABLED mode, the system checks its successors. (It should be clear that if a task has two predecessors or super-tasks or some combination, all must be in the proper state.)

A useful service provided by the system is that as soon as all the input pvars of a task are known, whether or not there are other gating conditions remaining unsatisfied, a formula of the form

[/:TASK-ACTION |task name| |action|]

is recorded in the data base.

Figure II.4 also shows the transition of a problematic task from being "*unreduced*" to being "*reduced.*" When a task has been completely replaced by subtasks, the proposition

[/:REDUCED |task name|]

is supposed to hold true of it. The system will not bother to reduce a task if such a formula has already been deduced; this enables task networks to be built up entirely by forward deduction.

II.B.1.2 Executing Tasks

When a task has been selected, it must be executed. If its action is of the form [|*f*| ...], I call *f* its ***action function.*** The system can tell by looking in the data base or on the property list of *f* whether the task is primitive or problematic. If it is problematic, it must be reduced.

II.B.1.2.1 Primitive and Problematic Tasks

An action can be primitive in one of two ways: its action function can have a defining LISP function on its property list, or it can be defined by model-manipulation axioms. The latter are looked for first.

The interpreter calls STP to deduce formulas of the form

[/:MOD-MANIP |task name| |action| ?DELETELIST ?ADDLIST],

where the variables ?DELETELIST and ?ADDLIST are intended to become bound to tuples of "senses," or quoted facts. (See Appendix 1.) For example, we might have

```
[(ON ?X ?B) ⊃
 (/:MOD-MANIP ?TASK (MOVE ?X ?A) <'(ON ?X ?B)> <'(ON ?X ?A)>)]
```

in the BLOCKS world. The meanings of the addlist and deletelist are the traditional ones. (Fikes and Nilsson, 1971) The model (data pool) is to be updated in the obvious way: the formulas represented by the elements of the deletelist are deleted, and those represented by the addlist are added. These manipulations are called *model effects.*

If the primitive has a defining LISP function on its property list, that function will just be executed. It can do something, return a value, establish a policy, or annex a subnet. An example of the first kind is the action [GRABBA |property|] in the design world, which creates an object with the property. Values are returned by deductive actions like /:FIND, which call STP to retrieve data.

The most important kind of primitive is the "macro," which annexes a subnet. The typical member of this class is

[/:DO-SUBNET |plan schema| |vars-map|].

which is used to instantiate plan schemata and hang them off the net.

In the current implementation of NASL, plan schemata are not represented as identifiable objects. Instead, they are defined implicitly through statements of the form

```
[(/:PLAN-INSTANCE ?NAME |plan schema| ?SUPER-TASK)
 ⊃ (AND (/:TASK |subtask 1| ...)
        (/:TASK |subtask 2| ...)
         ...
        (/:SUBTASK |subtask 1| ?SUPER-TASK)
        (/:SUCCESSOR |subtask 1| |subtask 2|)
        -other connectivity relations-)]
```

by which nets of subtasks are created and linked together. Executing /:DO-SUBNET creates a new plan instance and records

[/:PLAN-INSTANCE |plan instance name| |plan schema| |super task|].

This will trigger the forward deduction of subtasks in the schema.

These subtasks will compute and use the values of plan variables ("pvars"), some of which the super-task network needs; the vars-map argument of /:DO-SUBNET tells how to map the schema's variable back to the calling plan. To make this work, all the pvars used by the tasks in the schema must be of the form [(|var name| |plan instance name|)]. (For an example of the use of /:DO-SUBNET, see the formulas +DESI-1 and +DESI-2 in Appendix 2.)

A macro-expanded task will be FINISHED when all its subtasks are. It will have enablement status SUCCS-ENABLED when all of its "main" subtasks are FINISHED. This device is intended to capture the idea of a task reducing to two kinds of subproblem: things which must be done before going on to the successors of the task, and things which can wait. An example is biasing one stage of a complex circuit (see Appendix 3); this will appear as a subtask of acquiring a circuit, but it should not be done when the circuit is first obtained; instead, it may become a successor of, e.g., coupling the circuit to something else. Subtasks labeled /:MAIN are those whose completion is a necessary condition for enabling a supertask's successors. (See Fig. II.5.)

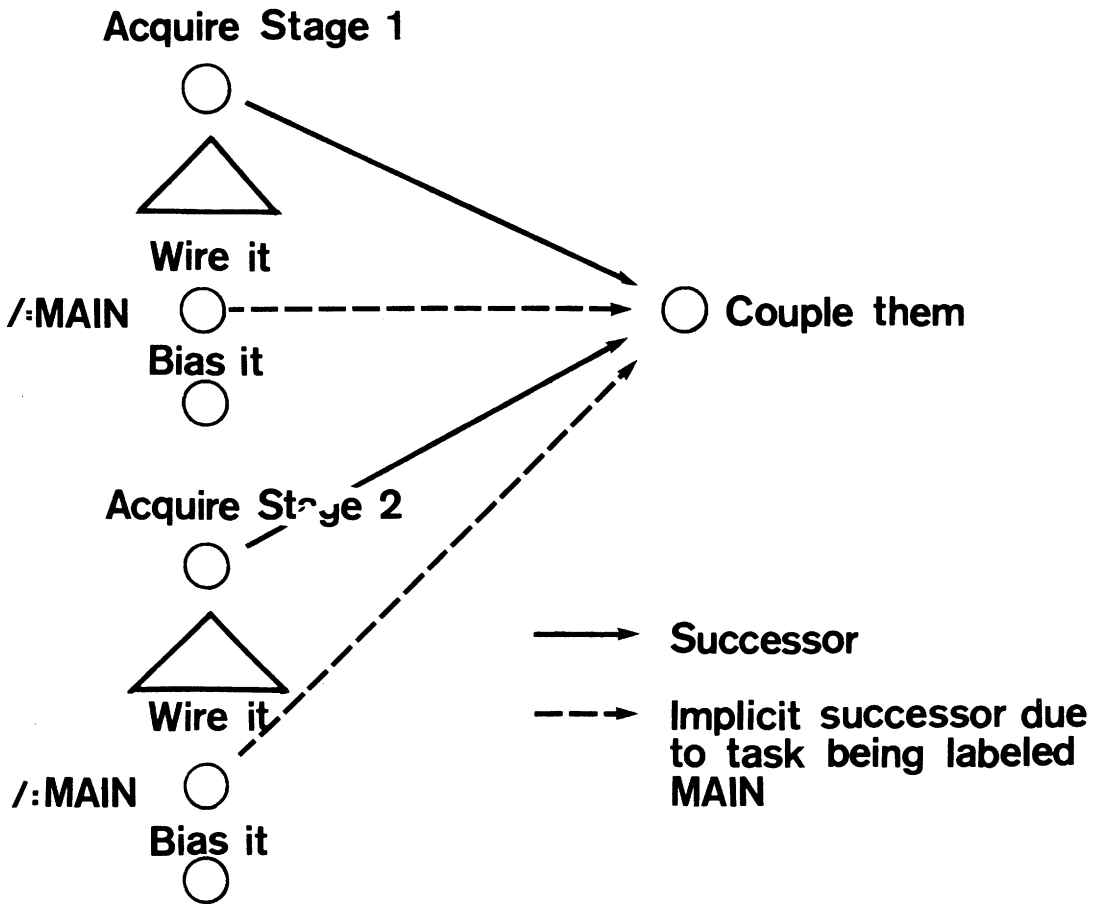

Figure II.5 Enablement Relations in Subnets

This is one of the ways in which the subtask relation differs from the usual relation between a program step and its program.

Other macro actions are described in Appendix 1.

This concludes my description of the execution of macros and other primitives. All other tasks have "problematic" actions. In such a case, NASL calls STP with the request

[/:TO-DO |task name| |action| < -output vars- > ?WAY]

If STP returns exactly one value for ?WAY, a new task for the new action is created, enabled, and made the /:MAIN subtask of the current task (which

becomes /:REDUCED). If STP does not return exactly one value, special things must occur which are the topic of Sect. II.C.

II.B.1.2.2 Primary and Secondary Tasks

Primary tasks are those which do something or infer something. Primitive primary tasks are those defined by /:MOD-MANIP and inferential functions. Secondary tasks ("*policies*") are those which influence the execution of primary tasks.

The principal primitive policy is

[/:MONITOR |formula| (λ (|*v*|) |action|)],

which does nothing unless some task removes the formula as a model effect. Then a new subtask will be created with the given action, with *v* bound to the task that did the removal. This is used to implement protection.

Policies may cause the "intermittent" execution of primary actions. A task with action [/:CONTINUE |policy task| |action|] will be executed in a nonstandard way. It causes a deduction of the form

[/:TO-CONTINUE |policy task| |action| <> ?WAY]

and the resulting sub-action is attached to the original policy task node of the task network. (See the implementation of protection described in Chapter III.) Thus a policy may occasionally cause execution of real actions in the process of executing /:CONTINUEs.

A problematic task may also be primary or secondary. This is not determined when the task is reduced, but after its /:MAIN subtasks have been set up. At that time, if any of its subtasks are discovered to be secondary, and to have a scope larger than it, it is declared to be a policy.

The main difference between the execution of primary and secondary tasks

is in how they are finished. A secondary primitive will not be finished until the task which is its scope is finished; then the interpreter executes [/:FINISH |policy|] to clean it up. Problematic tasks of both kinds are finished when all their subtasks are.

Here is a summary of the ways in which policies influence primary actions:

(1) The primitive policy /:MONITOR is used to implement things like "protection." (Sussman, 1975)

(2) The presence of formulas regarding the status of a task can license deductions of various sorts. The conclusions can be of the form [/:TASK ...] and [/:TO-DO ...], for example, and thus trigger things to do and ways of doing them.

(3) In particular, policies often influence the "choice protocol" described in the next section.

(4) The use of /:CONTINUE can cause intermittent execution of primary actions.

When policy-specifying formulas influence the interpreter's deductions, it will record their influence in the form of /:SUBTASK assertions. That is, when /:TASK formulas are deduced from policy task formulas, they become subtasks of those policies. (Sect. II.D) Thus, a natural structure evolves in which a task can be a subtask of "make a filter" (primary) and "keep the cost down" (secondary).

II.B.2 STP -- The Stupid Theorem Prover

STP is a backward-chaining, pattern-matching theorem prover. In R. Moore's (1975) phrase, it is a ***procedural deductive*** system. Such a system may best be thought of as a descendant of PLANNER (Hewitt, 1972) which emphasizes its logical aspects instead of emphasizing its programming-language features as most other descendants have done. By this I mean that it manipulates, not arbitrary list structures, but formulas that are supposed to represent

statements about entities. There are no side effects during deduction; the action system is completely divorced from the operation of the theorem prover. This means that the theorem prover can be optimized in various special ways. (See Appendix 4.)

STP is used by the system for two kinds of deductions: those about tasks and actions and those about the physics of the problem domain.

STP is not particularly bright; it is to be used for information retrieval, and it tends to balk at intricate reasoning. More sophisticated reasoning is done as inferential tasks. (There are things to regret about this organization. See Sect. VI.B.) Some kinds of reasoning do not naturally fit into the theorem-proving paradigm at all. These will be discussed in Sect. II.C.

Actually, "theorem prover" is a very misleading term. The "theorems" such programs prove would not be recognizable to a mathematician; the way in which they go about it would be even more incomprehensible. Nonetheless, I will continue to use this term, since by now AI people are unlikely to read anything very pretentious into it.

A theorem prover may be thought of as a problem-oriented interface between a problem solver and bare data-base machinery, such as that described in (McDermott, 1975). For example, an AI data-base manager implements the notion of "data pool." This will be used to implement the higher-level notions of "packet" and "reference point." (See below.) The calculations involved can be made invisible to the user, who thinks in the higher-level terms.

The basic data-base operations are three: putting things in, taking things out, and finding things. These are handled by the three primitive (LISP) operations RECORD, ERASE, and STP. RECORD puts a formula into a data pool. It also does forward deductions from that formula in a way to be described.

The results of these deductions are recorded also, and conclusions are linked by "***data dependencies***" to the formulas which support them. (See Sect. II.D.) ERASE flushes a formula and everything it supports from a data pool.

When proving theorems, STP works, like every other "theorem prover," by matching goal formulas against "knowledge" formulas, detaching the output, and repeating until a proof from atomic data is obtained. (Cf. (Bledsoe, 1975) For technical reasons (R. Moore, 1975), STP really attempts to refute the negations of goals. See Appendix 4.)

Formulas are stored in the data base in clause form. Clauses are implications whose format tells how they are to be used. The two most common forms are

[-/> C |*p*| |*q*|], meaning, "to prove *q*, prove *p*"
and [-/> A |*p*| |*q*|], meaning, "if *p* is recorded, record *q*."

(These correspond in an obvious way to Planner's consequent and antecedent theorems. (Hewitt, 1972, Moore, 1975)) The arguments *p* and *q* to these predicates can be clauses as well; my clauses have more pragmatic structure than those of a resolution theorem prover. (Robinson, 1965)

Internally, these clauses are stored as (pragmatic) ***disjunctions*** of the form

[/:CONSEQ |*q*| (NOT |*p*|)]
and [/:ANTEC (NOT |*p*|) |*q*|]

respectively. These forms are occasionally useful externally as well.

A third pragmatic disjunction is /:GEN. [/:GEN |*p*| |*q*|], when "recorded," really causes counterexamples to *p* to be found and, for each one found, an instance of *q* to be recorded. This may be also be expressed [-/> G (NOT |*p*|) |*q*|]. For example, recording

```
[-/> G (DEV-TYPE ?X COMMON-EMITTER)
       (DEV-TYPE ?X AMPLIFIER)]
```

calls STP to find all common-emitters and record that they are amplifiers. In this way backward deductions may be triggered in the midst of forward chaining.

STP is oriented toward the task of information retrieval. When given a goal with free variables, it doesn't interpret it as a request to prove that objects exist which would satisfy the formula if substituted in; instead, it considers it a request to find and return these objects. This is like PLANNER (Hewitt, 1972) and QA3 (Green, 1969a,b). (See Appendix 4.)

For example, given the clauses

```
[P A]
[P B]
[Q B]
[-/> C (AND (P ?X) (Q ?X))
       (R ?X)]
```

and the goal: Refute [NOT (R ?Y)],

STP chains backward through the consequent clause to generate as subgoal

Refute [NOT (AND (P ?Y) (Q ?Y))].

This becomes two "conjunctive goals"; "Refute [NOT (P ?Y)]" and "Refute [NOT (Q ?Y)]." STP finds Y → A and Y → B as answers to the first goal, and detaches [NOT (Q A)] and [NOT (Q B)] as alternative versions of the second. Only the latter of these succeeds. The returned answer is therefore Y → B.

The machinery to make this work reasonably well is described in Appendix 4.

Some other interesting features of STP are these:

(1) Ability to call LISP functions for low-level deductions. (Cf. (Nevins, 1974a,b).) I have made an effort to keep all such LISP-implemented concepts completely primitive and domain-independent. These are concepts for manipulating simple inequalities, predicates on embedded formulas, etc.

(2) "Non-monotonic" inference rules, which are implemented by having

certain predicates be evaluated by calling STP recursively. For example, [/:CONSISTENTLY '|pattern|] will be handled by calling STP to see if pattern can be refuted. (The single quote is used to flag an expression within which substitution of equals for equals is forbidden; such an expression is called a "sense.") McCarthy's "presumably" operator (McCarthy and Hayes, 1969) is defined as

[(PRESUMABLY '?P ?USE) = (-/> ?USE (/:CONSISTENTLY '?P) ?P)]

meaning, "if you can't prove ?P is false, assume it's true." Thus we have, [∀X (BIRD ?X) ⊃ (PRESUMABLY (CAN ?X FLY) C)], which means, "If X is a bird, then if you ever need to check if he can fly, assume he can if you can't prove he can't." (If the formula had had "G" instead of "C," the attempt to refute his ability to fly would be done at the time he was deduced to be a bird.)

(3) Pragmatic handling of equality. The usual predicate-calculus notion of equality does not correspond very closely to the programming notion of evaluation. If you ask a theorem prover, "Find ?x such that 2+2 = ?x," it will tell you, "?x = 2+2," which is true but useless. An action module communicating with a deductive system must have the concept of "useful expression." In the midst of problem solving, some data structures are inherently more oriented toward getting on with things. Consequently, STP works closely with an ***evaluator*** (see Fig. I.9), which applies ***rewriting rules*** found in the model to expressions. We have already seen these rules in action implementing pvars. They look like [=/> '(+ 2 2) 4]. The "sense" quote is a way of forbidding applying the rules to subexpressions. (Otherwise, the rule would rewrite itself as [=/> 4 4].)

The evaluator is used by the interpreter, by user plans which use the /:EVAL primitive, and by STP. (See Appendices 4 and 5.) Normal equality, [= |x| |y|], is used to express goals like, "prove two things are equal."

There is a "cheap" equality predicate called ":=". The only knowledge about it is [:= ?THING ?THING]. It is used in conjunctive subgoals to "set" variables for future use. That is, the goal [:= ?X (FOO BAR)] succeeds, setting ?X to [FOO BAR]. The system will not waste its time trying to prove a goal like this if it doesn't succeed immediately.

When an equality is recorded in the course of trying to prove x and y *unequal*, the system makes an effort to translate it into a rewriting rule; otherwise, it will never interact with other deductions. Cf. (Bledsoe and Tyson, 1975).

(4) "Packets." It often inconvenient to have to record a large conjunction as a consequence of some forward deduction. For example, in electronics, devices are of various types. If it is recorded that [DEV-TYPE DEV#73 COMMON-EMITTER], this might trigger the recording (*via* "-/> A") of scores of facts about DEV#73, most of which will never be looked at. This can be avoided by writing the relevant antecedent implication as

```
[-/> A (DEV-TYPE ?CE COMMON-EMITTER)
       (/:PKT CE-PKT (?CE)
            |fact 1|
            |fact 2|
              ...
            |fact n|)].
```

As explained in (McDermott, 1975), defining this formula will create a packet which plays the role of the large conjunction with one free variable ?##CE. It is actually implemented as a "data pool layer" which can be added cheaply to the current data pool. The individual facts will be closed and indexed only as they are accessed.

(5) A "modal" notation and inference mechanism. A general deductive system should be able to reason about hypothetical situations, other times, other creatures' beliefs, etc. These concepts are in the domain of "modal"

logic (Hughes and Cresswell, 1972), a difficult study with many problems. I have implemented a modest system for doing some very simple modal deductions, which uses the "data pool" mechanism to implement "reference points." (Montague, 1974, Rescher and Urquhart, 1971)

The basic modal notation in the NASL language is [T |reference point| |term|], which stands for the value of the term with respect to the given reference point. In principle, these reference points could be other creatures' minds, arbitrary points in time, or just "possible worlds." Under this last interpretation, logical necessity might be taken to mean [∀R (T ?R ...)], or "... is true in all possible worlds." However, this would require quantifying over reference points, a capability I have not had the time to pursue. Instead, DESI confines itself to the use of constant reference points. These are used (see Chapter IV) for things like the DC and sinusoidal-steady-state models of an electronic circuit.

This sort of mechanism is just a convenient notation for data pools (i.e., "contexts") from within the logical language. To make it work, I have introduced some notation for "frame" axioms. (Hayes, 1973a) A reference point is often defined in terms of the differences between itself and some set of super reference points from which it inherits most of its contents. These definitions are written thus:

[FRAME |reference point| < -reference points- >] means that a statement is to be assumed true in the given reference point if it is true in one of the other reference points and cannot be proved false. The given reference points are called *frames* of the new one. That is,

```
[(FRAME ?REF ?FRAME-REFS) ≡
 (∀P (∃F (?F ∈ ?FRAME-REFS) ∧ (T ?F ?P))
      ⊃ (PRESUMABLY '(T ?R P) C))].
```

Of course, it isn't implemented in this way. Instead, a new data pool is constructed using the FRAME axioms when it is required. This data pool has as superiors the data pools corresponding to its frames.

[N |reference point| '|fact|] means that the given fact is *not* inherited from the reference point's frames.

Formulas of the form [T |reference point| |fact|] are used in constructing new reference points. Any such propositions lying around have their facts shoved into the new data pool.

Examples of the use of these formulas are given in Chapter IV.

II.C Choice and Rephrasing

As sketched so far, NASL resembles some more familiar problem solvers. Except for the imposed distinction between deduction and action, it is a lot like PLANNER. (Hewitt, 1972) The main difference is that it does no backtracking past model manipulations. Since it is more disciplined in many ways, it is better able to explain its actions.

However, it suffers from some of the same problems as PLANNER-like systems. In particular, a certain amount of the additivity I wanted will not be found in this organization. Even though it is easy to add a new plan schema to a body of facts, the interactions of this new material with the old are not so easily handled.

For example, if acquiring a common-collector amplifier is known to be a good way of achieving high input impedance, this fact might be lying around in a formula of the form

```
["high input impedance required"
 ⊃  (/:TO-DO ?T (MAKE AMPLIFIER) <?N>
             (MAKE COMMON-COLLECTOR))].
```

(For a precise version of this, see Appendix 3.) Now, say that the system is to be told about field-effect transistors (FETs). Since they have a high input impedance, an exactly similar fact will be recorded regarding the FET common-source amplifier.

Now a request to make an amplifier will cause both these facts to be retrieved. What can be done? (We have already ruled out just trying one until it fails.) One approach would be to force the user to revise one or both of the formulas to check for information that will distinguish between the two cases. However, this will lead to large, impenetrable implications. Furthermore, in some cases of such confusion, neither choice is preferred, but some synthesis of the two. We need a way to represent such "differential diagnosis" and "partial-solution composition" knowledge in an additive manner.

The solution is to face up to the necessity for treating "choice between alternatives" as a basic situation of problem solving, and to create new pragmatic predicates for handling it. This is the subject of Sect. II.C.1.

The complementary problem that this brings to mind is when the deductive pattern-based backward chaining of STP is unable to retrieve *any* possible plans. This might be because there aren't any, and the user must provide new information, but it also might be because the relevant retrieval strategy depends upon pattern-manipulation operations which are less disciplined than unification. For example, we might want to express, "If the problem mentions MHz, try special high-frequency heuristics." Here the traditional AI language solution is to allow arbitrary list-processing operations upon formulas. (The traditional predicate-calculus solution is to do aimless equality substitution.) Thus, in CONNIVER (McDermott and Sussman, 1973) a method with pattern (LAMBDA !>X !>Y) can match the calling pattern (LAMBDA (X) (F (G X))) and do anything it likes with the pieces so generated. This is somewhat abhorrent, since it tends to destroy the notion that formulas mean anything. Who can rule on the consistency of a set of formulas that do things like that?

My approach to this problem is to try to impose some discipline on this kind of manipulation. The idea is to signal explicitly when the system is

allowing itself to do things like that, and to impose restrictions on its behavior and the results it computes. This idea is developed into the "rephrasing" protocol of Sect. II.C.2.

II.C.1 The Choice Protocol

Under some circumstances, STP is asked to return all the answers it can find (cf. Appendix 4), but it can be asked to return just one. In this situation, if more than one answer is found, the system performs a ritual invocation of information about choosing between them. This is called the ***choice protocol.*** For example, this protocol is called when DESI finds more than one possible circuit for a general concept like "amplifier." In that case, detailed information about the various types of amplifier interacts with information about what is required of this amplifier to force a choice.

The first thing the chooser does is to create an (abstract) choice situation name and record in the data pool

[/:CHOICE |name| |context| |goal formula|]

(The context is the inferential task for which more than one answer is found. In the case of the interpreter trying to deduce how to do something, this is just the symbol "EXEC.") For example, in trying to choose an amplifier, it would record

```
[/:CHOICE C#535 EXEC
    [/:TO-DO TSK#437 (MAKE AMPLIFIER) <'(STAGE1 CKT#747)>
        ?WAY]]
```

The formal resemblance of this to /:TASK formulas is suggestive; we have in effect added a new kind of entity, the choice. The intent is that this formula will trigger forward deductions of the kinds to be described in a moment. The packet machinery of (McDermott, 1975) will allow the system to

bring in large packets of what may loosely be called "advice" appropriate to this situation.

The use of "brackets inside brackets" is our first encounter with the concept of "***embedded formula.***" (See Appendix 1). The system is treating the goal here as a data structure to be analyzed.

For each of the possible answers, a formula of the form

[/:OPTION |choice name| |option name| |answer formula|]

is recorded in the data pool. For our amplifier example, we might have

```
[/:OPTION C#535 A#450
    [/:TO-DO TSK#437 (MAKE AMPLIFIER) <'(STAGE1 CKT#747)>
        (MAKE COMMON-COLLECTOR)]]

[/:OPTION C#535 A#451
    [/:TO-DO TSK#437 (MAKE AMPLIFIER) <'(STAGE1 CKT#747)>
        (MAKE FET-COMMON-SOURCE)]]
```

Recording these formulas will trigger the deduction of formulas of the form

```
    [/:RULE-OUT |option name|],
    [/:RULE-IN |option name|],
or  [/:RULE-TOGETHER < -option names- > |new answer formula|].
```

The system first searches for conclusions of the form [/:RULE-OUT ...]. This is a call to STP, of course. If any are found, the options ruled out are removed from consideration. Next, the system looks for conclusions of the form [/:RULE-IN ...]. If any of these are found, the system throws away all options ***except*** those mentioned. Finally, it looks for /:RULE-TOGETHERs. If one of ***these*** occurs, the options it mentions are discarded in favor of the new answer formula.

If at any stage all options but one are eliminated, the protocol stops with a winner. If all the options are ruled out, the system enters an error protocol to show the user what it did and ask for corrections of its misinformation. If more than one option survives, the system records

[/:QUIESCENCE |choice name|]

in an effort to trigger more forward deductions.

The intent of these devices is clear. Differential diagnosis is to be performed by the first two kinds of formula, while /:RULE-TOGETHERs are intended to be one locus of composition of partial solutions in the NASL system. (The others are problem reduction (see above) and error correction (see Sect. III.D) in the context of patching electronic circuits.) The /:QUIESCENCE trick enables the user to encode advice of the form, "All other things being equal...," as a forward implication like

[-/> A (/:QUIESCENCE ?C) ...].

The choice protocol keeps track of the rules which contribute to weeding out all but one option. These rules are used in building data dependencies (sect. II.D). In addition, when a policy is used in choosing a way to do something, the choice is made a subtask of that policy. For example, say there is a policy of the form "keep costs low," plus a deductive rule like,

> "When trying to make a device, and trying to keep costs low, then, all other things being equal, if a circuit with inductors is competing as an option against a circuit without them, the one with inductors is ruled out."

Now if the task of constructing some circuit is elaborated into a device chosen on the basis of this rule, the task of acquiring the device is subtask of both the construction task and the costs policy. This leads to clear explanations by the system of its behavior. (Sect. V.A)

(The choice protocol was inspired by the design of Marcus's (1973, 1975) "wait-and-see" parser, which does similar things in choosing directions in which to parse.)

II.C.2 Rephrasing

I now turn to one of the most important and least elegant subsystems of NASL, the ***rephrasing protocol***. This is the system which is invoked when STP is unable to find a reduction of a task. Rephrasing consists in treating the recalcitrant problem as an object to be transformed into a new problem. The pious hope is that the new one is easier. This, of course, is precisely the object of task reduction in the first place. So rephrasing may be thought of as taking extraordinary measures to reduce a task.

The way this works is as follows. When the system is unable to find a way /:TO-DO something, a task

```
[/:TASK |name| <>
    (λ () (/:REPHRASE |task| |action formula| < -output pvars- >)) <>]
```

is created, and made a predecessor of the losing task. This task is allowed to carry out arbitrary inferences in order to reduce the unreduceable task.

For example, the design task DES#78, with action

```
[DESIGN (λ (X) (AND (IS AMPLIFIER ?X)
                    (= (VOLTAGE-GAIN ?X) 100))   )]
```

is unlikely to trigger an indexed solution. Instead, it must be rephrased as some set of simpler actions, by the use of electronics knowledge. So the task

```
[/:TASK T#849 <>
     (λ () (/:REPHRASE DES#78
                [DESIGN (λ (X) (AND (IS AMPLIFIER ?X)
                                    (= (VOLTAGE-GAIN ?X) 100))   )]
                <|result pvar|>)   )
     <>]
```

will be put in the task network as a predecessor of the design task. Its effect will be to reduce task DES#78.

The rephrasing protocol must exist in order to provide for deductions beyond the scope of STP's simple strategies. These fall into two categories,

one more elegant than the other. First, because it uses the interpreter, it can take advantage of the choice protocol, flexible planning and policy making, and even recursive rephrasing. Thus, for example, one can make finer choices than is allowed by just running the chooser on a set of possible reductions.

Second, and less happily, the rephrasing protocol manipulates the action formula as an "embedded formula," and so is allowed to perform any operation on its representation. So one can write rephrasing plans which check to see if the given action refers to "WIDGETS" anywhere. In the next chapter, I will show how, in the course of rephrasing design problems, λ -expressions are routinely dismembered. This seems to be indispensable, but it would be nice if we could insist that the pieces be put back together in a legitimate way.

This is a special case of the more general problem of making sure that the interpreter and inference mechanisms actually do what they are supposed to do. The difficulty is in specifying what the object of a task or protocol is. For choice, the object is fairly clear: eliminate all but one option. (Inelegancy creeps in with /:RULE-TOGETHERS.) Elsewhere in the interpreter, I have ignored this very important problem, except for token checks such as that /:FINISH actually leave its task finished. In the case of rephrasing, the problem is especially acute; rephrasing can be thought of as a device for extending the pattern matcher by allowing arbitrary deductions about formulas. Something like this is necessary, but it should be better constrained.

As it is, there are only a few restrictions on the use of rephrasing: all the actions undertaken as subtasks of a rephrasing must be inferential, not worldly; the rephrasing task must leave its target task /:REDUCED; and the subtasks resulting from a rephrasing must be syntactically legal (i.e., not contain λ's in funny positions or have any free variables, etc.).

The rephrasing knowledge for the design domain, which I present in the next chapter, is an example of what rephrasing ought to be. The formulas involved are reduced to pieces by one task, and parsed together again by others. I hope that this will prove to be an instance of some more general recognition strategy that is more constrained than what the system now allows.

I have now described every module in Fig. I.9. The search paradigm that I have developed may be summarized as: let the theorem prover search, but not too far or too deeply. All searches are intended to be short and sweet; the search is used for exactly those spots where there is no applicable knowledge. These short searches are organized by a plan interpreter, which decides what sort of knowledge is to be accessed. It can ask for answers to questions about how to do things, the physics of the domain, choosing among alternatives, or transforming its own problem statements. Thus, as far as the paradigm has been developed, it is in accordance with R. Moore's (1975) observation that theorem provers are most naturally applicable to information retrieval problems, and that other control structures are needed for more sophisticated tasks.

II.D Dependencies Among Data and Tasks

It is becoming generally realized that AI systems must record their reasons for their conclusions and actions. (McDermott, 1974a, Stallman and Sussman, 1976, Shortliffe, 1976) These records have many uses:

(1) They can be used to explain reasoning and actions to a human user.
(2) They guide the system in undoing faulty deductions.
(3) They are a guide to correcting the effects of misguided actions.
(4) They can be used in assigning the blame to incorrect rules.

The basic relation among data is the ***deductive data dependency.***

(McDermott, 1975) Every time STP or RECORD does a deduction, it attaches such a dependency to the conclusion and the premisses; the latter become the ***supporters*** of the dependency, the former, the ***supportee***. (See Appendix 4.) When the system does an ERASE, all the supportees of the erased item are erased themselves if they have no remaining supporters.

These support relations are accessible to the problem solver as a set of LISP-implemented predicates. In particular,

[/:SUPPORT < -formula names- > |formula name|]

is supposed to be true when the indicated dependency holds.

These support dependencies are also created by the inferential action /:INFER (see Appendix 1). Other inferential tasks call STP and let it build dependencies.

These devices account for the second in the list of uses of dependencies among data and tasks. The others are more complicated, because they involve the relation between action and the world model. Here are examples of the kinds of relations that can occur:

(1) A task can have model effects. The relation between the task and its effects is non-deductive because erasing a task is not sufficient to undo its effects. (Besides, some of the effects are erasures.)

(2) A task or pvar value can be based on choice information. We want to record this relation, but erasing the basis of a choice does not erase the choice, although it calls the wisdom of the choice into question.

(3) Facts in the current model can support task statements. A fact about circuit topology supports a constraint on the physical quantities it influences. Erasing such a model-effect formulas should cause the task formulas to be erased too,

(4) Facts in the current model can *trigger* tasks. This is a quite different situation from (3). NASL implements the common AI mechanism of "demon" or "pattern-triggered interrupt" by allowing /:TASK and /:SUBTASK formulas to be deduced. For example, a BLOCKS-world system may for a time have a policy to the effect that a certain block B#72 is to have a clear top. This gets translated into the principle,

```
[∀X (ON ?X B#72)
     ⊃ (∃T (/:TASK ?T <> (λ () (REMOVE ?X B#72)) <>))]
```

Let B#74 appear on B#72. This will create a task to take it off. A model effect of this task is the erasure of (ON B#74 B#72) and with it the task! This contradicts common sense, since once the interpreter starts to work on something, its success should not erase it. There may even be serious errors as a result of such an erasure, since the erased task may not have been completed yet. In any case, the user may want to ask questions about tasks, without worrying about which ones erased themselves.

It is clear that this problem has to do with the treatment of time. An activity can *become* a task for one reason, but *stay* a task for another. This is handled by the use of the modal operator S, defined as follows: [S '|fact|] means "fact *starts* to be true." The conclusion of the given implication should be [...(S (∃T (/:TASK ?T...)))]. Exactly the same fact will end up in the data base, but the supporting data dependency will be different. (It is a bit wishful to call this a "modal operator" instead of a "patch." If the modal machinery were better developed, it could be supported by axioms like

```
[T ?R ((S '?P) ∧ ((TIME) = ?t))
      ⊃ (∃i (∀Q (T ?Q (?t < (TIME) < ?t+?i)
                      ⊃ ?P)))],
```

but it isn't.)

To represent these nuances, the structure of data-dependencies must be made more flexible. Before, the supporters list of a dependency was just a list of data; now we make it a "labeled tree" of tuples of data. Each label explains the role the supporters play in the dependency. For example, a BLOCKS-world program might execute the task

```
[/:TASK (FIND-DUMP) <>
    (λ () (:FIND (λ (X) (IS PLACE ?X)  ))  )
    <'(DUMP)>]
```

in order to find a place to get rid of a nuisance block. If it chooses X = TABLE because of a choice principle C, the result

```
[=/> '(DUMP) TABLE]
```

will be supported with the two labeled dependencies:

```
(DD-CHOICE ([IS PLACE TABLE]) (DD-CPRIN (C))) and
(DD-INFERER ([FIND-DUMP]))
```

where DD-CHOICE, DD-CPRIN, and DD-INFERER label the roles of the formulas they dominate in their trees. (I am being a little casual about the format of

these structures; when they are attached to the data, pointers to the supporting data themselves appear in place of their formulas.)

Here are some of the implemented labels:

(DD-ACT-RESULT (|task datum|)
(DD-APRIN -action principles-)
(DD-ATRIGGER -action triggers-))
relates a task to its results. The action principles are general formulas (found in the main data pool GENERAL-DP*); the action triggers are formulas that were true (perhaps transiently) when the action occurred. Erasing the latter will not disturb the supportee of the data dependency.

(DD-CHOICE (-inferential supporters-)
(DD-CPRIN -choice principles-)
(DD-CTRIGGER -choice triggers-))
records an inference for which an answer had to be chosen. The rules which contributed to this selection are sorted into triggers and principles just the way they are for actions, but, for choices, the supportee is immune from disturbances to either of the kinds of choice formula.

(DD-S (-triggers-))
labels supporters whose erasure does not affect the supportee. Deducing [S '|f|] will record [|f|] with the supporters insulated by a DD-S label.

(DD-INFERER (|task datum|))
is attached to formulas deduced or inferred by inferential tasks. This is used by other inferential tasks to refer to those formulas.

(DD-ISTATE (-data-))
is used to label formulas, like /:TASK formulas and pvar value assertions, which define the state of the interpreter. These formulas are "incorrigible," and are never erased.

(DD-EXEC (|task datum|) (-other data-))
records other miscellaneous relations between a task and a formula

(DD-T |data pool| (-data-))
links data across reference points. The intent is to record that the presence of the data in the foreign data pool are responsible for the presence of the supportee (which may be a DD-T itself).

This information can be dumped out in a revealing form, as described in Chapter V.

II.E Handling Mistakes

Consider situations like the following:

You are dialing a telephone number. Halfway through, you feel your hand slip and you know you have misdialed.

There is a power failure. You wonder if the refrigerator will be damaged. You flick the kitchen light switch on to have a closer look. Nothing happens.

Someone asks you to design an amplifier with a certain high gain-bandwidth product. You confidently pick a familiar circuit topology and begin to compute the required component values. You discover there are no component values that will do the trick.

All of these are examples of "mistakes." (A finer classification is possible. Cf. (Nilsson, 1973).) They all have in common, in the terms I have been developing, that the plan for accomplishing a certain task has been shown not to work. In each case, it is wholly or partly useless to continue on the plotted course.

Not enough work has been done in AI on correcting such mistakes. (But see Nilsson, 1973, Philip Hayes, 1975, Sacerdoti, 1975.) Instead, we have spent a lot of effort on seemingly similar search problems in which "blind alleys" are searched, and real mistakes never occur. I discussed this briefly at the beginning of this chapter. The problem with even the most sophisticated of mechanisms for searching through blind alleys (Stallman and Sussman, 1976) is that they rely on the ability to restore previous choice points. Previous discussion of the problems associated with this (e.g., McDermott and Sussman, 1972) has focused on the difficulty in choosing a choice point to restore; here I wish to call attention to the impossibility of restoring most choice points in any useful way. The problem is that the range of choices previously available may be obsolete. Sometimes this is because some of the choices have been ruled out by other processes. This is handled nicely by Stallman and

Sussman's EL (1976). A worse problem is that non-monotonic inferences made at the time of the old choice may have been rendered incorrect by further discoveries or changes since the old choice. (McDermott, 1974a) For example, the range of choices available for instantiating an amplifier can change dramatically after adjacent stages are instantiated. There is no way to return to one choice point without considering all the choices and actions in between.

The alternative scheme I am about to outline has not been implemented, although many of the pieces are in place.

The idea is to treat correcting a mistake as a task like any other. The mistake is given a description by the primitive that failed. (For example, if a constraint cannot be satisfied, the mistake is described as [CONSTRAINT-COLLAPSE |losing constraint|].) The system sets itself the task

[/:GET-RID-OF |mistake description|].

Often it will be necessary to re-describe the situation; this is a job for the rephrasing protocol. A typical electronics-domain redescription might be

[IMPROVE '(GAIN (STAGE#89))].

Plans are retrieved to carry this out. (Cf. Chapter I.)

The difference between this and and a routine situation is that the task network must be corrected in some way. Some of the tasks that existed before the mistake are still "healthy," else there would be no reason to go on living, but some of the subtasks are now "rotten," and may be replaced. A subtask of a /:GET-RID-OF task is allowed to alter certain parts of the task network.

Making the network-editing machinery work is the hardest part of implementing this scheme. The kinds of edits that must be allowed include

> Adding new subtasks to correct the problem. The commonest reaction to

an accidental "protection violation" (Sussman, 1975) is to re-establish the protected fact without further fuss.

> Restarting old subtasks. For example, the string of tasks involved in dialing the first digits of a misdialed telephone number must be resurrected.

> Detaching and redescribing old subtasks. For example, introducing too much feedback can cause oscillation; its old description (that it did something useful) must be discarded, and it must be seen as part of the problem instead of part of the solution. Its old supertask must be marked un-/:REDUCED again, and a new way must be found to solve it.

> Terminating active subtasks, especially policies, of a rotten task. In electronics, constraints derived from circuit diagrams must be removed when an IMPROVE task is executed and changes the topology of the circuit diagram.

The information about what edits are legal must be part of the mistake handler. For example, the plans regarding constraint collapse (see Chapter III) must specify that the highest task that is the scope of some of the collapsed constraints is still healthy; some lower-level task (probably associated with a particular canned circuit diagram) must be declared rotten and its policies abandoned.

The reason why this scheme has not been implemented is that it depends on the data-dependency machinery I described, which is still relatively untested itself. Undoubtedly both of these systems will grow together.

II.F Programmer's Guide

As I said, NASL is not exactly a programming language, but it's not a natural language either, so it is probably best for the programmer to approach it first as the kind of formal language he understands best. To help with this, I include "programmer's manuals" in each of these three tough chapters.

NASL has two interpreters-- the theorem prover (STP) through which all NASL formulas must pass, and the plan interpreter (NASL proper) which takes some conclusions to be instructions to act. The first design decision in

expressing a new set of facts in NASL is whether to rely entirely on STP or to cast them as rules which create and manipulate tasks.

In principle, everything could be handled by the theorem prover. For example, axioms could be introduced defining a space of electronic circuits, and constructively proving

```
[EXISTS (X) (AND (ELECTRONIC-CIRCUIT ?X)
                 ([P] ?X))]
```

could replace the action [DESIGN |P|].

As we all know, however, all theorem provers of STP's class rely heavily on the generate-and-test problem-solving method. Generating all circuits is obviously ridiculous.

Here are some more general criteria for deciding whether to represent a body of facts as axioms or plans:

(1) As R. Moore (1975) has pointed out, it is a strong clue that a theorem prover is out of place when side effects enter naturally into the statement of a body of knowledge; this is certainly true for design. Any irreversible action, such as asking a question or wiring a circuit, rules out the use of a raw theorem prover.

(2) If you wish to take advantage of information relevant to a choice point, the choice must come up as the choice of a way to do a task or of the answer to a /:FIND. (You should verify that the information is worth the trouble.)

(3) If subgoals arise which must interact, you must put the goals in the data pool, i.e., make them tasks. Similarly, if you wish to manipulate goals as data structures, you must add rephrasing knowledge for tasks of that type.

Only if it appears that only brute-force deduction is necessary or feasible should you cast the knowledge as pure axioms. An example is the theory of frequency-picture manipulations developed in Chapter IV. Commonly a class of tasks will be associated with a "mini-theory" of some characteristic criterion for choosing between them; this little theory is expressed in terms of pure axioms. For example, the theory of ordering the selection of component values with respect to other tasks (Chapter III) is a small set of

axioms. (The merits of this "clever cogitation directed by brute-force retrieval" organization will be discussed in Chapter VI.)

II.F.1 Predicate-Calculus Techniques

Even after you have decided to represent a body of knowledge as a set of facts about tasks, these facts must be expressed as predicate-calculus implications. The approach to this that I have found useful is to think of them independently of their use first, concentrating on what they are to mean. Once this is done, the pragmatic content can be added. This approach forces you to think about what you really mean to express. For example, when you write an implication of the form [|P| ⊃ (/:TASK ...)], do you really intend that this task exist only while P is true?

There are three pragmatic decisions to make: whether to express implication as /:CONSEQ, /:ANTEC, or /:GEN; where to use packets; and which version (/:=, =, or =/>) of equality to use;

The first decision is often simple. Systems of predicate-calculus rules develop in such a way that one layer of rules "feeds" the next during forward and backward deduction. The rules usually work together to record in a forward fashion up to a point; then backward (consequent) rules work their way from deductive goals to the formulas recorded by forward rules. Generative ("-/> G") rules are useful in mixing these processes up. So, for instance, it is no use having an antecedent rule if no one records an expression matching its left-hand side. R. Moore (1975) has given some useful hints in deciding which way implications can be used.

/:PKT should be used instead of AND on the right-hand side of an /:ANTEC when much of the contents of the conjunction are not looked at for most

instantiations, or if it is not necessary that they trigger further /:ANTECs immediately. This is true, for example, of circuit diagrams, where information about the purposes of components is not always accessed; but not true of plan schemata, where all the tasks and subtask relations are going to be recorded anyway (and the interpreter must notice every task).

It is usually clear which version of equality to use. Goals are usually phrase in terms of "=," but if you know there is only one simple answer, use "/:=," which merely matches the two sides against each other. Simple "=" will work harder in the case where they don't match. Often "=/>" does not have to mentioned in the rules where it it is used; if rules like [=/> '(F A) B] are around, they will be applied when the right-hand sides of implications like

[-/> A (P ?X) (Q (F ?X))]

are detached with the variables bound. That is, recording [P A] will cause [Q B] to be recorded.

Finally, remember that it is not always enough to supply axioms about ***proving*** propositions with a certain predicate; if you ever wish to ***disprove*** such propositions, you must supply appropriate axioms. Often disproof information can be summarized with a single PRESUMABLY statement. For example, in the world of blocks, we might have

```
[-/> C (AND (ON ?X ?Y) (ABOVE ?Y ?Z)) (ABOVE ?X ?Z)]
[-/> C (ON ?X ?Y) (ABOVE ?X ?Y)]
[PRESUMABLY '(NOT (ABOVE ?X ?Y)) C]
```

The effort to prove [NOT (ABOVE A B)] will cause (*via* /:CONSISTENTLY) an effort to prove A *is* above B; if it fails, the conclusion is taken as true.

II.F.2 NASL Programming Techniques

In applying NASL to a new problem domain, one must supply model-manipulation statements to actually get things done, and indexed plan schemata to orchestrate them.

Tasks may be reduced in a forward or backward way. In the former, the presence of a task can trigger deductions of subtasks. For example, in the world of blocks, one could specify a plan to the clear the top of a block thus:

```
[-/> A (/:TASK ?N <> (λ () (CLEAR ?X))  <>)
       (-/> A (=/> '(/:TASK-STATUS ?N) ACTIVE))
              (FORALL (Y)
                 (-/> A (ON ?Y ?X)
                        (S '(EXISTS (T)
                               (/:TASK ?T <>
                                   (λ () (PUTON ?Y TABLE)   )
                                   <>)   )))   ))]
```

(Notice the use of "S" to indicate that these tasks are being triggered, not supported, by the statement [=/> '(/:TASK-STATUS |task|) ACTIVE].)

In backward reduction, plan schemata are instantiated *via* /:DO-SUBNET calls. This requires a couple of formulas. In the same blocks world, we might have the formulas

```
[/:TO-DO ?TSK (ACHIEVE '(ON ?X ?Y)) <>
    (/:DO-SUBNET (ACH-ON ?X ?Y) <>)]

[-/> A (/:PLAN-INSTANCE ?PI (ACH-ON ?X ?Y) ?SUPER-TASK)
       (AND (/:TASK (CLEARER-1 ?PI) <> (λ () (CLEAR ?X)   ) <>)
            (/:TASK (CLEARER-2 ?PI) <> (λ () (CLEAR ?Y)   ) <>)
            (/:TASK (PUTTER ?PI) <> (λ () (PUTON ?X ?Y)   ) <>)
            (/:SUCCESSOR (CLEARER-1 ?PI) (PUTTER ?PI))
            (/:SUCCESSOR (CLEARER-2 ?PI) (PUTTER ?PI)))]
```

The interpreter, when it has decided to reduce [ACHIEVE '(ON ...)] using the first rule, will create an instance of the schema [ACH-ON ...]; the second rule will then trigger the creation of several subtasks.

A corpus of NASL rules is often written as an incomplete set of plans and axioms, which is then debugged by adding "interaction terms," i.e., knowledge which influences the application of the first-order rules. This occurs through the medium of these kinds of rules:

> Rephrasing rules which redescribe actions, usually by breaking them into pieces and putting them back together.

> Choice rules which influence the way in which tasks are reduced.

> Rules specifying /:SUCCESSOR relations.

> Policies to watch for interactions between tasks or to influence choices.

We will see plenty of examples of NASL plans and rules in the following chapters.

III Design of Hierarchical Systems

Design is the production of an object to satisfy certain requirements. The requirements may describe the desired object closely ("A stick 10 inches long"), or they may be very remote from what is finally produced ("Something to make this room look more friendly.")

Of course, designing does not mean actually manufacturing an object; what is actually produced is a detailed description of one. In fact, design might be described as the process of adding detail to a description until "full detail" is reached relative to some basis.

In what follows, I will elaborate this theory, and then explain how it is implemented as a set of NASL rules. (A close relative of this theory was outlined by Freeman and Newell (1971) in a paper called "A Model for Functional Reasoning in Design.")

The best way to explain it is to start at the bottom, near the "basis." The basis for a design domain is a set of ***primitive artifacts***. For example, if sticks are primitive, designing a stick 10 inches long is merely a matter of "instantiating the stick primitive." "Instantiation" means creating a symbol, such as X043, and recording that it denotes a stick. That is not all, however. Associated with the primitive "stick" are attributes such as its length, width, material, color, etc., which must be fixed for a concrete instance of it. Because it is a primitive, we may assume that fixing a stick's qualities is merely a matter of choosing them. (Wooden sticks are cheaper than platinum, but I will not consider cost explicitly in this paper. I emphasize finding any solution to a design problem, not finding the best solution.)

So designing a stick is just a matter of picking a name, a width, and a

length. (Assuming brown wooden sticks from now on.) If the length is constrained to be 10 inches, that is clearly the length to pick. The width, if unconstrained, may be picked arbitrarily, subject to the reasonable constraint on all sticks that their width be no more than 10% of their length.

For a primitive artifact, then, "adding detail" is just selecting values for its "***control attributes***," such as length and width.

This theory of design will not account for the design of "something to make a room more friendly," mainly because "object that makes a room look friendly" is not a primitive artifact with a fixed set of attributes. In general, a requirement may be arbitrarily remote in structure from the kind of object that satisfies it.

So it is necesary to provide for for the ***indexing*** of partial solutions by their important features. That is, the theory must just provide for statements like,

> "Funny posters make a room more friendly."
> "Plants make a room more friendly."
> "If *x* makes a room more friendly, and *y* (distinct from *x*) makes a room more friendly, usually the combination of *x* and *y* makes a room more friendly."
> ***etc.***

A ***partial solution*** of this kind may be a primitive artifact, in which case the problem has been solved, but more generally it consists of a structure of design subproblems. These subproblems must be solved in much the same way as the original problem, and the solutions must be connected up. Eventually the original problem will have been completely reduced to primitives. (Fig. III.1)

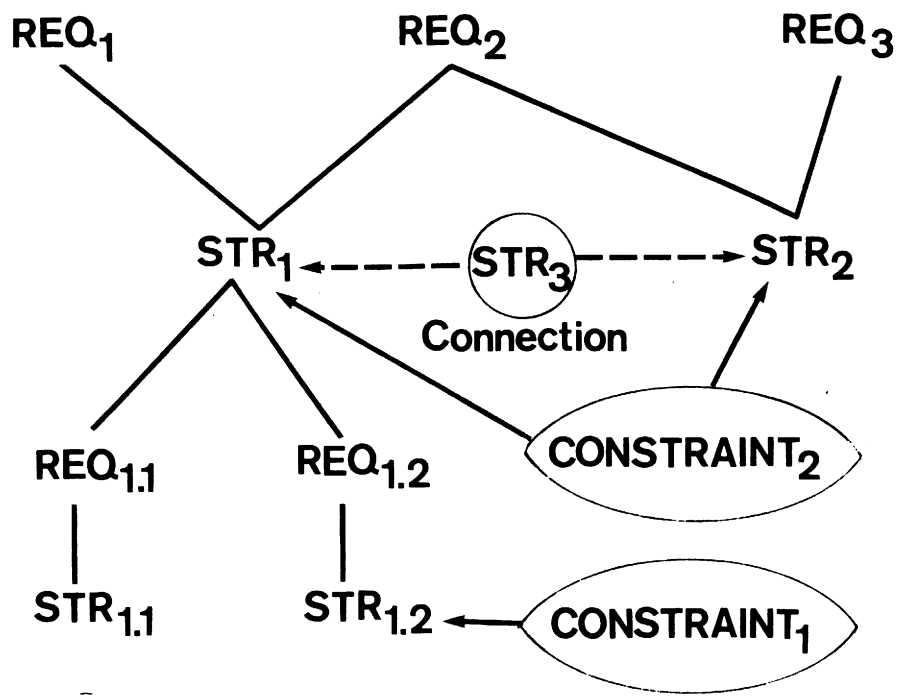

Figure III.1 Function-Structure Graph

These primitives will be ***connected*** and ***constrained***. Some of these constraints come from the problem (e.g., "Amplifier with gain = 10"), some from the partial solution ("A common-emitter's gain is ***beta*** X R_L/R_S"), some from connections ("The current from R_L is the current into the collector"), and some from descriptions of primitives ("The resistance must be positive"). As with the simple stick problem, the control attributes of the primitives must all be selected subject to the constraints.

The design process suggested by Fig. III.1 neglects several complications having to do with pragmatic knowledge of partial solutions. Some of these will be easier to talk about after I introduce the NASL implementation of this design theory. Before I do that, I should say more about the "indexing" of partial solutions.

If a design requirement is very simple, it is is plausible to imagine it as calling to mind a partial solution tagged with "specs" which match the requirement. For example, the design problem "Make a common-emitter amplifier" could plausibly match the specs on the common-emitter circuit exactly.

For more complicated problems, this will not work. The description might contain conjunctions, disjunctions, or quantifiers. It might consist of simple pieces whose solutions can be composed. It may be cluttered with numbers which have to be described more suggestively, as in the example of Chapter I, in which "gain = 10" was replaced by "moderate gain." Finally, the description might just be in the wrong terms; a common example in electronics is the translation between time-domain and frequency-domain descriptions of signals.

So the theory must provide for manipulation of problem descriptions, before the first partial solution can be proposed. This manipulation is aimed at transforming a description into a form suitable for retrieving stored partial solutions.

A version of this theory has been encoded in NASL. As coded, it is independent of electronics, although enough restrictions have been placed on it to keep me from claiming it is a complete general design theory. It is meant to be a theory of engineering design, for which, to first order, all effects can be thought of as local interactions among connected modules, each of which is designed to accomplish some part of an overall objective. It is biased toward systems whose interactions can be described numerically. I will call this domain "design of hierarchical systems."

It is straightforward to express in NASL most of the concepts I have outlined. The first step is to implement the notions of "requirement" and

"structure fulfilling it" as tasks and subtasks. That is, a design problem is expressed as a task, and the terminal nodes of the function-structure graph are to be identified with primitive tasks of the form "grab a (primitive) component." For example, a first-pass analysis of an electronics problem may generate this structure:

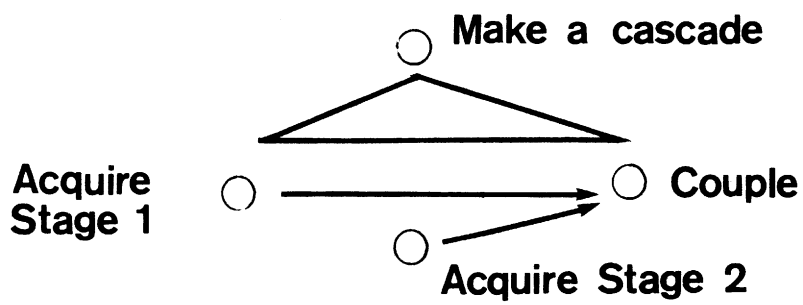

Figure III.2 A Two-Stage Cascade

Later elaboration will instantiate the coupling task:

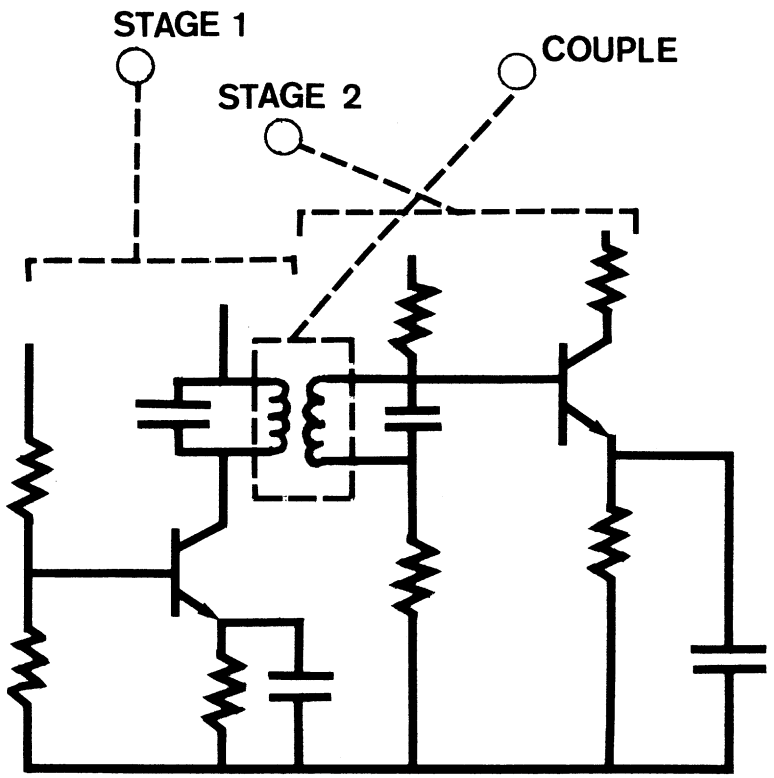

Figure III.3 An LC-coupled Amplifier

"Partial solutions" are implemented as a kind of plan schema. A particularly important kind of partial solution is a ***device type***, a packet of facts clustered around a concept like "amplifier," or "operational amplifier," or "resistor." Some of these facts describe the structure of the devices of the given type, but many of them are concerned with how such devices are used in solving larger design problems. This last set of facts defines a set of tasks for elaborating a device and connecting it to its peers.

Primitive devices are those with no internal structure, whose elaboration consists mainly of selecting values for their control attributes. The system represents these obligations as a set of "SELECT-VALUE" tasks. The constraints that accumulate during a design are implemented as policies which influence the execution of SELECT-VALUE tasks.

Because we are using the NASL interpreter, all design subproblems are represented explicitly in the data base as tasks. Partial solution plans are recovered, as for all tasks, by using STP to retrieve them. Choice rules are used to choose among or compose sets of partial solutions. Simultaneous subproblems are represented by simultaneously active tasks. There are frequent cases where it is important to start on one problem before another, because the solution to the first will influence the choice of approach to the other. This can be arranged by writing rules to cause the deduction of /:SUCCESSOR formulas.

The manipulation of requirement descriptions when routine indexing fails to retrieve a partial solution is handled by a special design rephrasing plan. It says to turn a recalcitrant design task into a task network of the following kind (cf. Fig. III.8): make a device of a known type, and constrain it. The plan is to do this by tearing the given problem into pieces called "shards" (usually conjuncts from the design requirement), each of which is classified as specifying either the device type or a constraint. The plan succeeds only if every shard is accounted for in one of these ways. It is generally the responsibility of rules from domain-dependent plans to make sure this is true. In the electronics domain, as we shall see, there are many rules for manipulating shards, ranging from those which convert shards regarding gain into control-quantity constraints, to those which change signal-conversion shards from the time domain to the frequency domain.

This is a broad outline of the design theory encoded in the formal theory of DESI. (Appendix 2) A point to notice in its exposition is that I appealed to innate control concepts to explain notions of structure, purpose, and constraint. It will be seen that appeals like this, appropriately formalized, are the only kinds of knowledge of these concepts that DESI has. In a

primitive way, the program exhibits "machinomorphism," the inclination to understand other systems in terms of its own kinds of motives. This allows a certain computational economy, and makes assimilation of new information more reliable by enforcing a small vocabulary. A complicated and delicate (or electronics-dependent) theory of purpose and commitment does not have to be added by the user.

Before turning to a detailed exposition of the DESI implementation, I should mention three issues I will have little to say about: learning, search, and creativity. The last of these may seem the most important. Many people would probably be skeptical about the ability of a machine to do design, because creativity seems to be absent from machines and vital to design. Indeed, "design" and "creativity" seem almost to be defined in terms of each other. If this issue bothers you, let me call your attention to the distinction between "routine" and "imaginative" design. Routine design is the production of an artifact in a field (such as electronics) such that anyone else with an ordinary mastery of the field could have produced the same thing. This kind of design is the only kind I can claim to have a theory of.

DESI does not learn anything from doing a design. Although at the beginning of this chapter I described designing as adding more detail to a description, the description at the end of a design is not represented the same way as the problem description. The problem description is essentially a λ-expression, but the final result is a set of statements in the data base about "X043," or whatever symbol was chosen to represent the target device. To learn, DESI would have to gather these statements together into a new plan, and index it under a useful generalization of the problem. Doing this is difficult. (Cf. Sussman, 1975)

Other kinds of learning are also possible. One can imagine a program

learning how to order certain kinds of subproblems, or how to choose and compose partial solution. These are examples of "trial and error" learning; to attack them requires a theory of search.

DESI, like all NASL systems, tries never to make a choice at random, and never backs up to undo a choice. (See the discussions in Chapter II.) Thus, it can be said not to search at all. This is the right organization, but it needs to be combined with a learning system that proposes new choice principles by watching what happens after it does make an arbitrary choice. For example, if one amplifier circuit is chosen from several that satisfy the known choice principles, and later its impedance is discovered to be too high, the system should not back up, but should make up a new choice principle to rule that circuit out in case high impedance is required.

The system currently does none of these things. If its rules get it into trouble, it will look for a correction plan that fits the situation, but will do the same thing all over again if the next problem is similar. This is a serious, but (I hope) temporaray, defect in the theory, since it seems clear that people learn something new in the course of all but the most routine design tasks.

As I describe in detail DESI's design theory, I will point to the more formal exposition of Appendix 2. By looking there, you will be able to judge the power of the NASL control language. It will be seen exactly how often it is directed and flexible, and how often clumsy, arbitrary, or inextensible. The important point at issue during this otherwise tedious exercise is one's ability to represent various specialized control and debugging strategies using the framework of tasks, data-dependencies, and conflicts described in Chapter II. In what follows, references to the formulas of Appendix 2 are indicated thus: <*formula-name*>.

III.A The Representation of Knowledge about Devices

Much of design is the manipulation of ***devices.*** A device is any manipulable, "physical" object in a design domain. (Thus, signals will not be devices, but nodes will be.) Familiar classes of devices that are useful are called ***device-types.*** These classes may be formed in several ways. (Sect. III.A.1) Each device in a class is described by a set of formulas arranged in certain standard ways. (Sect III.A.2) A set of formulas describing a device type is instantiated to form a particular device's description. Knowledge about a device type is therefore conveniently represented as a "***packet***" (McDermott, 1975) of facts which is instantiated when a particular example is considered. This packet is called a ***device schema.*** (Unfortunately, Brown and Sussman (1974, A. Brown, 1975) have used the term "plan" for this purpose. This conflicts with the usual range of meanings of this term in AI. I have used this term in the more traditional meaning already in discussing the interpreter.)

III.A.1 Hierarchies of Device Types

Device types which have a recognizable function and circuit diagram (or symbol) are called ***basic.*** Basic device types may be lumped into loose classes called ***superordinate device types.*** <*DEVICE-CLASSES> (This terminology is borrowed from (Bobrow and Winograd, 1976), but I am not sure I mean the same thing by it that they do.) Sometimes such a higher class exists just because people have a name for it and use it to specify problems. An example is "amplifier." Sometimes there is some class of facts it is convenient to store together, as for "2-terminals." (See Chapter IV.)

Kinds of Device Type

Basic
- Primitive (e.g., resistor)
- Composite (e.g., common-emitter amplifier)
  - General
  - Specialized
- Ideal (e.g., current source)

Superordinate (e.g., amplifier, 2-terminal)

Figure III.4 A Hierarchy of Types of Device Types

Basic device types may be further classified <*BASIC-DEVICE-CLASSES> as primitive, composite, and ideal. Primitive devices are the terminals of a complete function-structure graph. (See below.) Ideal devices such as current and voltage sources behave as primitive devices, but must be "implemented."

Canned devices that are made up of simpler components are called ***composite device types.*** Textbook diagrams of things like Hartley oscillators and common-emitter amplifiers may be taken as standard examples of composite device types. Often these textbook diagrams leave implicit what I take as an important feature, that they exist in ***general*** and ***specialized*** versions. (Fig. III.4) <*GENERAL-DEFN> The general common-emitter amplifier, for instance, is just a transconductance treated in a certain way, whereas the "typical common-emitter" has biasing resistors hung all over it. (Cf. Watson, 1970) This is expressed as

[DERIVED TYPICAL-CE GENERAL-CE].

The importance of this relation will be brought out shortly.

Thus a particular device will be at the bottom of a hierarchy of general and superordinate devices. (Fig. III.5)

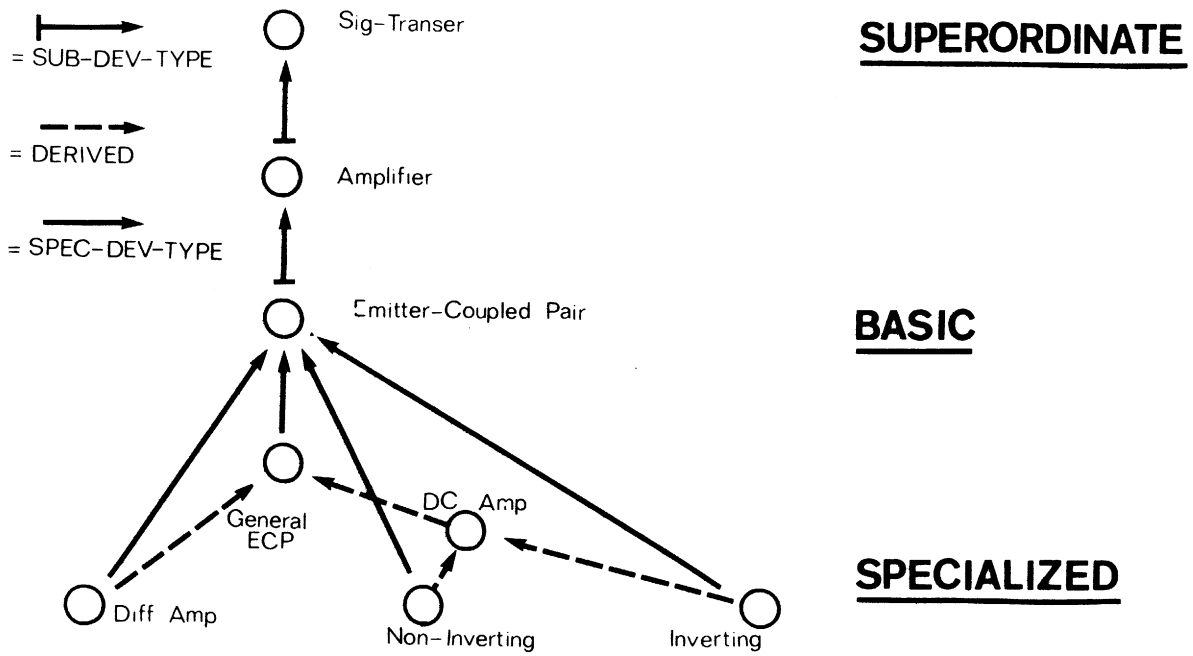

Figure III.5 Devices in The Type Hierarchy

The relation between the devices above the BASIC level in Fig. III.5 is [SUB-DEV-TYPE |dev type| |superordinate dev type|]. Below that level, the relation is [SPEC-DEV-TYPE |specialized dev type| |dev type|]. Thus we would write [SUB-DEV-TYPE COMMON-EMITTER AMPLIFIER] and [SPEC-DEV-TYPE TYPICAL-CE COMMON-EMITTER]. (The DERIVED relation will be explained below, Sect. III.A.2.)

A device will be of several device types, written [DEV-TYPE |dev| |type|]. There is usually one, its MAIN-DEV-TYPE, which is the most specific category it is known to fall in.

III.A.2 The Representation of Device Diagrams

A device is either primitive or composite, depending on its main device type. A device is specified with several kinds of information (most of them are not necessary for primitive and ideal devices):

(1) The ***components*** of devices of that type. This is kept in formulas of the form

[COMPONENTS |device| < -component names- >]

Each component is itself a device, whose main device type is expressed by a separate formula. For example, for a voltage divider VD#21 we might have

[COMPONENTS VD#21 <(R1 VD#21) (R2 VD#21)>]
[MAIN-DEV-TYPE (R1 VD#21) RESISTOR]
[MAIN-DEV-TYPE (R2 VD#21) RESISTOR]

(2) ***Connections*** and ***constraints*** between components. There can be no domain-independent notion of connection between physical objects, since any physical medium can be exploited. The only completely general thing that can be said is that connecting devices "constrains" them in some way. (Otherwise, there would be no point in connecting them. Cf. "CONSTRAINT2" in Fig. III.1.) As we shall see below, there is a rich theory of constraints built into DESI.

(3) ***Control quantities.*** These are numerical-valued attributes of the device that the designer has complete or partial control over. They are represented by formulas like

[CONTROL |attribute| |device| |range| |degree of control|].

This declares attribute to be a ***control attribute***; it means that the quantity [|attribute| |device|] may be assigned any value from the set of numbers range. Since real components often vary from their nominal values, the formula specifies the degree of control of the attribute, which is the quotient of the highest and lowest possible true values compatible with the selected value that appears in the data base. This value is actually the (geometric) mean of highest and lowest values. These uncertainties will be taken into account in reconciling constraints. As an example, in electronics for a transistor Q#173 we might have

[CONTROL BETA Q#173 (INTERVAL 10 500) 10],

since the beta of a transistor is controllable only to within an order of magnitude; while

[CONTROL POLARITY Q#173 <+1 -1> 1],

since every transistor is unambiguously PNP or NPN.

A distinction must be made between ***immediate*** control quantities and ***derived*** control quantities, corresponding roughly to attributes of primitive and composite devices. There are several relevant formula types for expressing information about these matters:

(a) [IMMEDIATE-CQ '|control quantity|]
Example: [IMMEDIATE-CQ '(RESISTANCE R#21)]

(b) [DERIVED-CQ '|control quantity|]
Example: [DERIVED-CQ '(V-GAIN AMP#34)] The actual function relating the V-GAIN of the amplifier to the values of its components' control quantities can be derived from constraints found in the description of AMP#34. Often these will be found in the device schema of which AMP#34 is an instance.

(c) [GENERIC-CA |attribute|]
Example: [GENERIC-CA THEV-R] It would be tedious and wasteful to derive a formula for each device or device schema relating its Thevenin resistance to its components' values. (A change of topology, for instance, would force a recomputation.) Instead, some control attributes can be declared *generic*, meaning the system knows how to compute them and will when they are needed. (The current system can handle this to the point of enqueueing a CALCULATE task, but the computational techniques required have not been implemented.)

(4) *Task information.* Every device has a cloud of tasks floating around it. These will be of various sorts:

> The purposes of a device and its components are represented by a set of finished tasks associated with acquiring them.

> Many devices will not function as they are supposed to without further work ("subrequirements" in Fig. III.1); these active tasks are called ***expansion obligations.*** These ride along on most composite devices and even some primitive ones; a transistor, for example, must be biased into its intended mode.

> A device carries along monitors on the topology of the connections inside it and from it to other devices; some of these monitors are for protection of important relationships, and some just to notice when something must be recomputed.

These are characteristics of devices. A ***device schema*** is merely a canned set of such formulas with a free variable to be bound to a particular device when it is made. Device schemata are used to represent device types. They are usually implemented as "packets." (McDermott, 1975) For example, the packet for "voltage divider" will include the formula

[COMPONENTS ?##VD <(R1 ?##VD) (R2 ?##VD)>]

from which the formula given above for VD#21 will be derived. The prefix "?##" specifies that ?##VD is a variable loosely bound to an "abstract voltage divider." The formula says, "the typical VD ?X has components [R1 ?X] and [R2

?X]."]

The tasks that will be liberated when a device schema is instantiated are called ***frozen tasks***, and the liberation process is called "thawing." Frozen tasks may be thought of as mummified remnants of actions that were (conceptually) executed when the device schema was first put together. (Device schemata are to be thought of as the result of previous designing activity followed by summarizing what was learned, but this is just to help your imagination; such a learning scheme doesn't exist yet.) One thing that must be left around in a schema is a record of why the various components were acquired and connected as they are now found; in other words, the ***purposes*** of the components. (If for no other reason, these are necessary in case they have to be undone during mistake correction.)

The simplest way to accomplish this is to keep the tasks that were known at the end of the (imagined) design episode in a frozen state. Some of these will have been FINISHED; for example, the tasks that acquired the components are there just to record why they were acquired. Others are still active. For example, there will be ACTIVE constraints and protected model manipulations.

By way of illustration, a voltage divider may be first thought of as a way of setting a bias voltage in a particular amplifier design problem. A voltage divider found in a schema must be associated with a policy of keeping that bias voltage set.

An advantage of the frozen policy solution compared to a more specialized implementation of purpose comments is that one mechanism is used to handle local cooperation and conflict of tasks as well as interactions of new actions with old purposes. This is an example of the "machinomorphism" I mentioned at the beginning of this chapter.

Specialized device types are arranged in a hierarchy according to the DERIVED relation. This is a more complex relation than SUB-DEV-TYPE and SPEC-DEV-TYPE (cf. Fig. III.5), which are used mainly to cause properties of higher types to be inherited by lower. <*SUB-DEV-TYPE-1, SPEC-DEV-TYPE-1> Most of the properties of a general circuit, such as its topology and components, are not to be inherited by its specializations. However, there is an important class of properties which must be accessible from the specialization: the frozen tasks of its more general counterpart. The relation between a general circuit and its specialization is precisely that the expansion obligations of the general circuit are fulfilled by the structure of the specialization.

To represent this relation, we need some more equipment. Every device thawed from a schema has a "deep freeze" of frozen tasks, which are collected for convenience as the subtasks of an abstract task called the [DEEP-FREEZE |device|]. If dev-type 1 is derived from dev-type 2, then a device of type 1 will have a "SOUL" which is a device of type 2. <*SOUL-ON-ICE> The important relation between them is that every subtask of the soul's deep-freeze is to be a subtask of the original device's deep-freeze. This inheritance is done *via* /:CONSEQ deduction, since it is not important to see every frozen task during normal operation; most of them will have been reduced anyway. They are mainly valuable in explaining the purposes of components.

For many examples of device schemata, see Appendix 3.

III.B Design Actions and Plans

I can go no further in talking about devices without talking about design actions. This is because devices' purposes are so intimately associated with the purposes of their designer, in this case DESI; and DESI's purposes are expressed as tasks.

Design actions fall naturally into these classes (see Fig. III.6):

(1) "Design something with property *p*": Starting with no structure or hint of it, one is to produce such a thing.

(2) "Make an *x*": Here *x* is a device type, an example of which is to be created. This kind of action breaks down into subtypes, depending on what kind of device type *x* is. Making a basic type tends to be a matter of choosing which version along the specialization scale to use, then plugging in its frozen tasks. Making a superordinate type requires more involved and careful choice, since the sub-types to choose from usually have incompatible properties.

(3) "Constrain something": Things that can be constrained are not devices, but ***quantities***. There are two classes: ***physical quantities*** such as voltages and currents; and the ***control quantities***, such as resistances and power gains, that I described above.

(4) "Change a device": Given a structure, it can be altered in various ways. These actions include fixing a physical quantity, biasing a circuit, adding feedback to improve stability, and coupling two stages. The actions are defined by plan schemata that often come in specialization hierarchies like those of devices. A major subdivision of these are actions which involve changing the previously reigning plan network as well, for example, altering a control quantity which is already fixed by constraints.

This list is derived by common sense, and from perusal of ***100 Ideas for Design*** (Electronic Design, 1964), among other works. (Senturia and Wedlock, 1975)

Design Actions

1 DESIGN

2 MAKE
- superordinate
- primitive
- ideal
- general
- specialized

3 Constrain
- CONSTRAIN
- SELECT-VALUE

4 Change
- FIX quantity
- BIAS
- COUPLE
- Patch
  - IMPROVE gain, input-Z, selectivity,...

Figure III.6 Design Action Taxonomy

The design problems which appear in books such as these include

"Design a power amplifier..." (Type 2)
"Increase the current..." (Type 4)
"Isolate two connected devices" (Type 4)
"Make the quiescent output voltage 40V" (Type 3)
"Design a circuit with a high gain-bandwidth product" (Type 1)
"Avoid loading" (Type 3)

(There are other kinds of actions, such as "simplify a circuit," which I have not counted as among these types. I hope they can be added, but I have no plans to do so.)

III.B.1 DESIGN

There is only one action in this class.

[DESIGN |prop|] ==> [<|name|>]

This action usually arises at the top level. Successful execution of such an action creates a model of a device that has property prop. In easy cases,

this reduces quickly to an action of Type <DESIGN>

In the hard cases, processing turns upon a large body of domain-dependent rephrasing knowledge, directed by the rephrasing plan DESI-REPHRASE-PLAN. <«+DESI-1, +DESI-2> This plan network may be graphically expressed as follows:

Figure III.5 Design Action Taxonomy

The design problems which appear in books such as those include

"Design a power amplifier..." (Type 2)
"Increase the current..." (Type 4)
"Isolate two connected devices" (Type 4)
"Make the quiescent output voltage 48V" (Type 3)
"Design a circuit with a high gain-bandwidth product" (Type 1)
"Avoid loading" (Type 3)

(There are other kinds of actions, such as "simplify a circuit," which I have not counted as among these types. I hope they can be added, but I have no plans to do so.)

III.8.1 DESIGN

There is only one action in this class.

(DESIGN [prop]) => (<plan>)

This action usually arises at the top level. Successful execution of such an action creates a model of a device that has property prop. In easy cases,

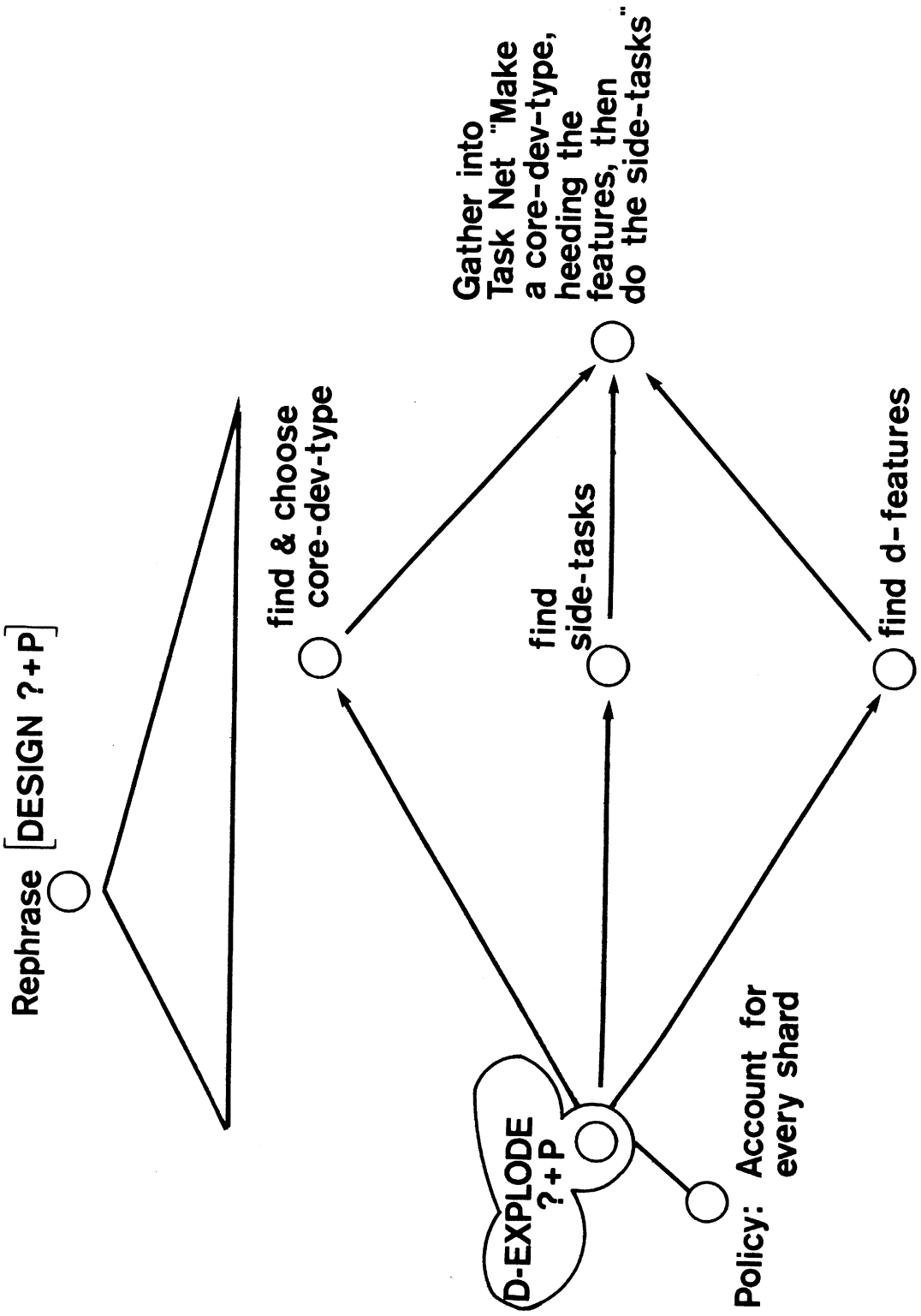

Figure III.7 Design Rephrasing Plan Schema

This is a plan to manipulate the design problem as an object. The plan network is set up using a formula <*+DESI-1> which extracts the desired

predicate as the value of ?+P. In this context, the embedded formulas, prefixed with the character "_", are being used essentially as SNOBOL patterns (Farber *et. al.*, 1964) to tear the goal to be rephrased into manageable pieces. So ?+P will have value [[|prop|]]. (See Appendix 1.)

Remember that the final aim of a rephrasing task is a revealing reduction of its target task. A detailed analysis of the problem may be postponed; the important thing in rephrasing is to make it "familiar." The goal in the case of the design rephrasing plan is to reduce the design problem to the following net:

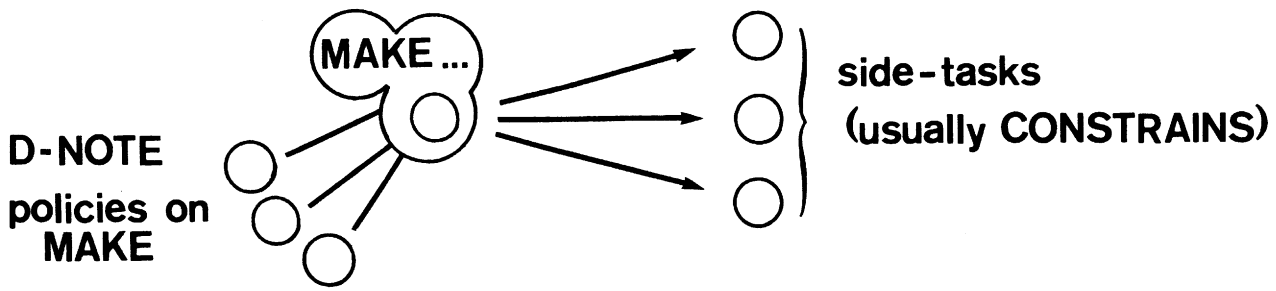

Figure III.8 Rephrased Design

The strategy of the designer is to "explode" the given predicate into "d-shards," which are conjuncts of the original predicate. Discovering d-shards is occasionally straightforward <*D-SHARD>, but usually depends upon formulas for the domain involved. (See Chapter IV.)

The d-shards are valuable only insofar as they lead to one of three things:

(1) A *core-device-type* the MAKing of which is the simple first step of the rephrased design plan (Fig. III.8);
(2) *Side-tasks*, typically to enforce numerical constraints discovered as d-shards;
(3) *D-features*, qualitative predicates used as policies in making a core-device-type.

This fact is expressed as the policy task ACCOUNT-FOR-ALL, which says to

make sure that every d-shard leads to one of these three things. In the current implementation, it is an error if a miscreant shard is discovered. A more sophisticated implementation would know how to try harder and attempt to learn from its efforts.

The only other feature of interest in the general design rephrasing plan is the step CORE-FINDER, during which NASL must find the core device-type to be used. The core device type is often clear from a d-shard of the form [[λ (|v|) (DEV-TYPE ?|v| |dev-type|)]] <*CORE-DT-1>. However, rules from particular domains can and do suggest device-types based on more elaborate d-shard processing. The interpreter must choose one. It is the responsibility of the writer of this knowledge to provide choice rules to get out of this situation. However, there is one rule <*CORE-DT-CHOOSE> which is domain-independent: if one device type is subordinate to another, reject it. (It should be suggested later anyway by the policies that grow out of d-features.)

This rephrasing method may be compared with the proposal of Moore and Newell (1974) for the MERLIN program. The idea there was to be able to "view" any conceptual structure as another by a process of mapping the pieces of one into the pieces of the other. DESI tries to view any design problem as "making a ..., while noticing hints +++, then doing ---"; this template may be seen as a three-slotted structure, such that every piece ("d-shard") of a design problem goes into one of these slots. The process is more structured than MERLIN; in particular, this kind of rephrasing is not an operation which can always be done by definition; it is capable of failing. The analogy may, however, be revealing. (It was suggested by Marvin Minsky.) I suspect that many rephrasing problems can be put in this paradigm form, and that the rephrasing protocol can be made more specific. However, currently DESI does things with rephrasing which cannot be seen as an instance of this paradigm.

(An example is equation solving.)

III.B.2 Making Things

(1) [MAKE |device type|] ==> [<|name|>]

The intent of this action is to create ("buy") a device of the indicated type. If the device type is basic <*MAKE-BASIC-PLAN>, a task net is set up with a primitive GRABBA action, which will just generate a new symbol and make its main-dev-type be the basic type. Extra tasks are hung on the network, depending on whether the device is primitive <*MAKE-PRIM>, composite <*MAKE-COMPOSITE>, or ideal <*MAKE-IDEAL>. In the case of primitive devices, the only commitment enqueued is to select the values of its control quantities. In the case of composite devices, the plan subnetwork includes a subtask to expand the device at some time in the future. Ideal devices receive a commitment to be implemented. (These new tasks are not marked /:MAIN in their task networks, so they do not have to be finished before the successors of their supertasks are begun; hence, they amount to future commitments.)

Expansion of a composite circuit means wiring up a circuit diagram for it. Usually this just means selecting a specialized device type and declaring the circuit to be that type; this is called "specializing" the circuit. <*SPECIALIZE-DEFN> The system has a choice of circuit diagrams from the specialization hierarchy. If one circuit is DERIVED from another (and hence is "specialized"; see Figs. III.4 and III.5), it will do the same task, but may depend on more specialized assumptions. It is the user's job to write rules that suggest circuit versions to match requirements, but the system knows about two peculiar specializations of a circuit: its "most general" specialization and its default specialization. <*MOST-GENERAL-DEFN, DEFAULT-

SPEC-DEFN> If either of these is available, it will be suggested. Generally, only one specialized device type of some basic type will come up in a given context. The user must make sure that good choice rules are available when more than one appears. The trade-offs should be clear: a general schema involves more work on expansion-obligations, but using a specialized version runs the risk of having to correct the circuit topology when some assumption proves unjustified.

If the user's rules do not sufficiently disambiguate, the system uses the two rules <*SPEC-DEV-BETTER, TWO-SPEC-DEVS-WORSE-THAN-ONE>. The first encourages the use of a more specialized device type if it has been suggested; the second overrides this one by ruling out conflicting suggested specializations.

Basic device types are just canned diagrams; the choice is which version of essentially the same circuit to take. That is why these special cases can be distinguished. In choosing among superordinate devices, rules for zeroing in on basic sub-types are entirely up to the user.

(2) [ACQUIRE |device type|] ==> [<|name|>]

MAKE a device type, unless there is already one around which can be used. <*ACQUIRE-DO-1, ACQUIRE-DO-2> For example, you should always re-use old voltage sources instead of making a new one; you should never re-use a transistor; and you should look around to see if you can bum a node before making a new one. (This last information is purely domain-dependent. See Appendix 3.)

(3) [EXPAND |device|]

This action is required for devices which are not fully specified by their circuit diagrams. (See below, Sect. III.A.2.) This task doesn't require elaboration, but accumulates subtasks by deduction. For example, it picks up ***expansion obligations*** from composite device schemata. These are actions which become subtasks of the task of EXPANDing each instance of the device. <*EXPANSION-OBLS-DO> Other tasks that are created involve finding all GENERIC-CA's of the device that have been constrained and deriving the formulas which define them. <*GENERIC-CAS-DO> (See Sect. IV.B.4.)

(4) [CONFIG < -types- >
(λ (-vars-) < -actions- >)]

This is a "macro-action" which is an abbreviation for "ACQUIRE the types, then bind the vars to the resulting devices to generate a list of actions to perform." <*CONFIG-DEFN> The LISP program SET-UP-CONFIG which elaborates it is not shown in Appendix 2. CONFIG is used as an abbreviation in Appendix 3.

III.B.3 Constraints

(1) [CONSTRAIN < -quantities- > |pred|]

Executing this action commits the interpreter to making pred hold true of the various physical quantities and control quantities. Thus, it is naturally a policy. CONSTRAIN is a peculiar mixture of ACHIEVE and ASSUME. If a quantity is under control, you are permitted to ASSUME it has any value that doesn't contradict what is already known about it. So, when CONSTRAINing, it is often permissible to record any equalities deducible immediately from its constraint plus other constraints and equalities to be found in the data base.

(Cf. (Sussman and Stallman, 1975).) If these constraints and equalities contradict the new constraint, the action fails. (See Sect. III.B.)

In detail <*CONSTRAIN-DO>, this is how CONSTRAINing is done: if the number of unknowns is exactly one, and the main connective of the constraining predicate is "=," then the system is to try to solve the equation immediately. <*CONSTRAINT-RESOLVE-DO> If it is solvable, the result is to be ***protected***. (See below.) In either case, the CONSTRAIN remains an established policy (a "constraint").

Algebraic Symbol Manipulation

Several times in discussions of constraint and equality manipulation I have assumed some sophisticated symbol-manipulation ability by the program. The various tasks that I take for granted include solving equations, choosing values to satisfy rather arbitrary constraints (including inequalities), and finding the precise way that a one control quantity depends on the variation in another (see Sect. III.B). In the long run, it would be enlightening to see whether the structure of the NASL system is sufficiently flexible for this information to be encoded as a set of NASL formulas. Preliminary indications are that this is quite feasible; very simple equations are already solved by the tasks generated by <*EQN-SOLVE-DO>. (Other equations might be handled by the methods of (Bundy, 1975).) My judgment has been that to explore this byway in more depth would bog me down. This decision is not obvious; after all, flexibility of application is one of the main design goals of my system. However, having an implementation of one domain is, for now, much preferable to having curious fragments of several. Therefore, I depend upon calls to an expert symbol-manipulation system to do this work for DESI. I might have used some symbol-manipulation system like MACSYMA (Mathlab, 1974), but, for simplicity, I chose myself as this expert subsystem. Whenever a non-trivial symbol-manipulation problem needs solving, the system types out a request for a solution and waits for its human interlocutor to supply it.

It is to be hoped that this is not a permanent trend in computer science, since it tends to reverse the usual practice of having humans do the creative work while machines do the tedious chores.

(2) [SELECT-VALUE |control attribute| |primitive device|]

This action is to be postponed until all tasks of Types 1, 2 and 4 are finished. <*SELECT-POSTPONE> DESI makes all SELECT-VALUEs subtasks of a task SELECT-EM-ALL which is a successor of every "topology-changing task." It is assumed that this kind of task is recognizable from its action function. (MAKE and FIX (-quantity) are the only built-in topology-changers.)

When the system gets to a SELECT-VALUE task, either the control attribute for this device already has a value; or DESI must pick a value within the range of the control attribute which fits all the known constraints on it <*SELECT-VALUE-DO>, and impose model effect

[=/> '(|control attribute| |primitive device|) |value|].

It then PROTECTs the fact that the value satisfies the constraints.

If there is no such value, the system is faced with a "constraint collapse" (see Sect. III.B); that is, it has made a mistake.

Executing SELECT-VALUE tasks causes the resolution of all remaining constraints of a network.

(3) [PROTECT '|proposition|]

The intent of this action is that the system should become alarmed, i.e., realize it has made a mistake, when the fact is no longer true in the model. This is not as easy as it sounds or as it is usually implemented (Sussman, 1975), because the given fact may be only indirectly related to atomic facts.

Because data dependencies are maintained by the system, it might be possible in principle to make this a built-in action. That is, the system could just wait for propagating erasures to wipe out a fact. Something special would have to be done for the results of non-monotonic inference (that is, using /:CONSISTENTLY), because in this case it is *recording* a fact that

could upset the protected fact.

For this reason (and for the weak introspective reason that protection does not seem to be a fool-proof operation), I have DESI treat protection as a problematic action to be reduced to different subtasks in different cases. This decision is under review.

In the design domain, we have need primarily of protecting values which are derived from consideration of constraints. This is done <*QVAL-PROTECT> by /:MONITORing the fact that the quantity has a value and /:CONTINUing the protection policy when the value is removed. <*PROTECT-CONTINUE>

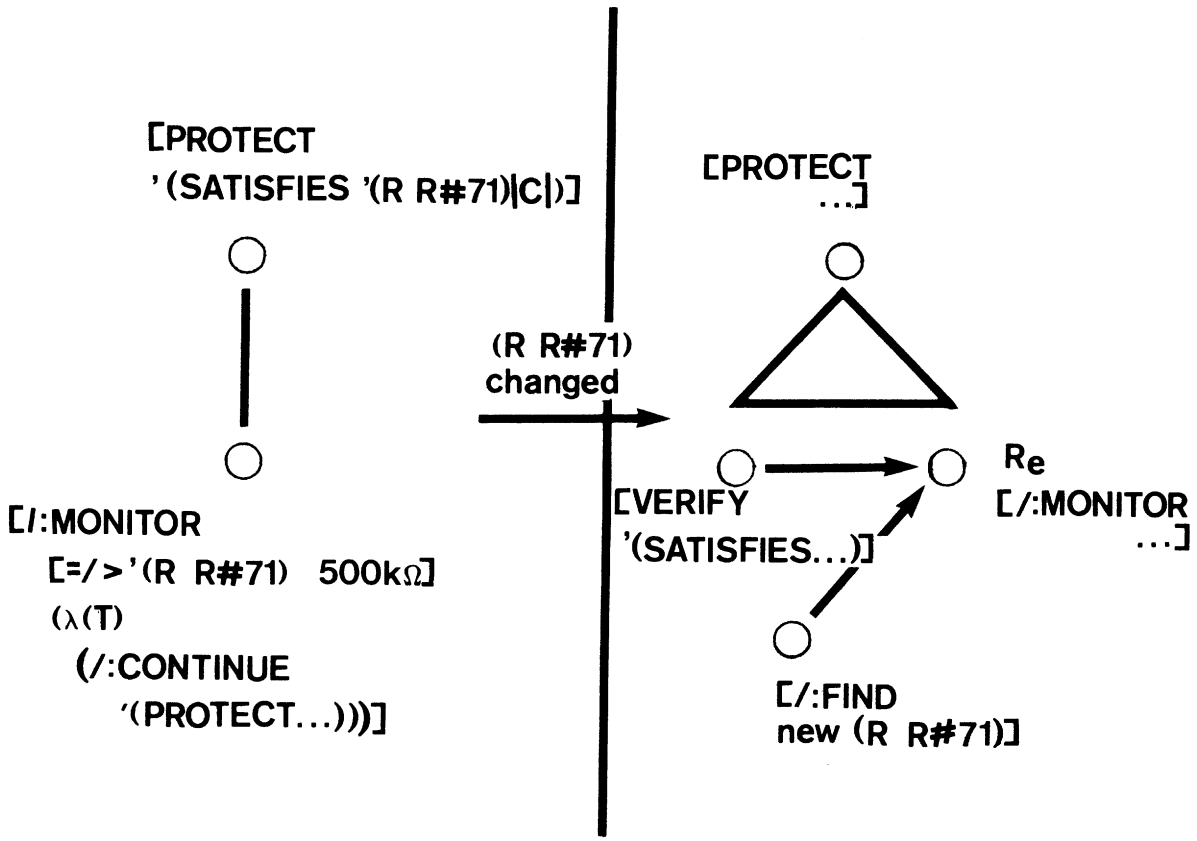

Figure III.9 Quantity-Value Protection Plan Schema

Notice that the variable may be getting a new value which satisfies the constraints, so the system cannot jump to the conclusion that a protection

violation has occurred. The decision to continue the policy is generally postponed. Upon continuing it, if a violation *has* occurred, DESI realizes its mistake.

III.B.4 Changing Devices

This is a catch-all category which includes biasing, feedback, coupling, and fixing the value of a physical quantity. These actions are described in the next chapter. Other domains would have other actions.

The last action in this list is the only one I feel there is anything to be said about at the rarefied level of general design. Even for that one, about all that can be said is that, for almost any domain, there exist ways of fixing physical quantities, bringing them under control, creating "boundary conditions." For electronics, this is done with sources, voltage dividers, etc. In mechanical engineering, it might include fastening things down, hooking up motors, etc. Perhaps it is worth saying that "FIXing a physical quantity means turning it into a control quantity," but I'm not sure how to say that. It is likely that the way one's knowledge of this sort of regularity is used is in assimilating a new domain; namely, everyone knows that when he is learning about meta-hydraulic engineering he should be sure to ask what methods there are for fixing meta-hydraulic quantities.

Besides these actions, which arise in the course of normal problem solving, there are actions (subsumed under "patch" in Fig. III.6) required to change a circuit because it is failing to meet its specifications. These are described in Sect. III.D; they are not implemented, because the mistake-correction machinery to support them does not exist.

Many design plan schemata are arranged in specialization hierarchies

similar to hierarchies of specialized composite device types. (Sect. III.A) This is especially true of the circuit-alteration plans discussed in the next chapter. The reason for this is that a circuit-alteration plan for an action like "bias ..." is close to *being* a device type. The difference is that the procedural component is larger and the plan is more anonymous; the resistors used for biasing do not become components of a "biasing device," but of the circuit that is being biased.

Just as devices may be related by the predicate SPEC-DEV-TYPE, plan schemata may be related by

[SPEC-SCHEMA |plan schema 1| |plan schema 2|].

Any instance of the specialized schema (1) is an instance of the general schema (2). <*SPEC-SCHEMA-DEFN> Choice rules are provided for these plan schemata which are completely analogous to the rules (Sect. III.B.2) for choosing among specialized device types. <*SPEC-IS-BETTER, TWO-SPECS-WORSE-THAN-ONE>

Two other useful control predicates defined in the file DESI are STASK <*STASK-DEFN> and REDUCE. <*REDUCE-DEFN> The first is used to abbreviate in the common case where a task is defined, and made a subtask of something else, in the same breath. The second is used to express a common interaction among design plans: when one plan accomplishes part of the function of another. In that case, saying [REDUCE < -reducers- > |reducee|] means "the reducer tasks are all the subtasks of the reducee task; don't bother to try to execute it any further." (Cf. Sect. II.B.1.)

III.C Composition of Partial Solutions

One of the most interesting and complex events that can happen during the career of any problem solver is the failure of the labels attached to its canned plans to match all of the requirements of some task. In a design task, this situation allows unlimited scope for the study of creativity. Of course, our knowledge of such matters is as yet very slight, so that the approach my system takes to the handling of such problems is not terribly brilliant.

When a DESIGN task does not immediately succeed, an attempt is made (Sect. III.B) to break it down into a core task plus several constraints and "features." For example, the knowledge in ZORCH is sufficient to discover that an amplifier is required given a wide range of requests for designs. Then further choice information from ZORCH is used to select one that is likely to meet all the amplifier constraints so generated. As mentioned in Sect. II.A, this sequence is called the "recognition protocol." Finally, the constraints are resolved.

Often this approach fails. It may fail in more than one way:

(1) More than one "core device-type" (see Sect. III.A) may be discovered.

(2) There may be more than one way to implement a core device type. (See above, especially the discussions of "superordinate" device classes and abstract device-types.)

(3) The constraints generated may not actually be satisfiable.

All of these problems may call for composition of solutions to subproblems. The last problem is peculiar in some ways; I devote section III.D to describing constraint resolution.

The others are related by having to do with the choice protocol. For example, problem (2) may arise if more than one amplifier fits some of the requirements, and none fits well enough to exclude the others. In this case,

the response of the system when no further exclusions can be made is to record [QUIESCENCE |choice name|] in the choice data pool.

It is here that choice rules with conclusions of the form [/:RULE-TOGETHER...] are important. They allow options to be superseded by new options. This is the most natural and painless place for composition to occur in a NASL-based system.

This is the closest DESI comes to a universal composition method. This is a defect in the system as it exists. A better system would recognize the *need* for a /:RULE-TOGETHER and propose one; future episodes would exercise and modify it. For example, some of the choice rules for amplifiers discussed in the next chapter contain almost-general principles regarding cascading a buffer amplifier to another amplifier when the first was picked for its input impedance. It is not hard to see a more general principle regarding inputs and outputs lurking behind this specific one. It lurks there still.

For now, you must be content with the specialized composition rules in Chapter IV.

III.D Constraint Collapse

As a design proceeds, the current data pool fills up with formulas specifying components, connections, quantity values, and constraints. If everything proceeds smoothly, eventually there will be a value for every primitive component and the problem will be solved. The most common thing to go wrong with this scenario is to discover that some subset of constraints cannot be satisfied. DESI counts this as a mistake in the sense of Sect. II.E. All its previous machinations were based on the assumption that the functional requirements could be reduced to constraints and satisfied. When

this turns out not to be the case, the task network must be altered to reflect what it should have been doing. On the other hand, as much as possible of the network must be salvaged.

As I pointed out in Sect. II.E, my theory of mistakes is as yet poorly developed. This section must be taken as a continuation of the detailed proposal I made there, not as a description of existing program.

As a concrete example, say that a low-pass filter is required, which filters out one radio station at 700KHz to leave the signal of another, equally strong station at 500kHz.

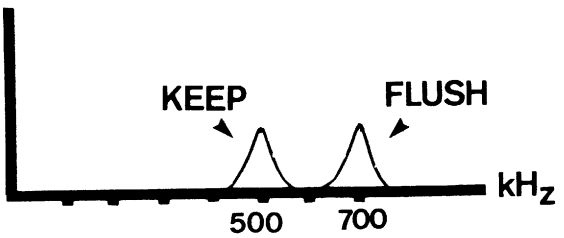

Figure III.10 Radio Spectrum With Two Stations

This problem can be represented easily using the "frequency picture" language I will develop in Chapter IV. I will continue to use simple English in what follows.

The chosen solution is an RC low-pass filter, an instance of the schema represented by Fig. I.6. This schema and the frequency-domain methods used to find it (Sect IV.B.1) generate these constraints and equalities:

CON1 (from rephrasing of the problem):

$$V_o(700\ \text{kHz}) < .1V_o(500\text{kHz})$$

CON2 (from schema for RC filter or by analysis):

$$H(s) = \frac{1}{1+RCs}$$

CON3 (from knowledge of filters):

$$\text{selectivity}(f_1, f_2) = \frac{|H(j2\pi f_1)|}{|H(j2\pi f_2|}$$

CON4 (from knowledge of linear devices):

$$|H(j2\pi f)| = \frac{V_o(f)}{V_i(f)}$$

Figure III.11 Relevant Constraints

From these constraints and the statement of the problem, we can build up the following "*constraint network*":

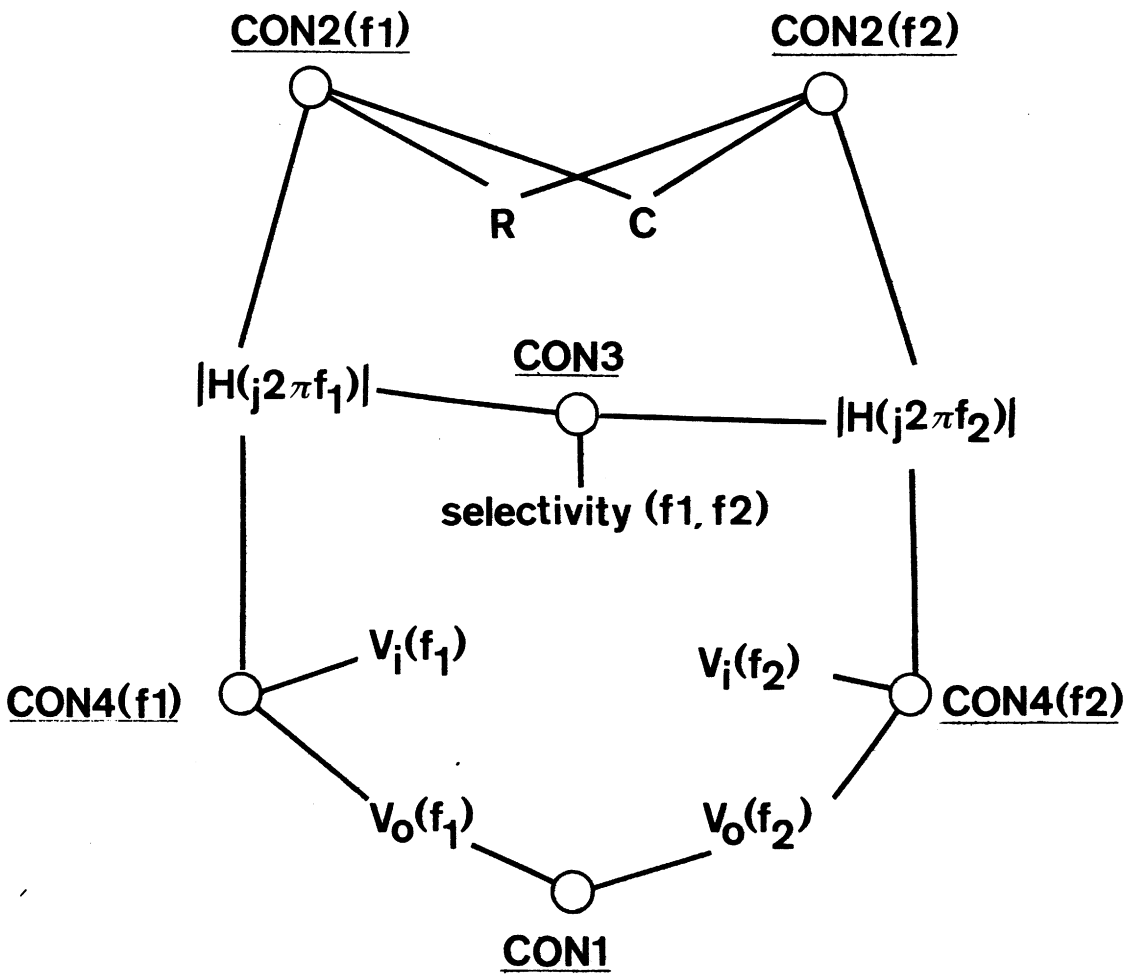

Figure III.12 Constraint Network

(Cf. <*CQ-CLO> in Appendix 2.)

If all the constraints are satisfiable, that is, if the output amplitude ratio can be made small enough that the second station has negligible output compared to the first, everything is fine. But, as it happens, there are no values of R and C which can be picked in order to bring this off. This is referred to as "constraint collapse." This will be noticed when one of the constraints proves unsatisfiable. Which constraint it will be is unpredictable; the problem is clearly a problem of the whole network rather than any task.

Recall from Sect. II.E that fixing a mistake involves altering the current

world model and the task network. In this case, there are several current tasks that have caused the problem: the choice of an RC circuit to implement the low-pass filter, and the various CONSTRAINs, some thawed from the RC schema, which led to the trouble. DESI has a record of every choice it made in the process of setting this network up, so it could find a choice point that would make a difference, restore the state of the world at that point, and try something else; I have already discussed and rejected this in Chapter II.

The design knowledge of DESI provides us with a better method of solving this problem. This method amounts to the following English summary:

(1) Find a control-quantity in the collapsed task network such that changing it would get rid of the problem. This is not as easy as it sounds. It is counterproductive to consider "making V_0(500kHz) larger," for instance. Any method for doing that will probably make V_0(700kHz) larger as well. In the example, the proper answer is "selectvity." I assume symbol manipulation powerful enough to handle this. (Sect. III.B.3)

(2) Introduce a new task [IMPROVE '|losing control quantity| |direction and magnitude|]. A task of type IMPROVE, unlike previous control-quantity manipulators, has the aim of changing some already-set object rather than fixing one that has yet to be set. To execute the IMPROVE task requires some domain-dependent rephrasing, choice, etc., which by now are routine. By one route or another, a plan like that of Fig. I.7 is recovered.

(3) The actual tasks associated with the IMPROVE plan perform the acquisition and insertion of the new pieces, an amplifier, capacitor, and resistor, and the renaming of the output port. The resulting change in topology flushes the old constraints on the system function and hence the selectivity of the device and enables the design to be completed. This is not too painful, since the control-quantity [(H ?DEV) ?F] is marked GENERIC-CA in the data pool; when the old stored value is flushed, a task is enqueued to recalculate it.

Except for the initial symbol manipulation, this seems fairly simple; the IMPROVE task is no different from any other task, such as BIAS or COUPLE, which alters the topology and part names of a circuit. The difference, of course, is that the policies associated with the IMPROVE plan must specify exactly what parts of the old task network encircling the RC filter are to be

preserved. The difficulty of implementing this scheme revolve around the careful undoing of protections and other policies.

III.E Programmer's Guide

DESI is a skeletal theory of design within which the user's domain-dependent rules operate. These rules will fall into three classes: rephrasing rules, device definitions, and device-choice rules.

The user's rephrasing rules can be very simple. Any declaration that a function is a CONTROL-ATTRIBUTE will cause the cq-shard machinery to turn a d-shard of the form [[λ (X) (= (|attribute| ?X) |value|)]] into a side-task to CONSTRAIN the given control quantity.

More complicated rules can create entire inferential subtasks to make finer discriminations. Examples will be given in the next chapter.

In making up device schemata, the user will have to use his intuitions. Superordinate device types are convenient slots to put inheritable characteristics into. A basic device type is one with a "diagram" common to every member of the type. (I assume that every domain DESI is likely to be applied to will have the concept of diagram.) The "diagram" (device schema) will not be attached to the basic device type directly; instead, each node in the DERIVED tree below the basic type (see Fig. III.5) will have its own schema. (Remember that the details of the diagrams for a device type and one of its specializations are likely to be inconsistent; the task networks of the two will be related *via* the SOUL device.)

In writing choice rules, the user has a certain amount of freedom. There are three stages in picking a device type or design plan: deducing possibilities, running choice rules before QUIESCENCE, and running choice

rules after QUIESCENCE. (See Sect. II.C.1.) There is as yet no iron-clad semantics for what each of these stages means, mainly because I lack experience in interfacing rules.

Typically, the "possibility" rules are very lax; if a device could be appropriate, given some piece of the current situation, some rule should suggest it. Throwing it away or incorporating it into a larger structure is the job of the choice rules. (Cf. CHOOSE-AMP, in Appendix 3, described in the next chapter.)

The user should be careful in his use of /:RULE-OUTs if this is the structure he chooses. If there are two relevant variables in a situation, and one is, strictly speaking, incompatible with a suggested device or plan, this is not necessarily a reason to rule it out. After all, it would not be a living option in the choice protocol if the other variable had not caused it to be suggested. For example, input-impedance considerations may suggest a common-collector amplifier, while gain considerations suggest something else. It is clearly silly to throw away the common-collector suggestion on the basis of gain. Instead, a /:RULE-TOGETHER is probably appropriate.

/:RULE-OUTs are useful mainly as a device for expressing "differential diagnosis" information. Such a rule mentions two or more options, and throws one of them away. Remember that the order of examination of these predicates is /:RULE-OUT, then /:RULE-IN, then /:RULE-TOGETHER, and that the choice protocol quits as soon as fewer than two options remain.

QUIESCENCE has no fixed meaning. Sometimes the system uses it to express rules which are intended to take over if the user's rules can't make up their minds; this is the case with the rules for choosing among SPEC-DEV-TYPEs. (Sect. III.B.2.) This freedom probably represents a deficiency in the choice machinery.

IV Electronics

Electronics knowledge falls naturally into three categories: the physics and mathematics of electronic components, devices, and signals; the knowledge necessary to do design, including rephrasing, composition, and patching; and a catalogue of circuit diagrams and plans.

In this chapter as in the last, the notation <**formula-name*> is a reference to a formula in the appendices, in this case usually Appendix 3. This Appendix is rather long, but has been laid out in an order which corresponds as closely as possible with the presentation of this chapter. This chapter is fairly dull. Given the vocabulary and conventions developed in Chapter III, it is routine to encode much of the information I will describe.

Appendix 3, long as it is, can only be described as "sketchy." For each important concept I will discuss, there is a representative circuit, plan, or rule set in the appendix, but often several of its siblings will be missing. I will describe these gaps in Sect. IV.D.

IV.A Physics

IV.A.1 Connections and Constraints on Components

Recall from Chapter III that connections in a design domain are physical configurations associated with constraints. In electronics, these configurations are called *nodes*.

Interfaces between electronic devices are of two types: terminals and ports. I will discuss ports later. A terminal is a wire coming out of a primitive or composite device. A group of terminals may be bunched into a

"node." One constraint on a node is "Kirchhoff's Current Law" (KCL) (Senturia and Wedlock, 1975), which says that the sum of the currents into a physical node is zero. My "logical" nodes are treated as terminals themselves at a "higher level," where they may themselves be joined into nodes. <*NODE-TRMIN> (See Fig. IV.1).) KCL for nodes therefore states that the current into the node, considered as a terminal, equals the sum of the currents out of the sub-terminals defining the node. <*KCL-2>

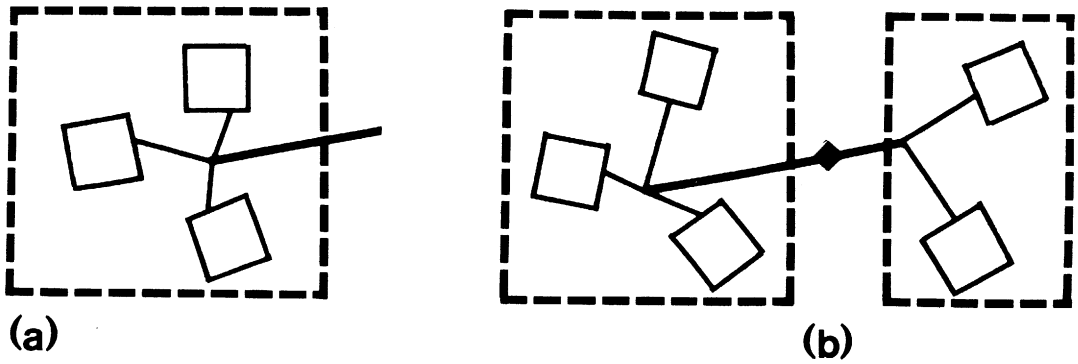

Figure IV.1 Terminals and Nodes

Devices also satisfy Kirchhoff's Current Law. <*KCL-1> A composite device's terminals are almost always nodes themselves. These will be declared with the predicate DEV-TERMINALS. <*KCL-3> (Fig. IV.2)

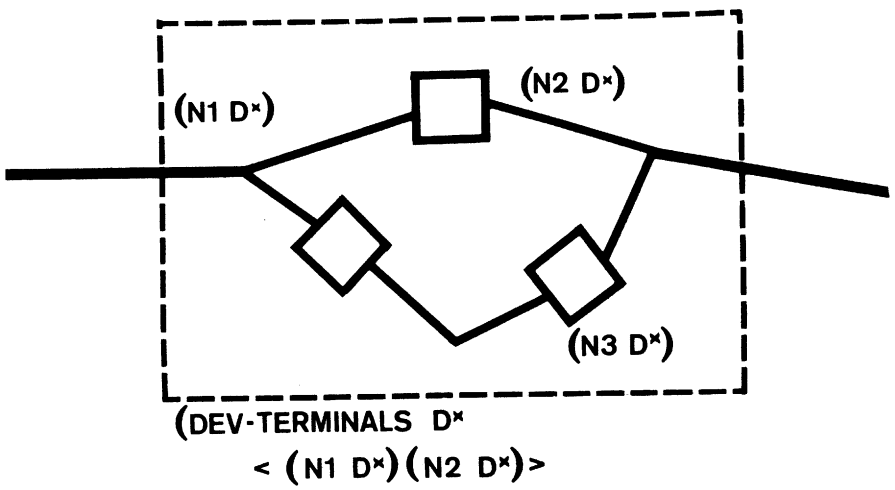

Figure IV.2 Composite Device Terminals

A fundamental model manipulation in the electronics domain is to merge two nodes, which makes them equal. <*NODES-MERGE-MANIP> A less easily visualized operation is DEV-INSERT <*DEV-INSERT-MANIP>, which breaks a node in two. (Fig. IV.3)

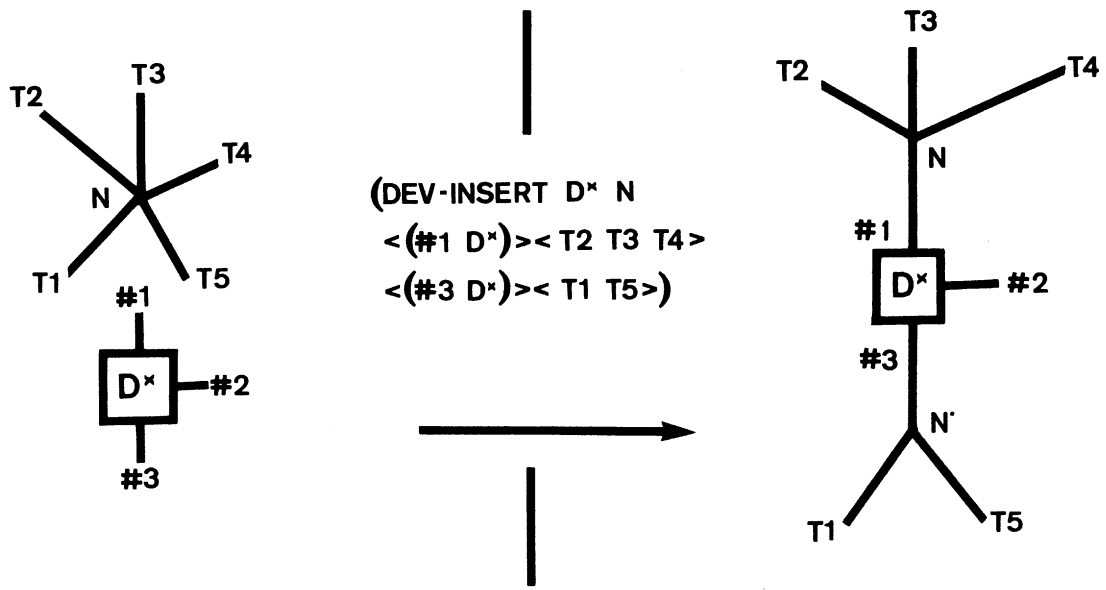

Figure IV.3 Inserting a Device into a Node

Ports are pairs of terminals (which are almost always nodes of composite devices), to be thought of as carrying signals. The signals may be

implemented either as currents or voltages. <*PORT-TAXONOMY> A set of voltage ports may be grouped into a *nest*, which is exactly analogous to a node formed by grouping terminals. (In particular, every nest is considered a higher-level port.) <*NEST-PORT> Ports may be combined into nests, and nests may be merged, just as terminals are combined into nodes. <*NESTS-MERGE-MANIP>

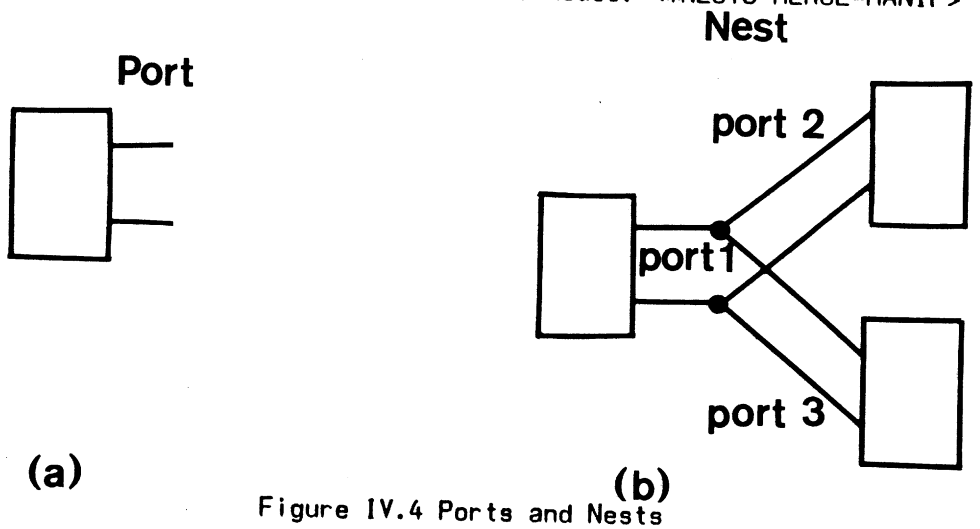

Figure IV.4 Ports and Nests

Kirchhoff also had a voltage law, which for our purposes merely amounts to the fact that every point in space can be assigned a conventional voltage. To enforce this, the rule <*KVL-1> constrains all the node and terminal voltages at a physical node to be the same.

All devices are given interface descriptions by the packets defining their device types. The interface description (e.g., "?X is a 2-terminal") interact with formulations of Kirchhoff's voltage and current laws to generate standard constraints. <*2-TERMINAL-DEFN> Composite device types (see Chapter III) may have terminal or port interfaces, or both.

An important class of devices are the "signal transmogrifiers" or SIG-TRANSERS, by which I mean any device one of whose primary functions is to take a signal in on its input port, or "INPORT," and put out a signal on its output port, or "OUTPORT." <*SIG-TRANSER-GLORIA-MUNDI>

(Much of the notation in this and the following section has been influenced by the notations devised by A. Brown (1975) and Stallman and Sussman (1976).)

IV.A.2 Signals

Signals are abstract objects with three components: a time function, a signal-medium (voltage or current), and a "home" (a port). "A ***signal*** is any physical variable whose magnitude or variation with time contains ***information***.... When we speak of 'signals,'... we refer ... to voltages and currents." (Senturia and Wedlock, 1975, p.2)

A "homeless" signal is not allowed in my notation. A signal may have more than one home, but only if all its homes are in the same nest. <*KVL-2> Given a port, the function PORT-SIGNAL should give its signal.

To make up for the absence of homeless signals, many actions manipulate signal ***descriptions***, lambda-expressions from signals to truth values. For example, the formula

[CONVERT |device| |signal description| |signal relation|]

means "device converts any signal appearing on its INPORT which satisfies the description into an OUTPORT signal which bears the relation to the input signal." An example of the use of this predicate appears in Chapter I. Another is,

```
[DESIGN (λ (X)
          (CONVERT ?X
             (λ (S1)
               (FORALL (F)
                  (IMPLIES (/> ?F 1MHz)
                           (= ((FOURIER-TRNSFRM (TFUN ?S1)) ?F)
                              0)))   )
             (λ (S1 S2)
               (FORALL (F)
                  (AND (IMPLIES (/< ?F 100kHz)
                                (= ((FOURIER-TRNSFRM (TFUN ?S2))
                                    ?F)
                                   0))
                       (IMPLIES (/> ?F 100kHz)
                                (= ((FOURIER-TRNSFRM (TFUN ?S2))
                                    ?F)
                                   ((FOURIER-TRNSFRM (TFUN ?S1))
                                    ?F)))))   ))))]
```

This formula illustrates the use of the Fourier transform of a signal.

The DESI+ZORCH system actually lacks any deep mathematical understanding of transforms. Instead, it summarizes the frequency-domain behavior of a signal with a ***frequency picture*** of its time function. A frequency-picture is a tuple of "frequency features." A feature is specified as [FF |freq| |landmark|]. <*FF-FREQ, FF-LANDMARK> A landmark may have a shape, height, and width, written FL-SHAPE, FL-HEIGHT, or FL-WIDTH. In general any particular characteristic is optional, but the assumption is that a feature has non-zero size. These are similar to the human conventions for such descriptions.

Signal characteristics can have these values:

SHAPE -- may be SPIKE, HUMP
WIDTH -- a HUMP may be SHARP or FUZZY
HEIGHT -- a number

The notation [SERIES |freq| |delta freq| |shape| |fun|] defines an infinite frequency picture consisting of a row of features of the given shape, at interval of delta freq, starting at the given frequency. The height of the nth landmark in the row is given by fun. ***This function must be decreasing*** (as it will be in all physical applications).

(This way of reasoning about the frequency domain is in many ways closer than straight Fourier transforms to the way humans think about it, and reveals more about the signals. This is a good illustration of the point that using a logical notation does not commit you to a "mathematical" treatment of a domain.)

For example, a square wave of frequency *f* offset by half its amplitude *A* has frequency picture picture

```
[<(FF 0Hz LANDMARK#79)
  !#(SERIES |f| (* 2 |f|)
       SPIKE (λ (N) (* |A| (// 4 (* ?N PI)))   ))>]
```

where

```
[FL-SHAPE LANDMARK#79 SPIKE]
```

and

```
[=/> '(FF-HEIGHT LANDMARK#79) (// |A| 2)].
```

The usual graphical notation for the Fourier transform of an offset square wave is, of course,

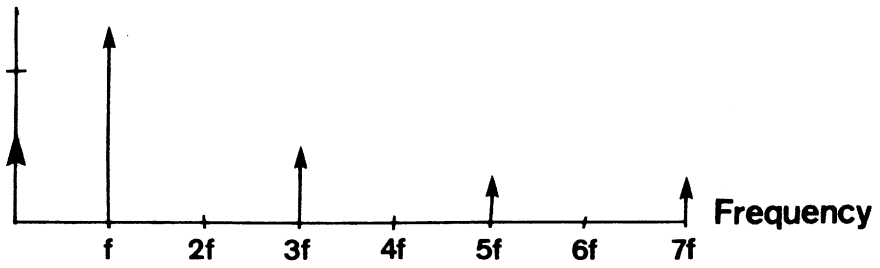

Figure IV.5 Fourier Transform of an Offset Square Wave

Time-domain signal attributes are also known to the system. A function may be periodic with a certain period, expressed as [PERIODIC |tfun| |period|]. If so, [ONE-PERIOD |tfun|] is a function which is zero everywhere except from -T/2 to T/2, where [(ONE-PERIOD |tfun|) ?T] = [|tfun| ?T]. (T is the period of the function.) Others are (following (A. Brown, 1975)) the DC ***offset*** of the signal; ***amplitude***, the height of a periodic signal; ***phase*** with respect to another signal; etc.

A large set of formulas is concerned with computing the frequency picture of periodic signal descriptions expressed in time-domain terms. <*PERIODIC-FREQ-PIC>

Besides these intrinsic descriptions, there are "extrinsic" signal properties like "carrier," or "modulation," or "distortion." These concepts would be necessary for a system that designed or redesigned radios, which are a level above that which I have focused on. (This is the chief difference in emphasis between Brown's approach to signal description and mine.) It appears to me that it would be easy and instructive to add this knowledge to DESI, but that it would be a slight detour.

Of course, the descriptions of signals are of interest only so far as they support comparison. We are interested in designing circuits which convert from one kind of signal to another; given two pictures, a good description of the transformation from one to the other will help us to retrieve useful plans. This will be described below, in Sect. IV.B.

IV.A.3 Multiple Models of Linear Systems

A great advantage of a linear domain such as elementary electrical engineering is that it may be profitably attacked by linear, time-independent methods in many cases. In addition, the fact that analysis is of a closed network makes a forward-deduction scheme practical. (Cf. (Nevins, 1974c, and Sussman and Stallman, 1975).)

The folowing types of quantities are to be dealt with by an electronics analyzer (see Sect. IV.C):

physical quantities like voltages and currents
component control quantities like resistances and capacitances

The kinds of questions to be answered are:

What is the the value of some physical quantity in a given circuit?
What are the values of derived control quantities like the Thevenin resistance or gain of a circuit?

Often these questions are with respect to a circuit of interest as given, but just as often explicit or implicit reference is made to a derived model of a circuit. For example,

The DC gain of a circuit is the gain of the DC model of that circuit

The Thevenin resistance of a circuit is the resistance of the circuit when it is disconnected from its environment and its independent sources are set to zero.

The impedance of a circuit is its (complex) resistance in its "sinusoidal steady state" model.

etc.

These models are generated by the use of FRAME, N, and T formulas (see Sect. II.B.2) in the data base. Many of these appear in the schemata for various devices, but the FRAME axioms occur separately. Together they define the models as follows:

(1) [(DC)] The DC model of a circuit is the same circuit with all frequency-dependent features nulled in a device-dependent way. Thus we have <*FRAME-DC> plus formulas like

```
[IMPLIES (IS CAPACITOR ?X)
         (AND (N (DC) '(IS CAPACITOR ?X))
              (T (DC) (IS OPEN ?X)))]
```

(cf. <*CAP-PKT>) for frequency-dependent components like capacitors. The (DC) of an already-DC model is itself, because these components can be nulled only once. <*DC-IDEM> Thus (see Chapter II), we have

```
[N (DC) '(T ?R (FRAME (DC) <(HERE)>))]
[T (DC) (=/> '(DC) (HERE))]
```

(2) [(INC)] The incremental model of a circuit. <*REF-INC, FRAME-INC, INC-IDEM>

(3) [SSS |s|] The sinusoidal-steady-state model for complex frequency s. <*REF-SSS, SSS-IDEM, FRAME-SSS> (In this model, all linear devices act like resistors with complex resistances or impedances.)

(4) [ISOLATE |trmin 1| |trmin 2|] The model obtained by disconnecting the given terminals from whatever nodes they appear in. <*ISOLATE-DEFN-1, 2, and 3> (These terminals are usually nodes in a composite device.) It is used for calculating Thevenin resistances. <*REF-ISO, FRAME-ISO, ISO-IDEM>

(5) [PASSIVE] The network with all active sources set to zero. Also used for calculating equivalents. <*REF-PASSIVE, FRAME-PASSIVE, PASSIVE-IDEM>

IV.B Electronic Design Knowledge

The knowledge in ZORCH is meant to mesh with the knowledge in DESI so as to bring about useful behavior. Generally DESI provides the plan framework and some quite general heuristics, while the solid stuff is in ZORCH. This is true of knowledge about design actions.

IV.B.1 Rephrasing Electronic Design Problems

When it comes to rephrasing design, DESI does little more than provide the concept of "d-shard" and the policy that every shard in the initial explosion must lead to something useful. (Chapter III) Most of the knowledge is in ZORCH. Here are defined the interesting control quantities of this domain: voltage gain <*V-GAIN-SHARD>, and input and output impedance <*INPUT-Z-SHARD, OUTPUT-Z-SHARD>. These lead to side-tasks which constrain control quantities (*via* <*CQ-SHARD> in DESI), but they also lead to domain-dependent "d-features" regarding the ranges of these quantities. These all end up affecting the way in which device schemata are selected.

Besides this sort of control-value redescription, more devious things can

happen. If a d-shard of the form [[λ (|v|) (CONVERT ...)]] appears in the middle of the explosion, it causes an inferential subtask to appear. <*CVT-EXPLODE> This subtask mimics the supertask for exploding design properties, in that it manipulates formulas describing signals, breaking them into "signal shards." These signal shards are then parsed into d-shards and ultimately into d-features, side-tasks, and core device types.

There are two complementary paths this deduction can take. One <*FREQ-DOM-REPHRASE> tries to compute the frequency pictures of the input and output, then find a transformation ("FREQ-PIC-TRANS") between them. The output of this deduction is an object of the form [LOW-PASS |cutoff|], [HIGH-PASS |cutoff|], [MODULATE |freq|], etc. This transformation becomes a signal shard. This language for describing frequency transformations is not as general as it could be. On the other hand, it is quite extensible, and reflects everything I know about the subject.

The other rephrasing method <*TIME-DOMAIN-REPHRASE> merely searches for certain simple functional relationships between the time values of the input and output. (This has not been implemented yet.)

The system tries to choose <*CVT-CHOICE> between these two ways of rephrasing on the basis of simple criteria. For example, if the input predicate to CONVERT doesn't mention the input at all, the transformation is more likely to be a function of signal values, so the time-domain is ruled in.

IV.B.2 Reconciling Partial Solutions

This kind of information arises in two places: generating and choosing core device-types, and choosing ways of making devices. (As in Sect. III.D, I am not including information about patching failing constraints.)

It is important to note that computation regarding a circuit is to be postponed whenever possible until after the design rephrasing plan. This is because (a) the ideological intent is for this phase to be concerned with ***problem***, not ***solution***, manipulation; and (b) the pieces of problem are lying around in the wrong form to be noticed during deductions.

So, during the d-explosion and subsequent re-parsing, DESI is more interested in seeing the pieces than putting them together. This orientation means that ordinarily the generation of two core device-types is an error; the error will be revealed by the choice protocol for the CORE-FINDER step of Fig. III.7.

ZORCH makes up for this by allowing the design rephraser to pass the buck under certain circumstances. <*LINEAR-GROUPING> If one of the competing core device types is linear, ZORCH rules them together into the artificial device-type [GROUP < -device types- >]. GROUPing means CASCADing in some as yet undetermined order. <*MAKE-GROUP-1, MAKE-GROUP-2, MAKE-GROUP-3> The idea is to postpone deciding among them or composing them until the constraints and features have been sorted out. After all, you can be pretty sure a linear device will not interfere too badly with what it is connected to. ("Cascading" two device-types means making one of each and coupling them. See below.)

This is the somewhat bedraggled method by which core-device-type choice can give rise to cascades. This should be a rare event. Normally, the device classes introduced do their job harmoniously enough so that one superordinate or basic category is natural for describing a desired circuit. In that case, the category will wind up as part of the central MAKE task of a design network. (See Fig. III.8.) Here cascading will reappear in a much more disciplined way as the choice of which subordinate or specialized device type

to use.

In choosing a way to MAKE a known circuit type such as an amplifier, often more than one suggested subtype comes to mind. This will be because various subtypes are indexed by associated D-NOTES. <*MOD-V-GAIN, HIGH-V-GAIN, etc.> If more than one device is triggered by a constellation of D-NOTES, it may be for two standard reasons. First, a control quantity like gain may fall into two ranges. (<*V-GAIN-SHARD> and its ilk specify overlapping ranges, for example.) If so, a value falling in the ambiguous region may necessitate differential diagnosis in the subsequent choice situation. One might have to decide between an op-amp or common-emitter amplifier on the basis of cost, convenience, etc.

Second, the two suggestions may answer to different subrequirements, in which case they must be combined in some way (or one subrequirement must be foregone).

Rules for performing these functions for the device type AMPLIFIER are given in the packets defined by <*CHOOSE-AMP>. This contains principles like, "If high bandwidth is required, prefer a multi-stage amplifier to a single high-gain stage"; and, "If one option (e.g., a common-collector) has been proposed because of its input impedance, and another for some other reason, cascade them in that order."

The fact that in using the rules of <*CHOOSE-AMP> the system is restricted to a well-defined situation helps to make those rules concise and to the point. This attests to the organizing power of the concept of "superordinate device type." It is difficult to think of a natural choice situation in which the statement of the problem does not bring to mind a set of rules organized around some such type. It appears that a large part of the training of technicians and engineers is the accumulation of these sets.

Cascading two device types is MAKE-ing an object of the abstract type [CASCADE |type 1| |type 2|]. <*MAKE-CASCADE> This consists of a straightforward plan to MAKE a device of each type and COUPLE them. The components of the result are the two devices. In cascading two device types, it is wise <*COUPLE-GENERAL-1, COUPLE-GENERAL-2> to use the most general specializations of the device types involved. These are defined by the predicate MOST-GENERAL-SPEC (defined in Appendix 2). An example is the device type GENERAL-CE, which is the most general common-emitter circuit.

Coupling comes under the heading of a circuit-change operation. (Cf. Sect. III.B.4)

IV.B.3 Changing Circuits

Except in the dullest circumstances, a circuit schema cannot be instantiated merely by binding a few variables. After it has been plugged in, its "expansion obligations" become active. (In the case of a productive "device type" like [CASCADE ...], these obligations are part of the plan that makes a device of that type.) These expansion obligations are the normal place in which circuit-changing tasks arise. (If failure-correction were implemented, this would also be a common source of circuit changes, of course.)

The circuit-changing actions defined so far include biasing, coupling, and fixing voltages. These actions come in specialization hierarchies, and they interact in interesting ways.

Biasing bipolar junction transistors (BJTs) is defined by <*BJT-BIAS-NET> as three important tasks: fixing the collector current (I_C), reverse-biasing the collector-base junction, and fixing the base-emitter voltage (V_{BE}). For a

typical one-stage transistor amplifier, these tasks are subsumed by the specific suggestions in <*TYPICAL-BJT-ONE-STAGE-BIAS-PLAN>.

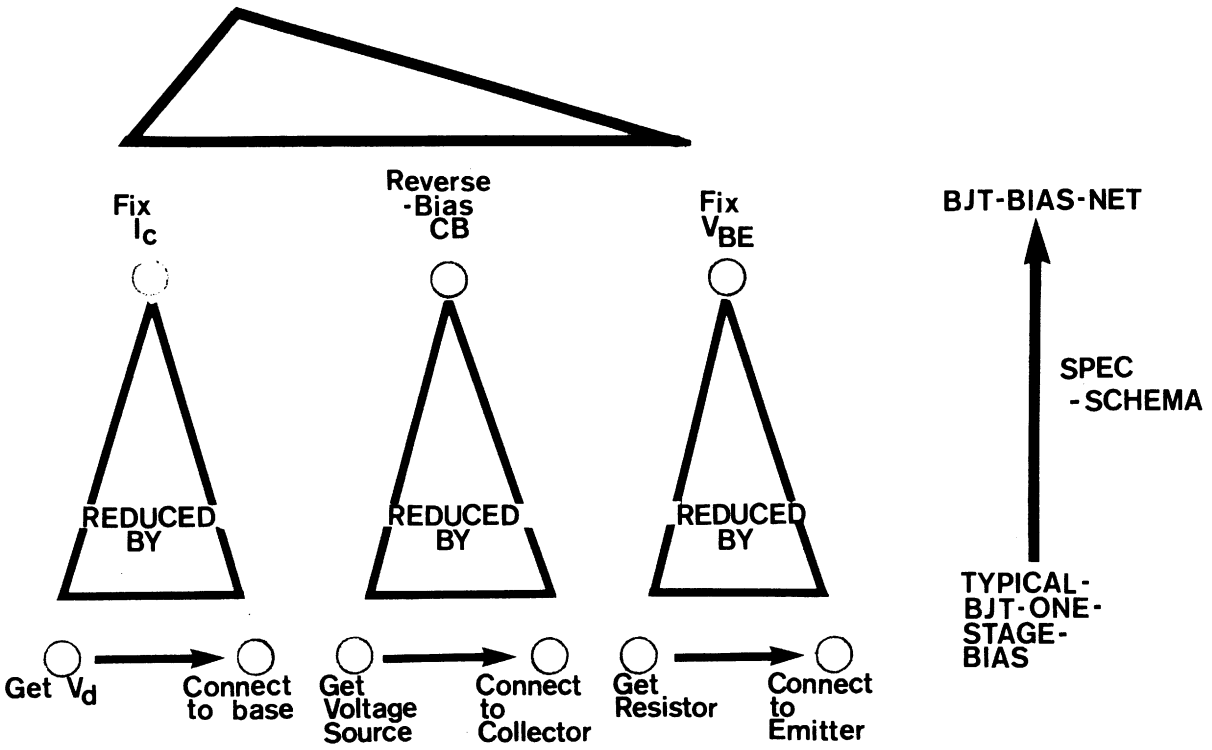

Figure IV.6 Bias Plans

This plan defines the bias networks of Fig. I.2.

The biasing plan interacts with the coupling plan for BJTs <*COUPLE-NET>, which itself has several SPEC-SCHEMAs. The general plan is shown in Fig. IV.7.

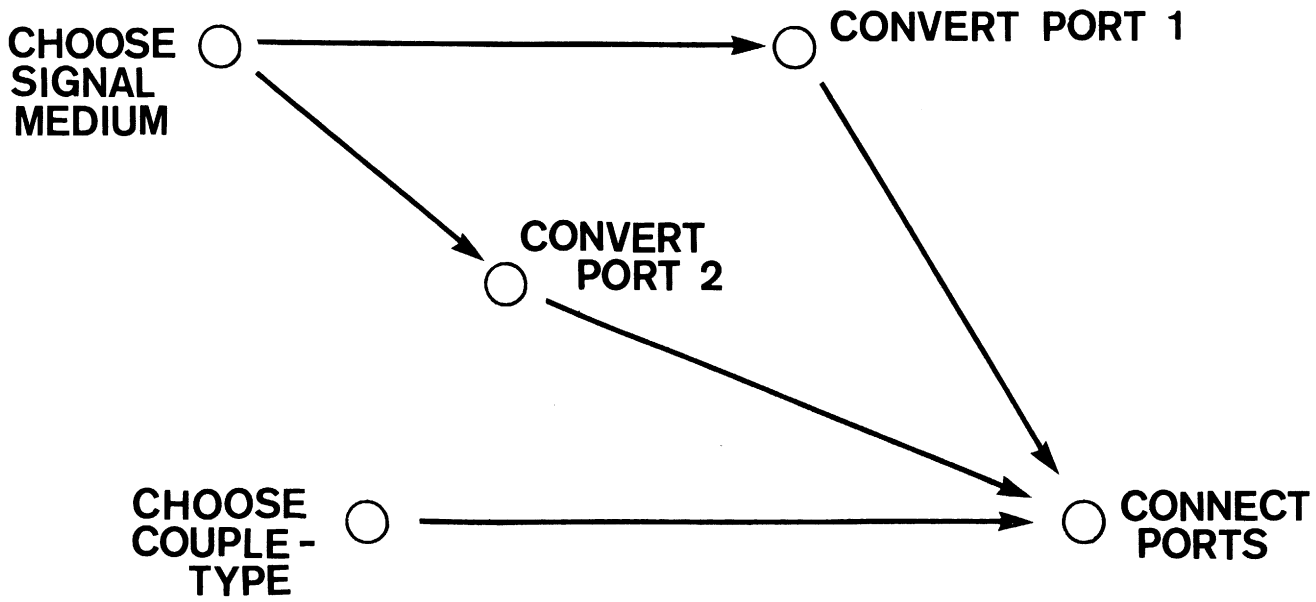

Figure IV.7 General BJT Coupling Plan

The interaction starts with the rule that coupling is considered before biasing. <*COUPLE-BEFORE-BIAS> (Cf. Fig. II.5.) The reasons for this rule are that decisions made during coupling often influence the way in which biasing is done; and that coupling components perform many biasing functions. These interactions depend upon the particular coupling network chosen. The rule packets <*COUPLE-CE-X-HINTS, COUPLE-CC-X-HINTS> suggest particular subnets to be used. One of these <*CE-DIR-VOL-COUPLE-PLAN> is shown in Fig. IV.8. (The others are as yet unwritten.)

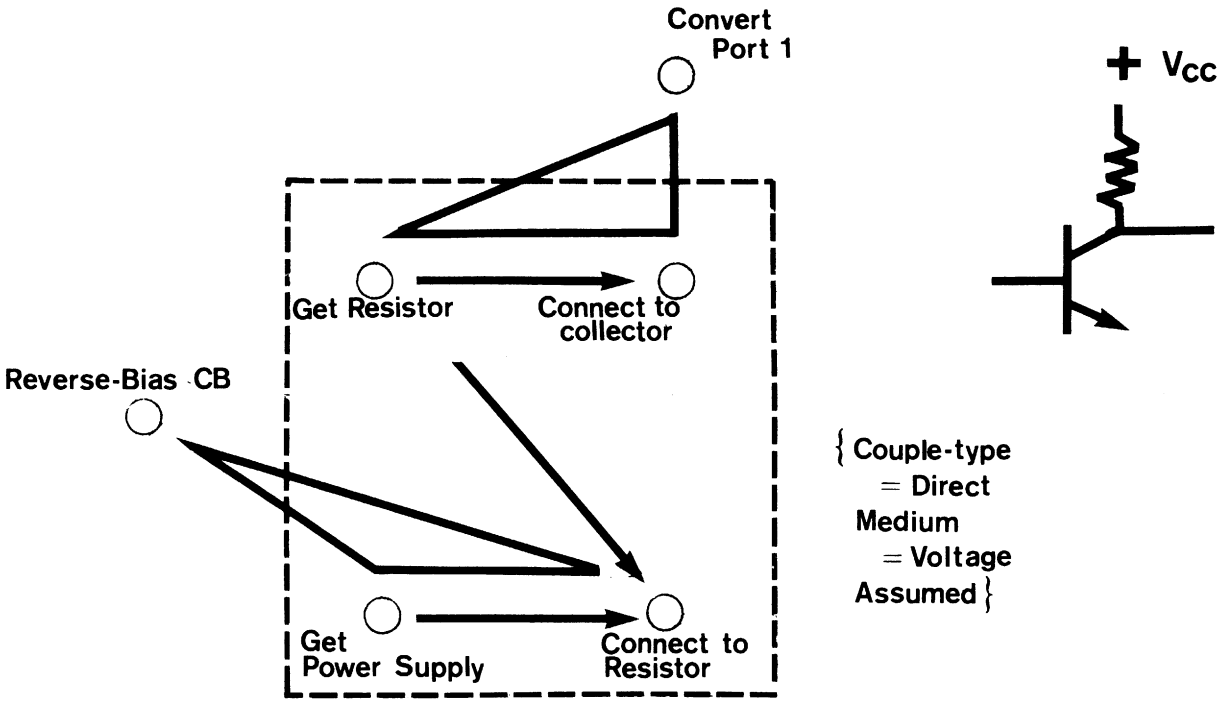

Figure IV.8 Common-Emitter Direct Voltage Coupling

Choosing this plan amounts to settling on direct coupling with voltage as the medium. Picking it reduces tasks from the bias network and main coupling plan without further examination.

Two plans for fixing voltages is shown in Appendix 3. <*VS-FIX-V, VD-FIX-V> The first is used to set voltages absolutely. The second is used for setting voltages, such as the base voltage of an transistor, which are to be allowed to change incrementally.

IV.B.4 Electronics Analysis Knowledge

In the usual case, there is no special electronics analysis knowledge. Device properties are expressed by numerical constraints (Chapter III), which interact with each other and SELECT-VALUE tasks to produce results. In the ideal case, the deductions involved are always "one-step deductions" (Stallman and Sussman, 1976) of the value of one variable at a time. (I have taken much of the analysis knowledge practically verbatim from Stallman and Sussman's data base.)

Besides calculations of physical quantities and component values, there is also the problem of computing values of "generic control quantities" (Sect. III.A.1). These are quantities like "voltage gain," which is defined generically as, "The value of the voltage on the outport over the value on the inport," but whose symbolic and numerical values depend on the circuit involved. When a generic attribute value is constrained, a task will be created to calculate it. <*GENERIC-CAS-DO in Appendix 2>

How this is done depends on the quantity to be calculated. For example, to calculate the Thevenin impedance of a circuit from two terminals, you must enter the reference point [ISOLATE |trmin 1| |trmin 2|], fix the voltage at the two terminals, and calculate the current. (This technique is probably beyond the current competence of NASL. For a precise account of how to do it, see Doyle, 1977.)

Very little electronics analysis knowledge has been implemented. (See Sect. IV.D. below.) My implementation has focussed on qualitative reasoning about design; it complements the work of Stallman and Sussman (1976) on quantitative analysis.

IV.C Device Schemata

The last part of Appendix 3 is a bag of device schemata. These include primitive components and a few composite devices. Most of the primitive components are defined entirely by one or two constraints and some T and N formulas describing their behavior from various derived models. The transistor schema <*BJT-DEFN>, as you might expect, has a more complicated structure. Besides the physical constraints on its terminal voltages and currents, every transistor must be biased into its desired region. Thus, a composite device schema that specifies that its transistor is active will automatically acquire an expansion obligation to bias the transistor.

There are several device schemata for composite devices defined at the end of Appendix 3:

GENERAL-CE -- The most abstract common-emitter circuit
TYPICAL-CE -- The circuit shown most often in textbooks, which is derived from the more abstract one
GENERAL-ECP -- The most abstract emitter-coupled pair
ECP-DC-AMP -- One of many circuits that can be derived from the ECP
VD -- The humble voltage divider
RC -- The filter, not the cola.

The relation between the ECP-DC-AMP and its "soul," the GENERAL-ECP, is shown in Fig. IV.9

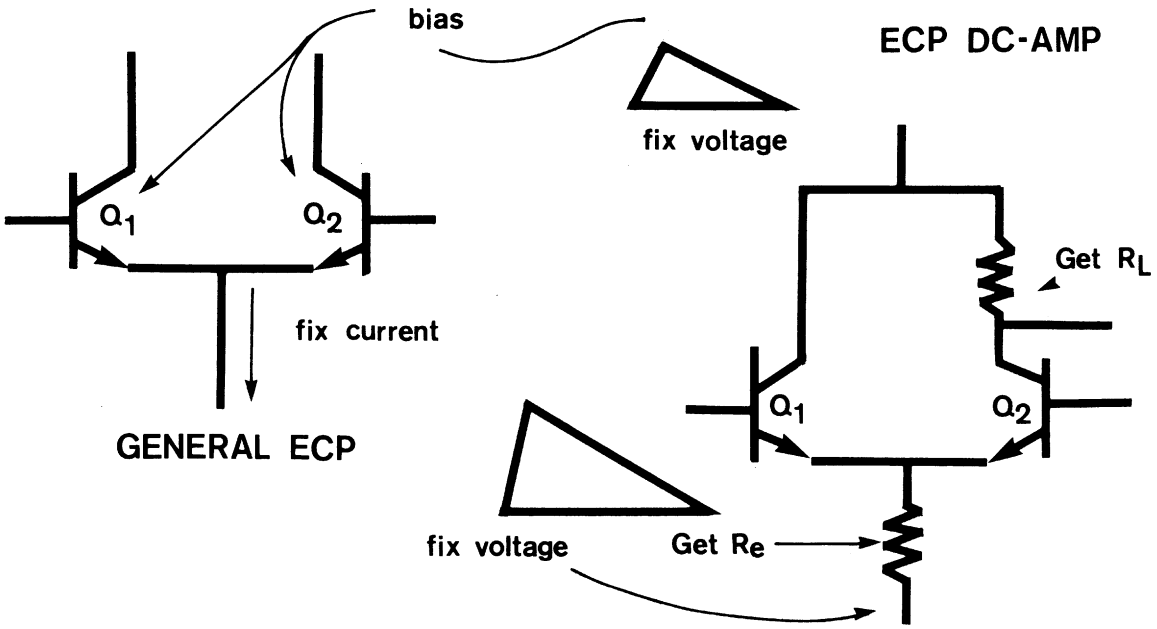

Figure IV.9 General and Specialized Emitter-Coupled Pairs

A similar relation exists between GENERAL-CE and TYPICAL-CE. In defining these relations, extensive use is made of the predicates REDUCE and FUNCTION, defined in Appendix 2. REDUCE declares that some set of tasks is a complete subnetwork of the reduced task. [FUNCTION |dev| |reduced task|] is a special case, used in device schemata, which declares the existence of an abstract task for acquiring the device dev, and specifies that this task is sufficient to accomplish the task to be reduced.

The function [OBL |dev| |action|] denotes an expansion task created by the use of rule <*EXPANSION-OBLS-DO>, defined in Appendix 2.

IV.D Programmer's Guide

As I have pointed out in several places, the information in ZORCH is sketchy. I hope this hasn't harmed the clarity of my presentation; I have provided an example of every kind of construct I discussed. The knowledge shown is still in the process of being debugged and assimilated into DESI; there has not been time to be complete. As it is, implementation and debugging cannot keep up with generation of new formulas. (See Chapter V.)

This section is included as a guide to anyone who wishes to add information to the DESI system (or its successor). It consists of a list of all the kinds of information that are missing. (Filling in these holes should be a matter of following the principles of Sect. III.E and imitating the formulas that already exist.)

> Rephrasing can be extended. More principles are needed for comparing frequency pictures; some knowledge of Fourier transforms (as opposed to frequency pictures) is necessary in order to generate precise side tasks.

> The time-domain information is non-existent in Appendix 3. The time-domain rephrasing problem comes down to trying to find a functional relationship between two axiomatically described objects. This is not easy.

> Many more circuits are needed. Most of the amplifier types mentioned in <*CHOOSE-AMP> and its satellites are undefined; power amplifiers are a glaring omission. The superordinate category of "filter" is entirely absent. There should be a theory of LC filters and when to use one of them. This should work with a theory of matching impedances during coupling of power stages. Time-domain rephrasing will be useless unless it is used to index clippers, rectifiers, and other "operational" circuits.

> Coupling circuits are many and varied. Only one specialized circuit is shown in Appendix 3. (See Fig. IV.8.) One difficulty I didn't mention in discussing coupling is choosing polarities of bipolar transistors. These are normally NPN, but coupling them directly often constrains them to be of opposite polarities.

> Some of the information shown hanging off circuit diagrams in Chapter I, especially interface constraints, has not been represented in the circuit schemata of Appendix 3. For example, the system function of the RC filter will not be what it claims to be if something is connected to it.

These problems are the ones I've thought about, and sketched answers for. There are many more I haven't thought of.

The major result of this research to date is the existence of MASL, DESI, and ZORCH. The experience I have had in defining the MASL notation and applying it to electronic circuit design is enough by itself to suggest certain conclusions. These are the substance of Chapter VI.

Of course, the claims I make there will be not be decisively proven until the program has been debugged and the ZORCH knowledge base has been completed. These activities are proceeding, and preliminary results are reported in Sect. V.B below.

Since DESI is intended to be a working system, and since people may wish to experiment with it or with the MASL interpreter, these are described from a practical point of view in Sect. V.A. I would appreciate comments from those who try it out.

V.A Using DESI

V.A.1 Loading and Running the Program

To run my programs on the MIT AI Lab time-sharing system, type

```
:l dvm;
at DOT, then
(load nasl)
```

at LISP. This will load the interpreter and leave you in the LISP read-eval-print loop. (The reader and printer are not the standard LISP mechanisms, but that shouldn't matter.) Now you can load MASL assertions from any file you choose (using DEFFILA; see Appendices 2 and 3). To load the designer in, type (load desi) instead of or in addition to the above to load the electronics

V Results

The major result of this research to date is the existence of NASL, DESI, and ZORCH. The experience I have had in defining the NASL notation and applying it to electronic circuit design is enough by itself to suggest certain conclusions. These are the substance of Chapter VI.

Of course, the claims I make there will be not be decisively proven until the program has been debugged and the ZORCH knowledge base has been completed. These activities are proceeding, and preliminary results are reported in Sect. V.B below.

Since DESI is intended to be a working system, and since people may wish to experiment with it or with the NASL interpreter, these are described from a practical point of view in Sect. V.A. I would appreciate comments from those who try it out.

V.A Using DESI

V.A.1 Loading and Running the Program

To run my programs on the MIT AI Lab time-sharing system, type

```
:l dvm;
```

at DDT, then

```
(load nasl)
```

at LISP. This will load the interpreter and leave you in the LISP read-eval-print loop. (The reader and printer are not the standard LISP mechanisms, but that shouldn't matter.) Now you can load NASL assertions from any file you choose (using DEFMLA; see Appendices 2 and 3). To load the designer in, type (load desi) instead of or in addition to the above; to load the electronics

knowledge, type (load zorch). (Or load the binary file TS DESI, to save a lot of time.)

To run NASL, type

(start |action|)

at LISP, where action is a formula or list structure of the form [/:TASK ...] or [|action function|...]. (In the latter case, if the action has outputs, type (start |action| |outputs|).) NASL will begin execution, adding the new action to its task network. When the network is empty, it will return the output variables of action if any. It will also remain in the data pool of the action performed, typically a DESIGN, so that you can run new tasks in the same environment.

V.A.2 DESI Talks to You

When NASL runs into trouble, or needs the answer to a symbol-manipulation problem, it stops and asks you questions to try to help itself out. "Trouble" is defined as the failure of the choice or rephrasing protocol to achieve its aims. In either case, the system stops and tells you its trouble, then asks various questions. For example, if it cannot make up its mind after applying all the known choice rules, it will ask if it should choose at random. Yes/no questions are answered by typing "yes," "y," "no," or "n." Other questions will be requests for formulas. (If you type an S-expression where a formula is wanted, NASL will convert it.)

Three standard questions and the reactions to their answers are:

> "Do you want to see the reasoning involved?" Answering affirmatively causes NASL to re-run the most recent relevant deduction, with the switches STP-TRACE-DEPTH* and RECORD-SEE-DEPTH* set to values that let you see the steps. (See below.)

> "Break?" If you answer "yes," the system will give you a LISP break loop. This is usually used for adding new information to the system with further DEFMLAs. (DEFMLA may be used to redefine old formulas, too.) When you return from the break, the next question in NASL's list will be asked.

> "Try again?" Don't say "yes" unless you've taken advantage of the break loop and added a new choice principle, a missing axiom that was needed by the last deduction, or some other rule.

V.A.3 You Talk to DESI

There are several useful programs for getting debugging information or explanations of behavior from DESI.

(TASK-NET-DUMP) causes the entire active or pending task network under your original request to be printed out in a somewhat readable fashion, with indentation, successor pointers, etc. (This function takes an optional task-name argument; if it is omitted, your original request is used as a default.)

(WHY |task|) causes the task network ***above*** the named task to be dumped. Notice that asking this question about a frozen task will show you part of the teleological structure of the device it appears in.

(SUPPORT |fact|) prints the supporters of the fact.

(PURPOSE |device|) prints out the supertasks associated with MAKE-ing or ACQUIRE-ing the device (depending on what kind of records of the history of the device have been kept).

I am being a little vague about the precise formats of these functions, both because the system encourages you to be vague (it will try to figure out whether you are referring to formulas by name or pattern); and because the features these functions support are changing rapidly.

Some useful statistics-gathering functions are defined in the file SNAP. The statistics gathered include running times overall, time spent doing matching and indexing (important low-level functions of a rule-based system), and success of the indexer in keeping garbage from reaching the matcher. Typing "(SNAP)" at LISP causes these statistics to be printed out. "(SNAP RESET)" resets them; this must be done once to turn them on. (Collecting

these statistics slows things down somewhat.)

There are several switches whose settings affect the verbiage produced by the NASL system. These switches are printed by the function SHOW-SWITCHES, defined in SNAP. The switch NASL-TRACE-DETAIL*, an integer from 0 to about 5, defines how much the interpreter will tell you about its every move. The switch STP-TRACE-DEPTH* controls printing of major sub-goal information by STP; if zero, nothing is printed. Similarly, RECORD-SEE-DEPTH* controls printing of data-base alterations. Both of these switches are normally zero; the system sometimes turns them on to display inferences or model effects. (As in Sect. V.B, below.)

V.B Experimental Results

The DESI system is still being debugged. Consequently, it has never designed a circuit all the way through. It has done simple choice analysis, trivial equation solution, and some rephrasing of simple conjunctive design problems.

Most of the time, the system runs well. Forward deduction and task reduction occur in a reasonable amount of time; watching the trace on the screen is a pleasant experience. The NASL language is easy to program in; it encourages an incremental style of programming in which partial plans interact. If an unforeseen interaction occurs, the presence of all task information in the data base usually makes the trouble easy to find; fixing it is usually a matter of adding a new rule. When it isn't, the problem usually turns out to be a syntactic error in a rule; I strongly recommend to future designers of predicate-calculus systems with large rule bases that they include checks of syntax (such as proper number and type of arguments to

predicates) at the time a rule is read. In such a system, a wrong rule is often overlooked entirely.

Calls to the theorem prover tend to slow things down. This not because of any combinatorial explosion, but probably because of the theorem prover's carefulness in checking subsumption and other special cases which are normally irrelevant. The theorem prover is probably too complex for its own good; the next such program I write will be much more like Conniver. (McDermott and Sussman, 1973)

Furthermore, the evaluation mechanism (Appendix 4), which is normally quite efficient, wastes much time in other circumstances. The evaluator is called whenever the right-hand side of an implication is detached during deduction. It tries to apply reduction rules to every new subexpression of the detached formula. This isn't nearly as expensive as it sounds, because one quick index call is enough to check the (very common) case where there are no applicable rules. Unfortunately, in detaching the right-hand side of a large implication like <*+DESI-2> in Appendix 2, there is an embarassing pause. I am putting up with this for the time being, but it is clear that this is a job for some further pragmatic-predicate mechanism.

As it stands, the system with ZORCH loaded occupies a huge amount of core. As incomplete as it is, it takes up about 220K of 36-bit word storage. About 50K of this is the LISP system and my low-level utilities, and 130K is list space, 70% (about 90K) of which is occupied. Of this, about 30K consists of circuit diagrams for the five device types defined in ZORCH! I am sure this can be brought down by judicious rearrangement of consequent vs. antecedent deduction; the problem appears to be overenthusiastic forward reasoning while setting up device schemata. However, a lot of storage is required to set up and index packets, whether they are used later or not. To duck this probelm,

in the sample runs given below, large conjunctions of the form [/:PKT ...] were actually implemented as ordinary formulas like [AND ...]. In the long run, we are going to have to confront the question of organizing secondary storage for use by AI programs.

There are three experimental results to demonstrate. First (Sect. V.B.1), I present DESI's attempt to design a simple amplifier. Then I show its pitiful approach to the filter-design problem I described in Chapter I. In both these cases, the program crashed due to bugs before going as far as it is, in theory, capable. Finally, in Sect. V.B.3, I discuss some research with Jon Doyle into the relation between NASL and NOAH. (Sacerdoti, 1975)

V.B.1 A Simple Amplifier

Here is a sample output from a run of DESI on the first problem in Chapter I, with NASL-TRACE-DETAIL* set to 3. My comments start with ";". The output has been edited to relieve the tedium. The dots "..." indicate omissions. In this and the next section, apparently random "!'s" in the output are caused by garbage collection, one exclamation point per collection. Long strings of these marks are indicative of long pauses between outputs.

```
;The system was started with by typing
;      (start '[design ...] '[<'(winner)>]:
(CREATING TASK
   [:TASK (*DES*) <> (LAMBDA NIL
                        (DESIGN (LAMBDA (X)
                                   (AND (DEV-TYPE ?X AMPLIFIER)
                                      (= (V-GAIN ?X) 5)
                                      (= (INPUT-Z ?X) 30000)))))
      <'(WINNER)>])
(ENABLED [(*DES*)])
(EXECUTING [(*DES*)] ...)
(TASK [(*DES*)] BEING REDUCED)
(TASK [(*DES*)] TO BE REPHRASED)
(CREATING TASK [:TASK (REPHRASER (*DES*))
                  <>
```

```
                    (LAMBDA NIL
                       (:REPHRASE (*DES*) [DESIGN (LAMBDA (X)
                                                    (AND ...))]
                          <'(WINNER)>))
                    <>])

(ENABLED [REPHRASER (*DES*)])
(EXECUTING ![REPHRASER (*DES*)]
   [:REPHRASE (*DES*) [DESIGN (LAMBDA (X)
                                 (AND (DEV-TYPE ?X AMPLIFIER)
                                    (= (V-GAIN ...) 5)
                                    (= (INPUT-Z ...) 30000)))]
      <'(WINNER)>]
   [<>])
(TASK [REPHRASER (*DES*)] BEING REDUCED)
(TASK [REPHRASER (*DES*)] REDUCED TO
   [:DO-SUBNET (DESI-REPHRASE-PLAN
                  [(LAMBDA (X) (AND ...))] (*DES*) <'(WINNER)>)
               <>])
                                          <>])
;Here are the tasks from the DESIGN rephrase plan <*+DESI-2>
(CREATING TASK [:TASK (EXPLODER PLAN#380)
                  <>
                  (LAMBDA NIL
                     (D-EXPLODE [(LAMBDA (X)
                                   (AND ...))]))
                  <>])
(CREATING TASK [:TASK (ACCOUNT-FOR-ALL PLAN#380)
                  <>
                  (LAMBDA NIL
                     (ACCOUNT-FOR-ALL-SHARDS [(LAMBDA (X)
                                                (AND ...))]))
                  <>])
(CREATING TASK [:TASK (CORE-FINDER PLAN#380)
                  <>
                  (LAMBDA NIL
                     (:FIND (LAMBDA (+DT)
                                (CORE-DEV-TYPE [...] ?+DT))))
                  <'(CORE-DT PLAN#380)>]) !
(CREATING TASK [:TASK (MAIN-TASK-INFERER PLAN#380)
                  <'(CORE-DT PLAN#380)>
                  (LAMBDA (+DT)
                     (:INFER '(AND (STASK (MAKER ...) (*DES*)
                                    <>
                                    (LAMBDA NIL
                                       ...)
                                    <...>)
                                 (:MAIN (MAKER ...) (*DES*)))
                        <(CORE-FINDER PLAN#380)>))
                  <>])
(CREATING TASK [:TASK (SIDE-TASKS-FINDER PLAN#380)
                  <>
                  (LAMBDA NIL
```

```
                    (:INFER '(FORALL (+ST) (-> G (SIDE-TASK ...)
                                                 (EXISTS ...)))
                       <>))
                  <>])
(CREATING TASK [:TASK (FEATURES-FINDER PLAN#380)
                  <>
                  (LAMBDA NIL
                    (:INFER '(FORALL (+FT) (-> G (D-FEATURE ...)
                                                 (EXISTS ...)))
                       <>))
                  <>])
(CREATING TASK [:TASK (GATHERER PLAN#380)
                  <>
                  (LAMBDA NIL
                    (:INFER '(:REDUCED (*DES*))
                       <(CORE-FINDER PLAN#380)
                        (SIDE-TASKS-FINDER PLAN#380)
                        (FEATURES-FINDER PLAN#380)>))
                  <>]) !
(BLOCKED [MAIN-TASK-INFERER PLAN#380])
(BLOCKED [CORE-FINDER PLAN#380])
(ENABLED [ACCOUNT-FOR-ALL PLAN#380])
(BLOCKED [EXPLODER PLAN#380])
(BLOCKED [GATHERER PLAN#380])
(BLOCKED [FEATURES-FINDER PLAN#380])
(BLOCKED [SIDE-TASKS-FINDER PLAN#380])

;The only task which is enabled is the policy-setup for shard
;accounting--

(EXECUTING [ACCOUNT-FOR-ALL PLAN#380]
   [ACCOUNT-FOR-ALL-SHARDS [(LAMBDA (X)
                              (AND (DEV-TYPE ?X AMPLIFIER)
                                   (= (V-GAIN ...) 5)
                                   (= (INPUT-Z ...) 30000)))]]
   [<>]) !
(TASK [ACCOUNT-FOR-ALL PLAN#380]
   BEING REDUCED)
(TASK [ACCOUNT-FOR-ALL PLAN#380]
   REDUCED TO [:PRIM *SETUP])

;But the first real task is the exploder

(ENABLED [EXPLODER PLAN#380])
(EXECUTING [EXPLODER PLAN#380]
   [D-EXPLODE [(LAMBDA (X)
                 (AND (DEV-TYPE ?X AMPLIFIER)
                      (= (V-GAIN ...) 5)
                      (= (INPUT-Z ...) 30000)))]]
   [<>])
(TASK [EXPLODER PLAN#380] BEING REDUCED)
(TASK [EXPLODER PLAN#380] REDUCED TO
    [:INFER '(D-SHARD [(LAMBDA (X) (AND ...))]
```

```
                        [(LAMBDA (X) (AND ...))]]
            <>]]

;The system prints out a lengthy list of deductions from this inferred
;   "d-shard":

(INFERENCES MADE BY [EXPLODER PLAN#380]
   --)
(RECORDING [D-SHARD [(LAMBDA (X)
                        (AND (DEV-TYPE ?X AMPLIFIER)
                           (= (V-GAIN ...) 5)
                           (= (INPUT-Z ...) 30000)))]
               [(LAMBDA (X)
                   (AND (DEV-TYPE ?X AMPLIFIER)
                      (= (V-GAIN ...) 5)
                      (= (INPUT-Z ...) 30000)))]]
   0)

;It turns the elements of the conjunction into shards
(RECORDING [:GEN (NOT (ELT ?+C^4 <[DEV-TYPE ?X AMPLIFIER]
                                  [= (V-GAIN ...) 5]
                                  [= (INPUT-Z ...) 30000]>))
               (D-SHARD [(LAMBDA (X)
                           (AND (DEV-TYPE ... AMPLIFIER)
                              (= ... 5)
                              (= ... 30000)))]
                  [(LAMBDA (X) _?+C^4)]]]
   0) !

;The first of three major shards:

(RECORDING [D-SHARD [(LAMBDA (X)
                        (AND (DEV-TYPE ?X AMPLIFIER)
                           (= (V-GAIN ...) 5)
                           (= (INPUT-Z ...) 30000)))]
               [(LAMBDA (X) (= (INPUT-Z ?X) 30000))]]
   0)
 ...
;The "=" tells DESI one side might be a control quantity:

(RECORDING [POS-CQ-SHARD [(LAMBDA (X)
                              (AND (DEV-TYPE ?X AMPLIFIER)
                                 (= (V-GAIN ...) 5)
                                 (= (INPUT-Z ...) 30000)))]
               [X] [INPUT-Z ?X] [30000]]
   0)
(RECORDING [:GEN (NOT (AND (NOT (CONTAINS [30000] [?? X]))
                           (=> '(DEN ...) ?F^9)
                           (CONTROL-ATTRIBUTE ?F^9)
                           (=> '(DEN ?+F^9) ?F^9)))
               (SIDE-TASK [(LAMBDA (X)
                              (AND (DEV-TYPE ... AMPLIFIER)
                                 (= ... 5)
```

```
                                    (= ... 30000))))]
                    [(LAMBDA (X) (CONSTRAIN <...> (LAMBDA ...)))]]]
   0)
(RECORDING [POS-CQ-SHARD [(LAMBDA (X)
                              (AND (DEV-TYPE ?X AMPLIFIER)
                                   (= (V-GAIN ...) 5)
                                   (= (INPUT-Z ...) 30000))))]
               [X] [30000] [INPUT-Z ?X]]
   0)
(RECORDING [:GEN (NOT (AND (NOT (CONTAINS [INPUT-Z ...]
                                          [?? X])))
                           (=> '(DEN ...) ?F^9)
                           (CONTROL-ATTRIBUTE ?F^9)
                           (=> '(DEN ?+F^9) ?F^9)))
               (SIDE-TASK [(LAMBDA (X)
                              (AND (DEV-TYPE ... AMPLIFIER)
                                   (= ... 5)
                                   (= ... 30000)))]
                    [(LAMBDA (X) (CONSTRAIN <...> (LAMBDA ...)))]]]
   0) !

;Having noticed that INPUT-Z is a control attribute, it uses the rule
;  INPUT-Z-SHARD to classify the desired impedance:

(RECORDING [:GEN (NOT (= 30000 ?Z^7))
                 (AND (:GEN (NOT (> ?Z^7 300000.0))
                            (D-FEATURE [(LAMBDA ...)]
                                       [RANGER INPUT-Z VERY-HIGH]))
                      (:GEN (NOT (AND (> ?Z^7 1500.0)
                                      (< ?Z^7 500000.0)))
                            (D-FEATURE [(LAMBDA ...)]
                                       [RANGER INPUT-Z HIGH]))
                      (:GEN (NOT (AND (> ?Z^7 500)
                                      (< ?Z^7 2000.0)))
                            (D-FEATURE [(LAMBDA ...)]
                                       [RANGER INPUT-Z MODERATE]))
                      (:GEN (NOT (< ?Z^7 1000))
                            (D-FEATURE [(LAMBDA ...)]
                                       [RANGER INPUT-Z LOW])))]
   0)
(RECORDING [AND (:GEN (NOT (> 30000 300000.0))
                   (D-FEATURE [(LAMBDA (X) (AND ...))]
                      [RANGER INPUT-Z VERY-HIGH]))
                (:GEN (NOT (AND (> 30000 1500.0) (< 30000 500000.0)))
                   (D-FEATURE [(LAMBDA (X) (AND ...))]
                      [RANGER INPUT-Z HIGH]))
                (:GEN (NOT (AND (> 30000 500) (< 30000 2000.0)))
                   (D-FEATURE [(LAMBDA (X) (AND ...))]
                      [RANGER INPUT-Z MODERATE]))
                (:GEN (NOT (< 30000 1000))
                   (D-FEATURE [(LAMBDA (X) (AND ...))]
                      [RANGER INPUT-Z LOW]))]
   0)
```

```
...
;The winning feature is "high input impedance":

(RECORDING [D-FEATURE [(LAMBDA (X)
                          (AND (DEV-TYPE ?X AMPLIFIER)
                               (= (V-GAIN ...) 5)
                               (= (INPUT-Z ...) 30000)))]
           [RANGER INPUT-Z HIGH]]
  0)
...
;A similar deduction is done for the case of voltage gain:

(RECORDING [D-SHARD [(LAMBDA (X)
                       (AND (DEV-TYPE ?X AMPLIFIER)
                            (= (V-GAIN ...) 5)
                            (= (INPUT-Z ...) 30000)))]
           [(LAMBDA (X) (= (V-GAIN ?X) 5))]]
  0)
...
(RECORDING [POS-CQ-SHARD [(LAMBDA (X)
                            (AND (DEV-TYPE ?X AMPLIFIER)
                                 (= (V-GAIN ...) 5)
                                 (= (INPUT-Z ...) 30000)))]
           [X] [V-GAIN ?X] [5]]
  0)
...
(RECORDING [POS-CQ-SHARD [(LAMBDA (X)
                            (AND (DEV-TYPE ?X AMPLIFIER)
                                 (= (V-GAIN ...) 5)
                                 (= (INPUT-Z ...) 30000)))]
           [X] [5] [V-GAIN ?X]]
  0)
...
(RECORDING [SIDE-TASK [(LAMBDA (X)
                         (AND (DEV-TYPE ?X AMPLIFIER)
                              (= (V-GAIN ...) 5)
                              (= (INPUT-Z ...) 30000)))]
           [(LAMBDA (X)
              (CONSTRAIN <'...> (LAMBDA (G1) (= ... 5))))]]
  0)
...
(RECORDING [AND (:GEN (NOT (> 5 1000)) (D-FEATURE [(LAMBDA (X)
                                                     (AND ...))]
                                                  [RANGER V-GAIN VERY-HIGH]))
           (:GEN (NOT (AND (> 5 50) (< 5 5000)))
              (D-FEATURE [(LAMBDA (X) (AND ...))]
                 [RANGER V-GAIN HIGH]))
           (:GEN (NOT (AND (> 5 1) (< 5 100)))
              (D-FEATURE [(LAMBDA (X) (AND ...))]
                 [RANGER V-GAIN MODERATE]))
           (:GEN (NOT (=< 5 1)) (D-FEATURE [(LAMBDA (X) (AND ...))]
```

```
                                        [RANGER V-GAIN LOW]))]
   0)
 ...
(RECORDING [D-FEATURE [(LAMBDA (X)
                          (AND (DEV-TYPE ?X AMPLIFIER)
                               (= (V-GAIN ...) 5)
                               (= (INPUT-Z ...) 30000)))]
              [RANGER V-GAIN MODERATE]]
   0)
 ...

;The last d-shard gives us the core device type:

(RECORDING [D-SHARD [(LAMBDA (X)
                       (AND (DEV-TYPE ?X AMPLIFIER)
                            (= (V-GAIN ...) 5)
                            (= (INPUT-Z ...) 30000)))]
              [(LAMBDA (X) (DEV-TYPE ?X AMPLIFIER))]]
   0) !
(RECORDING [CORE-DEV-TYPE [(LAMBDA (X)
                             (AND (DEV-TYPE ?X AMPLIFIER)
                                  (= (V-GAIN ...) 5)
                                  (= (INPUT-Z ...) 30000)))]
              [AMPLIFIER]]
   0)
(*INFERENCES DONE*)

;This concludes the inferences of the exploder
;Now the other tasks of the rephrasing plan assemble the features, core
;device type, and constraints into a new task network:

(ENABLED [FEATURES-FINDER PLAN#380])
(ENABLED [CORE-FINDER PLAN#380])
(EXECUTING [CORE-FINDER PLAN#380]
   [:FIND (LAMBDA (+DT)
            (CORE-DEV-TYPE [(LAMBDA (X)
                              (AND ..))]
              ?+DT))]
   [<'(CORE-DT PLAN#380)>])
(TASK [CORE-FINDER PLAN#380] PRIMITIVE) !
(OLD TASK [MAIN-TASK-INFERER PLAN#380]
   HAS ACTION [:INFER '(AND (STASK (MAKER (*DES*))
                                   (*DES*)
                                   <>
                                   (LAMBDA NIL (MAKE (DEN ...)))
                                   <'(WINNER ...)>)
                            (:MAIN (MAKER (*DES*))
                                   (*DES*)))
                      <(CORE-FINDER PLAN#380)>])
(ENABLED [MAIN-TASK-INFERER PLAN#380])
(FINISHED [CORE-FINDER PLAN#380])

;Retrieval of the core device type enables the system to infer
```

```
;the main task:
(EXECUTING [MAIN-TASK-INFERER PLAN#380]
   [:INFER '(AND (STASK (MAKER (*DES*)) (*DES*)
                        <>
                        (LAMBDA NIL (MAKE (DEN ...)))
                        <'(WINNER ...)>)
                 (:MAIN (MAKER (*DES*)) (*DES*)))
      <(CORE-FINDER PLAN#380)>]
   [<>])
(TASK [MAIN-TASK-INFERER PLAN#380]
   PRIMITIVE)
(INFERENCES MADE BY [MAIN-TASK-INFERER PLAN#380]
   --)
...
(RECORDING [:TASK (MAKER (*DES*)) <> (LAMBDA NIL (MAKE AMPLIFIER))
              <'(WINNER (*DES*))>]
   0)
(CREATING TASK [:TASK (MAKER (*DES*)) <>
                   (LAMBDA NIL (MAKE AMPLIFIER))
                   <'(WINNER (*DES*))>])
(RECORDING [:SUBTASK (MAKER (*DES*)) (*DES*)]
   0)
(RECORDING [:MAIN (MAKER (*DES*)) (*DES*)]
   0)
(*INFERENCES DONE*) !
;End of inferences by main task inferer

(ENABLED [SIDE-TASKS-FINDER PLAN#380])
(FINISHED [MAIN-TASK-INFERER PLAN#380])
(BLOCKED [MAKER (*DES*)])

;Now the features of the desired device cause policies to be created:
(EXECUTING [FEATURES-FINDER PLAN#380]
   [:INFER '(FORALL (+FT) (-> G (D-FEATURE [...] ?+FT)
                              (EXISTS (T) (AND (STASK ...)
                                               (:SCOPE ...)
                                               (:SUCCESSOR ...)))))
      <>]
   [<>])
(TASK [FEATURES-FINDER PLAN#380]
   PRIMITIVE)
(INFERENCES MADE BY [FEATURES-FINDER PLAN#380]
   --)

(RECORDING [:GEN (NOT (D-FEATURE [(LAMBDA (X) (AND ...))]
                         ?+FT))
              (AND (STASK T!400/1165 (*DES*)
                      <>
                      (LAMBDA NIL (D-NOTE (DEN ?+FT)))
                      <>)
                   (:SCOPE T!400/1165 (MAKER (*DES*)))
                   (:SUCCESSOR T!400/1165 (MAKER (*DES*))))]
   0)
```

```
...
(RECORDING [:TASK T!400/1691 <> (LAMBDA NIL
                                  (D-NOTE (RANGER INPUT-Z HIGH)))
            <>]
   0)
(CREATING TASK [:TASK T!400/1691 <> (LAMBDA NIL
                                      (D-NOTE (RANGER INPUT-Z HIGH)))
                <>])
(RECORDING [:SUBTASK T!400/1691 (*DES*)]
   0)

;Noting the SCOPE of the task triggers several rules for
;suggesting ways to make an amplifier:

(RECORDING [:SCOPE T!400/1691 (MAKER (*DES*))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER V-GAIN MODERATE))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE CE))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER V-GAIN HIGH))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE N-STAGE))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER V-GAIN VERY-HIGH))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE OP-AMP))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER FREQ-OP VERY-LOW))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE DIFF-AMP))]
   0) !
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER INPUT-Z HIGH))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE CC))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY T!400/1691
                          (D-NOTE (RANGER P-GAIN HIGH))))
              (AND (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER)
                     <?DEV^29>
                     (MAKE COMP-SYM))
                   (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER)
                     <?DEV^29>
                     (MAKE PUSH-PULL)))]
   0)
(RECORDING [:ANTEC (NOT (:POLICY ?PTASK^29 (MAKER (*DES*))
                          (D-NOTE LINEAR)))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
```

```
                (MAKE CE))]
   0)
(RECORDING [:SUCCESSOR T!400/1691 (MAKER (*DES*))]
   0)
...
(RECORDING [:TASK T!400/2154 <> (LAMBDA NIL
                                  (D-NOTE (RANGER V-GAIN MODERATE)))
            <>]
   0)
(CREATING TASK [:TASK T!400/2154 <>
                  (LAMBDA NIL (D-NOTE (RANGER V-GAIN MODERATE)))
                 <>])
(RECORDING [:SUBTASK T!400/2154 (*DES*)]
   0)
(RECORDING [:SCOPE T!400/2154 (MAKER (*DES*))]
   0)

;Similarly for this feature:
(RECORDING [:ANTEC (NOT (:POLICY T!400/2154
                           (D-NOTE (RANGER V-GAIN MODERATE))))
              (:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER) <?DEV^29>
                 (MAKE CE))]
   0)
;The same rules will be found again (a slight non-optimality
; in the phrasing of the these rules)
 ...
 ...
(RECORDING [:SUCCESSOR T!400/2154 (MAKER (*DES*))]
   0)
(*INFERENCES DONE*) !
;End of inferences by features finder.

(FINISHED [FEATURES-FINDER PLAN#380])
(BLOCKED [T!400/1691])
(BLOCKED [T!400/2154])

;The side-tasks finder similarly turns side-task shards into CONSTRAIN
; tasks:
(EXECUTING [SIDE-TASKS-FINDER PLAN#380]
   [:INFER '(FORALL (+ST) (-> G (SIDE-TASK [...] ?+ST)
                              (EXISTS (T) (AND (STASK ...)
                                               (:SUCCESSOR ...)))))
      <>]
   [<>])
(TASK [SIDE-TASKS-FINDER PLAN#380]
   PRIMITIVE)
(INFERENCES MADE BY [SIDE-TASKS-FINDER PLAN#380]
   --)

(RECORDING [:GEN (NOT (SIDE-TASK [(LAMBDA (X)
                                    (AND ...))]
                           ?+ST))
              (AND (STASK T!401/6707 (*DES*)
```

```
                        <'(WINNER (*DES*))>
                        (DEN ?+ST)
                        <>)
                    (:SUCCESSOR (MAKER (*DES*))
                      T!401/6707)))]
   0)
 ...
(RECORDING [:TASK T!401/3406 <'(WINNER (*DES*))>
               (LAMBDA (X)
                  (CONSTRAIN <'(V-GAIN ?X)>
                     (LAMBDA (G1) (= ?G1 5))))
               <>]
   0)
(CREATING TASK [:TASK T!401/3406 <'(WINNER (*DES*))>
                   (LAMBDA (X)
                      (CONSTRAIN <'(V-GAIN ?X)>
                         (LAMBDA (G1) (= ?G1 5))))
                   <>])
(RECORDING [:SUBTASK T!401/3406 (*DES*)]
   0)
(RECORDING [:SUCCESSOR (MAKER (*DES*))
               T!401/3406]
   0)
(*INFERENCES DONE*)
;End of inferences by side-tasks finder.

(ENABLED [GATHERER PLAN#380])
(FINISHED [SIDE-TASKS-FINDER PLAN#380]) !
(BLOCKED [T!401/3406])

;The "gatherer" just marks the design task reduced:
(EXECUTING [GATHERER PLAN#380]
   [:INFER '(:REDUCED (*DES*)) <(CORE-FINDER PLAN#380)
                                (SIDE-TASKS-FINDER PLAN#380)
                                (FEATURES-FINDER PLAN#380)>]
   [<>])
(TASK [GATHERER PLAN#380] PRIMITIVE)
(INFERENCES MADE BY [GATHERER PLAN#380]
   --)
(RECORDING [:REDUCED (*DES*)] 0)
(*INFERENCES DONE*)
;The interpreter will see this message in a second.

(ENABLED [(*DES*)]) !
(FINISHED [GATHERER PLAN#380])

;Now the task is reattempted:
(EXECUTING [(*DES*)] [DESIGN (LAMBDA (X)
                               (AND (DEV-TYPE ?X AMPLIFIER)
                                  (= (V-GAIN ?X) 5)
                                  (= (INPUT-Z ?X) 30000)))]
   [<'(WINNER)>])
(TASK [(*DES*)] ALREADY REDUCED) ;The /:REDUCED formula is seen
```

```
(ENABLED [T!400/1691])
(ENABLED [T!400/2154])

;The policies are put into effect:
(EXECUTING [T!400/2154] [D-NOTE (RANGER V-GAIN MODERATE)]
   [<>])
(TASK [T!400/2154] BEING REDUCED)
(TASK [T!400/2154] REDUCED TO [:PRIM *SETUP]) !
(EXECUTING [T!400/1691] [D-NOTE (RANGER INPUT-Z HIGH)]
   [<>])
(TASK [T!400/1691] BEING REDUCED)
(TASK [T!400/1691] REDUCED TO [:PRIM *SETUP])

;Recording these policies causes two of the rules inferred above to
;fire...

(ENABLED [MAKER (*DES*)])
(EXECUTING [MAKER (*DES*)] [MAKE AMPLIFIER]
   [<'(WINNER (*DES*))>])
(TASK [MAKER (*DES*)] BEING REDUCED)

;...so there are two ways, common emitter and common collector,
;to make an amplifier
(MAKING A CHOICE)
(RECORDING [:CHOICE CHOICE#402 EXEC(
               [:TO-DO (MAKER (*DES*))
                      (MAKE AMPLIFIER) <'(WINNER (*DES*))>
                      ?WAY]]
   0)
;The system records first the choice, then the options.
;Recording the choice causes a flock of choice rules to be
;instantiated:
(RECORDING [:GEN (NOT (:= (MAKER (*DES*)) ?AMP-TASK^3))
              (AND (CHOOSE-AMP-PKT CHOICE#402 ?AMP-TASK^3
                      [MAKER (*DES*)] ['(WINNER ...)] [?WAY])
                 (:GEN (NOT (:SCOPE ?PTSK^3 ?AMP-TASK^3))
                    TRUE))]
   0)
 ...
;The choice rules come in a packet:
(RECORDING [CHOOSE-AMP-PKT CHOICE#402 (MAKER (*DES*))
              [MAKER (*DES*)]
              ['(WINNER (*DES*))]
              [?WAY]]
   0)
;This odd-looking formula is intended to trigger antecedent
; rules in the packet which would otherwise not be noticed
(RECORDING [:GEN (NOT (:SCOPE ?PTSK^4 (MAKER (*DES*))))
              TRUE]
   0) !
; ... such as this one:
(RECORDING [:ANTEC (NOT (:SCOPE ?PTSK1 (MAKER (*DES*))))
              (:GEN (NOT (AND (:POLICY ?PTSK1 (D-NOTE LINEAR))
```

```
                                (:SCOPE ?PTSK2 (MAKER (*DES*)))
                                (:POLICY ?PTSK2
                                         (D-NOTE (RANGER P-GAIN HIGH)))
                                (=> '(DEN ...) ?PTSK2)))
                        (:ANTEC (NOT (:OPTION CHOICE#402 ?A1 [:TO-DO ...])))
                          (:GEN (NOT (OPT-SUPPORT ?A1 ...))
                             (:RULE-TOGETHER <?A1> [:TO-DO ...])))))]
   0)
(RECORDING [:SCOPE T!400/2154 (MAKER (*DES*))]
   0)
(RECORDING [:SCOPE T!400/1691 (MAKER (*DES*))]
   0)

;Here is the first option
(RECORDING [:OPTION CHOICE#402 OPT#403 [:TO-DO (MAKER (*DES*))
                                               (MAKE AMPLIFIER)
                                               <'(WINNER (*DES*))>
                                               (MAKE CC)]]
   0)
;It finds a rule
(RECORDING [:ANTEC (NOT (:OPTION CHOICE#402 ?A1
                            [:TO-DO (MAKER (*DES*))
                               (MAKE AMPLIFIER) <_?+N>
                               (MAKE _?+DT1)]))
                (:GEN (NOT (OPT-SUPPORT ?A1
                              [:POLICY _?+PTASK (D-NOTE ...)]))
                   (:ANTEC (NOT (:OPTION CHOICE#402 ?A2 [:TO-DO ...]))
                      (:GEN (= ?A1 ?A2) (:RULE-TOGETHER <?A1 ?A2>
                                           [:TO-DO ...])))))]
   0)
;and checks the support for the options to see if input impedance
; was relevant
(RECORDING [:GEN (NOT (OPT-SUPPORT OPT#403
                          [:POLICY _?+PTASK^3
                               (D-NOTE (RANGER INPUT-Z _...))]))
                (:ANTEC (NOT (:OPTION CHOICE#402 ?A2^3
                               [:TO-DO (MAKER ...) (MAKE AMPLIFIER)
                                  <...>
                                  (MAKE _...)]))
                   (:GEN (= OPT#403 ?A2^3)
                         (:RULE-TOGETHER <OPT#403 ?A2^3>
                             [:TO-DO (MAKER ...) (MAKE AMPLIFIER) <...>
                                 (MAKE ...)]))))]
   0) !
;It was
(RECORDING [:ANTEC (NOT (:OPTION CHOICE#402 ?A2^4
                            [:TO-DO (MAKER (*DES*))
                               (MAKE AMPLIFIER) <'...>
                               (MAKE _?+DT2^4)]))
                (:GEN (= OPT#403 ?A2^4)
                      (:RULE-TOGETHER <OPT#403 ?A2^4>
                          [:TO-DO (MAKER (*DES*))
                               (MAKE AMPLIFIER) <'...>
```

```
                                (MAKE (CASCADE CC _...))))))]
   0)
(RECORDING [:GEN (= OPT#403 OPT#403) (:RULE-TOGETHER <OPT#403 OPT#403>
                                        [:TO-DO (MAKER (*DES*))
                                           (MAKE AMPLIFIER)
                                           <'(WINNER ...)>
                                           (MAKE (CASCADE CC CC))]]]
   0)
;The rule excludes cascading something with itself, so this line of
; inference dies.

;Here is the second option:
(RECORDING [:OPTION CHOICE#402 OPT#404
              [:TO-DO (MAKER (*DES*))
                  (MAKE AMPLIFIER) <'(WINNER (*DES*))>
                  (MAKE CE)]]
   0)

;It is checked for input impedance being relevant
(RECORDING [:GEN (NOT (OPT-SUPPORT OPT#404 [:POLICY _?+PTASK^3
                                              (D-NOTE (RANGER INPUT-Z _
                                                 ...)))]))
              (:ANTEC (NOT (:OPTION CHOICE#402 ?A2^3
                              [:TO-DO (MAKER ...) (MAKE AMPLIFIER)
                                 <...>
                                 (MAKE _...)])))
                 (:GEN (= OPT#404 ?A2^3) (:RULE-TOGETHER <OPT#404
                                                          ?A2^3>
                                           [:TO-DO (MAKER ...)
                                              (MAKE AMPLIFIER)
                                              <...>
                                              (MAKE ...)])))]
   0) !
;It isn't.  However, the /:ANTEC derived from the other option
;triggers,

(RECORDING [:GEN (= OPT#403 OPT#404)
                 (:RULE-TOGETHER <OPT#403 OPT#404>
                     [:TO-DO (MAKER (*DES*))
                           (MAKE AMPLIFIER) <'(WINNER ...)>
                           (MAKE (CASCADE CC CE))]]]
   0)
; and the appropriate cascade is suggested:
(RECORDING [:RULE-TOGETHER <OPT#403 OPT#404>
              [:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER)
                 <'(WINNER (*DES*))>
                 (MAKE (CASCADE CC CE))]]
   0) !
(RECORDING [:OPTION CHOICE#402 NEWOPT#405
               [:TO-DO (MAKER (*DES*))
                   (MAKE AMPLIFIER) <'(WINNER (*DES*))>
                   (MAKE (CASCADE CC CE))]]
   0)
```

```
;This new option goes through the mill also
(RECORDING [:GEN (NOT (OPT-SUPPORT NEWOPT#405
                          [:POLICY _?+PTASK^3
                             (D-NOTE (RANGER INPUT-Z _...)]]]]
              (:ANTEC (NOT (:OPTION CHOICE#402 ?A2^3
                             [:TO-DO (MAKER ...) (MAKE AMPLIFIER)
                                <...>
                                (MAKE _...)]]]
                 (:GEN (= NEWOPT#405 ?A2^3)
                    (:RULE-TOGETHER <NEWOPT#405 ?A2^3>
                       [:TO-DO (MAKER ...) (MAKE AMPLIFIER)
                          <...>
                          (MAKE ...)]]]]]]
   0)

;A rather unpromising cascade is suggested:
(RECORDING [:GEN (= OPT#403 NEWOPT#405)
              (:RULE-TOGETHER <OPT#403 NEWOPT#405>
                 [:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER)
                    <'(WINNER ...)>
                    (MAKE (CASCADE CC (CASCADE CC CE)))]]]]
   0)
(RECORDING [:RULE-TOGETHER <OPT#403 NEWOPT#405>
              [:TO-DO (MAKER (*DES*)) (MAKE AMPLIFIER)
                 <'(WINNER (*DES*))>
                 (MAKE (CASCADE CC (CASCADE CC CE)))]]]
   0)
;but for some reason vanishes.  (Notice that the rule should be,
; "If x was suggested because of its input impedance and y wasn't,
; cascade them."  Then this cascade would never have been suggested.)

;The /:RULE-TOGETHER causes the old options to be flushed
(FLUSHED [:CONSEQ (:OPTION CHOICE#402 OPT#404 [:TO-DO (MAKER (*DES*))
                                                  (MAKE AMPLIFIER)
                                                  <'(WINNER ...)>
                                                  (MAKE CE)]]
            FALSE])
(FLUSHED [:CONSEQ (:RULE-TOGETHER <OPT#403 OPT#404>
                     [:TO-DO (MAKER (*DES*))
                        (MAKE AMPLIFIER)
                        <'(WINNER ...)>
                        (MAKE (CASCADE CC CE))]]
            FALSE])
(FLUSHED [:CONSEQ (:OPTION CHOICE#402 OPT#403
                     [:TO-DO (MAKER (*DES*))
                             (MAKE AMPLIFIER) <'(WINNER ...)>
                             (MAKE CC)]]
            FALSE])
(FLUSHED [:ANTEC (NOT (:OPTION CHOICE#402 ?A2^4
                        [:TO-DO (MAKER (*DES*))
                           (MAKE AMPLIFIER) <'...>
                           (MAKE _?+DT2^4)]]]
            (:GEN (= OPT#403 ?A2^4)
```

```
                    (:RULE-TOGETHER <OPT#403 ?A2^4>
                         [:TO-DO (MAKER (*DES*))
                              (MAKE AMPLIFIER) <'...>
                              (MAKE (CASCADE CC _...))]]]]]]
(FLUSHED [:CONSEQ (:RULE-TOGETHER <OPT#403 NEWOPT#405>
                       [:TO-DO (MAKER (*DES*))
                            (MAKE AMPLIFIER) <'(WINNER ...)>
                            (MAKE (CASCADE CC (CASCADE CC CE)))]]]
             FALSE]]
(CHOICE CHOICE#402 DONE)

;The choice successfully reduced the design problem:

(TASK [MAKER (*DES*)] REDUCED TO [MAKE (CASCADE CC CE)])
(CREATING TASK [:TASK G0248 <> (LAMBDA NIL
                                    (MAKE (CASCADE CC CE)))
                    <'(WINNER (*DES*))>])
(NEW TASK [G0248] HAS ACTION [MAKE (CASCADE CC CE)])
(ENABLED [G0248])
(EXECUTING [G0248] [MAKE (CASCADE CC CE)]
   [<'(WINNER (*DES*))>])
(TASK [G0248] BEING REDUCED)

;There is a standard plan for doing cascades:
(TASK [G0248] REDUCED TO [:DO-SUBNET (CASCADE-PLAN CC CE)
                              <CASCADE-NAME>]) !
(CREATING TASK [:TASK (MAKER-1 PLAN#406)
                    <> (LAMBDA NIL (MAKE CC))
                    <'(FIRST-DEV PLAN#406)>])
(CREATING TASK [:TASK (MAKER-2 PLAN#406)
                    <> (LAMBDA NIL (MAKE CE))
                    <'(SECOND-DEV PLAN#406)>])
(CREATING TASK [:TASK (GRABBER PLAN#406) <>
                    (LAMBDA NIL
                       (GRABBA (LAMBDA (X)
                                    (MAIN-DEV-TYPE ?X (CASCADE CC CE)))))
                    <'(CASCADE-NAME PLAN#406)>])
(CREATING TASK [:TASK (COUPLER PLAN#406)
                    <'(FIRST-DEV PLAN#406) '(SECOND-DEV PLAN#406)>
                    (LAMBDA (D1 D2) (COUPLE ?D1 ?D2))
                    <>]) !
```

At this point a bug in the specification for the cascade plan caused the system to crash. (This would have been caught by a syntax checker of the kind I mentioned above.) In any case, the system currently lacks knowledge of common-collector circuits and constraint analysis, so it could not have gone much further.

V.B.2 Converting a Square Wave into a Sine Wave

In this section I present the somewhat more disappointing behavior of DESI on the job of converting a 1 kHz square wave into sine wave of the same frequency, expressed as follows

```
(design
   (\ (ckt)
     (convert ?ckt
        (\ (in)
          (and (periodic (tfun ?in) 1.0E-3)
               (forall (t)
                  (and (implies (/< ?t 0)
                                (= ((one-period (tfun ?in)) ?t)
                                   1))
                       (implies (not (/< ?t 0))
                                (= ((one-period (tfun ?in)) ?t)
                                   -1)))   ))   )
        (\ (in out)
          (= (tfun ?out)
             (\ (t) (sin (* 2000 pi ?t))   ))   ))   ))   )
     <'(filter)>])

;The initial part of the sequence is just as in Sect. V.B.1
(CREATING TASK
     [:TASK (*DES*) <>
        (LAMBDA NIL
           (DESIGN (LAMBDA (CKT)
                     (CONVERT ?CKT (LAMBDA ...) (LAMBDA ...)))))
                <'(FILTER)>]) !
(ENABLED [(*DES*)])
(EXECUTING [(*DES*)]...)
(TASK [(*DES*)] BEING REDUCED)
(TASK [(*DES*)] TO BE REPHRASED)
(CREATING TASK [:TASK (REPHRASER (*DES*))
                   <>
                   (LAMBDA NIL
                      (:REPHRASE (*DES*) [DESIGN (LAMBDA ...)]
                        <'(FILTER)>))
                   <>])
(ENABLED [REPHRASER (*DES*)])
(EXECUTING [REPHRASER (*DES*)]
   [:REPHRASE (*DES*) [DESIGN (LAMBDA (CKT)
                                (CONVERT ?CKT (LAMBDA ...)
                                   (LAMBDA ...))))]
      <'(FILTER)>]
   [<>])
(TASK [REPHRASER (*DES*)] BEING REDUCED)
(TASK [REPHRASER (*DES*)] REDUCED TO
```

```
   [:DO-SUBNET (DESI-REPHRASE-PLAN
                  [(LAMBDA (CKT) (CONVERT ...))] (*DES*) '(FILTER))
                   <>]) !
 ...
;I have elided the messages regarding setting up the rephrasing
; network.  (They are the same as for the preceding example.)
 ...
(EXECUTING [ACCOUNT-FOR-ALL PLAN#392]
   [ACCOUNT-FOR-ALL-SHARDS
      [(LAMBDA (CKT) (CONVERT ?CKT (LAMBDA ...) (LAMBDA ...)))]]
   [<>])
(TASK [ACCOUNT-FOR-ALL PLAN#392] BEING REDUCED)
(TASK [ACCOUNT-FOR-ALL PLAN#392] REDUCED TO [:PRIM *SETUP])
(ENABLED [EXPLODER PLAN#392]) !
(EXECUTING [EXPLODER PLAN#392]
   [D-EXPLODE [(LAMBDA (CKT) (CONVERT ?CKT (LAMBDA ...) (LAMBDA ...)))]]
   [<>])
(TASK [EXPLODER PLAN#392] BEING REDUCED)

;Explosion begins as before...
(TASK [EXPLODER PLAN#392] REDUCED TO [:INFER '(D-SHARD [(LAMBDA ...)]
                                                      [(LAMBDA ...)])
                                             <>])
(INFERENCES MADE BY [EXPLODER PLAN#392]
   --)
(RECORDING [D-SHARD [(LAMBDA (CKT)
                        (CONVERT ?CKT (LAMBDA ...) (LAMBDA ...)))]
               [(LAMBDA (CKT)
                   (CONVERT ?CKT (LAMBDA ...) (LAMBDA ...)))]]
   0)

;But this time the main shard is too hairy to be handled by STP,
; so a subtask is set up

(RECORDING [:TASK T1!379/2344 <>
              (LAMBDA NIL (CVT-EXPLODE [(LAMBDA ...)] [(LAMBDA ...)]))
             <>]
   0)
(CREATING TASK [:TASK T1!379/2344 <>
                   (LAMBDA NIL (CVT-EXPLODE [(LAMBDA ...)]
                                            [(LAMBDA ...)]))
                   <>])
(RECORDING [:SUBTASK T1!379/2344 (EXPLODER PLAN#392)]
   0)
(RECORDING [:MAIN T1!379/24 (EXPLODER PLAN#392)]
   0)

;The same rule <*CONVERT-EXPLODE> also sets up a rule to infer
; SIG-TRANS d-shards from the "signal features" that fall out
; of the convert explosion...
(RECORDING [:ANTEC (NOT (SIG-FEATURE [(LAMBDA ...)] [(LAMBDA ...)]
                             ?+FEATURE^47))
               (D-SHARD [(LAMBDA (CKT) (CONVERT ...))]
```

```
                [(LAMBDA (CKT) (SIG-TRANS CKT _...))]]
   0)
(*INFERENCES DONE*)

;Work begins on this subtask
(ENABLED [T1!379/2394])
(EXECUTING [T1!379/2395]
    [CVT-EXPLODE [(LAMBDA (IN) (AND (PERIODIC ... 0.001)
                                    (FORALL ...)))]
                 [(LAMBDA (IN OUT) (= (TFUN ...) (LAMBDA ...)))]]
   [<>]) !
(TASK [T1!379/2395] BEING REDUCED)

;But there is a choice whether to look for frequency-domain
; or time-domain features of the signals
(MAKING A CHOICE)
(RECORDING [:CHOICE CHOICE#410 EXEC
              [:TO-DO T1!379/4711
                  (CVT-EXPLODE [...] [...]) <>
                  ?WAY]]
   0)
(RECORDING [:ANTEC (NOT (:OPTION CHOICE#410 ?A1^3
                           [:TO-DO T1!379/4711 (CVT-EXPLODE ...)
                              <>
                              (FREQ-DOMAIN-REPHRASE ...)]))
              (:ANTEC (NOT (:OPTION CHOICE#410 ?A2^3
                              [:TO-DO ...]))
                 (AND (:GEN (NOT (AND ...)) (:RULE-IN ?A1^3))
                    (:GEN (NOT (AND ...)) (:RULE-IN ?A2^3))
                    (:GEN (NOT (AND ...))
                       (:RULE-OUT ?A2^3)))))]
   0)
(RECORDING [:OPTION CHOICE#410 OPT#411 [:TO-DO T1!379/4711
                                          (CVT-EXPLODE [...]
                                             [...])
                                          <>
                                          (TIME-DOMAIN-REPHRASE [...]
                                             [...])]]
   0)
(RECORDING [:OPTION CHOICE#410 OPT#412 [:TO-DO T1!379/4711
                                          (CVT-EXPLODE [...]
                                             [...])
                                          <>
                                          (FREQ-DOMAIN-REPHRASE [...]
                                             [...])]]
   0)
(RECORDING [:ANTEC (NOT (:OPTION CHOICE#410 ?A2^5
                           [:TO-DO T1!379/4711 (CVT-EXPLODE ...)
                              <>
                              (TIME-DOMAIN-REPHRASE ...)]))
              (AND (:GEN (NOT (AND (:= ...) (NOT ...)))
                     (:RULE-IN OPT#412))
                 (:GEN (NOT (AND (:= ...) (NOT ...)))
```

```
                    (:RULE-IN ?A2^5))
                 (:GEN (NOT (AND (:SUBTASK ...) (:TASK-ACTION ...)
                                 (:= ... ACTIVE)
                                 (D-FEATURE ...)))
                    (:RULE-OUT ?A2^5))))
   0)

;The choice depends upon what the signal-description predicates
; depend on.  (See <*CVT-CHOICE> in Appendix 3.)
(RECORDING [:GEN (NOT (AND (:= [...] [...]) (NOT (CONTAINS ?+BODY^6
                                                   ...))))
              (:RULE-IN OPT#412)]
   0) !!!

;Frequency-domain is indicated--
(RECORDING [:RULE-IN OPT#412] 0)

(RECORDING [:GEN (NOT (AND (:= [...] [...]) (NOT (CONTAINS ?+BODY^6
                                                   ...))))
              (:RULE-IN OPT#411)]
   0) !!!!!!!!
;The /:GEN found nothing in this case, so time domain gets no
; votes.
;(Checking CONTAINment took a long time, as the row of "!'s"
; attests.  This is a typical example of the slowness of STP
; on a straightforward problem in which it did absolutely no
; combinatorial search.)

;This rule also ended up with nothing:
(RECORDING [:GEN (NOT (AND (:SUBTASK (DEN ...) ?SUP^6)
                           (:TASK-ACTION ?SUP^6 (D-EXPLODE ?+P^6))
                           (:= (:ENAB-STATUS ?SUP^6)
                            . ACTIVE)
                           (D-FEATURE ?+P^6 [RANGER FREQ-OP VERY-HIGH])))
              (:RULE-OUT OPT#411)]
   0)

;So the vote for frequency-domain is decisive...
(CHOICE CHOICE#410 DONE)
(TASK [T1!379/2395] REDUCED TO
   [FREQ-DOMAIN-REPHRASE
      [(LAMBDA (IN) (AND (PERIODIC ... 0.001) (FORALL ...)))]
      [(LAMBDA (IN OUT) (= (TFUN ...) (LAMBDA ...)))]])
;...and execution proceeds
(CREATING TASK [:TASK G0241 <> (LAMBDA NIL
                                 (FREQ-DOMAIN-REPHRASE [(LAMBDA ...)]
                                    [(LAMBDA ...)])))
                  <>])
(ENABLED [G0241])
(EXECUTING [G0241]...)
(TASK [G0241] BEING REDUCED) !
(TASK [G0241] REDUCED TO
    [:SEQ (:FIND (LAMBDA (+FPT)
```

```
                    (EXISTS (FP1 FP2 FPT)
                       (FORALL (S1 S2) (IMPLIES ...)))))
          (LAMBDA (FPT) (:INFER '(SIG-FEATURE ...) <>))])
;The plan is to find frequency pictures and compare them...,
(CREATING TASK [:TASK ITASK#414 <> (LAMBDA NIL
                                     (:FIND (LAMBDA (+FPT)
                                              (EXISTS (FP1 FP2 FPT)
                                                 (FORALL ...)))))
                    <'(FPT#413)>])
;...then infer signal features.  (See <*FREQ-DOM-REPH-DO-1>.)
(CREATING TASK [:TASK MTASK#423 <'(FPT#413)>
                    (LAMBDA (FPT)
                       (:INFER '(SIG-FEATURE ...)
                          <>))
                    <>])
(ENABLED [ITASK#414]) !
 ...
(BLOCKED [MTASK#423])
(EXECUTING [ITASK#414]...)
(TASK [ITASK#414] PRIMITIVE)

;/:FIND is the user's way of calling STP.
;Here is what the STP trace looks like for this problem:
(STP TRACE 1 0 [NOT (IMPLIES (AND (IS SIGNAL S1!433/3330)
                                  (AND (PERIODIC (TFUN ...)
                                               0.001)
                                     (AND (IMPLIES ...) (IMPLIES ...)))
                                  (IS SIGNAL S2!434/3330)
                                  (= (TFUN S2!434/3330)
                                     (LAMBDA (T)
                                        (SIN ...))))
                           (AND (=> '(FREQ-PICTURE ...)
                                    ?FP1)
                                (=> '(FREQ-PICTURE ...) ?FP2)
                                (FREQ-PIC-TRANS ?FP1 ?FP2
                                   ?FPT)
                                (=> '(DEN ...) ?FPT)))]
  NIL) !!!!
```

Unfortunately, a bug in a very low-level routine caused an infinite recursion in the midst of this attempted proof. Therefore, the system never got to the point of actually generating or comparing frequency pictures.

V.B.3 NOAH in NASL

Jon Doyle and I have done some preliminary experiments toward a "free translation" of Earl Sacerdoti's (1975) NOAH program into NASL. As I discussed in Chapter I, NOAH and NASL are based on rather different presuppositions, so an exact translation would be somewhat contrived. NOAH is organized around repetitive execution of a strict sequence of steps of the form, "Expand the plan; criticize it." After the plan has been entirely reduced to primitives, it is executed. In carrying out these steps, the NOAH system assumes that all actions' effects are fully computable in advance; it reasons confidently about future states of the world. This assumption is false for many of the actions NASL tries to accomplish.

Nonetheless, the parallels between the two systems are tempting. We wondered if it was possible to encode NOAH "critics" as NASL "policies." The critics we concentrated on were "Resolve Conflicts," which orders actions to prevent one from undoing the prerequesites of another; and "Eliminate Redundant Preconditions," which attempts to prevent the same action being done twice for two different reasons.

We have done some preliminary coding (it only takes about 5 pages of NASL expressions), but the unsettled state of the interpreter has made this mainly a ***Gedanken*** experiment. The results so far are mixed. On the one hand, it takes very little effort to express as deductive "mini-theories" much of what is meant by concepts like "prerequisite" in a system like Sacerdoti's which has them built in.

On the other hand, we had some disappointments. It is more difficult than I had hoped to distinguish problem reduction from execution. NASL assumes that a network can be executed as soon as it is generated; to force it to be

more NOAH-like requires the user to write explicitly the theory of elaboration levels that is apparently built into the NOAH elaborate-criticize loop. The user must explicitly tell the system to postpone execution of lower levels until higher levels are reduced. In principle, there is nothing wrong with having to do this, since this is just another mini-theory. What made us a bit squeamish about it was the necessity of ignoring altogether NASL's use of /:MAIN tasks (Sect. II.B.1) in specifying what happens during task reduction, and replacing it with an explicit theory of /:SUCCESSOR relations.

I think it is fair to conclude from this "experiment" that NASL is an worthwhile alternative to NOAH, especially for problems where there is much user-supplied knowledge about plans, and only incomplete foreknowledge of the effects of the basic actions.

VI Conclusions

"This ... may seem trivial,
but I think it is not without importance."
-- Mary Warnock, ***Ethics since 1900***

I set out to construct a circuit designer so flexibly organized that it could respond to all relevant aspects of a design problem, yet directed enough to be efficient during its normal operation. I implemented a rule-based problem solver called NASL and have done preliminary experiments using as my main vehicle DESI and ZORCH, sets of rules embodying theories of design and electronics. The results are consistent with the hypothesis that the organization of NASL has the right kinds of power. As with many experiments in AI, the results are not unequivocal. Our conclusions rest largely on esthetic considerations.

The theories of design and electronics drew heavily on previous work by others. (Freeman and Newell, 1971, Stallman and Sussman, 1976, A. Brown, 1975) There are novel elements. The design process embodies a modified top-down strategy. Domain-dependent information, expressed in a modular way, orders design choices and controls their interaction. When the top-down elaboration reaches the level of canned "device schemata," the task structures stored there become integrated with it. The theories embodied in the programs that make this happen are further steps toward competing with human performance in these areas.

NASL has severe limitations. Due to limited time, I was unable to develop many domain-independent control features, because they were not needed for electronic design. (Some of these limits were encountered in our attempt to study NOAH with NASL. See Chapter V.) The logics of time and belief are practically absent. Hence, I cannot claim that the current system could be

just as easily used, e.g., to understand stories. Even some features important to electronic design, such as describing and correcting mistakes, could not be implemented in the time I had.

Second, the system's flexibility in principle is offset by its lack of patience and skepticism in assimilating what it hears. An untrained user could bring its operation to a halt by telling it the wrong things.

I have had some disappointing failures. The program is too big and slow to be practical, apparently because of the implementation of data-base operations, rather than because of any combinatorial explosion. More substantively, the division of labor between theorem prover and interpreter is in many ways wrong. The decision to use predicate calculus for representing and using knowledge was the major theoretical gamble in NASL's design. This gamble has had wildly equivocal results.

The style of knowledge encoding encouraged by NASL is, in my opinion, quite exciting. These features in particular stand out:

> All control information is explicitly represented in the data base.

> Most dependency information is automatically gathered by the system in a complete and convenient way.

> Plans can be described incrementally. Specification of order and choice depends on rules which can be combined in flexible ways.

> Predicate calculus is used as the notation for control and physical information.

I will discuss first my successes, then my failures, and which way research might go to overcome the limitations I have discovered.

VI.A Successes

In this section I want to put the NASL system in perspective, and argue that it is a step toward understanding mental functions. Fig. VI.1 shows a map of current artificial-intelligence research. It may also be taken as a map of mental functions, with the arrows taken as indicating the flow of information. Either way, the central box with the question mark is a major mystery. We know that this center is concerned with "understanding," "problem solving," and "learning," and we know that it contains many sub-boxes. Much of mainstream AI is concerned with the somewhat speculative pastime of proposing and testing pieces of this box.

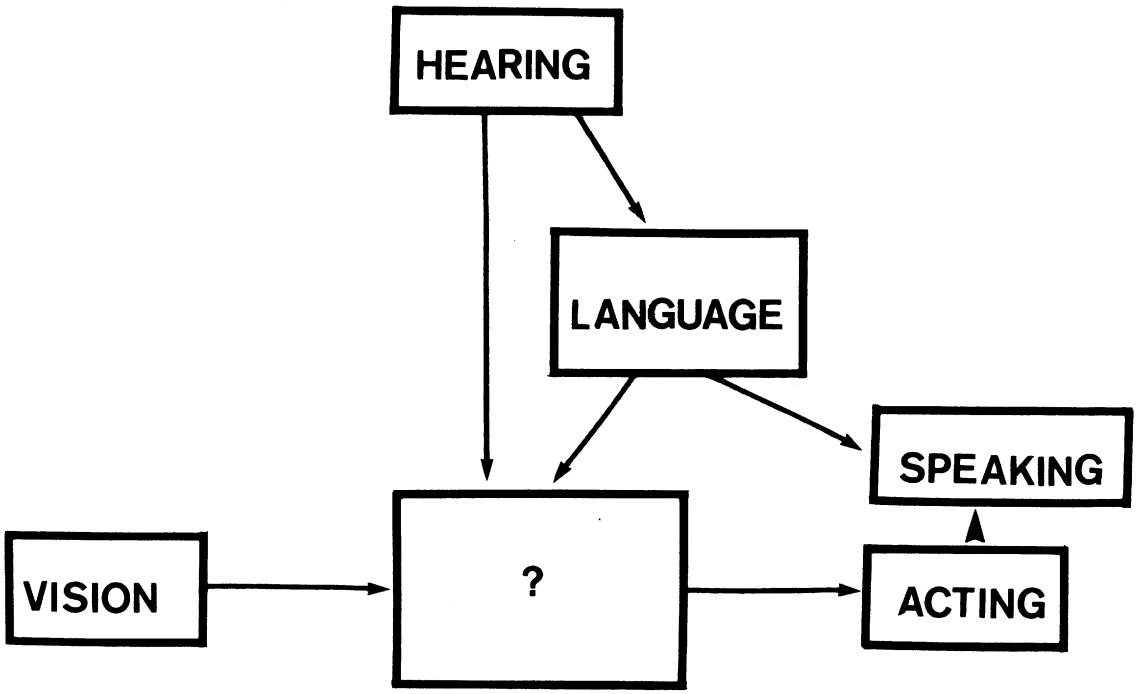

Figure VI.1 Provinces of Artificial Intelligence

The thesis that rule-driven problem solving is an important technique depends, I think, on a picture of the mystery box like that of Fig. VI.2. Normal cogitation is performed by a problem solver accessing a data base of rules. New rules are created by a rule generator; the most direct way is by translation from natural-language statements. The rules are not guaranteed "correct"; they must be revised if faulty by a debugger. (McDermott, 1974a, Sussman, 1975)

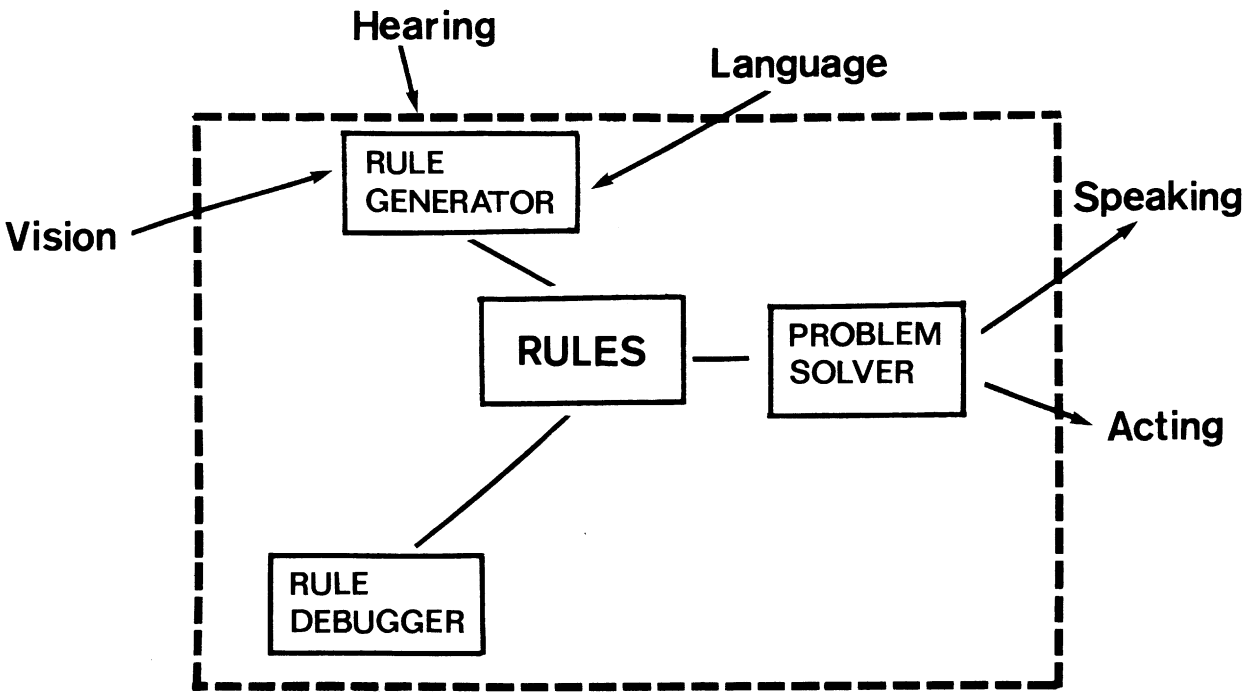

Figure VI.2 The Rule-Based Utopia

To some degree, putting these last two functions in neat boxes is wishful doodling, but the problem-solver box is intended to be real. The NASL system, or some future descendent, resides in this box. To what degree do features of NASL support progress on the rule-driven paradigm? In answering this question, I will survey in detail what I consider the strengths of the current system.

NASL is driven by a general predicate calculus that supports backward and forward deduction. This feature forces the user to think in terms of truth and falsehood, rather than in terms of, e.g., "demonic action." For example (see Chapter II), there is no way to "deduce the erasure" of a formula. Unlike a PLANNER-type system (Hewitt, 1972), NASL treats erasure entirely

differently from recording. This enables more revealing records to be kept. It forces the user to think in terms of ***true*** rules rather than apparently ***useful*** ones, like productions, which might have to be changed later, or might introduce arbitrary symbols with no presumed meaning. (Rychener, 1976, Newell, 1971) This half of Minsky's (1974) "monotonicity" problem is no problem at all, but a valuable kind of discipline.

The notational power of the predicate calculus strikes me as a tool we cannot do without. Much of this power depends on providing a good vocabulary; and, in the realm of control structure, I have done this. But the notation itself is good, in my experience, independently of what it is talking about. It allows you to think in terms of statements with truth values, its treatment of quantifiers doesn't cramp your style, it provides powerful and natural pattern matching, and it forces you to say what you mean.

This formal freedom is necessary to to support restrictions on the substance of rules. NASL formulas are not restricted to talking about clinical parameters or values of physical quantities in a network, but they ***are*** restricted (for practical purposes) in the way they talk about concepts like action, decision, policy, and problem transformation. In order to get something done in NASL, you must express yourself in terms of tasks, "rule-ins and rule-outs," rephrasing actions, etc. The task language is restricted to such a degree (there being no real loops, gotos, or conditionals) that many things can be done only ***via*** these mechanisms.

These conventions enforce "intelligibility" at a useful level. At every step, the system knows by way of quite shallow deductions almost everything there is to know about what it is doing, what its future tasks are, why it chose to do what it did, etc. This knowledge is used heavily in the rules for choosing, rephrasing, and mistake correction. Because the number of innate

concepts has deliberately been allowed to grow, shallow deductions are possible and required; NASL does not have or need a theory of "program understanding." (Waldinger and Levitt, 1974) I believe this property is essential to an intelligent program; it is no accident that the average person is a good planner and a bad programmer.

A most important example of this reliance on innate control concepts is the notion of "frozen tasks" which is so useful in the representation of device schemata. (Chapter III) The instantiation of such a schema causes information regarding the purposes of components to be activated, in the same notation that is used for expressing explicit goals. These old tasks then interact with new ones in determing the solution. In this way, DESI exhibits "machinomorphism." The purpose of a circuit is expressed as a frozen purpose of the machine. No new concept needs to be introduced, and the problem of relating the special-purpose teleology of each domain to the machine's concept of its own purposes is avoided.

This organization of predicate calculus plus large innate vocabulary is potentially of great value to the Rule Debugger module of Fig. VI.2. It is known that debugging a data base requires keeping records of the use of data which might be faulty. (McDermott, 1974a, Davis, 1976) The kinds of records kept by NASL, deductive dependencies and task relations, are just the kinds required.

Another powerful organization principle that emerged during this research was the notion of "packet." (McDermott, 1975) This device enables NASL to represent large chunks of data at a relatively low cost. It is used to represent circuit diagrams and sets of related problem-solving rules.

From a distance, a packet looks like a large chunk of knowledge; up close, it looks like many small pieces of knowledge. It may be used to implement

"frames" (Minsky, 1974) in an "extensible" way. The knowledge is organized by the dependencies I described before, but a new piece of knowledge can always be added without immediate fear of interactions.

This is partly because NASL is organized around the expectation of interactions. It expects that occasionally more than one or less than one rule will appear relevant. In these circumstances, it enters special protocols which first look for relevant higher-level rules, and then ask the user to supply them. Much remains to be done in this area. (Good work has been done already by Davis (1976).) The results so far indicate that the standard feeling that frames' contents are hopelessly "strongly coupled" (cf. Sussman, 1975) is too pessimistic.

The fact that NASL's facts come in large chunks of small data has implications for the design of the Rule Generator of Fig. VI.2. It is a very appealing model of learning by discovery that large bites of information are taken at one time, by copying an entire theory from one domain to another. (Minsky, in progress) What is nice about rule-based theories in particular is that they hint at the next step: altering particular rules that were faultily transformed during the "copy," and adding domain-specific interaction handlers.

The kind of chunking that NASL currently supports would not handle this kind of learning, but it is suggestive. It might be worth making the effort to recast the entire corpus of electronics information as an instantiation of a more abstract (and smaller) theory of, say, common-sense physics. If this were successful, a start at capturing any similar domain would be to reinstantiate this theory in a different way.

The conclusions I have drawn so far can be summarized as follows: (1) A flexible problem solver must have innate concepts of tasks, decisions, and

other similar control concepts; (2) predicate calculus is, at present, the best language for expression of propositions in these terms; (3) the rules expressed in the calculus must be tightly organized, linked by dependencies and bundled into packets.

VI.B Failures

The next questions are somewhat independent: Is a predicate-calculus theorem prover an effective mechanism for retrieval of information expressed this way? In particular, can it be interfaced effectively with the interpreter that uses it to decide how to act? With respect to these questions, the current version of NASL fails to provide satisfying answers.

As it stands now, NASL is organized as follows. (See Fig. I.9.) Control resides in the interpreter, which decides what to do and how to do it on the basis of (a) forward deduction triggered by plan and model assertions in the data pool; (b) backward deduction to find ways of breaking problems down and to answer questions in the domain of interest; (c) calls to the choice protocol, which consists of a ritual of deductions regarding which alternatives are good and which bad; and (d) calls to itself recursively with /:REPHRASE tasks (which are restricted to being inferential). The outstanding features of this organization are:

(1) The action system always calls the theorem prover, never vice versa.

(2) The system contains, in effect, two independent interpreters, one for plans, and one for implications (/:CONSEQ's and /:ANTEC's).

These features distinguish NASL rather sharply from the typical AI languages. (Bobrow and Raphael, 1974)

The strengths of this organization are easy to see. The two interpreters

are optimized separately. For example, the theorem prover does not have to worry about side effects, so it can re-order conjunctive goals and separate goals into classes which share variables for backtracking purposes. (See Appendix 4.) The interpreter, on the other hand, does no backtracking at all; handling a failed action is a ***problem*** for the interpreter, not part of its solution mechanism. It goes to great trouble to find reasons for its choices, rather than just trying one alternative now, and another later if that one fails.

These strengths are pleasing, but they do not outweigh the weaknesses, which are:

(1) The same knowledge must sometimes be put in two places, in two notations, one for each interpreter.

(2) The deduction machinery is unable to use information about choice and rephrasing.

(3) Additivity suffers from the user's uncertainty about which interpreter to use. If he guesses wrong, he may have to reorganize his data completely when the chickens come home to roost.

Consider, for example, the notion of equation solving. DESI has a weak ability to do this (see Appendix 2 and Chapter III), which could be stronger without too much effort. Notice that this information has been expressed as an "inferential plan" concerned with rephrasing manipulations of predicates on constrained quantities. This seems entirely proper, clear, and elegant. Now consider the following deductive goal:

[= (+ ?X 3) 5]

Plainly, this requires exactly the same knowledge. (Cf. Bundy, 1975)

It would be embarrassing enough to have to put the same information in two places, but in fact the situation is even worse: the necessary knowledge cannot be given to STP at all! (Which is why it appears in an inferential rephrasing plan.) It would have to be put into an ad hoc LISP program.

Rather than do this, I have tried to make sure that deductive goals of this sort never appear. The absence of a choice protocol inside the theorem prover hurts just as much. The most embarrassing consequence of this awkwardness is that the user must plan his advice a little more carefully than is desirable; he must decide what should be expressed as a task and what should be expressed as a deductive goal on the basis of his knowledge of the theorem prover's limitations; this requires an unacceptable degree of knowledge of the program's internal structure.

This problem evolved from seemingly innocuous beginnings; what started as a single interpreter fissioned. It has been clear from the start of this work (McDermott, 1974b) that the concepts of deduction and action were both going to be necessary. As design and implementation proceeded, these two categories became more and more closely identified with the independent categories of "blind search" and "careful mode," respectively. The theorem prover was written with fewer and fewer "careful" features, and more and more general optimization of the sort described above, while the action system absorbed the cleverness. This organization finally broke down when I realized that several sorts of "clever" forward deduction, such as constraint resolution (Stallman and Sussman, 1976), would not fit into the framework of a stupid theorem prover (STP). The inferential plan was created to fill this gap. It may turn out to be the right solution (see below), but if it does it will be an accident.

To some degree this failure is due to sloppy thinking, but I believe the problem is more fundamental. The only way to make "careful mode" work is to spend time asking and telling yourself what you're doing. If this asking and telling is itself "careful," things become intolerably slow.

It might look as if asking and telling could be ***potentially*** careful, in

the sense that they could normally proceed blindly, but, if trouble arises, turn themselves into ongoing plan-language plans. For example, with the conjunctive goal [AND (P ?X) (Q ?X)], if the system "runs into trouble" on [Q ?X], it could turn itself into a plan of the form "Find a P; then verify it is a Q," and use choice and rephrasing on the second subtask. There are two problems with this. First, it is not all that easy to decide that you're in trouble. The mere retrieval of two rules does not mean that a choice is in order; the two rules could be functioning as a COND, or there may be no intelligent criterion for selecting one or the other. Indeed, once one has the notion that the theorem prover is the locus of "blind search," he tends to write rule systems of just this sort. However, I believe that the "meta-rule" construct of MYCIN (Davis *et. al.*, 1975) would go far toward solving this problem cheaply.

Second, and much more difficult, is that the kind of sequencing done in a backtracking theorem prover is just not the same as that in a planner. Predicate calculus is a non-deterministic language; it does no good to translate it into a formally isomorphic construction in a deterministic language. Put another way, NASL is intelligible because many unintelligible constructs have been covered by deduction or other built-in protocols: "map" loops like those in LISP are done by generating items deductively and generating a (sub)task for each; many search loops are done by finding one such item; the choice protocol is a priceless piece of "canned loop" which replaces specialized discrimination nets. To ask that any of these constructs be translatable when necessary into NASL plans is to destroy this intelligibility.

The situation is not hopeless; I have just learned less about this aspect than I had hoped. Here are two possible routes for avoiding this problem:

(1) Elevate this disorganization to the status of a theoretical conjecture that "deductive information retrieval" is a separable activity that never requires anything as complicated as solving an equation or rephrasing a goal. (This is part of what R. Moore (1975) and Fahlman (1975) have been urging.) Inferential plans would be retained for these complex tasks.

(2) Provide deductive protocols analogous to those used by NASL. Dispense with inferential plans. This will require careful identification of situations where the protocol would be worth it (such as: choosing among answers to a conjunctive goal, ordering conjuncts, choosing splits, ordering implications to apply); and a way of efficiently noticing when there are no applicable rules, in which case brute-force deduction is to be used.

The substantive difference between these alternatives is that alternative (1) makes complex inference a kind of task, and hence deterministic, saving search for the stupid theorem prover; while alternative (2) makes even complex inference subject to backtracking, which is modified by the application of rules.

VI.C Further Work

Let me list, in order of increasing difficulty, projects that are worth doing to extend this research. Some of them I may do myself.

(1) Encode more electronics knowledge. The gaps in ZORCH are described at the end of Chapter III.

(2) Speed the system up. The system can undoubtably be made much faster by abandoning some of my more elegant programming techniques.

(3) Implement the error-handling machinery I described in Chapters II and III. This will require careful reconsideration of data dependencies.

(4) Unleash the logical calculus. There are restrictions on NASL's generality which I believe are due mainly to inadequate implementation of the logical calculus. There are many domains which are beyond its grasp because it lacks a notation for things like time, belief, and the actions and beliefs

of other people. To some degree these areas could be handled by the use of modal reference points for time intervals and other people's minds. The system could simulate other persons' thought processes with its own theorem prover, and avoid some of the problems associated with epistemic logic. (Hintikka, 1962) Broadening the syntax of "/:TASK" to include an agent slot would be a step toward representing other persons' purposes; the interpreter could use itself to simulate them as a way of understanding their actions. (Cf. McDermott, 1974a) However, there is an "abductive" component to such reasoning (Pople, 1973, Schank, 1975, Lehnert, 1975) that is beyond the capability of NASL without more substantive revision.

(5) Endow the system with more "patience and skepticism." The greatest weakness of NASL as it now stands is its credulity. It accepts wrong rules (even syntactically wrong ones) as readily as right ones. This is unsatisfactory for a system which already understands its domain thoroughly; surely one task it should be able to carry out is to estimate the plausibility of a new rule in the presence of its old ones.

In some simple cases, a new formula may be disproved. In that case, the system would be in an excellent position to claim that something was wrong. Unfortunately, this is not likely to happen very often. The less important reason for this is that STP is not built to do interesting proofs. The more important reason is that many theorems have conclusions defined only pragmatically, by their meaning to the NASL interpreter. These are the formulas of the form "A1 and A2 and ... and Ak imply C," where C is in terms of concepts having to do with choosing (e.g., "/:RULE-IN") or acting (e.g., "/:TASK"). These concepts are in a sense primitive; we want to define "good" and "feasible" in terms of these concepts rather than vice versa. Thus, I said that [/:TO-DO |task| |action| |outputs| |method|] meant, among other

things, that the method was an effective, feasible, and permitted way of carrying out the task. But, since there is no independent theory of these concepts, a /:TO-DO implication cannot, except in the most trivial cases, be disproved by showing its method would not be permitted (or feasible or effective) under the circumstances. Still another problem is that a typical conditional of this sort is counterfactual; one of the antecedents is probably false at the time the rule is heard, making a proof trivial. To disprove this rule, the system would have to prove a modal theorem to the effect that "there exists a 'possible world' in which the antecedents are true and the consequent false."

The solution to these problems is to integrate the theory of action failure with the theory of assimilation of new information. In the early stages, this will be probably require the co-operation of a human friend. (Shortliffe, 1976) The idea is to place a new formula or set of formulas on "probation." (McDermott, 1974a) When a contradiction, action failure, or inability to choose occurs, the system will check the formulas involved to see which are on probation and might contain errors. The idea is to see how things would work out differently if the formula were not there. If, for example, a choice fails because all alternatives are eliminated, and there is a formula on probation involved whose absence would have left some alternatives in, the system is justified in asking for clarification of the new rule.

Notice that this requires an "advice-taking protocol" for each class of rules, that is, for each pragmatic predicate the system knows. It would be attractive if these were plan networks; and if the advice-taking actions in certain circumstances could be framed as policies.

(6) Add a natural-language interface. This is difficult in itself, and,

in addition, its impact on the assimilation machinery I outlined is unclear. Users will make fewer mistakes of notation if they use their own language, but the language interface will inevitably pass ambiguities through for the assimilation machinery to worry about.

(7) Add a theory of learning so that the system will not forget its more brilliant insights.

Appendix 1 -- NASL Syntax and Informal Semantics

A ***formula*** is an S-expression enclosed in [brackets]. Redundant parentheses may be dropped. Thus [(IS RESISTOR R#21)] is written [IS RESISTOR R#21].

The leftmost element of a formula or subpattern is its ***function***. Functions with range {***true, false***} are ***predicates***. The Boolean functions AND, OR, IMPLIES, and NOT operate on truth values.

Besides functions and their arguments, there are "***variable binders***" whose job is to indicate the names and uses of variables in formulas. These are the universal quantifier FORALL, the existential quantifier EXISTS, and LAMBDA, which defines functions and is used for all other variable-binding chores. (Lambda may be typed as "\" ("backslash") to my LISP system. I will use this symbol instead of "λ" throughout the appendices.) Thus the following are NASL expressions:

```
[FORALL (X) (EXISTS (Y) (LOVES ?X ?Y))]
[FORALL (X) (SATISFIES ?X
                (\ (Y) (EXISTS (Z) (P ?X ?Y ?Z)   ))))]
```

Variables are flagged with a "?" where used, but not where bound.

Many variables are not bound at all. As in most predicate calculus-oriented systems, all formulas are Skolemized (Nilsson, 1971) before being put in the data base, so that there are no quantifiers at the "top level" of an expression. (Expressions remain quantified inside lambda expressions and as arguments to function.) Free (universally quantified) variables remain prefixed with a question mark. A "***skolem form***" represents an existentially quantified variable. Skolem forms look like

```
[SK |var| |id number| -dominating universals-].
```

For example, [FORALL (X) (EXISTS (Y) (LOVES ?X ?Y))] is internally represented as [LOVES ?X (SK Y 70 ?X)]; while [EXISTS (Y) (FORALL (X) (LOVES ?X ?Y))] is represented as [LOVES ?X (SK Y 71)]. The program generally abbreviates the general skolem form to "|var|!|id number|" on output; e.g., (SK Y 70 ?X) is printed Y!70. I occasionally use this loose notation. (To avoid collision, a hash number derived from the skolem-form arguments is usually printed following the variable.)

Because quantifiers are retained inside lambda expressions, the example of a lambda expression above is skolemized to

```
[SATISFIES ?X (\ (Y) (EXISTS (Z) (P ?X ?Y ?Z)   ))]
```

An important concept in predicate-calculus systems is ***matching***, or ***unification***, of two formulas. (Robinson, 1965) Two formulas are said to ***match*** if there is a substitution for their variables which makes them equal. The variables are to be imagined subscripted with the name of the formulas they came from, to avoid confusion. Thus [P ?X (F ?V)] matches [P (F ?X) ?X] with the substitution

$$X_1 \rightarrow [F\ (F\ ?V_1)]$$
$$X_2 \rightarrow [F\ ?V_1]$$

Internally, substitutions and subscripts are handled using a method derived from (Boyer and Moore, 1972). (See Appendix 4.)

There are two special cases of matching. F_1 ***subsumes*** F_2, if F_2 is equal to the result of performing a substitution on F_1. F_1 and F_2 are ***variants*** if they subsume each other; alternatively, if renaming the variables of F_1 makes it equal to F_2.

These concepts are essential to the operation of the deductive system. (Appendix 4.)

The matcher is not intended to be a complete unification algorithm for *n*th-order logic, typed λ-calculus logic, etc. A lambda expression will match another lambda expression if their variables differ only in name. Of course, free variables may not become bound to fragments of lambda expressions containing bound variables. Thus [P (\ (X Y) (F ?X (G ?Y)))] will not match [P (\ (U V) (F ?U ?W))]. The matcher will not create lambda-expressions without prodding. (See below.) Thus, [?F A] doesn't match [G (H A A)] with F → [\ (X) (G (H ?X ?X))] or any of the alternatives.

The language allows formulas to refer to other formulas. Thus [ABSENT [BROWN COW#22]] expresses a property of [BROWN COW#22]. This is one of two ways in which NASL expressions may refer to other expressions. (It may be considered equivalent to, but more convenient than, the use of Goedel numbers of formulas.) It has the following variants. First, every user-defined formula has an atomic name. (See the description of DEFMLA in Appendix 2.) If FMLA#99 is the name of [BROWN COW#22], we may write [ABSENT FMLA#99]. Second, an embedded formula may have variable parts, called ***escape forms***, which are prefixed by "_"; this indicates that the value of the prefixed expression (which should be a formula) is to be used to construct the formula. For example, if FUN is a function that maps a formula into its first subformula, [FOO [BAR _(FUN [F A]))] = [FOO [BAR F]]. Escape forms are most useful in conjunction with variables. Thus [FOO [BAR _?FMLA]] says, "For all formulas ?FMLA, the formula obtained by making the pattern of ?FMLA the argument of BAR has property FOO." (Each such embedded formula is equivalent to some open term, such that substituting Goedel numbers for its free variables gives a closed term whose value is a Goedel number.)

Matching embedded formulas against formulas with escaped variables is a way of decomposing formulas. This is used by some of the "meta-systems" of NASL. For example, the result of matching [P [FOO _?X]] against [P [FOO BAR]] is the substitution X → [[FOO BAR]].

For ease of manipulation of formulas, the primitive function DEN is understood by STP to map formulas onto what they denote. Thus [DEN [+ 5 5]] = [+ 5 5]. One convention I use is that variables ranging over formulas start with the character "+"; thus ?+X might designate [[FOO]] and ?X designate [FOO]. This is purely a convention and not part of the language.

Notice that all NASL formulas in this paper are surrounded by [brackets]. For example, in the result of matching [P ?X] against [P (FOO BAR)], I write, "?X has value [FOO BAR]," even though ?X was originally matched against a subpattern without brackets. This enables you to tell unambiguously which formulas are being used and which quoted. (Actually, when it comes to atomic symbols, I rarely make the distinction between a symbol and a formula. I allow myself to drop the brackets in a formula like [FOO].)

The "sense" construct using single quote is the second way in which NASL expressions may refer to other NASL expressions. It allows one to refer to the "meaning" of an expression and not its value. For example, even if the value of [= NIXON PRESIDENT] is false, [POSSIBLY '(= NIXON PRESIDENT)] may be

true.

Substitution of equals is, of course, prohibited inside any embedded formula or sense.

The operator POSSIBLY is an example of a ***modal*** predicate. (Hughes and Cresswell, 1972, Bressan, 1972) The basic system-supported modal operator is [T |reference-point| |pattern|], meaning the value of pattern in "possible world" reference-point. (Prior, 1967) The second argument is implicitly quoted. Thus [T (1970) PRESIDENT] would have value NIXON; and [T (1970) (= NIXON PRESIDENT)] would have value ***true***.

NASL contains tuples like those of QA4 (Rulifson ***et. al.***, 1972). They are represented using <angle brackets>. Within a tuple, the prefix "!#" means that the value of what follows is to be considered spliced in instead of substituted directly. Such an expression is called a ***segment form.*** For example, if [FOO BAR] = [<A B C>], [<P (FOO BAR) Q !#(FOO BAR) R>] = [<P <A B C> Q A B C R>]. A similar notation, "!#_", is used inside embedded formulas. If [WHIZ BANG] = [<[A] [B] [C]>], [[P !#_(WHIZ BANG) Q]] = [[P A B C Q]].

Segment forms make matching more complicated. Strictly speaking, these two formulas

[P <!#?X !#?Y>] and [P <A B>]

should match three ways

{X → [<>], Y → [<A B>]},
{X → [<A>], Y → [<B>]},
and {X → [<A B>], Y → [<>]}.

My matcher is too lazy. Occasionally this means deductive formulas have to be framed in terms of list operations instead of in the most concise style.

Semantics

While I am in sympathy with Hayes's (1974) contention that the semantics of a representation is very important, the subject seems much too complicated for practical representation schemes. NASL is a modal calculus, which should have an attractive model theory like Bressan's (1972). However, operators like "/:CONSISTENTLY" ruin it. Furthermore, there is a pragmatic component to many predicates which could not be expressed model theoretically. For example, "/:CONSEQ" and "/:ANTEC" both mean "OR," but they are used in different ways. Consequently, the most precise description of the meaning of the language is a description of STP (Appendix 4) to account for the strictly semantic "meaning" of a symbol; and the following index of pragmatic predicates with an informal description of the pragmatic meaning the system assigns to each one.

Here is a list of built-in, pragmatic predicates, with an informal description of each.

Predicates and Functions with Meanings to the Interpreter

Task Specification and Relation

[/:TASK |name| < -input pvars- >
(\ (-vars-) |action|)
< -output pvars- >]

means task name, which consists of doing action with the values of the input pvars substituted for the lambda-variables, is worth doing. It will produce output values to be bound to the output pvars.

[/:SUBTASK |task name 1| |task name 2|]
[/:SUCCESSOR |task name 1| |task name 2|]

These relate tasks. A task will not be started until all its super tasks have enablement-status SUBS-ENABLED and its predecessors have enablement-status SUCCS-ENABLED. These assertions may therefore be used to direct the flow of control.

[/:TASK-STATUS |task name|]
[/:ENAB-STATUS |task name|]
[/:TASK-ACTION |task name| |action|]
[/:REDUCED |task name|]
[/:ELABORATED |task name|]
These define the control state of a task as discussed in Sect. II.B.1.

[/:POLICY |task name| |action|]
[/:SCOPE |secondary task name| |primary task name|]
These functions are define policies. When a policy task has begun, it is declared to be a /:POLICY, usually with some /:SCOPE. It will be explicitly finished with a /:FINISH task. (See below.)

Primitives

[/:MOD-MANIP |task name| |action| |deletelist| |addlist|]
[/:MONITOR |formula| (\ (|v|) |action|)]
[/:SET '|expression| |value|]
These are the non-macro worldly primitives. /:MOD-MANIP defines the deletelist and addlist of the given action. /:MONITOR creates a policy of looking for the erasure of formula and creating a task with the given action. (The variable v will be bound to the task that did the erasing.) /:SET is used to change or set the value of the expression; this should be a model quantity like voltage or resistance, ***not*** a pvar. Its effects are supported as though they were model manipulations.

[/:INFER '|proposition| < -task names- >]
[/:FIND (\ (|v_1| ... |v_n|) |exp|)] ==> <|pv_1| ... |pv_n|>
[/:FIND-ALL |property|] ==> <|pvar|>

are assigned to the pvars. If a choice of answers is required, this will be reflected in the data-dependency supporting these values. /:FIND-ALL takes a property of one argument, and returns a tuple of all the objects which satisfy it. /:EVAL calls the evaluator and returns the value.

/:INFER is used to write special inference rules. The proposition given is recorded, supported by the propositions recorded by the specified task names. For example, the following task net does the obvious

```
[/:TASK MINOR <> (\ () (/:FIND (\ () (MAN SOCRATES)))) <>]
[/:TASK MAJOR <>
    (\ () (/:FIND (\ () (IMPLIES (MAN ?X) (MORTAL ?X))))) <>]
[/:TASK CONCLUSION <>
    (\ () (/:INFER '(MORTAL SOCRATES) <MINOR MAJOR>)) <>]
```

```
[/:OUTPUT < -vals- >] ==> < -pvars- >
[/:PRIM |type|]
```

/:PRIM defines a primitive action; type should be one of *FINISHED, *SETUP, or *BEGUN. *FINISHED means the action is done; *SETUP means the action is a policy whose successors may now be enabled; *BEGUN means the action is a policy whose successors may not be enabled until the policy is /:FINISHED. /:OUTPUT is like [/:PRIM *FINISHED], but in addition the values are returned to be made pvar values. Thus, the task [(FOO)] in

```
[/:TASK (FOO) <'(PV1)> (\ (V) (/:OUTPUT <?V>)   ) <'(PV2)>]
```

sets [(PV2)] to the value of [(PV1)] when it becomes available.

```
[/:CONTINUE |task name| |action|]
[/:FINISH |task name| |action|]
```

These functions are used to control policies. When all the primary subtasks of a policy's scope have been finished, a /:FINISH task will be created as a subtask of the policy; it is up to the user to supply rules to reduce it. The user may also execute actions of the form [/:CONTINUE |policy-task| |action|] to perform intermittent execution of the policy. (See Sect. II.B.1.)

Macro Primitives

```
[/:DO-SUBNET |plan schema| |vars map|]
[/:PLAN-INSTANCE |plan instance| |plan schema| |supertask|]
[/:MAIN |subtask| |supertask|]
```

As explained in Chapter II, these formulas are used in attaching standardized subnetworks to the current plan.

```
[/:SEQ |action 1| (\ ( -vars- ) |action 2|   )]
[/:FORK |action 1| (\ ( -vars- ) < -actions- >   )]
[/:WHILE |primary action| < -secondary actions- >]
[/:DO-ALL < -actions- > |action|]
```

These macros elaborate into various standard structures. /:SEQ turns into a net of two tasks, the first of which feeds pvars to the second; the outputs of the second are the outputs of the /:SEQ. /:FORK produces no pvar values; it sets up one task per action, action 1 being the predecessor of each of the other tasks. The values of action 1's outputs are fed to the successor tasks. /:WHILE starts all the secondary actions as policy tasks with /:SCOPE equal to the task for the primary action. /:DO-ALL carries out all the actions in no

particular order, outputting the values of action's pvars.

Task Reduction

[/:TO-DO |task name| |action 1| |output pvars| |action 2|]
means, "action 2 is an effective, permitted, and feasible way of doing the task named which consists of action 1 and outputs the given pvars." Deducing formulas of this kind is the first resort in reducing problematic tasks.

[/:REPHRASE |task name| |action formula| |output pvars|]
This action, which must be reduced by user-supplied rules, is set up when /:TO-DO deduction fails. See Sect. II.C.2. Its object is to leave the task /:REDUCED.

Predicates with Meanings to the Choice Protocol

[/:CHOICE |choice name| |context| |formula|]
means a task or the executive (context) requires a choice regarding answers to formula. The choices are recorded by formulas like
[/:OPTION |choice name| |option name| |answer|]

[/:RULE-OUT |option|]
[/:RULE-IN |option|]
[/:RULE-TOGETHER < -options- > |new formula|]
These three kinds of formulas are looked for repeatedly, ***in this order on each pass.*** So, for example, if a formula is /:RULED-OUT before the /:RULE-IN rules are looked at, it has lost its chance. See Sect. II.C.1.

[/:QUIESCENCE |choice name|]
is recorded when no activity occurred on a choice cycle. It is used to cause further forward deduction of /:RULE statements.

Functions and Predicates Defined by Built-in Axioms

[ELT |x| |tuple|] means x is an element of the tuple.
[SET-OF-ALL |prop|] denotes the set of all objects with the property prop.
[MAPCAR |f| |tuple|] denotes the tuple obtained by applying f to every element of the tuple.
[DEL |x| |tuple|] denotes the tuple obtained by deleting the first occurrence of x from tuple.
[SUBTUP |tup 1| |tup 2|] means every element of tuple 1 is an element of tuple 2.
[CONTAINS |formula 1| |formula 2|] is true if formula 2 occurs somewhere inside formula 1 (as a proper subexpression).
[F-IS-ATOM |formula|] true of atomic-symbol formula like [[A]]
[F-IS-VAR |formula|] true of formula of variable, like [[?X]]
[DEN |formula|] strips a layer of brackets off formula. (See above.)

[/:= |x| |y|] true if x and y match; else assumed false.

[= |x| |y|] true if x and y designate the same thing. To test this, the system first tries matching, then evaluating (*via* "=/>") x and y and trying the match again.

[PRESUMABLY '|proposition| |use|] if true, may assume the proposition from inability to disprove it. (See Sect. II.B.2.)

[XOR1 |pat| < -propositions- >] means exactly one of the propositions is true if the pattern is. E.g.,
```
[XOR1 (LIVING ?X)
      <(ANIMAL ?X) (VEGETABLE ?X)>]
```

[NFUN |n| |f|] denotes a function of n arguments which makes a list of them and applies f to it. E.g.,
```
[NFUN 3 (\ (L) (+ !#?L)   )]
= [\ (X Y Z) (+ ?X ?Y ?Z)]
```

[+ -args-] [- |x| |y|] [* -args-] [// |x| |y|]
arithmetic functions. These are simplified by built-in LISP functions called by the evaluator.

[/< |x| |y|] [/> |x| |y|] [=/< |x| |y|] [/>= |x| |y|]
arithmetic inequalities

Predicates with Meanings to the Theorem Prover

Pragmatic versions of "OR":
[/:CONSEQ |*p*| |*q*|]
[/:ANTEC |*p*| |*q*|]
[/:GEN |*p*| |*q*|]
The first two are used during backward chaining (a call to STP). The second is also used during forward chaining (a call to RECORD). /:GEN is really a call to STP in the midst of forward chaining. See Sect. II.B.2.

[/:PKT |name| |packet vars| -conjuncts-]
Like [AND -conjuncts-], but indexed differently, and more efficiently if most of the conjuncts will never be referenced.

[=/> '|left| |right|]
means [= |left| |right|], but it also means that any expression subsumed by left should be replaced by right wherever it appears (except inside a quoted expression). In practice, this replacement is done mainly in newly detached deductive conclusions.

[/:CONSISTENTLY '|proposition|]
is true iff the proposition cannot be refuted by STP in the current data base. If the proposition has free variables, they will be converted to Skolem forms before trying the refutation.

[/:DD |supporter| |path| < -supporters- > |supportee|]
[/:SUPPORT < -supporters- > |supportee|]
are used to access data dependencies as though they were stored in the data base. The /:DD formula is true if there is a data dependency linking the supporters to the supportee. These are tuple-fied versions of the labeled treelets described in Sect. II.D. In particular, the element supporter must appear in the supporters treelet, with labels in path. For example, we might have

```
[/:DD [/:TASK T#607 <> (λ () (PUTON A B)) <>]
      <DD-ACT-RESULT>
      <(DD-ACT-RESULT <[/:TASK T#607 <> (λ () (PUTON A B)) <>]>
                      <DD-APRIN <MOVE-DEFN>>)>
      [ON A B]]
```

/:SUPPORT is simplified version in which the supporters must be just a list of deductive supporters. It is equivalent to [FORALL (S) (IMPLIES (ELT ?S < -supporters- >) (/:DD ?S <> < -supporters- > |supportee|))].

[T |reference point| |term|]
[S '|proposition|]
These are the built-in modalities. The first is the value of term from the given reference point; the term is usually a fact with value *true* or *false*. [S '|fact|] means "S begins to be true"; it amounts to a special treatment of the data dependency that supports it.

[FRAME |ref point| < -ref points- >]
[N |ref point| '|fact|]
Computationally efficient ways of using modalities. When the system tries a deduction of a T-formula, it will try to smash the reference point to a data pool using these formulas. See Sect. II.B.2.

Predicates with Meanings to the Matcher

Formula	Meaning
[/?/? |sym|]	Inside an embedded formula, matches a variable with the symbol sym. Example: [[\ (_?+V) (F (/?/? _?+V))]] matches [[\ (X) (F ?X)]] with +V → [[X]].
[IDN |n| |k|]	The identity function of n args that returns arg k. Matches [\ ($X_1 \ldots X_{\|n\|}$) $?X_{\|k\|}$]
[K |n| |c|]	The constant function of n args with value c. Matches [\ ($X_1 \ldots X_{\|n\|}$) |c|]

[CMP |fun| < -funs- >]
The composition of fun with the funs. If there are n of them, each with m args, this matches
[\ ($X_1 \ldots X_{|m|}$) (|fun| -args-)],
where the ith "arg" is of the form
[|fun_i| ?X_1 ...?$X_{|m|}$].

Examples: [CMP SIN <?F>] matches [SIN] with F → [IDN 1 1].
[CMP FOO <?F1 ?F2>]
matches [\ (X Y) (FOO (+ ?X ?Y) (- ?Y ?X))]
with F1 → [+] and F2 → [\ (X Y) (- ?Y ?X)]

Appendix 2 -- A Listing of DESI

This is the current (June 27, 1977) version of the design knowledge. It is complete except for the definition of LISP functions defining macro-actions like CONFIG. (See Chapter III.)

In Appendices 2 and 3, NASL formulas are defined and added to the data base with the function DEFMLA, which is somewhat similar to MACLISP's DEFUN. The expression

[DEFMLA |name| |formula| |destination|]

names the formula and adds it to the data pool that is the value of destination. The destination is optional; if it is absent, the current pool CURRENT-DP* will be taken.

```
(DEFMLA STASK-DEFN [-/> A (STASK ?TSK ?SUPER ?I ?A ?O)
                          (AND (/:TASK ?TSK ?I ?A ?O)
                               (/:SUBTASK ?TSK ?SUPER))]
                GENERAL-DP*)

(DEFMLA DEVICE-CLASSES
       [XOR1 (IS DEVICE-TYPE ?D)
             <(BASIC-DEV-TYPE ?D)
              (SUPERORDINATE-DEV-TYPE ?D)>])


(DEFMLA BASIC-DEFN [EQUIV (BASIC-DEV-TYPE ?X)
                          (NOT (EXISTS (Y) (SUB-DEV-TYPE ?Y ?X)   ))])


(DEFMLA MAIN-DEV-TYPE [-/> A (MAIN-DEV-TYPE ?X ?DT) (DEV-TYPE ?X ?DT)])

(DEFMLA SUB-DEV-TYPE-1 [-/> A (SUB-DEV-TYPE ?DT1 ?DT2)
                               (-/> A (DEV-TYPE ?X ?DT1)
                                      (DEV-TYPE ?X ?DT2))])
```

```
(DEFMLA BASIC-DEVICE-CLASSES
        [XOR1 (BASIC-DEV-TYPE ?D)
              <(PRIMITIVE-DEV-TYPE ?D)
               (COMPOSITE-DEV-TYPE ?D)
               (IDEAL-DEV-TYPE ?D)>]
        GENERAL-DP*)



(DEFMLA COMPOSITE-DEVICE-CLASSES
        [XOR1 (COMPOSITE-DEV-TYPE ?D)
              <(GENERAL-DEV-TYPE ?D)
               (SPECIALIZED-DEV-TYPE ?D)>]
        GENERAL-DP*)

(DEFMLA GENERAL-DEFN [EQUIV (GENERAL-DEV-TYPE ?X)
                            (NOT (EXISTS (Y) (DERIVED ?X ?Y)   ))])

(DEFMLA SPEC-DEV-TYPE-1
        [-/> A (SPEC-DEV-TYPE ?DT1 ?DT2)
               (-/> A (DEV-TYPE ?X ?DT1) (DEV-TYPE ?X ?DT2))])

(DEFMLA SPEC-DEV-TYPE-2
        [-/> A (DERIVED ?DT1 ?DT2)
               (-/> A (SPEC-DEV-TYPE ?DT2 ?DT3)
                      (SPEC-DEV-TYPE ?DT1 ?DT3))])

(DEFMLA SPEC-DEV-TYPE-3
        [-/> A (SPEC-DEV-TYPE ?DT1 ?DT2) (SPECIALIZED-DEV-TYPE ?DT1)]
        GENERAL-DP*)

(DEFMLA SOUL-ON-ICE
        [-/> A (DERIVED ?DT ?G)
               (-/> A (MAIN-DEV-TYPE ?X ?DT)
                      (AND (MAIN-DEV-TYPE (SOUL ?X) ?G)
                           (-/> C (/:SUBTASK ?T (DEEP-FREEZE (SOUL ?X)))
                                  (/:SUBTASK ?T (DEEP-FREEZE ?X))))))])

(DEFMLA EASY-DESIGN
        [/:TO-DO ?TSK (DESIGN (\ (X) (DEV-TYPE ?X ?DT)   )) <?NAME>
                      (MAKE ?DT)])

(DEFMLA +DESI-1
        [/:TO-DO ?T (/:REPHRASE ?TSK [DESIGN _?+P] <?DEVNAME>)
                    <>
            (/:DO-SUBNET (DESI-REPHRASE-PLAN ?+P ?TSK ?DEVNAME) <>)]
```

```
        GENERAL-DP*)


(DEFMLA +DESI-2
   [-/> A (/:PLAN-INSTANCE ?PI
                  (DESI-REPHRASE-PLAN ?+P ?TSK ?DEVNAME)
                  ?SUP)
         (-/> A (/:= ?+P [\ (_?+V) _?+B])
                (AND
                  (/:TASK (EXPLODER ?PI) <>
                     (\ () (D-EXPLODE ?+P)) <>)
                  (/:SUBTASK (EXPLODER ?PI) ?SUP)

                  (/:TASK (ACCOUNT-FOR-ALL ?PI) <>
                      (\ () (ACCOUNT-FOR-ALL-SHARDS ?+P)   ) <>)
                  (/:SCOPE (ACCOUNT-FOR-ALL ?PI) (EXPLODER ?PI))
                  (/:SUCCESSOR (ACCOUNT-FOR-ALL ?PI) (EXPLODER ?PI))

                  (/:TASK (CORE-FINDER ?PI) <>
                     (\ () (/:FIND (\ (+DT)
                                      (CORE-DEV-TYPE ?+P ?+DT))))
                     <'(CORE-DT ?PI)>)
                  (/:SUBTASK (CORE-FINDER ?PI) ?SUP)

                  (STASK (MAIN-TASK-INFERER ?PI) ?SUP <'(CORE-DT ?PI)>
                     (\ (+DT) (/:INFER
                                 '(AND (STASK (MAKER ?TSK) ?TSK <>
                                         (\ () (MAKE (DEN ?+DT))   )
                                         <'(WINNER ?TSK)>)
                                       (/:MAIN (MAKER ?TSK) ?TSK))
                                 <(CORE-FINDER ?PI)>)   )
                     <>)

                  (/:SUCCESSOR (MAIN-TASK-INFERER ?PI)
                               (SIDE-TASKS-FINDER ?PI))

                  (STASK (SIDE-TASKS-FINDER ?PI) ?SUP <>
                     (\ () (/:INFER
                              '(FORALL (+ST)
                                  (-/> G (SIDE-TASK ?+P ?+ST)
                                         (EXISTS (T)
                                            (AND (STASK ?T ?TSK
                                                    <'(WINNER ?TSK)>
                                                    (DEN ?+ST)
                                                    <>)
                                                 (/:SUCCESSOR
                                                    (MAKER ?TSK)
                                                    ?T))   ))   )
                              <>)   )
                     <>)

                  (STASK (FEATURES-FINDER ?PI) ?SUP <>
                     (\ ()
```

```
                    (/:INFER
                       '(FORALL (+FT)
                           (-/> G (D-FEATURE ?+P ?+FT)
                                  (EXISTS (T)
                                     (AND (STASK ?T ?TSK <>
                                             (\ ()
                                               (D-NOTE (DEN ?+FT)) )
                                             <>)
                                          (/:SCOPE ?T (MAKER ?TSK))
                                          (/:SUCCESSOR
                                             ?T (MAKER ?TSK))) ))  )
                           <>)   )
                       <>)

                    (STASK (GATHERER ?PI) ?SUP <>
                       (\ () (/:INFER '(/:REDUCED ?TSK)
                                      <(CORE-FINDER ?PI)
                                       (SIDE-TASKS-FINDER ?PI)
                                       (FEATURES-FINDER ?PI)>)   )
                       <>)

                    (/:SUCCESSOR (EXPLODER ?PI) (CORE-FINDER ?PI))
                    (/:SUCCESSOR (EXPLODER ?PI)
                                 (SIDE-TASKS-FINDER ?PI))
                    (/:SUCCESSOR (EXPLODER ?PI) (FEATURES-FINDER ?PI))
                    (/:SUCCESSOR (MAIN-TASK-INFERER ?PI) (GATHERER ?PI))
                    (/:SUCCESSOR (SIDE-TASKS-FINDER ?PI)
                                 (GATHERER ?PI))
                    (/:SUCCESSOR (FEATURES-FINDER ?PI) (GATHERER ?PI))

                    (/:MAIN (GATHERER ?PI) ?SUP))))]
            GENERAL-DP*)


;THE INTENT OF D-EXPLODE IS TO DISCOVER D-SHARDS, WHICH GENERATE
;  (1) CORE-DEV-TYPE, THE KIND OF DEVICE WHICH HAS THE DESIRED PROPERTY
;  (2) D-FEATURES, WHICH WILL GUIDE (AS POLICIES) THE MAKER OF THE DEVICE
;  (3) SIDE-TASKS, WHICH TYPICALLY ARE CONSTRAINTS ON PROPERTIES OF THE
;      DEVICE


(DEFMLA D-EXPLODE
        [/:TO-DO ?T (D-EXPLODE ?+PROP) <>
                 (/:INFER '(D-SHARD ?+PROP ?+PROP) <>)])

(DEFMLA D-SHARD
        [-/> A (D-SHARD ?+P [\ (_?+V) (AND !#_?+CS)])
               (FORALL (+C) (/:GEN (NOT (ELT ?+C ?+CS))
                                   (D-SHARD ?+P [\ (_?+V) _?+C]))   )]
        GENERAL-DP*)


(DEFMLA ACCOUNT-FOR-ALL-DO
```

```
        [/:TO-DO ?T (ACCOUNT-FOR-ALL-SHARDS ?+P) <>
           (/:PRIM *SETUP)]
        GENERAL-DP*)

(DEFMLA ACCOUNT-FOR-ALL-FINISH
        [-/> A (/:TASK-ACTION ?FIN (/:FINISH (ACCOUNT-FOR-ALL ?PI)))
               (AND (/:REDUCED ?FIN)
                    (-/> G (AND (/:PLAN-INSTANCE ?PI
                                      (DESI-REPHRASE-PLAN ?+P ?+TSK
                                                          ?+DEVNAME
                                                          ?+WAY)
                                      ?SUP)
                                (D-SHARD ?+P ?+SHARD))
                           (STASK (SHARD-ACCOUNTANT ?+SHARD) ?FIN
                              <>
                              (\ ()
                                (ACCOUNT-FOR-SHARD ?+SHARD
                                                   ?+P)   )
                              <>)))])


(DEFMLA SHARD-ACCOUNT-DO
        [/:TO-DO ?TSK (ACCOUNT-FOR-SHARD ?+SHARD ?+P) <>
           (/:CALL (SHARD-ACCOUNT-CHEAT ?+SHARD ?+P))]
        GENERAL-DP*)
;THIS IS HANDLED BY A LISP FUNCTION (NOT SHOWN)
;IN THE CURRENT IMPLEMENTATION



(DEFMLA D-NOTE-DO
        [/:TO-DO ?TASK (D-NOTE ?PROPERTY) <> (/:PRIM *SETUP)])

(DEFMLA D-NOTE-FINISH
        [/:TO-DO ?TASK (/:FINISH ?PTASK (D-NOTE ?PROPERTY)) <>
            (/:PRIM *FINISHED)])

(DEFMLA CQ-FUNS-1 [/:ANTEC (NOT (DERIVED-CQ '(?F ?X)))
                           (CONTROL-ATTRIBUTE ?F)])

(DEFMLA CQ-FUNS-2 [/:ANTEC (NOT (IMMEDIATE-CQ '(?F ?X)))
                           (CONTROL-ATTRIBUTE ?F)])

(DEFMLA CQ-SHARD
        [/:ANTEC (NOT (D-SHARD ?+P [\ (_?+V) (= _?+EXP1 _?+EXP2)]))
                 (AND (POS-CQ-SHARD ?+P ?+V ?+EXP1 ?+EXP2)
                      (POS-CQ-SHARD ?+P ?+V ?+EXP2 ?+EXP1))])

(DEFMLA POS-CQ-SHARD
        [/:ANTEC (NOT (POS-CQ-SHARD ?+P ?+V ?+EXP1 ?+EXP2))
                 (/:GEN (NOT (AND (NOT (CONTAINS ?+EXP2 [(|??| _?+V)]))
                                  (=/> '(DEN [\ (_?+V) _?+EXP1])
                                       ?F)
```

```
                                (CONTROL-ATTRIBUTE ?F)
                                (=/> '(DEN ?+F) ?F)))
                        (SIDE-TASK ?+P
                           [\ (_?+V)
                             (CONSTRAIN <'(_?+F (|??| _?+V))>
                                (\ (X) (= ?X _?+EXP2)   ))   ])))]
        GENERAL-DP*)

(DEFMLA CORE-DT-1
        [/:ANTEC (NOT (D-SHARD ?+P [\ (_?+V) (DEV-TYPE (/?/? _?+V)
                                                       _?+DT))))
                 (CORE-DEV-TYPE ?+P ?+DT)])


;CHOOSING CORE-DEV-TYPES

(DEFMLA CORE-DT-CHOOSE
        [/:ANTEC (NOT (CHOICE ?C ANSWER [CORE-DEV-TYPE _?++P _?++D]))
                 (-/> G (AND (/:OPTION ?C ?A1
                                  [CORE-DEV-TYPE _?++P _?++D1])
                             (/:OPTION ?C ?A2
                                  [CORE-DEV-TYPE _?++P _?++D2])
                             (SUB-DEV-TYPE (DEN (DEN ?++D1))
                                           (DEN (DEN ?++D2))))
                        (/:RULE-OUT ?A1))])
;PLUS DOMAIN-DEPENDENT INFO IN ZORCH


(DEFMLA MAKE-BASIC
        [-/> A (BASIC-DEV-TYPE ?DEV-TYPE)
               (/:TO-DO ?TSK (MAKE ?DEV-TYPE) <?NAME>
                   (/:DO-SUBNET (MAKE-BASIC-NET ?DEV-TYPE) <DEVNAME>))])


(DEFMLA MAKE-BASIC-PLAN
        [-/> A (/:PLAN-INSTANCE ?PI (MAKE-BASIC-NET ?DEV-TYPE) ?SUP)
               (AND (/:TASK (GRABBER ?PI) <>
                         (\ ()
                           (/:CALL
                             (GRABBA (\ (X)
                                       (MAIN-DEV-TYPE ?X
                                                      ?DEV-TYPE)   )))   )
                         <'(DEVNAME ?PI)>)
                    (/:MAIN (GRABBER ?PI) ?SUP))]
        GENERAL-DP*)

(DEFMLA MAKE-PRIM
        [-/> A (PRIMITIVE-DEV-TYPE ?DEV-TYPE)
               (-/> A (/:PLAN-INSTANCE
                         ?PI (MAKE-BASIC-NET ?DEV-TYPE) ?SUP)
                      (FORALL (Q D C)
                         (-/> G
                            (FORALL (X)
```

```
                        (IMPLIES (DEV-TYPE ?X ?DEV-TYPE)
                                 (CONTROL ?Q ?X ?D ?C))   )
                     (EXISTS (SUB)
                        (STASK ?SUB ?SUP <'(DEVNAME ?PI)>
                           (\ (X) (SELECT-VALUE
                                    '(?Q ?X))   )
                           <>)   ))   ))))

(DEFMLA MAKE-COMPOSITE
        [-/> A (COMPOSITE-DEV-TYPE ?DEV-TYPE)
               (-/> A (/:PLAN-INSTANCE
                         ?PI (MAKE-BASIC-NET ?DEV-TYPE) ?SUP)
                      (EXISTS (SUB)
                         (STASK ?SUB ?SUP <'(DEVNAME ?PI)>
                            (\ (X) (EXPAND ?X)   )
                            <>)   ))))

(DEFMLA MAKE-IDEAL
        [-/> A (IDEAL-DEV-TYPE ?DEV-TYPE)
               (-/> A (/:PLAN-INSTANCE
                         ?PI (MAKE-BASIC-NET ?DEV-TYPE) ?SUP)
                      (EXISTS (SUB)
                         (STASK ?SUB ?SUP <'(DEVNAME ?PI)>
                            (\ (X) (IMPLEMENT ?X)   )
                            <>)   ))))



(DEFMLA COMPONENTS-NOTICE
        [-/> A (COMPONENTS ?X ?PARTS)
               (-/> G (ELT ?PART ?PARTS)
                      (-/> A (MAIN-DEV-TYPE ?X ?DT)
                             (EXISTS (PI)
                                (AND (/:PLAN-INSTANCE ?PI
                                        (MAKE-BASIC-NET ?DT)
                                        (DEEP-FREEZE ?X))
                                     (/:FINISHED (GRABBER ?PI)))   ))))]
        GENERAL-DP*)

;DEFINITION OF EXPAND

;THE MOST GENERAL SPECIALIZATION AND ANY DEFAULT SPECIALIZATIONS OF A
; DEVICE TYPE ?G ARE TREATED THE SAME HERE, BUT (A) THE CHOICE RULES
; BELOW WILL TAKE THE DEFAULT, OR (B) USER-SUPPLIED RULES WILL
; FAVOR THE GENERAL.

(DEFMLA MOST-GENERAL-DEFN
        [-/> A (MOST-GENERAL-SPEC ?G ?DT)
               (AND (SPEC-DEV-TYPE ?DT ?G)
                    (-/> C (MAIN-DEV-TYPE ?X ?G)
```

```
                        (/:TO-DO ?TASK (EXPAND ?X) <>
                           (SPECIALIZE ?X ?DT)  )))))


(DEFMLA DEFAULT-SPEC-DEFN
        [-/> A (DEFAULT-SPEC ?G ?DT)
               (-/> C (MAIN-DEV-TYPE ?X ?G)
                      (/:TO-DO ?TASK (EXPAND ?X) <>
                         (SPECIALIZE ?X ?DT)   ))))



(DEFMLA SPECIALIZE-DEFN
   [-/> C (MAIN-DEV-TYPE ?DEV ?OLD-DEV-TYPE)
          (/:MOD-MANIP (SPECIALIZE ?DEV ?DEV-TYPE)
                       <'(MAIN-DEV-TYPE ?DEV ?OLD-DEV-TYPE)>
                       <'(MAIN-DEV-TYPE ?DEV ?DEV-TYPE)>)))

;IF ONE DEVICE-TYPE IS A SPECIALIZED VERSION OF ANOTHER, TRY
; TO TAKE THE MORE SPECIFIC, OTHER THINGS EQUAL.

(DEFMLA SPEC-DEV-BETTER
        [-/> A (DERIVED ?DT1 ?DT2)
               (-/> G (AND (=/> '(DEN ?+DT1) ?DT1)
                           (=/> '(DEN ?+DT2) ?DT2))
                      (-/> A (/:OPTION ?C ?A1
                                 [/:TO-DO _?+TSK (EXPAND _?+DEV) <>
                                    (SPECIALIZE _?+DEV _?+DT1)])
                             (-/> A (/:QUIESCENCE ?C)
                                  (-/> G (/:OPTION ?C ?A2
                                             [/:TO-DO _?+TSK
                                                      (EXPAND _?+DEV)
                                                      <>
                                                   (SPECIALIZE
                                                      _?+DT _?+DT2)])
                                           (/:RULE-IN ?A1))))))]
       GENERAL-DP*)


(DEFMLA TWO-SPEC-DEVS-WORSE-THAN-ONE
   [-/> A (DERIVED ?DT1 ?DT0)
      (-/> A (DERIVED ?DT2 ?DT0)
         (-/> G (AND (NOT (/:= ?DT1 ?DT2))
                     (=/> '(DEN ?+DT0) ?DT0)
                     (=/> '(DEN ?+DT1) ?DT1)
                     (=/> '(DEN ?+DT2) ?DT2))
            (-/> A (/:OPTION ?C ?A1
                      [/:TO-DO _?+TSK (EXPAND _?+DEV)
                                  <>
                         (SPECIALIZE _?+DEV _?+DT1)])
                   (-/> A (/:QUIESCENCE ?C)
                      (-/> G (AND (/:OPTION ?C ?A2
                                    [/:TO-DO _?+TSK (EXPAND _?+DEV)
```

```
                                        <>
                                    (SPECIALIZE _?+DEV _?+DT2)])
                                (/:OPTION ?C ?A0
                                  [/:TO-DO _?+TSK (EXPAND _?+DEV)
                                        <>
                                     (SPECIALIZE _?+DEV _?+DT0)]))
                            (AND (/:RULE-OUT ?A1)
                                 (/:RULE-OUT ?A2))))))))])

;AUXILIARY SUBTASKS OF EXPAND
(DEFMLA EXPAND-LOOKAHEAD
        [-/> A (/:TASK-ACTION ?TSK (EXPAND ?DEV))
               (TO-BE-EXPANDED ?DEV ?TSK)])



(DEFMLA EXPANSION-OBLS-DO
        [/:ANTEC (NOT (EXPANSION-OBL ?DEV ?B))
                 (/:ANTEC (NOT (TO-BE-EXPANDED ?DEV ?TN))
                          (/:TASK (OBL ?DEV ?B) <> (\ () ?B) <>))])


(DEFMLA GENERIC-CAS
        [-/> A (GENERIC-CA ?CA)
               (AND (CONTROL-ATTRIBUTE ?CA)
                    (-/> A (/:POLICY ?TASK
                              (CONSTRAIN <!#?QS1 '(?CA ?DEV) !#?QS2>
                                         ?P))
                           (-/> G (ELT '(?CA ?X)
                                       <!#?QS1 '(?CA ?DEV) !#?QS2>)
                                (-/> A (TO-BE-EXPANDED ?DEV ?TSK)
                                       (STASK (CA-CALC ?CA ?DEV) ?TSK
                                          <>
                                          (\ ()
                                            (CALCULATE
                                               '(?CA ?DEV))   )
                                          <>))))))]
        GENERAL-DP*)

(DEFMLA ACQUIRE-DO-1
        [-/> C (AND (REUSABLE ?DEV-TYPE) (DEV-TYPE ?X ?DEV-TYPE))
               (/:TO-DO ?TSK (ACQUIRE ?DEV-TYPE) <?NAME>
                        (/:OUTPUT <?X>))])


(DEFMLA ACQUIRE-DO-2
        [/:TO-DO ?TSK (ACQUIRE ?DEV-TYPE) <?NAME>
                 (MAKE ?DEV-TYPE)])


(DEFMLA REUSABLE-1 [PRESUMABLY '(NOT (REUSABLE ?X)) C])
```

```
(DEFMLA REUSE-CETERIS-PARIBUS
        [-/> A (/:CHOICE ?C EXEC [/:TO-DO _?+TASK (ACQUIRE _?+DT) <_?+N>
                                                  _?+WAY])
                (-/> A (/:QUIESCENCE ?C)
                       (-/> A (/:OPTION ?C ?A1
                                  [/:TO-DO _?+TASK (ACQUIRE _?+DT) <_?+N>
                                                   (MAKE _?+DT)])
                              (/:RULE-OUT ?A1)))])


;IF SIMPLE EQUATION, TRY SOLVING IT
(DEFMLA CONSTRAIN-DO-1
        [-/> C (:= <?UNK>
                   (/:THFIND
                      (\ (U)
                        (AND (ELT ?U ?QUANTS)
                             (/:CONSISTENTLY
                                '(FORALL (VAL)
                                    (NOT (=/> ?U ?VAL))   )))   )))
                (/:TO-DO ?TASK (CONSTRAIN ?QUANTS (CMP = <?F1 ?F2>)) <>
                    (/:SEQ (CONSTRAINT-RESOLVE ?UNK ?F1 ?F2 ?QUANTS)
                           (\ (VAL)
                             (PROTECT
                                '(SATISFIES
                                    ?UNK (CMP = <?F1 ?F2>) ?QUANTS))
                            )))])


;ELSE, JUST ESTABLISH POLICY
(DEFMLA CONSTRAIN-DO-2
        [-/> C (OR (NOT (:= ?F =))
                   (FORALL (UNK)
                      (NOT (:= <?UNK>
                               (/:THFIND
                                  (\ (U)
                                    (AND (ELT ?U ?QUANTS)
                                         (/:CONSISTENTLY
                                            '(FORALL (VAL)
                                                (NOT (=/> ?U ?VAL))
                                             )))   ))))   ))
                (/:TO-DO ?TASK (CONSTRAIN ?QUANTS (CMP ?F ?P1)) <>
                   (/:PRIM *SETUP))])


;DEFINITION OF "CONSTRAINT" --
(DEFMLA CONSTRAINT-1
        [-/> A (CONSTRAINT ?QS ?P)
               (EXISTS (T) (/:POLICY ?T (CONSTRAIN ?QS ?P))   )])

(DEFMLA CONSTRAINT-2
        [-/> C (EXISTS (T) (/:POLICY ?T (CONSTRAIN ?QS ?P))   )
               (CONSTRAINT ?QS ?P)])
```

```
(DEFMLA CONSTRAINT-RESOLVE-REPH
        [/:TO-DO ?RTSK
           (/:REPHRASE ?TASK
             [CONSTRAINT-RESOLVE _?+UNK
                                 (\ (_?+VARS) _?+EXP1)
                                 (\ (_?+VARS) _?+EXP2)
                                 <!#_?QUANTS>]
               <?VALUE>)
           (/:SEQ (EQN-SOLVE ?+UNK (LAMBDA-APPLY (\ (_?+VARS) _?+EXP1)
                                                 ?+QUANTS)
                                   (LAMBDA-APPLY (\ (_?+VARS) _?+EXP2)
                                                 ?+QUANTS))
                  (\ (+ANS)
                    (/:INFER '(AND (STASK (SETTER ?TASK) ?TASK <>
                                          (\ () (/:SET (DEN ?+UNK)
                                                       (DEN ?+ANS))   )
                                          <>)
                                   (/:MAIN (SETTER ?TASK) ?TASK)
                                   (/:REDUCED ?TASK))
                             <>)   ))))


(DEFMLA EQN-SOLVE-DO-1
        [-/> C (AND (ONE-OCCURRENCE ?+LFT ?+UNK)
                    (NOT (CONTAINS ?+RGT ?+UNK)))
               (/:TO-DO ?TASK (EQN-SOLVE ?+UNKF ?+LFT ?+RGT) <?ANSV>
                  (ISOLATE ?+UNK ?+LFT ?+RGT))])

(DEFMLA EQN-SOLVE-DO-2
        [-/> C (AND (ONE-OCCURRENCE ?+RGT ?+UNK)
                    (NOT (CONTAINS ?+LFT ?+UNK)))
               (/:TO-DO ?TASK (EQN-SOLVE ?+UNKF ?+LFT ?+RGT) <?ANSV>
                  (ISOLATE ?+UNK ?+RGT ?+LFT))])

(DEFMLA EQN-SOLVE-DO-3
        [-/> C (NOT (ONE-OCCURRENCE [_?+LFT _?+RGT] ?+UNK))
               (/:TO-DO ?TSK (EQN-SOLVE ?+UNK ?+LFT ?+RGT) <?ANS>
                  (/:CALL (EQN-CHEAT ?+UNK ?+LFT ?+RGT)))])

;THE TERM "ISOLATE" IS FROM BUNDY'S MINI-THEORY OF EQUATION SOLVING.
; HIS NOTION OF "COLLECTION" IS ALSO APPROPRIATE, BUT NOT IMPLEMENTED.


(DEFMLA ISOLATE-DO-1
        [/:CONSEQ (/:TO-DO ?TASK (ISOLATE ?+UNKF ?+LFT ?+REL ?+RGT)
                     <?RV ?ANSV>
                     (OUTPUT <?+REL ?+RGT>))
                  (NOT (= ?+LFT ?+UNKF))])

(DEFMLA ISOLATE-DO-2
```

```
        [/:CONSEQ (/:TO-DO ?TASK (ISOLATE ?+UNKF ?+LFT ?+REL ?+RGT)
                     <?RV ?ANSV>
                     (/:SEQ (ISOLATE-ONE-STEP ?+UNKF ?+LFT ?+REL ?+RGT)
                          (\ (+NEW-LFT +NEW-RGT)
                            (ISOLATE ?+UNKF ?+NEW-LFT
                                     ?+REL ?+NEW-RGT)   )))
                  (= ?+LFT ?+UNKF)))

(DEFMLA ISOLATE-ONE-DO-+
        [/:CONSEQ (/:TO-DO ?TASK
                           (ISOLATE-ONE-STEP ?+UNKF [+ !#_?+ADDENDS]
                                             ?+REL ?+RGT)
                           <?LV ?RV>
                     (OUTPUT <?+TERM [- _?+RGT (+ !#_?+TERMS)]>))
                  (NOT (AND (ELT ?+TERM ?+ADDENDS)
                            (CONTAINS ?+TERM ?+UNKF)
                            (= ?+TERMS (DEL ?+TERM ?+ADDENDS))))))


(DEFMLA SELECT-VALUE-DO
        [-/> C (/:CONSISTENTLY
                  '(FORALL (VAL) (NOT (=/> ?QUANT ?VAL))   ))
                (/:TO-DO ?T (SELECT-VALUE ?QUANT) <>
                    (/:CALL (CHEAT ?QUANT (CQ-CLOSURE ?QUANT))))))
;SELECT-VALUE IS HANDLED BY A LISP FUNCTION
;IN THE CURRENT IMPLEMENTATION



(DEFMLA CQ-CLO-1
        [=/> '(CQ-CLOSURE ?Q)
             (/:THFIND (\ (C) (CQ-CLOSURE-ELT ?C ?Q)   ))])

(DEFMLA CQ-CLO-2
        [-/> C (CONSTRAINT <!#?QT1 ?Q !#?QT2> ?P)
               (CQ-CLOSURE-ELT (CONSTRAINT <!#?QT1 ?Q ?QT2> ?P)
                               ?Q)])

(DEFMLA CQ-CLO-3
        [-/> C (AND (EQ-CLOSURE-ELT (CONSTRAINT ?QS ?P) ?Q)
                    (ELT ?Q1 ?QS)
                    (EQ-CLOSURE-ELT ?C ?Q1))
               (CQ-CLOSURE-ELT ?C ?Q)])

(DEFMLA SELECT-POSTPONE
        [-/> A (/:TASK ?TSK ?I (CMP DESIGN <?PF>) <?DEV>)
               (-/> A (/:TASK ?S ?I (CMP SELECT-VALUE <?QF>) <>)
                      (AND (STASK (SELECT-EM-ALL ?TSK) ?TSK <>
                                  (\ () (DO-ALL-SELECT-VALUES ?TSK)   )
                                  <>)
                           (/:SUBTASK ?S (SELECT-EM-ALL ?TSK))
                           (/:REDUCED (SELECT-EM-ALL ?TSK))
                           (-/> A (TOPO-CHANGE-ACTION-FUN ?F)
```

```
                                (-/> A (/:TASK ?T ?INS
                                          (CMP ?F ?FS) ?OUTS)
                                       (/:SUCCESSOR
                                         ?T (SELECT-EM-ALL ?TSK)))))]
        GENERAL-DP*)


(DEFMLA MAKE-CHANGES-TOPOLOGY
        [TOPO-CHANGE-ACTION-FUN MAKE])

(DEFMLA FIX-CHANGES-TOPOLOGY
        [TOPO-CHANGE-ACTION-FUN FIX])

(DEFMLA QVAL-PROTECT
        [-/> C (AND (=/> ?Q ?VAL)
                    (=/> '(DEN ?+FACT) (=/> ?Q ?VAL)))
               (/:TO-DO ?TASK (PROTECT '(SATISFIES ?Q ?C ?QUANTS)) <>
                  (/:MONITOR ?+FACT
                      (\ (T)
                        (/:CONTINUE ?TASK
                            (PROTECT
                                '(SATISFIES ?Q ?C ?QUANTS)))   )))])


(DEFMLA PROTECT-CONTINUE
        [/:TO-DO ?TASK (/:CONTINUE ?PTASK
                           (PROTECT '(SATISFIES ?Q ?C ?QUANTS)))
                 <>
           (/:DO-SUBNET
              (PROTECT-CONTINUE-NET ?PTASK ?Q ?C ?QUANTS) <>)])


(DEFMLA PROTECT-CONTINUE-PLAN
        [-/> A (/:PLAN-INSTANCE ?PI
                  (PROTECT-CONTINUE-NET ?PTASK ?Q ?C ?QUANTS)
                  ?SUP)
               (AND (STASK (RECHECKER ?PI) ?SUP <>
                       (\ () (VERIFY '(SATISFIES ?Q ?C ?QUANTS))   ) <>)
                    (STASK (VALUE-FINDER ?PI) ?SUP <>
                       (\ ()
                         (/:FIND
                            (\ (+NEWMON)
                              (EXISTS (NEWVAL)
                                 (AND (=/> ?Q ?NEWVAL)
                                      (=/> '(DEN ?+NEWMON)
                                           (=/> ?Q ?NEWVAL)))   )))   )
                       <'(NEWMON ?PI)>)
                    (STASK (REMONITOR ?PI) ?SUP <'(NEWMON ?PI)>
                       (\ (+FACT)
                         (/:MONITOR ?+FACT
                             (\ (T)
                               (/:CONTINUE ?PTASK
                                   (PROTECT
```

```
                                        '(SATISFIES ?Q ?C
                                              ?QUANTS)))   ))   )))))

(DEFMLA VERIFY-DO
        [/:TO-DO ?TSK (VERIFY '?P) <>
           (/:FIND (\ () ?P)   )))

(DEFMLA SPEC-SCHEMA-DEFN
        [-/> A (SPEC-SCHEMA ?SCH1 ?SCH2)
               (-/> A (/:PLAN-INSTANCE ?PI ?SCH1 ?SUP)
                      (/:PLAN-INSTANCE ?PI ?SCH2 ?SUP))))

;THIS PREDICATE IS USEFUL IN RELATING A SCHEMA TO ITS SPECIALIZERS
(DEFMLA REDUCE-DEFN
        [-/> A (REDUCE ?TASKS ?TASK)
               (AND (-/> G (ELT ?T ?TASKS) (/:SUBTASK ?T ?TASK))
                    (/:REDUCED ?TASK))))


;THIS IS A USEFUL PREDICATE ON FROZEN TASKS
(DEFMLA FUNCTION-DEFN
        [-/> A (FUNCTION ?DEV ?TSK)
               (-/> A (MAIN-DEV-TYPE ?DEV ?DT)
                      (EXISTS (ACQ)
                         (AND (/:TASK ?ACQ ?I ?A <?DEV>)
                              (/:TASK-ACTION ?ACQ (ACQUIRE ?DT))
                              (/:REDUCED ?ACQ)
                              (REDUCE <?ACQ> ?TSK))   ))))

;IF ONE PLAN-SCHEMA IS A SPECIALIZED VERSION OF ANOTHER, TRY
; TO TAKE THE MORE SPECIFIC.  (THIS IS REALLY MORE GENERAL THAN
; THE WORLD OF DESIGN.)

(DEFMLA SPEC-IS-BETTER
        [-/> A (SPEC-SCHEMA ?SCH1 ?SCH2)
               (-/> G (AND (=/> '(DEN ?+SCH1) ?SCH1)
                           (=/> '(DEN ?+SCH2) ?SCH2))
                      (-/> A (/:OPTION ?C ?A1
                                  [/:TO-DO _?+TSK _?+ACT _?+OUTS
                                      (/:DO-SUBNET _?+SCH1 _?+VARS1)])
                             (-/> A (/:OPTION ?C ?A2
                                        [/:TO-DO _?+TSK _?+ACT _?+OUTS
                                           (/:DO-SUBNET _?+SCH2
                                                        _?+VARS2)])
                                  (/:RULE-IN ?A1))))))


(DEFMLA TWO-SPECS-WORSE-THAN-ONE
        [-/> A (SPEC-SCHEMA ?SCH1 ?SCH0)
               (-/> A (SPEC-SCHEMA ?SCH2 ?SCH0)
                      (-/> G (AND (NOT (/:= ?SCH1 ?SCH2))
```

```
                (=/> '(DEN ?+SCH0) ?SCH0)
                (=/> '(DEN ?+SCH1) ?SCH1)
                (=/> '(DEN ?+SCH2) ?SCH2))
           (-/> A
                (/:OPTION ?C ?A1
                   [/:TO-DO _?+TSK _?+ACT _?+OUTS
                      (/:DO-SUBNET _?+SCH1 _?+VARS1)])
                (-/> G (AND (/:OPTION ?C ?A2
                              [/:TO-DO _?+TSK _?+ACT
                                       _?+OUTS
                                 (/:DO-SUBNET _?+SCH2
                                              _?+VARS2)])
                            (/:OPTION ?C ?A0
                              [/:TO-DO _?+TSK _?+ACT
                                       _?+OUTS
                                 (/:DO-SUBNET _?+SCH0
                                              _?+VARS0)]))
                       (AND (/:RULE-OUT ?A1)
                            (/:RULE-OUT ?A2)))))))
```

Appendix 3 -- A Listing of ZORCH

This is the current version of DESI's electronics knowledge. Much of it interacts with the more general rules of the previous appendix.

```
(INTSECT-DISPARITY* := 1000)
(ALLOC '(LIST (100000. 150000. 0.6)))

;PHYSICAL KNOWLEDGE

;EVERY NODE IS A TERMINAL

(DEFMLA NODE-TRMIN
        [FORALL (X) (-/> A (DEV-TYPE ?X NODE) (DEV-TYPE ?X TERMINAL))])

;KCL FOR DEVICES
(DEFMLA KCL-1
        [-/> A (TERMINAL-NAMES ?X ?TRMIN-TUP)
               (CONSTRAINT (MAPCAR (LAMBDA (T) '(I (?T ?X))   )
                                   ?TRMIN-TUP)
                           (NFUN (LENGTH ?TRMIN-TUP)
                                 (LAMBDA (L) (= (+ !#?L) 0)))))])

;KCL FOR NODES
(DEFMLA KCL-2
        [-/> A (=/> '(NODE-TERMINALS ?NODE) ?TRMIN-TUP)
               (CONSTRAINT <'(I ?NODE)
                           !#(MAPCAR (\ (T) '(I ?T)   )
```

```
                                    ?TRMIN-TUP)>
                              (NFUN (+ (LENGTH ?TRMIN-TUP) 1)
                                    (LAMBDA (L) (= (+ !#?L) 0))))))

; ...AND COMPOSITE DEVICES
(DEFMLA KCL-3
        [-/> A (COMPOSITE-DEV-TYPE ?DT)
               (-/> A (DEV-TERMINALS ?DEV ?TRMIN-TUP)
                      (CONSTRAINT (MAPCAR (\ (T) '(I ?T)   )
                                          ?TRMIN-TUP)
                                  (NFUN (LENGTH ?TRMIN-TUP)
                                        (LAMBDA (L) (= (+ !#?L) 0)
                                           ))))))


;KVL FOR NODES
(DEFMLA KVL-1
        [-/> A (=/> '(NODE-TERMINALS ?NODE) ?TRMIN-TUP)
               (-/> G (ELT ?TRMIN ?TRMIN-TUP)
                      (CONSTRAINT <'(V ?TRMIN) '(V ?NODE)> =))))


;SOME TERMINALS HAVE NODES THAT THEY ARE TERMINALS OF

(DEFMLA NODE-OF-1 [-/> A (=/> '(NODE-TERMINALS ?N) ?TS)
                         (-/> G (ELT ?T ?TS) (NODE-OF ?T ?N))])

(DEFMLA NODE-OF-2 (PRESUMABLY '(NOT (NODE-OF ?T ?N)) C))


;NODES CAN BE MERGED
(DEFMLA NODES-MERGE-MANIP
        [-/> C (AND (=/> '(NODE-TERMINALS ?N1) ?T1)
                    (=/> '(NODE-TERMINALS ?N2) ?T2))
               (/:MOD-MANIP ?TASK (NODES-MERGE ?N1 ?N2)
                   <'(=/> '(NODE-TERMINALS ?N1) ?T1)
                    '(=/> '(NODE-TERMINALS ?N2) ?T2)>
                   <'(=/> '(NODE-TERMINALS ?N1) (UNION ?T1 ?T2))
                    '(=/> '?N2 ?N1)>)])


;TERMINALS CAN BE CONNECTED TO CREATE NEW NODES OR MERGE OLD ONES

(DEFMLA TRMINS-CONNECT-DO-1
        [-/> C (AND (NODE-OF ?T1 ?N1) (NODE-OF ?T2 ?N2))
               (/:TO-DO ?TASK (TRMINS-CONNECT ?T1 ?T2) <>
                  (NODES-MERGE ?N1 ?N2))])

(DEFMLA TRMINS-CONNECT-DO-2
        [-/> C (AND (NODE-OF ?T1 ?N1)
                    (CONSISTENTLY
                       '(FORALL (N) (NOT (NODE-OF ?T2 ?N))  ))
                    (=/> '(NODE-TERMINALS ?N1) ?TS1))
```

```
            (/:MOD-MANIP ?TASK (TRMINS-CONNECT ?T1 ?T2)
               <'(=/> '(NODE-TERMINALS ?N1) ?TS1)>
               <'(=/> '(NODE-TERMINALS ?N1) <?T2 !#?TS1>)>)]]

(DEFMLA TRMINS-CONNECT-DO-3
        [-/> C (AND (NODE-OF ?T2 ?N2)
                    (CONSISTENTLY
                       '(FORALL (N) (NOT (NODE-OF ?T1 ?N))   ))
                    (=/> '(NODE-TERMINALS ?N2) ?TS2))
               (/:MOD-MANIP ?TASK (TRMINS-CONNECT ?T1 ?T2)
                  <'(=/> '(NODE-TERMINALS ?N2) ?TS2)>
                  <'(=/> '(NODE-TERMINALS ?N2) <?T1 !#?TS2>)>)]]

(DEFMLA TRMINS-CONNECT-DO-4
        [-/> C (AND (CONSISTENTLY
                       '(FORALL (N) (NOT (NODE-OF ?T1 ?N))   ))
                    (CONSISTENTLY
                       '(FORALL (N) (NOT (NODE-OF ?T2 ?N))   )))
               (/:MOD-MANIP ?TASK (TRMINS-CONNECT ?T1 ?T2)
                  <>
                  <'(EXISTS (N)
                       (=/> '(NODE-TERMINALS ?N) <?T1 ?T2>)   )>)]]


;INSERTING A DEVICE INTO A NODE BREAKS IT INTO TWO NODES
(DEFMLA DEV-INSERT
        [-/> C (AND (=/> '(NODE-TERMINALS ?NODE) ?TS)
                    (= (SET ?TS1) (SET <!#?TS1 !#?TS2>)))
               (/:MOD-MANIP ?TASK
                  (DEV-INSERT ?D ?NODE ?T1 ?TS1 ?T2 ?TS2)
                   <'(=/> '(NODE-TERMINALS ?NODE) ?TS)>
                   <'(=/> '(NODE-TERMINALS ?NODE) <?T1 !#?TS1>)
                    '(EXISTS (NEWNODE)
                        (AND (DEV-TYPE ?NEWNODE NODE)
                             (=/> '(NODE-TERMINALS ?NEWNODE)
                                  <?T2 !#?TS2>))   )>)]]

(DEFMLA PORT-DEFN [PRIMITIVE-DEV-TYPE PORT])

;PORTS CARRY VOLTAGE OR CURRENT
(DEFMLA PORT-TAXONOMY [XOR1 (DEV-TYPE ?X PORT)
                            <(V-PORT ?X) (I-PORT ?X)>]
                      GENERAL-DP*)

(DEFMLA PORT-MEDIUM-1
        [-/> C (V-PORT ?X) (=/> '(PORT-MEDIUM ?X) VOLTAGE)])

(DEFMLA PORT-MEDIUM-2
        [-/> C (I-PORT ?X) (=/> '(PORT-MEDIUM ?X) CURRENT)])

;MOST PORTS ARE VOLTAGE PORTS
(DEFMLA PRES-V-PORT [-/> C (AND (DEV-TYPE ?X PORT)
                                (CONSISTENTLY '(V-PORT ?X)))
```

```
                        (V-PORT ?X)])

;A NEST IS MADE UP OF PORTS AND IS ITSELF A PORT

(DEFMLA NEST-PORT [-/> A (DEV-TYPE ?X NEST) (DEV-TYPE ?X PORT)])

(DEFMLA NEST-OF-1 [-/> A (=/> '(NEST-PORTS ?N) ?TS)
                         (-/> G (ELT ?T ?TS) (NEST-OF ?T ?N))])

(DEFMLA NEST-OF-2 [PRESUMABLY '(NOT (NEST-OF ?T ?N)) C])

;PORTS ARE HOMES FOR SIGNALS

;KVL FOR PORTS
(DEFMLA KVL-2
        [-/> A (=/> '(NEST-PORTS ?NEST) ?PORT-TUP)
               (-/> G (ELT ?PORT ?PORT-TUP)
                      (AND (-/> A (SIGNAL-HOME ?SIG ?PORT)
                                  (SIGNAL-HOME ?SIG ?NEST)))))])

(DEFMLA SIGNAL-HOME
        [-/> A (SIGNAL-HOME ?SIG ?PORT)
               (=/> '(PORT-SIGNAL ?PORT) ?SIG)])

;THESE ACTIONS ARE ANALOGOUS TO THE NODE ACTIONS

(DEFMLA NESTS-MERGE-MANIP
        [-/> C (AND (=/> '(NEST-PORTS ?N1) ?T1)
                    (=/> '(NEST-PORTS ?N2) ?T2))
               (/:MOD-MANIP ?TASK (NESTS-MERGE ?N1 ?N2)
                   <'(=/> '(NEST-PORTS ?N1) ?T1)
                    '(=/> '(NEST-PORTS ?N2) ?T2)>
                   <'(=/> '(NEST-PORTS ?N1) (UNION ?T1 ?T2))
                    '(=/> '?N2 ?N1)>)])


(DEFMLA PORTS-CONNECT-DO-1
        [-/> C (AND (NEST-OF ?T1 ?N1) (NEST-OF ?T2 ?N2))
               (/:TO-DO ?TASK (PORTS-CONNECT ?T1 ?T2 ?TYPE) <>
                  (NESTS-MERGE ?N1 ?N2))])

(DEFMLA PORTS-CONNECT-DO-2
        [-/> C (AND (NEST-OF ?T1 ?N1)
                    (CONSISTENTLY
                       '(FORALL (N) (NOT (NEST-OF ?T2 ?N))  ))
                    (=/> '(NEST-PORTS ?N1) ?TS1))
              (/:MOD-MANIP ?TASK (PORTS-CONNECT ?T1 ?T2 ?TYPE)
                 <'(=/> '(NEST-PORTS ?N1) ?TS1)>
                 <'(=/> '(NEST-PORTS ?N1) <?T2 !#?TS1>)>)])

(DEFMLA PORTS-CONNECT-DO-3
        [-/> C (AND (NEST-OF ?T2 ?N2)
```

```
                (CONSISTENTLY
                   '(FORALL (N) (NOT (NEST-OF ?T1 ?N))   ))
                (=/> '(NEST-PORTS ?N2) ?TS2))
             (/:MOD-MANIP ?TASK (PORTS-CONNECT ?T1 ?T2 ?TYPE)
                <'(=/> '(NEST-PORTS ?N2) ?TS2)>
                <'(=/> '(NEST-PORTS ?N2) <?T1 !#?TS2>)>)]]

(DEFMLA PORTS-CONNECT-DO-4
        [-/> C (AND (CONSISTENTLY
                      '(FORALL (N) (NOT (NEST-OF ?T1 ?N))   ))
                    (CONSISTENTLY
                      '(FORALL (N) (NOT (NEST-OF ?T2 ?N))   )))
                (/:MOD-MANIP ?TASK (PORTS-CONNECT ?T1 ?T2 ?TYPE)
                   <>
                   <'(EXISTS (N) (=/> '(NEST-PORTS ?N) <?T1 ?T2>)   )>)]]


(DEFMLA PORTS-CONNECT-DIRECT-DO
        [-/> C (AND (PORT-TERMINALS ?PRT1 <?TOP1 ?BOT2>)
                    (PORT-TERMINALS ?PRT2 <?TOP2 ?BOT2>)
                    (OR (/:MOD-MANIP ?TASK (TRMINS-CONNECT ?TOP1 ?TOP2)
                            ?DEL ?ADD)
                        (/:MOD-MANIP ?TASK (TRMINS-CONNECT ?BOT1 ?BOT2)
                            ?DEL ?ADD)))
                (/:MOD-MANIP ?TASK (PORTS-CONNECT ?PRT1 ?PRT2 DIRECT)
                   ?DEL ?ADD)]]


(DEFMLA PORTS-CONNECT-CAPACITIVE-DO
        [-/> C (AND (PORT-TERMINALS ?PRT1 <?TOP1 ?BOT1>)
                    (PORT-TERMINALS ?PRT2 <?TOP2 ?BOT2>))
                (/:TO-DO ?TSK (PORTS-CONNECT ?PRT1 ?PRT2 CAPACITIVE) <>
                   (CONFIG <CAPACITOR>
                      (\ (C)
                        <(TRMINS-CONNECT (#1 ?C) ?TOP1)
                         (TRMINS-CONNECT (#2 ?C) ?TOP2)>   )))]]


(DEFMLA PORTS-CONNECT-INDUCTIVE-DO
        [-/> C (AND (PORT-TERMINALS ?PRT1 <?TOP1 ?BOT1>)
                    (PORT-TERMINALS ?PRT2 <?TOP2 ?BOT2>))
                (/:TO-DO ?TSK (PORTS-CONNECT ?PRT1 ?PRT2 INDUCTIVE) <>
                   (CONFIG <TRANSFORMER>
                      (\ (LL)
                        <(TRMINS-CONNECT (#2 ?LL) ?TOP1)
                         (TRMINS-CONNECT (#4 ?LL) ?TOP2)>   )))]]

;SIGNALS

(DEFMLA FL-SHAPE [ELT (FL-SHAPE ?LM) <HUMP SPIKE>])


(DEFMLA FF-FREQ [=/> '(FF-FREQ (FF ?F ?LM)) ?F])
```

```
(DEFMLA FF-LANDMARK [=/> '(FF-LANDMARK (FF ?F ?LM)) ?LM])


(DEFMLA PERIODIC-FREQ-PIC
        [-/> C (PERIODIC ?S ?T)
               (=/> '(FREQ-PICTURE ?S)
                    (UNION (POSSIBLE-DC-SUB-PIC ?S)
                           (SERIES-SUB-PIC ?S)))])

(DEFMLA DC-FEATURE-1
        [-/> C (AND (PERIODIC ?TFUN ?T)
                    (= (OFFSET ?TFUN) ?V)
                    (/> (ABS ?V) 0))
               (=/> '(POSSIBLE-DC-SUB-PIC ?S)
                    <(FF 0 (DC-LANDMARK ?TFUN ?V))>)])

(DEFMLA DC-FEATURE-2
        [-/> C (AND (PERIODIC ?TFUN ?T)
                    (= (OFFSET ?TFUN) 0))
               (=/> '(POSSIBLE-DC-SUB-PIC ?S) <>)])

(DEFMLA DC-LANDMARK-1
        [=/> '(FL-SHAPE (DC-LANDMARK ?TFUN ?V)) SPIKE]
        GENERAL-DP*)

(DEFMLA DC-LANDMARK-2
        [=/> '(FL-HEIGHT (DC-LANDMARK ?TFUN ?V)) ?V]
        GENERAL-DP*)



(DEFMLA FREQ-PIC-ELTS [IMPLIES (ELT ?X (FREQ-PIC ?S))
                               (IS FREQ-FEATURE ?X)])

(DEFMLA SPIKES
        [-/> A (PERIODIC ?TFUN ?T)
               (FORALL (X) (-/> C (ELT ?X (FREQ-PICTURE ?TFUN))
                                  (= (FL-SHAPE ?X) SPIKE))   )])


(DEFMLA SINUSOIDAL-SPIKE
        [-/> C (AND (PERIODIC ?TFUN ?T)
                    (SINUSOIDAL ?TFUN)
                    (AMPLITUDE ?TFUN ?A))
              (=/> '(SERIES-SUB-PIC ?TFUN)
                   <(FF 0 (SIN-LANDMARK ?TFUN ?T ?A))>)])


(DEFMLA EVERYOTHER-SERIES
       [-/> C (AND (PERIODIC ?TFUN ?T)
                   (NOT (SINUSOIDAL ?TFUN))
                   (FORALL (X)
                      (= (?TFUN ?X) (?TFUN (+ ?X (// ?T 2))))   ))
```

```
            (=/> '(SERIES-SUB-PIC ?TFUN)
                 (SERIES (* 2 PI (// 1 ?T)) (* 4 PI (// 1 ?T))
                         SPIKE
                         (\ (N)
                           (INTEGRAL 0 ?T
                              (\ (U)
                                (* (?TFUN ?U)
                                   (COS (* 2 PI (- (* 2 ?N) 1)
                                           ?U (// 1 ?T))))   ))   )
                  ))))


(DEFMLA STRAIGHT-SERIES
        [-/> C (AND (PERIODIC ?TFUN ?T)
                    (NOT (SINUSOIDAL ?TFUN))
                    (EXISTS (X)
                       (NOT (= (?TFUN ?X)
                               (?TFUN (+ ?X (// ?T 2))))))   ))
               (=/> '(SERIES-SUB-PIC ?TFUN)
                    (SERIES (* 2 PI (// 1 ?T)) (* 2 PI (// 1 ?T))
                            SPIKE
                            (\ (N)
                              (INTEGRAL 0 ?T
                                 (\ (U)
                                   (* (?TFUN ?U)
                                      (COS (* 2 PI ?N
                                              ?U (// 1 ?T))))   ))   )
                     ))))

(DEFMLA SIN-LANDMARK-SHAPE
        [=/> '(FL-SHAPE (SIN-LANDMARK ?TFUN ?T ?A)) SPIKE])

(DEFMLA SIN-LANDMARK-HEIGHT
        [=/> '(FL-HEIGHT (SIN-LANDMARK ?TFUN ?T ?A)) ?A])


(DEFMLA SINU+ [-/> C (AND (ELT ?SIN ?FS)
                          (SINUSOIDAL ?SIN)
                          (FORALL (F)
                             (EXISTS (A)
                                (IMPLIES (ELT ?F (DEL ?SIN ?FS))
                                         (/:= ?F (K 1 ?A)))   )))
                     (SINUSOIDAL (CMP + ?FS))])

(DEFMLA SINU* [-/> C (AND (ELT ?SIN ?FS)
                          (SINUSOIDAL ?SIN)
                          (FORALL (F)
                             (EXISTS (A)
                                (IMPLIES (ELT ?F (DEL ?SIN ?FS))
                                         (/:= ?F (K 1 ?A)))   )))
                     (SINUSOIDAL (CMP * ?FS))])

(DEFMLA SIN-SIN [-/> C (LINEAR ?F)
```

```
                    (SINUSOIDAL (CMP SIN <?F>))))

(DEFMLA COS-SIN [-/> C (LINEAR ?F)
                    (SINUSOIDAL (CMP COS <?F>))))

(DEFMLA LIN1 [LINEAR (IDN ?N ?K)])

(DEFMLA LIN2 [-/> C (AND (ELT ?LIN ?FS) (LINEAR ?LIN)
                         (FORALL (K)
                            (EXISTS (A)
                               (IMPLIES (ELT ?K (DEL ?LIN ?FS))
                                        (/:= ?K (K 1 ?A)))   )))
                    (LINEAR (CMP * ?FS))])

(DEFMLA LIN3 [-/> C (FORALL (F) (IMPLIES (ELT ?F ?FS) (LINEAR ?F))   )
                    (LINEAR (CMP + ?FS))])

(DEFMLA LIN4 [-/> C (LINEAR (CMP + <?F1 (\ (X) (- (?F2 ?X))   )>))
                    (LINEAR (CMP - <?F1 ?F2>))])

(DEFMLA LIN5 [-/> C (AND (LINEAR ?F1)
                         (/:= ?F2 (K 1 ?A)))
                    (LINEAR (CMP // <?F1 ?F2>))])

(DEFMLA SIN-PERIODIC [PERIODIC SIN (* 2 PI)])

(DEFMLA COS-PERIODIC [PERIODIC COS (* 2 PI)])


(DEFMLA PRES-NOT-SIN [PRESUMABLY '(NOT (SINUSOIDAL ?F)) C])

(DEFMLA OFFSET-DEFN [-/> A (PERIODIC ?TFUN ?T)
                         (=/> '(OFFSET ?TFUN)
                              (// (INTEGRAL 0 ?T ?TFUN) ?T))])

;LINEAR-DERIVED MODELS

;(ISOLATE T1 T2) IS A WORLD IN WHICH THE CIRCUIT IS DECOUPLED

(DEFMLA REF-ISO [IS REF (ISOLATE ?TRMIN1 ?TRMIN2)])

(DEFMLA FRAME-ISO [(FRAME (ISO ?TRMIN1 ?TRMIN2) <(HERE)>)])

(DEFMLA ISOLATE-DEFN-1
        [-/> C (AND (=/> '(NODE-TERMINALS ?N1)
                         <!#?TS11 ?TRMIN1 !#?TS12>)
                    (=/> '(NODE-TERMINALS ?N2)
                         <!#?TS21 ?TRMIN2 !#?TS22>))
               (T (ISOLATE ?TRMIN1 ?TRMIN2)
                  (AND (=/> '(NODE-TERMINALS ?N1)
                            <!#?TS11 !#?TS12>)
                       (=/> '(NODE-TERMINALS ?N2)
                            <!#?TS21 !#?TS22>)))])
```

```
(DEFMLA ISOLATE-DEFN-2
        [-/> C (=/> '(NODE-TERMINALS ?N1)
                    <!#?TS11 ?TRMIN1 !#?TS12>)
               (N (ISOLATE ?TRMIN1 ?TRMIN2)
                  '(=/> '(NODE-TERMINALS ?N1)
                        <!#?TS11 ?TRMIN1 !#?TS12>))))

(DEFMLA ISOLATE-DEFN-3
        [-/> C (=/> '(NODE-TERMINALS ?N2)
                    <!#?TS21 ?TRMIN2 !#?TS22>)
               (N (ISOLATE ?TRMIN1 ?TRMIN2)
                  '(=/> '(NODE-TERMINALS ?N2)
                        <!#?TS21 ?TRMIN2 !#?TS22>))])



;FOR THE FOLLOWING REFERENCE-POINT DEFINITIONS,
; SEE CIRCUIT PACKETS FOR ALTERED COMPONENT PROPERTIES IN EACH REF

(DEFMLA REF-DC [IS REF (DC)])

(DEFMLA FRAME-DC [(FRAME (DC) <(HERE)>)])



(DEFMLA REF-SSS [IS REF (SSS ?FREQ)])

(DEFMLA FRAME-SSS [<FRAME (SSS ?S) <(HERE)>)])


(DEFMLA REF-INC [IS REF (INC)])

(DEFMLA FRAME-INC [(FRAME (INC) <(HERE)>)])


(DEFMLA REF-PASSIVE [IS REF (PASSIVE)])

(DEFMLA FRAME-PASSIVE [(FRAME (PASSIVE) <(HERE)>)])

;INTERACTIONS --

;THIS MEANS (T (DC) (DC)) = (DC)
(DEFMLA DC-IDEM [REF-IDEMPOTENT (DC)])
(DEFMLA ISOLATE-IDEM [REF-IDEMPOTENT (ISOLATE ?TRMIN1 ?TRMIN2)])
(DEFMLA SSS-IDEM [REF-IDEMPOTENT (SSS ?S)])
(DEFMLA PASSIVE-IDEM [REF-IDEMPOTENT (PASSIVE)])
(DEFMLA INC-IDEM [REF-IDEMPOTENT (INC)])

(DEFMLA DC-ISO [=/> '(T (DC) (ISOLATE ?TRMIN1 ?TRMIN2))
                    (T (ISOLATE ?TRMIN1 ?TRMIN2) (DC))])
```

```
(DEFMLA PASSIVE-DC [=/> '(T (PASSIVE) (DC)) (T (DC) (PASSIVE))])


;INFORMATION ABOUT REPHRASING ELECTRONIC DESIGN PROBLEMS

;D-SHARDS ARE FRAGMENTS OF DESIGN PROPERTIES; THOSE DEALING WITH
; CONTROL QUANTITIES ARE IMPORTANT

;THESE APPLY TO "IO" DEVICES

(DEFMLA CQ-V-GAIN [GENERIC-CA V-GAIN])
(DEFMLA CQ-P-GAIN [GENERIC-CA P-GAIN])
(DEFMLA CQ-IZ [GENERIC-CA INPUT-Z])
(DEFMLA CQ-OZ [GENERIC-CA OUTPUT-Z])



(DEFMLA V-GAIN-SHARD
        [-/> A (D-SHARD ?+P [\ (_?+V) (= (V-GAIN (|??| _?+V))
                                         _?+G)])
                (AND (SIDE-TASK ?+P
                                [\ (_?+V)
                                  (CONSTRAIN <'(V-GAIN (|??| _?+V))>
                                     (\ (G1) (= ?G1 _?+G)   ))])
                     (-/> G (= (DEN ?+G) ?G)
                            (AND (-/> G (/> ?G 1000)
                                        (D-FEATURE ?+P
                                           [RANGER V-GAIN VERY-HIGH]))
                                 (-/> G (AND (/> ?G 50)
                                             (/< ?G 5000))
                                        (D-FEATURE ?+P
                                           [RANGER V-GAIN HIGH]))
                                 (-/> G (AND (/> ?G 1)
                                             (/< ?G 100))
                                        (D-FEATURE ?+P
                                           [RANGER V-GAIN MODERATE]))
                                 (-/> G (=/< ?G 1)
                                        (D-FEATURE ?+P
                                           [RANGER V-GAIN LOW])))))))


;"GAIN" ALONE MEANS V-GAIN AND P-GAIN
(DEFMLA GAIN-SHARD
        [-/> A (D-SHARD ?+P [\ (_?+V) (= (GAIN (|??| _?+V))
                                         _?+G)])
                (AND (D-SHARD ?+P [\ (_?+V) (= (V-GAIN (|??| _?+V))
                                               _?+G)])
                     (-/> G (AND (= (* 20 (LOG (DEN _?+G))) ?PG)
                                 ;CONVERT TO DECIBELS
                                 (= (DEN ?+PG) ?PG))
                            (D-SHARD ?+P
```

```
                        [\ (_?+V) (= (P-GAIN (|??| _?+V))
                                     _?+PG)]))))])

(DEFMLA INPUT-Z-SHARD
        [-/> A (D-SHARD ?+P [\ (_?+V) (= (INPUT-Z (|??| _?+V))
                                         _?+Z)])
               (-/> G (= (DEN ?+Z) ?Z)
                      (AND (-/> G (/> ?Z 3.0E5)
                                  (D-FEATURE ?+P
                                     [RANGER INPUT-Z VERY-HIGH]))
                           (-/> G (AND (/> ?Z 1.5E3)
                                       (/< ?Z 5.0E5))
                                  (D-FEATURE ?+P
                                     [RANGER INPUT-Z HIGH]))
                           (-/> G (AND (/> ?Z 500)
                                       (/< ?Z 2.0E3))
                                  (D-FEATURE ?+P
                                     [RANGER INPUT-Z MODERATE]))
                           (-/> G (/< ?Z 1000)
                                  (D-FEATURE ?+P
                                     [RANGER INPUT-Z LOW])))))])

(DEFMLA OUTPUT-Z-SHARD
        [-/> A (D-SHARD ?+P [\ (_?+V) (= (OUTPUT-Z (|??| _?+V))
                                         _?+Z)])
               (-/> G (= (DEN ?+Z) ?Z)
                      (AND (-/> G (/> ?Z 1.0E6)
                                  (D-FEATURE ?+P
                                     [RANGER OUTPUT-Z VERY-HIGH]))
                           (-/> G (AND (/> ?Z 1.0E5)
                                       (/< 1.5E6))
                                  (D-FEATURE ?+P
                                     [RANGER OUTPUT-Z HIGH]))
                           (-/> G (AND (/> ?Z 1.0E4)
                                       (/< ?Z 1.5E5))
                                  (D-FEATURE ?+P
                                     [RANGER OUTPUT-Z MODERATE]))
                           (-/> G (AND (/> ?Z 100)
                                       (/< ?Z 1.5E4))
                                  (D-FEATURE ?+P
                                     [RANGER OUTPUT-Z LOW]))
                           (-/> G (/< ?Z 200)
                                  (D-FEATURE ?+P
                                     [RANGER OUTPUT-Z VERY-LOW])))))])
;SOURCE: WATSON (1970), P. 60

;SHARDS REGARDING SIGNAL CONVERSIONS MUST BE EXPLODED SPECIALLY

(DEFMLA CONVERT-EXPLODE
        [-/> A (/:TASK-ACTION ?T (D-EXPLODE ?+P))
```

```
                (-/> A
                     (D-SHARD ?+P
                        [\ (_?+V)
                          (CONVERT (|??| _?+V) _?+Q _?+R)])
                     (EXISTS (T1)
                        (AND (STASK ?T1 ?T <>
                                [\ () (CVT-EXPLODE ?+Q ?+R)   )
                                <>)
                             (/:MAIN ?T1 ?T)
                             (-/> A (SIG-FEATURE ?+Q ?+R ?+FEATURE)
                                    (D-SHARD ?+P
                                       [\ (_?+V)
                                         (SIG-TRANS _?+V
                                            _?+FEATURE)]))))   )))
        GENERAL-DP*)

;THERE ARE TWO WAYS TO DO THIS--

(DEFMLA CVT-EXPLODE-1
        [/:TO-DO ?TASK (CVT-EXPLODE ?+Q ?+R) <>
           (FREQ-DOMAIN-REPHRASE ?+Q ?+R)])

(DEFMLA CVT-EXPLODE-2
        [/:TO-DO ?TASK (CVT-EXPLODE ?+Q ?+R) <>
           (TIME-DOMAIN-REPHRASE ?+Q ?+R)])

; BASIS ON WHICH TO CHOOSE ONE OR THE OTHER
(DEFMLA CVT-CHOICE
        [/:ANTEC (NOT (/:CHOICE ?C EXEC
                          [/:TO-DO _?+TASK
                              (CVT-EXPLODE _?++Q _?++R) <>
                              _?++METHOD]))
                 (-/> A (/:OPTION ?C ?A1
                              [/:TO-DO _?+TASK
                                 (CVT-EXPLODE _?++Q _?++R)
                                 <>
                                 (FREQ-DOMAIN-REPHRASE
                                     _?++Q _?++R)])
                        (-/> A (/:OPTION ?C ?A2
                                [/:TO-DO _?+TASK
                                   (CVT-EXPLODE _?++Q _?++R)
                                   <>
                                   (TIME-DOMAIN-REPHRASE
                                      _?++Q _?++R)])
;     IF OUTPUT RELATION DOESN'T MENTION INPUT, TAKE FREQUENCY-DOMAIN
                               (AND (-/> G (AND (/:= (DEN ?++R)
                                                     [\ (_?+SV1 _?+SV2)
                                                       _?+BODY])
                                                (NOT (CONTAINS ?+BODY
                                                       [|??| _?+SV1])))
                                           (/:RULE-IN ?A1))
;     IF INPUT PREDICATE IS TRIVIAL, TAKE TIME-DOMAIN
                                    (-/> G (AND (/:= (DEN ?++Q)
```

```
                                                    [\ (_?+V) _?+BODY])
                                                (NOT (CONTAINS ?+BODY
                                                       [|??| _?+V])))
                                            (/:RULE-IN ?A2))
;     FOR VERY HIGH FREQUENCIES, TIME DOMAIN WON'T WORK
                                    (-/> G (AND (/:SUBTASK (DEN ?+TASK)
                                                           ?SUP)
                                                (/:TASK-ACTION ?SUP
                                                       (D-EXPLODE ?+P))
                                                (/:= (/:ENAB-STATUS
                                                         ?SUP)
                                                     ACTIVE)
                                                (D-FEATURE ?+P
                                                   [RANGER FREQ-OP
                                                           VERY-HIGH]))
                                            (/:RULE-OUT ?A2))))))]
        GENERAL-DP*)

; FREQUENCY-DOMAIN REPHRASING INFO

(DEFMLA FREQ-DOM-REPH-DO-1
        [/:TO-DO ?TASK (FREQ-DOMAIN-REPHRASE ?+Q ?+R) <>
           (/:SEQ
              (/:FIND
                 (\ (+FPT)
                   (EXISTS (FP1 FP2 FPT)
                      (FORALL (S1 S2)
                         (IMPLIES (AND (IS SIGNAL ?S1)
                                       ((DEN ?+Q) ?S1)
                                       (IS SIGNAL ?S2)
                                       ((DEN ?+R) ?S1 ?S2))
                                  (AND (=/> '(FREQ-PICTURE
                                                 (TFUN ?S1))
                                            ?FP1)
                                       (=/> '(FREQ-PICTURE
                                                 (TFUN ?S2))
                                            ?FP2)
                                       (FREQ-PIC-TRANS ?FP1 ?FP2
                                                       ?FPT)
                                       (=/> '(DEN ?+FPT) ?FPT)))
                    ))))
              (\ (FPT)
                (/:INFER '(SIG-FEATURE ?+Q ?+R [FREQ-TRANS _?+FPT])
                         <>)   )))]


;GENERATING FREQUENCY-DOMAIN TRANSFORMATIONS


(DEFMLA NFREQ-PICS-FILTER
        [/:CONSEQ (NOT (FREQ-PICS-FILTER ?FP1
                         <(FF ?FREQ ?LM2) !#?FP2> ?C))
```

```
                (NOT (FORALL (LM1)
                        (NOT (ELT (FF ?FREQ ?LM1) ?FP1))   ))))


(DEFMLA FREQ-TRANS-LOW
        [/:CONSEQ (FREQ-TRANS ?FP-IN ?FP-OUT (LOW-PASS ?M))
                  (NOT (AND (FREQ-PICS-FILTER ?FP-IN ?FP-OUT ?FP-GONE)
                            (FORALL (FL F)
                               (AND (EQUIV (ELT (FF ?F ?FL) ?FP-GONE)
                                           (FORALL (FL1 F1)
                                             (IMPLIES (ELT (FF ?F1 ?FL1)
                                                           ?FP-GONE)
                                                      (/> ?F ?F1))   ))
                                    (/:= ?M
                                         (MAX ?FP-GONE
                                              (\ (F G)
                                                (/> (FF-FREQ ?F)
                                                    (FF-FREQ ?G))
                                               ))))   ))))]

;SIMILARLY FOR HIGH-PASS AND BAND-PASS

(DEFMLA FREQ-PICS-FILTER-1
        [FREQ-PICS-FILTER ?FP1 <> ?FP1])

(DEFMLA FREQ-PICS-FILTER-2
        [/:CONSEQ (FREQ-PICS-FILTER ?FP1 <(FF ?FREQ ?LM2) !#?FP2> ?C)
                  (NOT (AND (ELT (FF ?FREQ ?LM1) ?FP1)
                            (= (FF-SHAPE ?LM1) (FF-SHAPE ?LM2))
                            (FREQ-PICS-FILTER (DEL (FF ?FREQ ?LM1) ?FP1)
                                              ?FP2  ?C)))])

;TYPES OF FREQ-TRANS: (LOW-PASS |CUTOFF|), (HIGH-PASS |CUTOFF|),
;  (BAND-PASS |CUTOFF|), (MIX |SIGNAL-PRED|) (MODULATE...)

(DEFMLA LOW-PASS
        [/:ANTEC (NOT (D-SHARD ?+P
                          [\ (_?+V) (SIG-TRANS (|??| _?+V)
                                               (FREQ-TRANS
                                                  (LOW-PASS _?+CUTOFF)))])))
                 (CORE-DEV-TYPE ?+P [LOW-PASS-FILTER _?+CUTOFF])])

(DEFMLA HIGH-PASS
        [/:ANTEC (NOT (D-SHARD ?+P
                          [\ (_?+V) (SIG-TRANS (|??| _?+V)
                                               (FREQ-TRANS
                                                  (HIGH-PASS _?+CUTOFF)))])))
                 (CORE-DEV-TYPE ?+P [HIGH-PASS-FILTER _?+CUTOFF])])



;CHOOSING DEV-TYPES --
```

```
(DEFMLA LINEAR-GROUPING
        [-/> A (/:CHOICE ?C ?TSK [CORE-DEV-TYPE _?++P _?++D])
               (-/> A (QUIESCENCE ?C)
                      (-/> G (AND (/:OPTION ?C ?A1
                                       [CORE-DEV-TYPE
                                          [__?++P] [__?++D1]])
                                  (/:OPTION ?C ?A2
                                       [CORE-DEV-TYPE [__?++P] [__?++D2]])
                                  (LINEAR-DEV-TYPE (DEN (DEN ?++D2)))
                                                  )
                             (/:RULE-TOGETHER <?A1 ?A2>
                                  [CORE-DEV-TYPE _?++P
                                     [GROUP <__?++D1 __?++D2>]])))))]

;GROUPS MAY BE EXPRESSED AS SEVERAL KINDS OF CASCADE

(DEFMLA MAKE-GROUP-1
        [/:CONSEQ
           (/:TO-DO ?T (MAKE (GROUP <?X ?Y !#?Z>)) <?NAME>
               (/:SEQ (ACQUIRE (GROUP ?DT-REST))
                      (\ (G) (MAKE (CASCADE ?DT1 ?G))   )))
           (NOT (AND (ELT ?DT1 <?X ?Y !#?Z>)
                     (/:= ?DT-REST (DEL ?DT1 <?X ?Y !#?Z>)))))]
        GENERAL-DP*)


(DEFMLA MAKE-GROUP-2
        [/:TO-DO ?T (MAKE (GROUP <?DT1 ?DT2>)) <?NAME>
           (MAKE (CASCADE ?DT1 ?DT2)))]

(DEFMLA MAKE-GROUP-3
        [/:TO-DO ?T (MAKE (GROUP <?DT1 ?DT2>)) <?NAME>
           (MAKE (CASCADE ?DT2 ?DT1)))]

;AMPLIFIER IS A WIDE ("SUPERORDINATE") CATEGORY, FOR WHICH THERE ARE
; SEVERAL SPECIFIC TYPES

(DEFMLA AMP-DEV-TYPE [SUPERORDINATE-DEV-TYPE AMPLIFIER])

(DEFMLA SUB-CE [SUB-DEV-TYPE CE AMPLIFIER])
(DEFMLA SUB-CC [SUB-DEV-TYPE CC AMPLIFIER])
(DEFMLA SUB-CB [SUB-DEV-TYPE CB AMPLIFER])

; IF MODERATE V-GAIN, COMMON EMITTER
(DEFMLA MOD-V-GAIN
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER V-GAIN MODERATE)))
                             (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                 (MAKE CE)))))]
        GENERAL-DP*)
```

```
; IF HIGH V-GAIN, SOME KIND OF N-STAGE
(DEFMLA HIGH-V-GAIN
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER V-GAIN HIGH)))
                             (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                          (MAKE N-STAGE)))))]
        GENERAL-DP*)


; IF VERY-HIGH, OP-AMP
(DEFMLA VERY-HIGH-V-GAIN
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER V-GAIN VERY-HIGH)))
                             (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                 (MAKE OP-AMP)))))]
        GENERAL-DP*)


; IF VERY LOW FREQ OF OPERATION (E.G., DC) -- DIFF AMP
(DEFMLA VERY-LOW-FREQ
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER FREQ-OP VERY-LOW)))
                             (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                          (MAKE DIFF-AMP)))))]
        GENERAL-DP*)


;IF HIGH INPUT Z, UP CC
(DEFMLA HIGH-INPUT-Z
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER INPUT-Z HIGH)))
                             (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                          (MAKE CC)))))]
        GENERAL-DP*)


; IF HIGH POWER GAIN, UP COMP-SYM AND PUSH-PULL
(DEFMLA HIGH-POWER-GAIN
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
               (-/> A (/:SCOPE ?PTSK ?TSK)
                      (-/> A (/:POLICY ?PTSK
                                 (D-NOTE (RANGER P-GAIN HIGH)))
                             (AND (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
```

```
                                        (MAKE COMP-SYM))
                                (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                        (MAKE PUSH-PULL))))))]
        GENERAL-DP*)


;IF LINEARITY REQUIRED, CE
(DEFMLA LINEARITY
        [-/> A (/:TASK-ACTION ?TSK (MAKE AMPLIFIER))
                (-/> A (/:SCOPE ?PTSK ?TSK)
                        (-/> A (/:POLICY ?PTASK ?TSK
                                        (D-NOTE LINEAR))
                                (/:TO-DO ?TSK (MAKE AMPLIFIER) <?DEV>
                                        (MAKE CE)))))]
        GENERAL-DP*)

;IF MORE THAN ONE TYPE APPEARS, A CHOICE IS IN ORDER

(DEFMLA CHOOSE-AMP
        [-/> A (/:CHOICE ?C EXEC
                    [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+NAME>
                            _?+METHOD])
                (-/> G (/:= (DEN ?+TASK) ?AMP-TASK)
                        (AND
                          (/:PKT CHOOSE-AMP-PKT
                                (?C ?AMP-TASK ?+TASK ?+NAME ?+METHOD)
                             (-/> A (/:QUIESCENCE ?C)
                                    (/:PKT QUI-CHOOSE-AMP-PKT
                                            (?C ?AMP-TASK ?+TASK
                                             ?+NAME ?+METHOD))))
                          (-/> G (/:SCOPE ?PTSK ?AMP-TASK)
                                TRUE))))]
        GENERAL-DP*)


;IF HIGH POWER AND LINEARITY ARE REQUIRED, REPLACE POWER-AMP
; OPTION WITH LINEARIZED POWER-AMP

(DEFMLA LINEAR-POWER
        [-/> A (/:SCOPE ?PTSK1 ?AMP-TASK)
                (-/> G (AND (/:POLICY ?PTSK1 (D-NOTE LINEAR))
                            (/:SCOPE ?PTSK2 ?AMP-TASK)
                            (/:POLICY ?PTSK2
                                    (D-NOTE (RANGER P-GAIN HIGH)))
                            (=/> '(DEN ?+PTSK2) ?PTSK2))
                        (-/> A (/:OPTION ?C ?A1
                                [/:TO-DO _?+TASK (MAKE AMPLIFIER)
                                        <_?+NAME> (MAKE _?+DT)])

                                (-/> G
                                    (OPT-SUPPORT ?A1
                                        [/:POLICY _?+PTASK2
```

```
                                                    _?+ABOVE-ACT])
                                        (/:RULE-TOGETHER <?A1>
                                            [/:TO-DO _?+TASK (MAKE AMPLIFIER)
                                                        <_?+NAME>
                                                (/:SEQ (MAKE _?+DT)
                                                       (\ (?X)
                                                         (/:DO-ALL
                                                           <(LINEARIZE ?X)>
                                                           (/:OUTPUT <?X>)) )
                        )]]]]]]
        CHOOSE-AMP-PKT)


;QUIESCENCE PKT
;DIFFERENTIAL DIAGNOSIS BETWEEN CE AND N-STAGE
(DEFMLA DIFF-CE-N-STAGE
  [-/> A (/:QUIESCENCE ?C)
        (-/> G (AND (/:OPTION ?C ?A1
                      [/:TO-DO _?+TASK (MAKE AMPLIFIER) <?_+NAME>
                           (MAKE CE)])
                    (/:OPTION ?C ?A2
                      [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+NAME>
                           (MAKE N-STAGE)]))
               (AND
                  (-/> G (AND (/:SCOPE ?PTSK ?AMP-TASK)
                              (/:POLICY ?PTSK
                                 (D-NOTE
                                   (RANGER BANDWIDTH HIGH))))
                         (/:RULE-OUT ?A1))
                  (-/> G (NOT (EXISTS (PTSK)
                                 (AND (/:SCOPE ?PTSK ?AMP-TASK)
                                      (/:POLICY ?PTSK
                                         (D-NOTE (RANGER BANDWIDTH
                                                        HIGH))))   ))
                         (/:RULE-OUT ?A2)))))
        CHOOSE-AMP-PKT)


;IF ONE OPTION WAS SUGGESTED BECAUSE OF ITS INPUT-Z (E.G.,
; COMMON-COLLECTOR), AND ANOTHER FOR SOMETHING ELSE,
; YOU MAY CASCADE THEM

(DEFMLA INPUT-CASCADE
        [-/> A (/:OPTION ?C ?A1
                  [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+N>
                               (MAKE _?+DT1)])
               (-/> G (OPT-SUPPORT ?A1
                         [/:POLICY _?+PTASK
                                   (D-NOTE (RANGER INPUT-Z _?+RAN))])
                      (-/> A
                         (/:OPTION ?C ?A2
                             [/:TO-DO _?+TASK (MAKE AMPLIFIER) <_?+N>
                                          (MAKE _?+DT2)])
```

```
                    (-/> G (NOT (= ?A1 ?A2))
                           (/:RULE-TOGETHER <?A1 ?A2>
                              [/:TO-DO _?+TASK (MAKE AMPLIFIER)
                                               <_?+N>
                                       (MAKE (CASCADE _?+DT1
                                                      _?+DT2
                                             ))))))))]
        CHOOSE-AMP-PKT)


(DEFMLA OPT-SUPPORT [-/> C (AND (/:OPTION-SUPPORT ?OPT ?FMLAS)
                                (ELT ?+F ?FMLAS)
                                (OR (:= ?+F ?+S)
                                    (SUPPORTS ?+S ?+F)))
                           (OPT-SUPPORT ?OPT ?+S)]
                    GENERAL-DP*)

(DEFMLA SUPPORT-DEFN [-/> C (AND (/:DO ?+S ?P ?I ?+F2)
                                 (OR (:= ?+F1 ?+S)
                                     (SUPPORTS ?+F1 ?+S)))
                            (SUPPORTS ?+F1 ?+F2)]
                     GENERAL-DP*)



(DEFMLA MAKE-CASCADE
        [/:TO-DO ?TASK (MAKE (CASCADE ?DT1 ?DT2)) <?NEWDEV>
           (/:DO-SUBNET (CASCADE-PLAN ?DT1 ?DT2) <CASCADE-NAME>)])

(DEFMLA CASCADE-PLAN-NET
        [-/> A (/:PLAN-INSTANCE ?PI (CASCADE-PLAN ?DT1 ?DT2) ?SUP)
               (AND (STASK (MAKER-1 ?PI) ?SUP <>
                       (\ () (MAKE ?DT1)   ) <'(FIRST-DEV ?PI)>)
                    (STASK (MAKER-2 ?PI) ?SUP <>
                       (\ () (MAKE ?DT2)   ) <'(SECOND-DEV ?PI)>)
                    (STASK (GRABBER ?PI) ?SUP <>
                       (\ ()
                         (GRABBA (\ (X)
                                   (MAIN-DEV-TYPE
                                      ?X (CASCADE ?DT1 ?DT2))   ))   )
                       <'(CASCADE-NAME ?PI)>)
                    (STASK (COUPLER ?PI) ?SUP
                       <'(FIRST-DEV ?PI) '(SECOND-DEV ?PI)>
                       (\ (D1 D2) (COUPLE ?D1 ?D2)   )
                       <>)
                    (STASK (COMPONENTS-NAMER ?PI)
                       <'(CASCADE-NAME ?PI)
                        '(FIRST-DEV ?PI) '(SECOND-DEV ?PI)>
                       (\ (C D1 D2)
                         (/:INFER '(COMPONENTS ?C <?D1 ?D2>) <>)   )
                       <>)
                    (/:MAIN (GRABBER ?PI) ?SUP))]
```

```
        GENERAL-DP*)


;USE THE MOST GENERAL VERSION OF A CIRCUIT IN A CASCADE

(DEFMLA COUPLE-GENERAL-1
        [-/> A (/:CHOICE ?C EXEC
                    [/:TO-DO (MAKER-1 _?+PI) (MAKE _?+DT) <_?+DEV>
                        _?+WAY])
               (-/> G (AND (=/> '(DEN ?+DT) ?DT)
                           (MOST-GENERAL-SPEC ?DT ?SPEC-DT)
                           (=/> '(DEN ?+SPEC-DT) ?SPEC-DT))
                      (-/> A (/:OPTION ?C ?A
                                  [/:TO-DO (MAKER-1 _?+PI) (MAKE _?+DT)
                                                           <_?+DEV>
                                     (MAKE _?+SPEC-DT)])
                             (/:RULE-IN ?A)))])


(DEFMLA COUPLE-GENERAL-2
        [-/> A (/:CHOICE ?C EXEC
                    [/:TO-DO (MAKER-2 _?+PI) (MAKE _?+DT) <_?+DEV>
                        _?+WAY])
               (-/> G (AND (=/> '(DEN ?+DT) ?DT)
                           (MOST-GENERAL-SPEC ?DT ?SPEC-DT)
                           (=/> '(DEN ?+SPEC-DT) ?SPEC-DT))
                      (-/> A (/:OPTION ?C ?A
                                  [/:TO-DO (MAKER-2 _?+PI) (MAKE _?+DT)
                                                           <_?+DEV>
                                     (MAKE _?+SPEC-DT)])
                             (/:RULE-IN ?A)))])

;CIRCUIT ALTERATION ACTIONS


(DEFMLA FIXING-CHANGES-TOPOLOGY [TOPO-CHANGE-ACTION-FUN FIX])


(DEFMLA VS-FIX-V
        [/:TO-DO ?TASK (FIX '(V ?NODE)) <>
           (CONFIG (VOLTAGE-SOURCE)
              (\ (VS)
                <(TRMINS-CONNECT (#1 ?VS) ?NODE)>   ))])


(DEFMLA VD-FIX-V
        [/:TO-DO ?TASK (FIX '(QUIESCENT (V ?NODE))) <>
             (CONFIG <VD NODE NODE>
                (LAMBDA (VD N1 N2)
                  <(CONSTRAIN <'(V ?N1) '(V ?NODE)>
                      (\ (V1 V) (/> ?V1 ?V)   ))
                   (CONSTRAIN <'(V ?N2) '(V ?NODE)>
                      (\ (V2 V) (/> ?V ?V2)   ))
```

```
                (NODES-MERGE ?N1 (TOP ?VD))
                (NODES-MERGE ?N2 (BOT ?VD))
                (NODES-MERGE ?NODE (MID ?VD))>   )))]
        GENERAL-DP*)


(DEFMLA DIF-FIX [/:TO-DO ?TASK (FIX '(- ?X1 ?X2)) <>
                 (/:DO-ALL <(FIX '?X1) (FIX '?X2)>)])

;BIASING

(DEFMLA BIAS-CHANGES-TOPOLOGY [TOPO-CHANGE-ACTION-FUN BIAS])

(DEFMLA BJT-BIAS
        [/:ANTEC (NOT (DEV-TYPE ?Q BJT))
                 (/:TO-DO ?TASK (BIAS ?Q ACTIVE) <>
                    (/:DO-SUBNET (GENERAL-BIAS-PLAN ?Q) ?TASK <>))])

(DEFMLA BJT-BIAS-NET
        [/:ANTEC (NOT (/:PLAN-INSTANCE ?PI (GENERAL-BIAS-PLAN ?Q) ?SUP))
                 (AND (STASK (VBE-FIXER ?PI) ?SUP <>
                         (\ () (FIX '(- (V (BASE ?Q)) (V (EMI ?Q))))   )
                         <>)

                      (STASK (CB-BIASER ?PI) ?SUP <>
                         (\ () (REVERSE-BIAS (CB-JUNCTION ?Q))   ) <>)
                      (STASK (IC-FIXER ?PI) ?SUP <>
                         (\ () (FIX '(I (COL ?Q)))   )      <>)
                      (/:MAIN (VBE-FIXER ?PI) ?SUP)
                      (/:MAIN (IC-FIXER ?PI) ?SUP))]
        GENERAL-DP*)

(DEFMLA TYPICAL-BJT-ONE-STAGE-BIAS
        [-/> A (/:PLAN-INSTANCE ?PI
                  (TYPICAL-BJT-ONE-STAGE-BIAS-PLAN ?Q) ?SUP)
               (AND
                  (STASK (BVD ?PI) ?SUP <>
                     (\ () (ACQUIRE VD)   ) <'(BVD ?PI)>)
                  (STASK (SUPPLY-POWER ?PI) ?SUP <>
                     (\ () (ACQUIRE VS)   ) <'(POWER-SUPPLY ?PI)>)
                  (STASK (RESIS-GETTER ?PI) ?SUP <>
                     (\ () (ACQUIRE RESISTOR)   ) <'(RE ?PI)>)
                  (STASK (BASE-SETTER ?PI) ?SUP <'(BVD ?PI)>
                     (\ (VD) (TRMINS-CONNECT (BASE ?Q) (MID ?VD))   )
                     <>)
                  (STASK (COLLECTOR-POWER ?PI) ?SUP
                     <'(POWER-SUPPLY ?PI)>
                     (\ (PS) (DEV-INSERT ?PS (CNODE ?Q)
                                 (#2 ?PS)
                                 (NODE-TERMINALS (CNODE ?Q))
                                 (#1 ?PS) <>)   )
                     <>)
                  (STASK (EMITTER-MUNGER ?PI) ?SUP <'(RE ?PI)>
```

```
                (\ (R) (DEV-INSERT ?R (ENODE ?Q)
                          (#1 ?R) <(EMI ?Q)>
                          (#2 ?R)
                          (DEL (EMI ?Q)
                               (NODE-TERMINALS (ENODE ?Q))))   )
                <>)
             (REDUCE <(BVD ?PI) (BASE-SETTER ?PI)>
                     (VBE-FIXER ?PI))
             (REDUCE <(SUPPLY-POWER ?PI) (COLLECTOR-POWER ?PI)>
                     (CB-BIASER ?PI))
             (REDUCE <(RESIS-GETTER ?PI) (EMITTER-MUNGER ?PI)>
                     (IC-FIXER ?PI))   )))


(DEFMLA SPEC-TYPICAL-BJT
        [SPEC-SCHEMA (TYPICAL-BJT-ONE-STAGE-BIAS-PLAN ?Q)
                     (GENERAL-BIAS-PLAN ?Q)])


;COUPLING HAS BEEN SPECIALIZED TO BJT AMPLS.

(DEFMLA COUPLING-CHANGES-TOPOLOGY [TOPO-CHANGE-ACTION-FUN COUPLE])

(DEFMLA COUPLE-DO-1
        [/:TO-DO ?TASK (COUPLE ?D1 ?PORT1 ?PORT2 ?D2) <>
           (/:DO-SUBNET (GENERAL-COUPLING-PLAN ?D1 ?PORT1 ?PORT2 ?D2)
                        ?TASK <>)])

(DEFMLA COUPLE-NET
        [/:ANTEC (NOT (/:PLAN-INSTANCE ?PI
                         (GENERAL-COUPLING-PLAN ?D1 ?PORT1 ?PORT2 ?D2)
                         ?SUP))
                 (AND (STASK (SIGNAL-MEDIUM-CHOOSE ?PI) ?SUP <>
                         (\ ()
                            (/:FIND
                               (\ (M)
                                  (ELT ?M <VOLTAGE CURRENT>)   ))   )
                         <'(MEDIUM ?PI)>)
                      (STASK (COUPLE-TYPE-CHOOSE ?PI) ?SUP <>
                         (\ ()
                           (/:FIND
                              (\ (CT)
                                 (ELT ?CT
                                      <DIRECT CAPACITIVE
                                       INDUCTIVE>)   ))   )
                         <'(COUPLE-TYPE ?PI)>)
                      (STASK (CONVERT-PORT-1 ?PI) ?SUP <'(MEDIUM ?PI)>
                         (\ (M) (PORT-CONVERT ?PORT1 ?M)   )
                         <>)
                      (STASK (CONVERT-PORT-2 ?PI) ?SUP <'(MEDIUM ?PI)>
                         (\ (M) (PORT-CONVERT ?PORT2 ?M)   )
```

```
                    <>)
                  (STASK (CONNECTOR ?PI) ?SUP <'(COUPLE-TYPE ?PI)>
                    (\ (TYPE)
                      (PORTS-CONNECT ?PORT1 ?PORT2 ?TYPE)   )
                    <>)
                  (/:SUCCESSOR (SIGNAL-MEDIUM-CHOOSE ?PI)
                               (CONVERT-PORT-1 ?PI))
                  (/:SUCCESSOR (SIGNAL-MEDIUM-CHOOSE ?PI)
                               (CONVERT-PORT-2 ?PI))
                  (/:SUCCESSOR (CONVERT-PORT-1 ?PI)
                               (CONNECTOR ?PI))
                  (/:SUCCESSOR (CONVERT-PORT-2 ?PI)
                               (CONNECTOR ?PI))
                  (/:MAIN (CONNECTOR ?PI) ?SUP)
                  )]
        GENERAL-DP*)

(DEFMLA COUPLE-BEFORE-BIAS
        [-/> A (/:TASK-ACTION ?CT (COUPLE ?D1 ?PRT1 ?PRT2 ?D2))
               (-/> G (AND (COMPONENTS ?D1 ?D1CS)
                           (COMPONENTS ?D2 ?D2CS)
                           (OR (ELT ?Q ?D1CS) (ELT ?Q ?D2CS))
                           (MAIN-DEV-TYPE ?Q BJT))
                      (/:SUCCESSOR ?CT (BIASER ?Q ?MODE)))])



;SPECIFIC SITUATIONS --
(DEFMLA COUPLE-CE-X-HINTS
        [-/> A (/:TASK-ACTION ?CT (COUPLE ?D1 (OUTPORT ?D1) ?PRT2 ?D2))
               (-/> G (DEV-TYPE ?D1 CE)
                      (/:PKT CE-COUPLE-HINTS (?CT ?D1 ?PRT2 ?D2)
                          (/:TO-DO ?CT (COUPLE ?D1 (OUTPORT ?D1)
                                               ?PRT2 ?D2) <>
                             (/:DO-SUBNET
                                (CE-DIR-VOL-COUPLE ?D1 ?PRT2 ?D2)
                                <>))
                          (/:TO-DO ?CT (COUPLE ?D1 (OUTPORT ?D1)
                                               ?PRT2 ?D2) <>
                             (/:DO-SUBNET
                                (CE-DIR-CUR-COUPLE ?D1 ?PRT2 ?D2)
                                <>))
                          (/:TO-DO ?CT (COUPLE ?D1 (OUTPORT ?D1)
                                               ?PRT2 ?D2) <>
                             (/:DO-SUBNET
                                (CE-CAP-VOL-COUPLE ?D1 ?PRT2 ?D2)
                                <>))))])


(DEFMLA COUPLE-CC-X-HINTS
        [-/> A (/:TASK-ACTION ?CT (COUPLE ?D1 (OUTPORT ?D1) ?PRT2 ?D2))
```

```
            (-/> G (DEV-TYPE ?D1 CC)
                   (/:PKT CC-COUPLE-HINTS (?CT ?D1 ?PRT2 ?D2)
                      (/:TO-DO ?CT (COUPLE ?D1 (OUTPORT ?D1)
                                           ?PRT2 ?D2) <>
                         (/:DO-SUBNET
                            (CC-DIR-VOL-COUPLE ?D1 ?PRT2 ?D2)
                            <>))))))

(DEFMLA CE-DIR-VOL-COUPLE-PLAN
       [-/> A (/:PLAN-INSTANCE ?PI
                 (CE-DIR-VOL-COUPLE ?D1 ?PRT2 ?D2) ?SUP)
              (AND (=/> '(MEDIUM ?PI) VOLTAGE)
                   (=/> '(COUPLE-TYPE ?PI) DIRECT)
                   (/:REDUCED (SIGNAL-MEDIUM-CHOOSE ?PI))
                   (/:REDUCED (COUPLE-TYPE-CHOOSE ?PI))

                   (STASK (GET-RESISTOR ?PI) ?SUP <>
                      (\ () (ACQUIRE RESISTOR)  ) <'(RESISTOR ?PI)>)
                   (STASK (GET-POWER-SUPPLY ?PI) ?SUP <>
                      (\ () (ACQUIRE VOLTAGE-SOURCE)   ) <'(VS ?PI)>)
                   (STASK (BJT-FINDER ?PI) ?SUP <>
                      (\ ()
                        (/:FIND
                           (\ (Q)
                             (EXISTS (CS)
                                (AND (COMPONENTS ?D1 ?CS)
                                     (ELT ?Q ?CS)
                                     (MAIN-DEV-TYPE ?Q BJT))   )))   )
                      <'(TRANSISTOR ?PI)>)
                   (STASK (WIRER-1 ?PI) ?SUP
                      <'(RESISTOR ?PI) '(TRANSISTOR ?PI)>
                      (\ (R Q) (TRMINS-CONNECT (#2 ?R) (COL ?Q))   )
                      <>)
                   (STASK (WIRER-2 ?PI) ?SUP
                      <'(RESISTOR ?PI) '(VS ?PI)>
                      (\ (R VS) (TRMINS-CONNECT (#1 ?R) (#2 ?VS))   )
                      <>)

                   (REDUCE <(GET-RESISTOR ?PI) (WIRER-1 ?PI)>
                           (CONVERT-PORT-1 ?PI))
                   (-/> A (=/> '(TRANSISTOR ?PI) ?Q)
                          (-/> A (/:PLAN-INSTANCE ?BIAS-PI
                                    (GENERAL-BIAS-PLAN ?Q) ?SUP-BIAS)
                                 (REDUCE <(GET-POWER-SUPPLY ?PI)
                                          (WIRER-2 ?PI)>
                                         (CB-BIASER ?BIAS-PI)))))))


(DEFMLA SPEC-CE-DIR-VOL
      [SPEC-SCHEMA (CE-DIR-VOL-COUPLE ?D1 ?PRT2 ?D2)
                   (GENERAL-COUPLING-PLAN ?D1 (OUTPORT ?D1)
```

```
                                        ?PRT2 ?D2)))


(DEFMLA PORT-CONVERT
        [-/> C (AND (I-PORT ?PRT)
                    (=/> '(PORT-TERMINALS ?PRT) <?TRMIN1 ?TRMIN2>)
                    (=/> '(NODE-TERMINALS ?TRMIN1) ?TS))
               (/:TO-DO ?TASK (PORT-CONVERT ?PRT CURRENT VOLTAGE) <>
                  (CONFIG <RESISTANCE>
                     (\ (R)
                       <(/:FORK
                          (DEV-INSERT ?R ?TRMIN1
                                      (#1 ?R) ?TS (#2 ?R) <>)
                          (\ ()
                            <(RELABEL-PORT ?PRT I-PORT V-PORT)
                             (CONSTRAIN <'(I (#1 ?R)) (I ?TRMIN1)>
                                (\ (IR IT)
                                  (/>/> (ABS ?IR)
                                        (ABS ?IT))   )>   )>   )))))


(DEFMLA RELABEL
        [/:MOD-MANIP ?TSK (RELABEL-PORT ?PRT ?OLD ?NEW)
           <'(?OLD ?PRT)> <'(?NEW ?PRT)>])


(DEFMLA ACQ-NODE [NOT (REUSABLE NODE)])

(DEFMLA PRIM-NODE [PRIMITIVE-DEV-TYPE NODE] GENERAL-DP*)


(DEFMLA 2-TERMINAL-DEFN
        [-/> A (DEV-TYPE ?X 2-TERMINAL)
               (/:PKT 2-TERMINAL-PKT (?X)
                  (TERMINAL-NAMES ?X <#1 #2>)
                  (CONSTRAINT <'(I (#1 ?X)) '(I (#2 ?X))>
                              (\ (I1 I2) (= (+ ?I1 ?I2) 0)   ))
                  (CONSTRAINT <'(V ?X) '(V (#1 ?X)) '(V (#2 ?X))>
                              (\ (V V1 V2)
                                (= ?V (- ?V1 ?V2))   ))))]
        GENERAL-DP*)


(DEFMLA PRIM-RESIS [PRIMITIVE-DEV-TYPE RESISTOR])

(DEFMLA  RESISTOR-DEFN
         [-/> A (DEV-TYPE ?X RESISTOR)
                (/:PKT RESISTOR-PKT(?X)
                       (DEV-TYPE ?X 2-TERMINAL)
                       (CONTROL ?X R REAL$ 1.0)
```

```
                (CONSTRAINT <'(R ?X)>
                            (LAMBDA (R) (/>= ?R 0.0)   ))
                (CONSTRAINT <'(R ?X) '(V (#1 ?X))
                             '(V (#2 ?X)) '(I (#1 ?X))>
                   (LAMBDA (R V1 V2 I1)
                      (= (* ?I1 ?R) (- ?V1 ?V2))   ))))]

      GENERAL-DP*)

(DEFMLA ACQ-RESISTOR [NOT (REUSABLE RESISTOR)])


(DEFMLA PRIM-OPEN [PRIMITIVE-DEV-TYPE OPEN])

(DEFMLA OPEN-DEFN
        [-/> A (DEV-TYPE ?X OPEN)
               (AND (DEV-TYPE ?X 2-TERMINAL)
                    (CONSTRAINT <'(I (#1 ?X))>
                                (LAMBDA (I) (= ?I 0)   ))
                    (CONSTRAINT <'(I (#2 ?X))>
                                (LAMBDA (I) (= ?I 0)   ))))]

        GENERAL-DP*)


 (DEFMLA PRIM-SHORT [PRIMITIVE-DEV-TYPE SHORT])

 (DEFMLA SHORT-DEFN
        [-/> A (DEV-TYPE ?X SHORT)
              (AND (DEV-TYPE ?X 2-TERMINAL)
                   (CONSTRAINT <'(V (#1 ?X)) '(V (#2 ?X))>
                               (LAMBDA (V1 V2) (= ?V1 ?V2)   ))))]

         GENERAL-DP*)



  (DEFMLA PRIM-CAP [PRIMITIVE-DEV-TYPE CAPACITOR])
  (DEFMLA ACQ-CAP [NOT (REUSABLE CAPACITOR)])

  (DEFMLA CAP-DEFN
        [-/> A (DEV-TYPE ?X CAPACITOR)
               (/:PKT CAP-PKT (?X)
                      (DEV-TYPE ?X 2-TERMINAL)
                      (CONTROL ?X C REALS 1.2)
                      (CONSTRAINT <'(C ?X)>
                                  (LAMBDA (C) (/>= ?C 0.0)   ))
                      (N (DC) '(DEV-TYPE ?X CAPACITOR))
                      (T (DC) (DEV-TYPE ?X OPEN))
                      (N (SSS ?S) '(DEV-TYPE ?X CAPACITOR))
                      (T (SSS ?S) (AND (DEV-TYPE ?X RESISTOR)
                                       (=/> '(R ?X)
                                            (// 1.0 (* (C ?X) ?S)))))
                      (N (HIGH-FREQ) (DEV-TYPE ?X CAPACITOR))
                      (T (HIGH-FREQ) (DEV-TYPE ?X SHORT))))]

          GENERAL-DP*)
```

```
(DEFMLA PRIM-INDUC [PRIMITIVE-DEV-TYPE INDUCTOR])

(DEFMLA INDUC-DEFN
        [-/> A (DEV-TYPE ?X INDUCTOR)
               (/:PKT INDUC-PKT (?X)
                      (DEV-TYPE ?X 2-TERMINAL)
                      (CONTROL ?X L REALS 1.5)
                      (CONSTRAINT <'(L ?X)>
                                  (LAMBDA (L) (/>= ?C 0.0)   ))
                      (N (DC) '(DEV-TYPE ?X INDUCTOR))
                      (T (DC) (DEV-TYPE ?X SHORT))
                      (N (SSS ?S) '(DEV-TYPE ?X INDUCTOR))
                      (T (SSS ?S) (AND (DEV-TYPE ?X RESISTOR)
                                       (=/> '(R ?X)
                                            (* (L ?X) ?S))))
                      (N (HIGH-FREQ) (DEV-TYPE ?X INDUCTOR))
                      (T (HIGH-FREQ) (DEV-TYPE ?X SHORT))))]
        GENERAL-DP*)


(DEFMLA COMP-XFMR [COMPOSITE-DEV-TYPE TRANSFORMER])

(DEFMLA XFMR-DEFN
        [-/> A (DEV-TYPE ?X TRANSFORMER)
               (/:PKT XFMR-PKT (?X)
                  (TERMINAL-NAMES ?X <#L1 #L2 #R1 #R2>)
                  (CONTROL ?X TURNS-RATIO REALS 1)
                  (CONSTRAINT <'(TURNS-RATIO ?X)>
                              (\ (N) (/> ?N 0)   ))
                  (CONSTRAINT <'(I (#L1 ?X)) '(I (#L2 ?X))>
                              (\ (I1 I2) (= (+ ?I1 ?I2) 0)   ))
                  (CONSTRAINT <'(I (#R1 ?X)) '(I (#R2 ?X))>
                              (\ (I3 I4) (= (+ ?I3 ?I4) 0)   ))
                  (CONSTRAINT <'(V (#L1 ?X)) '(V (#L2 ?X))
                               '(V (#R1 ?X)) '(V (#R2 ?X))
                               '(I (#L1 ?X)) '(I (#R1 ?X))>
                              (\ (VL1 VL2 VR1 VR2 IL IR)
                                (= (+ (* (- ?VL1 ?VL2) ?IL)
                                      (* (- ?VR1 ?VR2) ?IR))
                                   0)   ))))])

; A TRANSFORMER MAY CONSIST OF TWO INDUCTORS

(DEFMLA DERIVED-XFMR
        [/:TO-DO ?TASK (MAKE TRANSFORMER) <?NAME>
            (/:DO-SUBNET (BUM-TWO-INDUCTORS) <XFMR>)]
        GENERAL-DP*)

(DEFMLA BUM-INDUCTORS-NET
   [-/> A (/:PLAN-INSTANCE ?PI (BUM-TWO-INDUCTORS) ?SUP)
          (AND (STASK (GET-INDUC-1 ?PI) ?SUP <>
```

```
                (\ () (ACQUIRE INDUCTOR)) <'(L1 ?PI)>)
             (STASK (GET-INDUC-2 ?PI) ?SUP <>
                (\ () (ACQUIRE INDUCTOR)) <'(L2 ?PI)>)
             (STASK (ABRA ?PI) ?SUP <'(L1 ?PI) '(L2 ?PI)>
                (\ (L1 L2) (GRABBA
                             (\ (X)
                               (MAIN-DEV-TYPE ?X TRANSFORMER)   )))
                <'(XFMR ?PI)>)
             (STASK (CADABRA ?PI) ?SUP
                    <'(L1 ?PI) '(L2 ?PI) '(XFMR ?PI)>
                (\ (L1 L2 XFMR)
                   (/:INFER '(COMPONENTS ?NAME <?L1 ?L2>)
                            <>)   )
                <>)
             (/:MAIN (ABRA ?PI) ?SUP)))]
  GENERAL-DP*)

(DEFMLA IDEAL-VS [IDEAL-DEV-TYPE VOLTAGE-SOURCE])

(DEFMLA REUSABLE-VS [REUSABLE VOLTAGE-SOURCE])

(DEFMLA VS-DEFN
        [-/> A (DEV-TYPE ?X VOLTAGE-SOURCE)
               (AND (DEV-TYPE ?X 2-TERMINAL)
                    (CONTROL V ?X REALS 1)
                    (CONSTRAINT <'(V ?X) '(V (#1 ?X)) '(V (#2 ?X))>
                       (\ (V V1 V2) (= ?V (- ?V1 ?V2))   )))])

(DEFMLA PRIM-BJT [PRIMITIVE-DEV-TYPE BJT])

(DEFMLA BJT-DEFN
  [-/> A (DEV-TYPE ?Q BJT)
         (/:PKT BJT-PKT (?Q)
                (TERMINAL-NAMES ?Q <BASE EMI COL>)
                (CONTROL POLARITY ?Q <+1 -1> 1) ;NPN VS PNP
                (CONTROL BETA ?Q (INTERVAL 10 500) 10.)
                 ;BETA CONTROLLABLE UP TO ORDER OF MAGNITUDE
                (CONTROL RPI ?Q REALS 1.5)
                 ;INCREMENTAL BASE RESISTANCE

                (-/> A (MODE ?Q ?M)
                       (STASK (BIASER ?Q ?M) (DEEP-FREEZE ?M) <>
                          (\ () (BIAS ?Q ?M)   ) <>))

                (-/> A (MODE ?Q ACTIVE)
                       (AND
                           (CONSTRAINT <'(I (BASE ?Q))
                                       '(I (COL ?Q)) '(BETA ?Q)>
                                       (LAMBDA (IB IC BETA)
                                          (= ?IC (* ?BETA ?IB))   ))
                           (CONSTRAINT <'(I (COL ?Q)) '(I (EMI ?Q))>
                                       (LAMBDA (IC IE)
```

```
                                    (= (+ ?IC ?IE) 0.0)   ))
                        (CONSTRAINT <'(I (BASE ?Q))> ZEROP)
                        (CONSTRAINT <'(V (BASE ?Q)) '(V (EMI ?Q))
                                    '(POLARITY ?Q)>
                           (LAMBDA (VB VE S)
                               (= (- ?VB ?VE) (* 0.6 ?S))   ))
                        (INEQ (V (BASE ?Q)) (V (EMI ?Q))
                              (POLARITY ?Q))
                        (INEQ (V (COL ?Q)) (V (BASE ?Q))
                              (POLARITY ?Q))
                        (INEQ (I (COL ?Q)) 0 (POLARITY ?Q))
                        (INEQ (I (BASE ?Q)) 0 (POLARITY ?Q))
                        (INEQ 0 (I (EMI ?Q)) (POLARITY ?Q))))

                (-/> A (MODE ?Q CUTOFF)
                       (AND
                           (CONSTRAINT <'(I (COL ?Q))> ZEROP)
                           (CONSTRAINT <'(I (BASE ?Q))> ZEROP)
                           (CONSTRAINT <'(I (EMI ?Q))> ZEROP)))

                (-/> A (MODE ?Q SATURATED)
                       (CONSTRAINT <'(V (COL ?Q)) '(V (EMI ?Q))
                                   '(POLARITY ?Q)>
                          (LAMBDA (VC VE S)
                             (= ?VC (+ ?VE (// (* 0.6 ?S) 2.0)))   )))

                (N (INC) '(CONSTRAINT <'(V (BASE ?Q)) '(V (EMI ?Q))
                                      '(POLARITY ?Q)>
                            (LAMBDA (VB VE S)
                                (= (- ?VB ?VE) (* 0.6 ?S))   )))
                (T (INC) (CONSTRAINT <'(V (BASE ?Q)) '(V (EMI ?Q))
                                     '(I (BASE ?Q)) '(RPI ?Q) >
                             (\ (VB VE IB R)
                               (= (- ?VB ?VE) (* ?IB ?R))   )))

))
  GENERAL-DP*)

(DEFMLA INEQ-1
        [-/> A (INEQ ?X ?Y ?S)
               (AND (-/> C (/> ?S 0) (/> ?X ?Y))
                    (-/> C (/< ?S 0) (/< ?X ?Y)))])

;COMPOSITE DEVICES

(DEFMLA SIG-TRANSER-GLORIA-MUNDI
        [-/> A (DEV-TYPE ?X SIG-TRANSER)
               (PORTS ?X <(INPORT ?X) (OUTPORT ?X)>) ])


(DEFMLA NODES-DECL-DEFN
        [-/> A (NODES ?DEV ?NODE-TUP)
               (-/> G (ELT ?N ?NODE-TUP) (MAIN-DEV-TYPE ?N NODE))]
```

```
        GENERAL-DP*)


(DEFMLA PORTS-DECL-DEFN
        [-/> A (PORTS ?DEV ?PORT-TUP)
               (-/> G (ELT ?PRT ?PORT-TUP) (MAIN-DEV-TYPE ?PRT PORT))])

(DEFMLA AMP-SIG-TRANS [SUB-DEV-TYPE AMPLIFIER SIG-TRANSER])


(DEFMLA CE-BASIC [BASIC-DEV-TYPE CE])

(DEFMLA CE-AMP [SUB-DEV-TYPE CE AMPLIFIER])

(DEFMLA MOST-GEN-CE [MOST-GENERAL-SPEC CE GENERAL-CE])

(DEFMLA CE-DEFN
  [-/> A (DEV-TYPE ?CE GENERAL-CE)
         (/:PKT CE-PKT (?CE)
                (COMPONENTS ?CE <(Q ?CE)>)
                (NODES ?CE <(BNODE ?CE) (ENODE ?CE) (CNODE ?CE)>)

                (DEV-TYPE (Q ?CE) BJT)
                (MODE (Q ?CE) ACTIVE)
                (/:SUBTASK (BIASER (Q ?CE) ACTIVE) (DEEP-FREEZE ?CE))
                (/:TO-DO (BIASER (Q ?CE) ACTIVE) ?A <>
                   (/:DO-SUBNET
                      (TYPICAL-BJT-ONE-STAGE-BIAS (Q ?CE)) <>))


                (=/> '(NODE-TERMINALS (BNODE ?CE)) <(BASE (Q ?CE))>)
                (=/> '(NODE-TERMINALS (ENODE ?CE)) <(EMI (Q ?CE))>)
                (=/> '(NODE-TERMINALS (CNODE ?CE)) <(COL (Q ?CE))>)

                (I-PORT (INPORT ?CE))
                (PORT-TERMINALS (INPORT ?CE)
                                <(BNODE ?CE) (ENODE ?CE)>)
                (I-PORT (OUTPORT ?CE))
                (PORT-TERMINALS (OUTPORT ?CE)
                                <(CNODE ?CE) (ENODE ?CE)>)
              )]
  GENERAL-DP*)


(DEFMLA DEFAULT-CE [DEFAULT-SPEC CE TYPICAL-CE])

(DEFMLA DERIVATION-TYP-CE [DERIVED TYPICAL-CE GENERAL-CE])

(DEFMLA TYPICAL-CE-DEFN
  [/:ANTEC (NOT (DEV-TYPE ?TYP-CE TYPICAL-CE))
           (/:PKT TYPICAL-CE-PKT (?TYP-CE)
                (DEV-TYPE ?TYP-CE CE)
                (COMPONENTS ?TYP-CE <(Q ?TYP-CE) (R1 ?TYP-CE)
```

```
                        (R2 ?TYP-CE) (RE ?TYP-CE)
                        (RL ?TYP-CE)>)

(MAIN-DEV-TYPE (Q ?TYP-CE) BJT)
(MODE (Q ?TYP-CE) ACTIVE)
(MAIN-DEV-TYPE (R1 ?TYP-CE) RESISTOR)
(MAIN-DEV-TYPE (R2 ?TYP-CE) RESISTOR)
(MAIN-DEV-TYPE (RE ?TYP-CE) RESISTOR)
(MAIN-DEV-TYPE (RL ?TYP-CE) RESISTOR)

(NODES ?TYP-CE <(BNODE ?TYP-CE) (CNODE ?TYP-CE)
                (ENODE ?TYP-CE) (GND ?TYP-CE)
                (PNODE ?TYP-CE)>)

(=/> '(NODE-TERMINALS (BNODE ?TYP-CE))
   <(BASE (Q ?TYP-CE)) (#2 (R1 ?TYP-CE))
    (#1 (R2 ?TYP-CE))>)
(=/> '(NODE-TERMINALS (ENODE ?TYP-CE))
     <(EMI (Q ?TYP-CE)) (#1 (RE ?TYP-CE))>)
(=/> '(NODE-TERMINALS (CNODE ?TYP-CE))
     <(COL (Q ?TYP-CE)) (#2 (RL ?TYP-CE))>)
(=/> '(NODE-TERMINALS (PNODE ?TYP-CE))
     <(#1 (R1 ?TYP-CE)) (#1 (RL ?TYP-CE))>)
(=/> '(NODE-TERMINALS (GND ?TYP-CE))
     <(#2 (R2 ?TYP-CE)) (#2 (RE ?TYP-CE))>)

;FROZEN TASKS
(/:PLAN-INSTANCE (FROZEN-BIAS-PLAN ?TYP-CE)
   (TYPICAL-BJT-ONE-STAGE-BIAS (Q ?TYP-CE))
   (BIASER (Q ?TYP-CE) ACTIVE))
(/:REDUCED (BIASER (Q ?TYP-CE) ACTIVE))

(FUNCTION (R1 ?TYP-CE) (BVD (FROZEN-BIAS-PLAN ?TYP-CE)))
(FUNCTION (R2 ?TYP-CE) (BVD (FROZEN-BIAS-PLAN ?TYP-CE)))
(FUNCTION (RE ?TYP-CE)
          (RESIS-GETTER (FROZEN-BIAS-PLAN ?TYP-CE)))


(STASK (PORT-CVT ?TYP-CE) (DEEP-FREEZE ?TYP-CE) <>
   (\ () (CONVERT-PORT (OUTPORT ?TYP-CE)
                       CURRENT VOLTAGE)  )
   <>)
(FUNCTION (RL ?TYP-CE) (PORT-CVT ?TYP-CE))

(EXPANSION-OBL ?TYP-CE (FIX '(V (PNODE ?TYP-CE))))
(/:SUBTASK (OBL ?TYP-CE (FIX '(V (PNODE ?TYP-CE))))
           (BIASER (Q ?TYP-CE) ACTIVE))

(CONSTRAINT <'(POLARITY (Q ?TYP-CE))
             '(SIGN (V (PNODE ?TYP-CE)))>
            =)
(CONTROL V-GAIN ?TYP-CE (INTERVAL -50 50) 2)
(CONSTRAINT <'(V-GAIN ?TYP-CE) '(R (RL ?TYP-CE))
```

```
                        '(R (RE ?TYP-CE))>
                        (LAMBDA (AV RL RE) (= ?AV (// ?RL ?RE))   ))
            ))
  GENERAL-OP*)



(DEFMLA BASIC-VD (BASIC-DEV-TYPE VD))

(DEFMLA VD-DEFN
       (-/> A (DEV-TYPE ?VD VD)
            (/:PKT VD-PKT (?VD)
                   (COMPONENTS ?VD <(R1 ?VD) (R2 ?VD)>)
                   (NODES ?VD <(TOP ?VD) (MID ?VD) (BOT ?VD)>)

                   (MAIN-DEV-TYPE (R1 ?VD) RESISTOR)
                   (MAIN-DEV-TYPE (R2 ?VD) RESISTOR)

                   (=/> '(NODE-TERMINALS (TOP ?VD)) <(#1 (R1 ?VD))>)
                   (=/> '(NODE-TERMINALS (MID ?VD))
                        <(#2 (R1 ?VD)) (#1 (R2 ?VD))>)
                   (=/> '(NODE-TERMINALS (BOT ?VD))
                        <(#2 (R2 ?VD))>)

                   (DEV-TERMINALS ?VD
                                  <(TOP ?VD) (MID ?VD) (BOT ?VD)>)

;FROZEN TASKS
                   (EXPANSION-OBL ?VD (FIX '(V (TOP ?VD))))
                   (EXPANSION-OBL ?VD (FIX '(V (BOT ?VD))))

                   (CONSTRAINT <'(V (TOP ?VD)) '(V (BOT ?VD))
                               '(V (MID ?VD))
                               '(R (R1 ?VD)) '(R (R2 ?VD))>
                      (\ (V1 V2 V R1 R2)
                        (= ?V (// (+ (* ?R2 ?V1) (* ?R1 ?V2))
                                  (+ ?R1 ?R2)))   ))

                   (CONSTRAINT <'(I (MID-NODE ?VD))
                               '(I (#2 (R1 ?VD)))>
                      (\ (I I1) (/</< (ABS ?I) (ABS ?I1))   ))
                   (CONSTRAINT <'(I (MID-NODE ?VD))
                               '(I (#1 (R2 ?VD)))>
                      (\ (I I2) (/</< (ABS ?I) (ABS ?I2))   ))
))
       GENERAL-OP*)

(DEFMLA ACQ-VD (NOT (REUSABLE VD)))
```

```
;AS WITH CE, THERE IS AN ABSTRACT ECP WHICH IS A CURRENT AMPLIFIER

(DEFMLA BASIC-ECP (BASIC-DEV-TYPE ECP))

(DEFMLA ECP-IS-AMP (SUB-DEV-TYPE ECP AMPLIFIER))

(DEFMLA MOST-GENERAL-ECP (MOST-GENERAL-SPEC ECP GENERAL-ECP))

(DEFMLA ECP-DEFN
        (-/> A (DEV-TYPE ?ECP GENERAL-ECP)
             (/:PKT ECP-PKT (?ECP)
                  (COMPONENTS ?ECP <(Q1 ?ECP) (Q2 ?ECP)>)
                  (NODES ?ECP <(ENODE ?ECP)
                              (BNODE1 ?ECP) (BNODE2 ?ECP)
                              (CNODE1 ?ECP) (CNODE2 ?ECP)>)

                  (MAIN-DEV-TYPE (Q1 ?ECP) BJT)
                  (MODE (Q1 ?ECP) ACTIVE)
                  (MAIN-DEV-TYPE (Q2 ?ECP) BJT)
                  (MODE (Q2 ?ECP) ACTIVE)

                  (=/> '(NODE-TERMINALS (ENODE ?ECP))
                       <(EMI (Q1 ?ECP)) (EMI (Q2 ?ECP))>)
                  (=/> '(NODE-TERMINALS (BNODE1 ?ECP))
                       <(BASE (Q1 ?ECP))>)
                  (=/> '(NODE-TERMINALS (BNODE2 ?ECP))
                       <(BASE (Q2 ?ECP))>)
                  (=/> '(NODE-TERMINALS (CNODE1 ?ECP))
                       <(COL (Q1 ?ECP))>)
                  (=/> '(NODE-TERMINALS (CNODE2 ?ECP))
                       <(COL (Q2 ?ECP))>)

                  (PORTS ?ECP <OUTPORT-1 OUTPORT-2>)

                  (V-PORT (INPORT ?ECP))
                  (PORT-TERMINALS (INPORT ?ECP)
                                  <(BNODE1 ?ECP) (BNODE2 ?ECP)>)

                  (I-PORT (OUTPORT-1 ?ECP))
                  (PORT-TERMINALS (OUTPORT-1 ?ECP)
                       <(CNODE1 ?ECP) (ENODE ?ECP)>)
                  (I-PORT (OUTPORT-2 ?ECP))
                  (PORT-TERMINALS (OUTPORT-2 ?ECP)
                       <(CNODE2 ?ECP) (ENODE ?ECP)>)

                  (/:SUBTASK (BIASER (Q1 ?ECP) ACTIVE)
                             (DEEP-FREEZE ?ECP))
                  (/:SUBTASK (BIASER (Q2 ?ECP) ACTIVE)
                             (DEEP-FREEZE ?ECP))
                  (EXPANSION-OBL ?ECP (FIX '(I (ENODE ?ECP))))))))

(DEFMLA DEFAULT-ECP (DEFAULT-SPEC ECP ECP-DC-AMP))
```

```
(DEFMLA DERIVED-ECP-DC-AMP (DERIVED ECP-DC-AMP ECP))

(DEFMLA ECP-DC-AMP-DEFN
        (/:ANTEC (NOT (DEV-TYPE ?TYP-ECP ECP-DC-AMP))
                 (/:PKT ECP-DC-AMP-PKT (?TYP-ECP)
                    (COMPONENTS ?TYP-ECP
                       <(Q1 ?TYP-ECP) (Q2 ?TYP-ECP)
                        (RL ?TYP-ECP) (RE ?TYP-ECP)>)
                    (DEV-TYPE (RL ?TYP-ECP) RESISTOR)
                    (DEV-TYPE (RE ?TYP-ECP) RESISTOR)
                    (DEV-TYPE (Q1 ?TYP-ECP) BJT)
                    (DEV-TYPE (Q2 ?TYP-ECP) BJT)
                    (MODE (Q1 ?TYP-ECP) ACTIVE)
                    (MODE (Q2 ?TYP-ECP) ACTIVE)

                    (NODES ?TYP-ECP
                      <(ENODE ?TYP-ECP) (LOWNODE ?TYP-ECP)
                       (HIGHNODE ?TYP-ECP)
                       (C2NODE ?TYP-ECP) (B1NODE ?TYP-ECP)
                       (B2NODE ?TYP-ECP)>)
                    (=/> '(NODE-TERMINALS (ENODE ?TYP-ECP))
                      <(EMI (Q1 ?TYP-ECP)) (EMI (Q2 ?TYP-ECP))
                       (#1 (RE ?TYP-ECP))>)
                    (=/> '(NODE-TERMINALS (LOWNODE ?TYP-ECP))
                      <(#2 (RE ?TYP-ECP))>)
                    (=/> '(NODE-TERMINALS (HIGHNODE ?TYP-ECP))
                      <(COL (Q1 ?TYP-ECP)) (#1 (RL ?TYP-ECP))>)
                    (=/> '(NODE-TERMINALS (C2NODE ?TYP-ECP))
                      <(COL (Q2 ?TYP-ECP)) (#2 (RL ?TYP-ECP))>)
                    (=/> '(NODE-TERMINALS (B1NODE ?TYP-ECP))
                      <(BASE (Q1 ?TYP-ECP))>)
                    (=/> '(NODE-TERMINALS (B2NODE ?TYP-ECP))
                      <(BASE (Q2 ?TYP-ECP))>)

;THESE ARE WAYS OF DOING TASKS IN ABSTRACT ECP...
                    (EXPANSION-OBL ?TYP-ECP
                                   (FIX '(V (LOWNODE ?TYP-ECP))))
                    (EXPANSION-OBL ?TYP-ECP
                                   (FIX '(V (HIGHNODE ?TYP-ECP))))

                    (REDUCE <(OBL ?TYP-ECP
                                  (FIX '(V (LOWNODE ?TYP-ECP))))>
                            (OBL (SOUL ?TYP-ECP)
                                 (FIX '(I (ENODE (SOUL ?TYP-ECP))))))

                    (REDUCE <(OBL ?TYP-ECP
                                  (FIX '(V (HIGHNODE ?TYP-ECP))))>
                            (BIASER (Q1 (SOUL ?TYP-ECP)) ACTIVE))
                    (REDUCE <(OBL ?TYP-ECP
                                  (FIX '(V (HIGHNODE ?TYP-ECP))))>
                            (BIASER (Q2 (SOUL ?TYP-ECP)) ACTIVE))

                    (STASK (CVT-PORT ?ECP) (DEEP-FREEZE ?TYP-ECP)
```

```
                    <> (\ () (PORT-CONVERT (OUTPORT ?TYP-ECP)
                                CURRENT VOLTAGE)  ) <>)
                (FUNCTION (RL ?TYP-ECP) (CVT-PORT ?ECP))

                (FUNCTION (RE ?TYP-ECP)
                          (OBL (SOUL ?TYP-ECP)
                               (FIX '(I (ENODE ?ECP)))))

                (CONSTRAINT <'(I (RE ?TYP-ECP))>
                            (\ (I) (= ?I 0.002)  ))
                (CONSTRAINT <'(I (COL (Q1 ?TYP-ECP)))
                             '(I (COL (Q2 ?TYP-ECP)))>
                            =))))


(DEFMLA RC-DEV-TYPE
   [-/> A (DEV-TYPE ?RC RC-FILTER)
          (/:PKT RC-PKT (?RC)
               (DEV-TYPE ?RC SIG-TRANSER)
               (COMPONENTS ?RC <(R1 ?RC) (C1 ?RC)>)
               (NODES ?RC <(NODE1 ?RC) (NODE2 ?RC) (NODE3 ?RC)>)
               (=/> '(NODE-TERMINALS (NODE1 ?RC)) <(#1 (R1 ?RC))>)
               (=/> '(NODE-TERMINALS (NODE2 ?RC))
                   <(#2 (R1 ?RC)) (#1 (C1 ?RC))>)
               (=/> '(NODE-TERMINALS (NODE3 ?RC)) <(#2 (C1 ?RC))>)

               (V-PORT (INPORT ?RC))
               (PORT-TERMINALS (INPORT ?RC) <(NODE1 ?RC) (NODE3 ?RC)>)
               (V-PORT (OUTPORT ?RC))
               (PORT-TERMINALS (OUTPORT ?RC) <(NODE2 ?RC) (NODE3 ?RC)>)

               (CONTROL CUTOFF-FREQ ?RC POS-REALS 1)
               (CONTROL (H ?S) ?RC COMPLEX 1.2)

               (CONSTRAINT <'(CUTOFF-FREQ ?RC)
                           '(R (R1 ?RC)) '(C (C1 ?RC))>
                   (\ (F R C) (= ?F (// 1 (* ?R ?C)))  ))

               (CONSTRAINT <'((H ?S) ?RC) '(R (R1 ?RC)) '(C (C1 ?RC))>
                   (\ (H R C) (= ?H (// 1 (+ 1 (* ?R ?C ?S))))  ))
              )))
```

Appendix 4 -- Details of STP for Theorem Provers

The main requirement for an information-retrieval theorem prover is that it halt. This is hard because it must return as many answers as possible. If the theorem prover were the top level, as is usually the case, we could just let it run until we ran out of money or patience, but the problem solver above it needs its answers in a finite time. STP has been written with this emphasis in mind; in its design, I have sacrificed "completeness" to this "finiteness" requirement.

STP is organized as a backward-chaining PLANNER-like system. (Moore, 1975) Given a goal, it finds implications to back through. For example, with the goal "Refute [NOT (P ?X)]," it might back through [-/> C (AND (Q ?X) (R ?X)) (P ?X)], which is internally stored as the disjunction [/:CONSEQ (P ?X) (NOT (AND (Q ?X) (R ?X)))]]. The reason implications are internally disjunctions is that STP wants to retrieve /:ANTECs in the same fetch as /:CONSEQs, so it must put the index pattern in the same place. It is for this reason that atomic data are stored as [/:CONSEQ |pat| FALSE].

This backward chaining creates a conjunction of subgoals, each of which is treated similarly to the way the top-level one was. The other subgoals are held in abeyance while the chosen one is worked on. This is called "***splitting***" for reasons I will explain below. The chosen subgoal can sprout new subgoals, so the conjunction grows.

When a subgoal is reduced to FALSE, the resulting answer substitution is applied to the remaining conjuncts before they are split. When there aren't any more, the substitution is the final result. I will say more about "answer substitutions" below.

Backtracking is necessary because there may be more than one split, and because there may be more than one answer to try on remaining conjuncts. Backtracking is implemented by saving the theorem-prover state when such a choice arises; and restoring such states when branches run out of choices, and after each top-level answer has been found.

To limit backtracking, failures are not allowed to cause backtracking to an irrelevant choice point. Splits, when generated or augmented, are partitioned into "split groups," each member of which is a conjunction which shares no free variables with the others. (Ernst, 1973, Moore, 1975) For example, if [-/> C (AND (P ?x) (Q ?y)) (R ?x ?y)], the goal [NOT (R ?u ?v)] gives rise to two independent subgoals [NOT (P ?u)] and [NOT (Q ?v)]. All the answers are found to each, and the result is the "Cartesian product" of the answer sets of each. That is, if [P a] and [P b], and [Q c] and [Q d] are present, [R a c], [R a d], [R b c], and [R b d] are deducible.

A request to STP with free variables is interpreted as a demand for values of those variables which refute the request. In the example I just gave, the request [NOT (R ?u ?v)] is satisfied by four such sets. These are called ***answer substitutions.*** They are computed from the history of the matches from the original request to the instances of [FALSE] that are ultimately derived. For example, given the request [NOT (P ?x ?y)] and the axioms [/:CONSEQ (P ?u B) (NOT (Q ?u))] and [/:CONSEQ (NOT (Q A)) FALSE], the first backward chain to [NOT (Q ?x)] constrains ?y to be [B]; the chain to [FALSE] then sets ?x to [A].

This bookkeeping is handled in STP by use of the Boyer-Moore representation of clauses. (Boyer and Moore, 1972) The idea is to represent every clause as a pattern plus substitution. The substitution is called an ***environment.*** A formula represented this way is called a ***closure.*** Matching

two closures creates a new, more constrained environment which describes the new substitution as a further specification of the two old ones. In this way, each environment really specifies a binary tree of ***super-environments*** which parallel its deductive history. Boyer and Moore explain how a numerical "environment id" or ***envid*** can specify any branch of this tree. Giving two envids specifies the node where the two branches meet.

STP uses this device to keep track of answers to goals. The variable ANS-ENVID* specifies the envid of the current main goal. The variable MAIN-MAX* specifies the size of the tree above the main goal's environment; that is, ANS-ENVID* and ANS-ENVID* + MAIN-MAX* are two envids which uniquely specify the environment of the main goal. The function ENV-COLLAPSE takes the environment of a terminal [FALSE] and these two numbers and produces an environment with all the discovered constraints recorded. This mechanism makes unnecessary the "answer-predicate" construct of (Green, 1969).

Of course, this mechanism is more specialized than Green's, since it is not able to return "disjunctive" substitutions. Thus, it will not work if the request [NOT (P ?X)] appears with [/:CONSEQ (P A) (P B)], even though these formulas are provably inconsistent. (The [P B] detached by the first resolution matches the original goal.) It won't work because there is no assignment of one value to X which refutes [NOT (P ?X)]. This is not really a deficiency in doing information retrieval, but we must be careful to detect it. I will say how this is done after a short digression.

STP is a refutation-driven theorem prover. This means that it doesn't just back through implications, it also records the formulas it is trying to refute so that they can take part in deductions. When the prover returns, all of these effects must disappear. This is accomplished by "pushing" the current data pool, doing recording formulas in it, and "popping" at the end. (McDermott and Sussman, 1973) This is done for every subgoal as well.

Now the machinery can allow several kinds of interactions between goals. The kind I mentioned two paragraphs ago is called "befuddlement." This is when two matching subgoals specify conflicting substitutions. This is handled by a program (TP-STATUS-RECONCILE), which forces agreement between two conflicting lines of deduction; it insists that just one ANS-ENVID* be passed along.

Another kind of interaction is subsumption. If a new goal is subsumed by a fact not in the pushed data pool, it is abandoned. For example, even if there are axioms for proving [P ?X], there is no point in trying to prove [P A] (that is, refute [NOT (P A)]), if [NOT (P A)] is in the data base. If you succeeded in deriving such a proof, it would be of little value, since it would just prove the inconsistency of the data base with regard to this question.

More interesting is the case where the subsumer is another goal. This case must be noticed, since the subsumer may be a super-goal of the current one, and therefore an infinite recursion may be impending. In any case, there is usually no point in proceeding, so STP abandons this goal. (This puts the program even further from deductive completeness.) However, there is an important case in which mere abandonment is not enough. If the super-goal is a main super-goal which is a variant of the current one, the answers to the super-goal must be applied to this one. For example, given the axioms

```
[/:CONSEQ (ABOVE ?X ?Y) (NOT (ON ?X ?Y))]
[/:CONSEQ (ABOVE ?X ?Z)
          (NOT (AND (ABOVE ?X ?Y) (ON ?Y ?Z)))]
```

(cf. (Moore, 1975)), the request "Refute [NOT (ABOVE A ?V)]" will create a subgoal [NOT (ABOVE A ?Y)] which is subsumed by it. The resulting infinite recursion is unimportant to a plain theorem prover, but important to us. The solution is to connect the supergoal and subgoal in such a way that all answers, past and future, to the supergoal are translated into subgoal answers. Thus, if the data base contains [ON A B], [ON B C], and [ON C D], the given request will first generate V → [B]. The repeated subgoal will be noticed, and this first answer will be used, causing the detachment of subgoal [NOT (ON B ?V)]. When this succeeds, the answer V → [C] will have been found. Now this answer is used to reawaken the repeated subgoal again, this time detaching [NOT (ON C ?V)] and given answer V → [D]. The final goal [ON D ?V] produces no new answers.

The ability to notice and use repeated subgoals depends on the calculation of answers to subgoals as well as the main theorem-prover goal. This is surprisingly difficult to accomplish in conjunction with the split-group sorting mechanism. Because goals can be reordered, it is not obvious when the last subgoal of a goal has been finished. Every goal structure must store a stack of its ancestors. The program ANSWERS-RECORD checks this stack to see, for each ancestor, whether there are any outstanding sibling goals. If not, an answer for that ancestor may be recorded.

While working on one split of a goal, STP does not record competing splits in the data base. Thus, some of the more devious kinds of reasoning discussed by R. Moore (1975) are not noticed. This could be changed without too much trouble.

Other features:

Equality -- STP uses equalities of the form [=/> '|x| |y|] routinely, much as a programming language does. (Cf. Bledsoe and Tyson, 1975) The function FMLA-CLOSE-AND-DP-EVAL creates a Boyer-Moore closure, and attempts to evaluate subexpressions which have been changed by the new environment. Evaluation is a call to STP with a request to refute [=/> '|new pat| ?VAL]. Before this call is made, the variables in the pattern are "marked universal," meaning they are not allowed to be set by the matching done by STP; this turns out to be equivalent to the marking done in packets. (McDermott, 1975) If it were not done, more than one "value" could be derived for the new pattern.

This evaluation is done whenever a pattern is detached. For example, given the axioms

```
[=/> '(F (G A)) B]        [=/> '(F (G C)) D]
[/:CONSEQ (P (G ?X) ?Y) (NOT (Q ?X ?Y))]
[/:CONSEQ (Q A ?Y) (NOT (R ?Y))]
[R B]
```

and the request, "Refute [NOT (P ?U (F ?U))]," the system detaches first [NOT (Q ?X (F (G ?X)))], with U → [G ?X]. The attempt to evaluate [F (G ?X)] fails because X would have to be bound to A and/or C. The next subgoal is [NOT (R (F (G A)))], which is evaluated to become [NOT (R B)], which succeeds. The final answer is U → [B].

The same kind of substitution is done in a more limited way when equalities are to be refuted. For example, STP is told to prove [FORALL (X) (IMPLIES (ELT ?X <A B C>) (P ?X))] by asking it to refute [IMPLIES (ELT X!69 <A B C>) (P X!69)], where X!69 is a skolem form. It assumes [ELT X!69 <A B

C>] and [NOT (P X!69)]. The first subgoal becomes, *via* the definition of ELT, [OR (=/> 'X!69 A) (=/> 'X!69 B) (=/> 'X!69 C)]. When this is split, the first subgoal generated is [=/> 'X!69 A]. The only effect this can have is by substitution, so the system finds all formulas that mention X!69 and does the indicated replacements. (There won't be very many, because X!69 is a new skolem form.) In this case, the subgoal [NOT (P A)] will be generated. Similar things will happen in the other two branches of this split. To make this work requires a bit of mechanism not usually implemented in theorem provers; the data-base machinery of (McDermott, 1975) must be augmented so that from any pattern one can retrieve all the formulas which contain it. This is done by keeping track of all the positions an atom appears in, and intersecting the index buckets corresponding to atom positions compatible with the pattern.

Modality -- Complications are introduced by the use of data pools to implement "reference points." (Sect. II.B.2) When the system has a goal of the form [T |ref| |fact|], it attempts to "coerce" the ref into a data pool. It then pushes this data pool as it did the calling one, and puts the fact into it for refutation.

Data Dependencies -- STP keeps track of the data that support its conclusions. Its caller will use these to build data dependencies as described in Sect. II.D. The only tricky part to this is to make sure the data pools are kept straight. Whenever working on a [T...] expression causes a jump to a new pool, the supporters will be packaged up in the form (DD-T |pool name| |fact|). These ultimately end up in this form in the dependencies. (See Sect. II.D.)

Life with Boyer and Moore -- J Moore has informed me (personal communication) that several people have used their representation for clauses. (1972) For the benefit of those tempted to use it, I should report my experiences with it.

My original motivation for wanting to represent formulas as closures was to preserve a perspicuous representation of goals for interaction with advice. The usual predicate-calculus theorem prover renames all the variables in two formulas before unifying them; this is called "standardizing them apart." Boyer and Moore's motivation was to save storage by representing clauses incrementally, as the differences between input clauses and output clauses of deductions.

Neither of these reasons turned out to be important, so that this is a classic example of the dangers of bottom-up programming, or anticipating problems that never arise or are swamped by other effects. As I discussed in Sect. VI.B, my deductive goals never interact with advice anyway; and Boyer and Moore's effort to save storage is wasted in a system, like mine, which must index each clause atom by atom. (Actually, the system is not quite that stupid, but I think any savings incurred are minute.)

On the other hand, the representation has proven to have several very natural uses. The packet machinery of (McDermott, 1975) "actualizes" potential items by just replacing the potential environment with the real one. The answer calculation described above is completely natural in a system that represents clauses by their deductive histories. The evaluation machinery that tries to evaluate only subexpressions with newly instantiated variables operates by traversing the expression to be evaluated, comparing variable

values in the old environment with those in the new.

On balance, my conclusion is that these advantages do not outweigh the disadvantages, which are considerable. The principal one is that CAR and CDR are useless for operating on closures. You cannot take the first step into a closure without setting up its environment. Every function that manipulates them must treat assigned variables as though they were transparent (i.e., go straight to their values); and correctly handle envid's that direct attention to the proper parts of the environment tree. Boyer and Moore show in their paper that it is easy to write a unification algorithm for closures. Indeed, it is easy to write any one algorithm, but the fact that *every* manipulation has to take the same tedious cases into account is something I didn't foresee. I solved this problem to some degree by writing a set of functions (with names like FMAP and FHACK) which (analogously to LISP's MAP functions) map operations over Boyer-Moore data structures. But these functions have to bind special variables and use MACLISP functional arguments, so they are of necessity quite slow. To get the CAR of a formula, you have to write (FHACK (FUNCTION CAR) FMLA); this binds two special variables, does an uncompilable property-list lookup on CAR, and still returns an object (such as "(#F-CLOSURE (P ?X) 3)") which is meaningless without further bit-picking in the formula's environment tree.

Alas, these problems are complicated an order of magnitude by interactions with segment forms and embedded formulas. (See Appendix 1.) Processing an embedded formula often requires setting up a stack of environments, which is pushed and popped as you go into a pair of brackets or encounter an escape form.

Finally, it is very hard to develop intuitions about the structure of environment trees. A buggy program that manipulates envids doesn't do anything radically different from a correct program; it's just looking at the wrong parts of the environment tree. The envids it uses are just little integers with little meaning. Debugging such a program usually comes down to experimenting with different numbers until something works!

Bibliography

Abelson, Robert (1975) "Concepts for Representing Mundane Reality in Plans," in Bobrow and Collins (1975).

Alexander, C. (1964) ***Notes on the Synthesis of Form***, Cambridge: Harvard University Press.

Asimow, M. (1962) ***Introduction to Design***, Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Bledsoe, W.W. (1975) ***Non-Resolution Theorem Proving***, The University of Texas at Austin, Departments of Mathematics and Computer Sciences Automatic Theorem Proving Project Memo ATP 29. Based on a Tutorial talk given at ***IJCAI 4***.

Bledsoe, W.W. and Mabry Tyson (1975) ***The UT Interactive Prover***, The University

of Texas at Austin, Departments of Mathematics and Computer Sciences Automatic Theorem Proving Project Memo ATP 17.

Bobrow, Daniel G. and Allan M. Collins (1975) (eds) ***Representation and Understanding***, New York: Academic Press.

Bobrow, Daniel G. and Bertram Raphael (1974) New Programming Languages for Artificial Intelligence Research, ***Computing Surveys 6***, No. 3, p. 155.

Bobrow, Daniel G. and Terry Winograd (1976) ***An Overview of KRL, A Knowledge Representation Language***, unpublished paper, version of May 28, 1976.

Boyer, R.S. and J.S. Moore (1972) "The Sharing of Structure in Theorem-Proving Programs," in Meltzer and Michie (1972).

Brand, Myles (1970) ***The Nature of Human Action***, Glenview, Illinois: Scott, Foresman, and Company.

Bressan, Aldo (1972) ***A General Interpreted Modal Calculus***, New Haven: Yale University Press.

Brown, A. (1975) ***Qualitative Knowledge, Causal Reasoning, and the Localization of Failures,*** Cambridge: unpublished MIT Ph.D. thesis.

Brown, A. and G.J. Sussman (1974) "Localization of Failure in Radio Circuits--A Study in Causal and Teleological Reasoning," Cambridge: MIT AI Lab Memo 319.

Buhl, H.R. (1962) ***Creative Engineering Design***, Ames, Iowa: The Iowa State University Press.

Buchanan, Bruce, Georgia Sutherland, and E.A. Feigenbaum (1969) "HEURISTIC DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry," in Meltzer and Michie (1969), p. 209.

Bundy, Alan (1975) "Analyzing Mathematical Proofs (Or Reading Between the Lines)," in ***Proc. IJCAI 4.***

Charniak, Eugene (1972) ***Toward a Model of Children's Story Comprehension***, Cambridge, Massachusetts: MIT AI Lab TR 266.

Charniak, Eugene (1975) "A Partial Taxonomy of Knowledge about Actions," ***Proc. IJCAI 4***, p. 91.

Chohan, V.C. and J.K. Fidler (1974) "Computer Aided Design of Filters for Data Transmission Using Frequency Modulation," ***Proc. Int. Conf. on CAD 1974.***

Danto, Arthur C. (1965) "Basic Actions," ***Am. Phil. Quart. 2***, p. 141. Also in Brand (1970), p. 255.

Darlington, J.L (1969) "Theorem Proving and Information Retrieval," in Meltzer and Michie (1969), p. 173.

Davis, Randall (1976) ***Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of a Large Knowledge Bases,*** Stanford AI Lab Memo AIM-283.

Davis, Randall, B.G. Buchanan, and Edward H. Shortliffe (1975) ***Production Rules as a Representation for a Knowledge-Based Consultation Program,*** Stanford University AI Laboratory Memo AIM-266.

Davis, Randall and Jonathan King (1975) ***An Overview of Production Systems,*** Stanford University AI Laboratory Memo AIM-271.

Director, S.W. (1974) "Towards Automatic Design of Integrated Circuits," in Spillers (1974), p. 303.

Doyle, Jon (1976) ***Analysis by Propagation of Constraints in Elementary Geometry Problem Solving.*** Cambridge: MIT AI Lab Working Paper 108.

Doyle, Jon (1977) ***Truth Maintenance Systems for Problem Solving,*** Cambridge, Mass.:MIT AI Laboratory Report TR-419.

Eastman, Charles M. (1968) ***Explorations of the Cognitive Processes in Design.*** unpublished Ph.D. dissertation, Carnegie-Mellon University.

Eastman, Charles M. (1969) "Problem-Solving Strategies in Design," ***Proc. Env. Des. Res. Ass. Conf.***

Electronic Design (1964) ***400 Ideas for Design,*** compiled by the editors of ***Electronic Design*** Magazine. New York: Hayden Book Company, Inc.

Elithorn, Alick and David Jones (1973) ***Artificial and Human Thinking.*** Amsterdam: Elsevier Scientific Publishing Company.

Ernst, George W. (1971) "The Utility of Independent Subgoals in Theorem Proving," ***Information and Control,*** April.

Ernst, George W. (1973) "A Definition-Driven Theorem Prover," ***Proc. IJCAI 3,*** p. 51.

Ernst, George W. and Allen Newell (1969) ***GPS: A Case Study in Generality and Problem-Solving,*** New York: Academic Press.

Fahlman, Scott (1973) ***A Planning System for Robot Construction Tasks,*** Cambridge: MIT AI Lab Technical Report 283.

Fahlman, Scott (1975) ***Thesis Progress Report: A System for Representing and Using Real-World Knowledge,*** Cambridge: MIT AI Lab Memo 331.

Farber, D.J., R.E. Griswold, and I.P. Polonsky (1964), "SNOBOL, A String Manipulation Language," ***JACM 11,*** p. 21.

Feigenbaum, E.A. and J. Feldman (1963) ***Computers and Thought,*** New York: McGraw-Hill Book Company.

Fikes, Richard and Nils J. Nilsson (1971) "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Proc. IJCAI 2*, p. 608.

Fletcher, A.J. (1974) "EUREKA -- A System for the Automatic Layout of Single-Sided Printed Circuit Boards," *Proc. Int. Conf. on CAD 1974.*

Floyd, Robert (1967) "Assigning Meanings to Programs," in J.T. Schwartz (ed.), *Mathematical Aspects of Computer Science*, p. 19.

Freeman, P. and Allen Newell (1971) "A Model for Functional Reasoning in Design," *Proc. IJCAI 2*, p. 621.

Furman, T.A. (1970) (ed.) *The Use of Computers in Engineering Design*, London: English Universities Press.

Glegg, Gordon Lindsay (1973) *The Science of Design*, Cambridge University Press.

Goldman, Alvin I. (1970) *A Theory of Human Action*, Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Grason, John (1970) *Methods for the Computer-Implemented Solution of a Class of "Floor Plan" Design Problems*, unpublished Ph.D. dissertation, Carnegie-Mellon University.

Gray, P.E. and C.L. Searle (1969) *Electronic Principles: Physics, Models, and Circuits*, New York: John Wiley & Sons, Inc.

Green, C. Cordell (1969a) *The Application of Theorem Proving to Question-Answering Systems*, Standford: Stanford University Computer Science Department Report CS-138.

Green, C. Cordell (1969b) "Theorem-Proving by Resolution as a Basis for Question-Answering Systems," in Meltzer and Michie (1969).

Haney, Frederick Marion (1968) *Using a Computer to Design Computer Instruction Sets*, Pittsburgh: Carnegie-Mellon University Computer Science Department Ph.D. thesis.

Hayes, Patrick J. (1973a) "The Frame Problem and Related Problems in Artificial Intelligence," in Elithorn and Jones (1973).

Hayes, Patrick J. (1973b) "Computation and Deduction," *Proc. MFCS Symp.* , Czech Acad. of Sciences.

Hayes, Patrick J. (1974) "Some Problems and Non-Problems in Representation Theory," Sussex: *Proc. AISB*, p. 63.

Hayes, Philip (1975) "A Representation for Robot Plans," *IJCAI 4.*

Hayt, William H. and Gerold W. Neudeck (1976) *Electronic Circuit Analysis and Design*, Boston: Houghton Mifflin Company.

Hewitt, Carl (1972) ***Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot***, Cambridge: MIT AI Lab TR-258.

Hintikka, Kaarlo Jaako Juhani (1962) ***Knowledge and Belief: an Introduction to the Logic of the Two Notions***, Ithaca: Cornell University Press.

Hoare, C.A.R. (1969) "An Axiomatic Basis for Computer Programming," *CACM 12*, p. 576.

Hughes, G.E. and M.J. Cresswell (1972) ***An Introduction to Modal Logic***, London: Methuen and Co Ltd.

King, J. (1969) ***A Program Verifier***, Pittsburgh: Carnegie-Mellon University Ph.D. thesis.

Kowalski, Robert (1973) ***Predicate Logic as Programming Language***, Edinburgh: University of Edinburgh Department of Computational Logic Memo 70.

Kowalski, Robert (1974) ***Logic for Problem Solving***, Edinburgh: University of Edingburgh Department of Computation Logic Memo 75.

Kowalski, Robert (1975) "A Proof Procedure Using Connection Graphs," *J. ACM 22*, p. 572.

Kuo, F.F. and W.G. Magnuson (1969) (eds.) ***Computer-Oriented Circuit Design***, Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Langford, Glenn (1971) ***Human Action***, Garden City, New York: Doubleday & Company, Inc.

Latombe, Jean-Claude (1976) ***Artificial Intelligence in Computer-Aided Design: The "TROPIC" System***, Menlo Park: Stanford Research Institute Artificial Intelligence Center Technical Note 125.

Lehnert, Wendy (1975) ***Question Answering in a Story Understanding System***, New Haven: Yale University Computer Science Research Report 57.

Marcus, M. (1973) ***Wait-and See Strategies for Parsing Natural Language***, Cambridge: M.I.T. A.I. Lab Working Paper 75.

Marcus, M. (1975) "Diagnosis as a Notion of Grammar," Pre-prints of Workshop on Theoretical Issues in Natural Language Processing, June 10-13, 1975, M.I.T.

Mason, Matthew (1976) ***Qualitative Simulation of Swine Production***, Cambridge: unpublished MIT bachelor's thesis.

Mathlab (1974) ***MACSYSMA Reference Manual***, Cambridge: MIT Artificial Intelligence Laboratory.

McCarthy, John (1959) "Programs with Common Sense," ***Proc. Symposium on Mechanization of Thought Processes I***, London: Her Majesty's Stationery

Office. Also in Minsky (1968), p. 403.

McCarthy, John and Patrick J. Hayes (1969) "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Meltzer and Michie (1969), p. 463.

McDermott, D. (1974a) *Assimilation of New Information by a Natural Language-Understanding System*, Cambridge: MIT AI Lab TR 291.

McDermott, D. (1974b) *Advice on the Fast-Paced World of Electronics*, Cambridge: MIT AI Lab Working Paper No. 71.

McDermott, D. (1975) *Very Large Planner-Type Data Bases*, Cambridge: MIT AI Laboratory Memo 339.

McDermott, D. and G.J. Sussman (1973) *The Conniver Reference Manual*, Cambridge: MIT AI Lab Memo 259a.

Meltzer, Bernard, and Donald Michie (1969) (eds.) *Machine Intelligence 4*, New York: American Elsevier Publishing Company, Inc.

Meltzer, Bernard, and Donald Michie (1972) (eds.) *Machine Intelligence 7*, New York-Toronto: John Wiley & Sons.

Michie, Donald (1968) *Machine Intelligence 3*, New York: American Elsevier Publishing Company, Inc.

Minsky, Marvin (1968) *Semantic Information Processing*, Cambridge, Massachussets: MIT Press.

Minsky, Marvin (1974) *A Framework for Representing Knowledge*, Cambridge: MIT AI Lab Memo 306. Revised version in Winston (1975).

Moore, J. and Allen Newell (1974) "How Can Merlin Understand?" in Gregg, L. (ed.) *Knowledge and Cognition*, Potomac, Maryland: Lawrence Erlbaum Associates.

Moore, Robert C. (1975) *Reasoning from Incomplete Knowledge in a Procedural Deductive System*, Cambridge, Mass.: MIT AI Laboratory TR 347.

Nevins, Arthur (1974a) "A Human Oriented Logic for Automatic Theorem-Proving," *JACM 21*, no. 4, p. 606.

Nevins, Arthur (1974b) *A Relaxation Approach to Splitting in an Automatic Theorem Prover*, Cambridge, Massachusetts: MIT AI Lab Memo 302. Also *Artificial Intelligence 6*, p. 25.

Nevins, Arthur (1974c) *Plane Geometry Theorem Proving Using Forward Chaining*, Cambridge: MIT AI Lab Memo 303.

Newell, Allen (1962) "Some Problems of Basic Organization in Problem-Solving Programs," in Yovitts, M., G.T. Jacobi, and G.D. Goldstein (eds.) *Self-Organizing Systems--1962*, New York: Spartan.

Newell, Allen (1973a) "Production Systems: Models of Control Structures," in Chase, W.C. (ed.) ***Visual Information Processing*** (New York: Academic Press), p. 463.

Newell, Allen (1973b) "Artificial Intelligence and the Concept of Mind," in Schank and Colby (1973), p. 1.

Nilsson, Nils J. (1971) ***Problem-Solving Methods in Artificial Intelligence***, New York: McGraw-Hill Book Company.

Nilsson, Nils J. (1973) ***A Hierarchical Robot Planning and Execution System***, Menlo Park, California: SRI Artificial Intelligence Center Technical Note 76.

Pople, H.E., Jr. (1973) "On the Mechanization of Abductive Logic," ***Proc. IJCAI 3.***

Powers, Gary J. (1972) "Computer Aided Synthesis of Chemical Processing Systems," ***Proc. 6th Princeton Conf. on Information Sci. and Systems***, p. 42.

Powers, Gary J. (1973) "Non-Numerical Problem Solving Methods in Computer Aided Design," in Vlietstra and Wielinga (1973).

Powers, Gary J. and Dale F. Rudd (1974) "A Theory for Chemical Engineering Design," in Spillers (1974).

Prior, Arthur N. (1957) ***Time and Modality***, Oxford: Clarendon Press.

Prior, Arthur N. (1967) ***Past, Present, and Future***, Oxford: Clarendon Press.

Rescher, Nicholas and Alasdair Urquhart (1971) ***Temporal Logic***, New York: Springer-Verlag.

Rieger, Charles (1976) "An Organization of Knowledge for Problem Solving and Language Comprehension," ***Artificial Intelligence 7***, No. 2, p. 89.

Robinson, J.A. (1965) "A Machine-oriented Logic Based on the Resolution Principle," ***JACM 12.***

Rosenbrock, H.H. (1974) ***Computer-Aided Control System Design***, London: Academic Press.

Rulifson, J.F., J.A. Derksen, and R.J. Waldinger (1972) ***QA4: A Procedural Calculus for Intuitive Reasoning***, Menlo Park: SRI Technical Note 73.

Rychener, Michael D. (1975) ***The Studnt Production System: A Study of Encoding Knowledge in Production Systems***, Pittsburgh: Carnegie-Mellon University Department of Computer Science.

Rychener, Michael D. (1976) ***Production Systems as a Programming Language for Artificial Intelligence Applications***, Pittsburgh: Carnegie-Mellon University Department of Computer Science, in preparation.

Sacerdoti, Earl D. (1975) ***A Structure for Plans and Behavior***, SRI Artificial Intelligence Center Technical Note 109.

Schank, Roger (1975) "The Structure of Episodes in Memory," in Bobrow and Collins (1975), p. 237.

Schank, Roger and Robert Abelson (1975) "Scripts, Plans, and Knowledge," ***Proc. IJCAI 4.***

Schank, Roger and Kenneth Mark Colby (1973) ***Computer Models of Thought and Language***, San Francisco: W.H. Freeman and Company.

Senturia, Stephen D. and Bruce D. Wedlock (1975) ***Electronic Circuits and Applications***, New York: John Wiley and Sons, Inc.

Shortliffe, Edward H. (1976) ***Computer-Based Medical Consultations: MYCIN***, New York: American Elsevier Publishing Company, Inc.

Siklossy, L. and J. Roach (1973) "Proving the Impossible is Impossible is Possible: Disproofs Based on Hereditary Partitions," in ***Proc. IJCAI 3***, p. 383.

Slagle, James R. (1971) ***Artificial Intelligence: The Heuristic Programming Approach***, New York: McGraw-Hill Book Company.

Spillers, William R. (1974) (ed.) ***Basic Questions of Design Theory***, New York: American Elsevier Publishing Company, Inc.

Srinivasan, Chitoor V. (1976a) ***Introduction to the Meta Description System***, New Brunswick, N.J.: Rutgers University Dept. of Computer Science SOSAP-TR-18.

Srinivasan, Chitoor V. (1976b) "The Architecture of Coherent Information System: A General Problem Solving System," ***IEEE Trans. on Computers C-25***, no. 4, p 390.

Stallman, Richard M. and Gerald J. Sussman (1976) ***Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis***, Cambridge: MIT AI Lab Memo 380.

Stickel, Mark E. (1975) "A Complete Unification Algorithm for Associative-Commutative Functions," ***Proc. IJCAI 4.***

Suppes, Patrick (1957) ***Introduction to Logic***, New York: Van Nostrand Reinhold Company.

Sussman, Gerald J. (1975) ***A Computer Model of Skill Acquisition***, New York: American Elsevier Publishing Company.

Sussman, Gerald J. and D.V. McDermott (1972) "From PLANNER to CONNIVER -- A Genetic Approach," ***Proc. FJCC 41***, p. 1171.

Sussman, Gerald J. and Richard M. Stallman (1975) "Heuristic Techniques in

Computer-Aided Circuit Analysis," *IEEE Trans. on Circuits and Systems 22*, p. 857.

Tarnlund, Sten-Ake (1975) "An Interpreter for the Programming Language Predicate Logic," *Proc. IJCAI 4*, p. 601.

Tate, Austin (1975) "Interacting Goals and Their Use," *Proc. IJCAI 4*.

Travis, Larry, Charles Kellogg, and Philip Klahr (1972) *Inferential Question-Answering: Extending CONVERSE*, mimeo.

Tulving, Endel and Wayne Donaldson (1972) *Organization of Memory*, New York: Academic Press.

Vlietstra, J. and R.F. Wielinga (1973) (eds.) *Computer-Aided Design*, Amsterdam: North-Holland Publishing Company, Inc.

Waldinger, Richard J. and K.N. Levitt (1974) "Reasoning about Programs," *Artificial Intelligence*.

Warren, David H.D. (1974) *WARPLAN: A System for Generating Plans*, Edinburgh: University of Edinburgh Department of Computational Logic Memo. No. 76.

Watson, J. (1970) *Semiconductor Circuit Design: for a.f. and d.c. Amplification and Switching*, London: Adam Hilger Ltd.

Winston, Patrick H. (1975) *The Psychology of Computer Vision*, McGraw-Hill.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 3 / 21 / 96

Report # AI-TR-402

Each of the following should be identified by a checkmark:
Originating Department:

- [x] Artificial Intellegence Laboratory (AI)
- [ ] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other:

Document Information

Number of pages: 266 (274-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

- [ ] Single-sided or
- [x] Double-sided

Intended to be printed as :

- [ ] Single-sided or
- [x] Double-sided

Print type:

- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [ ] Unknown
- [x] Other:

Check each if included with document:

- [x] DOD Form (2)
- [x] Funding Agent Form
- [x] Cover Page
- [ ] Spine
- [ ] Printers Notes
- [ ] Photo negatives
- [ ] Other:

Page Data:

Blank Pages (by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description :	Page Number:
IMAGE MAP!	(1-266) UN#'ED TITLE PAGE, UN# BLANK, 2-6, UN# BLANK, 7-263, UN# BLANK. (267-274) SCANCONTROL, COVER, FUNDING AGENT, DOD (2), TRGT'S (3)

Scanning Agent Signoff:

Date Received: 3 / 21 / 96 Date Scanned: 3 / 29 / 96 Date Returned: 4 / 4 / 96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-402	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Flexibility and Efficiency in a Computer Program for Designing Circuits		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Drew Vincent McDermott		8. CONTRACT OR GRANT NUMBER*(s)* N00014-75-C-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE June 1977 13. NUMBER OF PAGES 263
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Logic
Problem Solving
Circuit design
Artificial Intelligence
Representation
Data Base

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report is concerned with the problem of achieving flexibility (additivity, modularity) and efficiency (performance, expertise) simultaneously in one AI program. It deals with the domain of elementary electronic circuit design. The proposed solution is to provide a deduction-driven problem solver with built-in control-structure concepts. This problem solver and its knowledge base in the application areas of design and electronics are described. The program embodying it is being used to explore the solution of some modest problems in circuit design. It is concluded that shallow reasoning about

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

problem-solver plans is necessary for flexibility, and can be implemented with reasonable efficiency.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/01/1996

M.I.T. Libraries
Document Services