Report 83-45
Stanford -- KSL

A Framework for Circuit Design.
Christopher Tong,
Dec 1983

Heuristic Programming Project
HPP-83-45

December 1983

A Framework for Circuit Design

Christopher Tong

Department of Computer Science
Stanford University

To appear in the digest of papers
for COMPCON84
February, 1984

Stored: {Phylum}<Tong>Talks>CompConPaper.bravo HPP-83-45
Last revised: December 13, 1983 2:00 PM
First draft: October 24, 1983

A Framework for Circuit Design

Christopher Tong

Computer Science Department
Stanford University
Stanford, California 94305

to appear in
the digest of papers for COMPCON84
February, 1984

Abstract. Designers have many concerns. The process of circuit design is complex largely because the required knowledge takes many forms. We present a framework that contains such *design descriptions* as components, plans, goals, and tradeoffs. The design process is represented by *tasks*, which synthesize and revise descriptions, and *principles* that should be upheld by descriptions.

Control of the circuit design process involves sequencing the creation and execution of tasks and the maintainence of principles. Control is knowledge-intensive; different design processes are represented by such *control descriptions* as strategies and metagoals.

We provide examples of design tasks, principles, and control descriptions. Finally, we describe a computer program based on this framework.

1. An example

A digital system designer requires a FIFO stack in her system architecture. She recalls several alternatives for refining her abstract *component* description. The stack memory cells could be serially linked, or they could be accessed in parallel. She must choose a particular stack *refinement alternative.* The designer recognizes that a relevant *tradeoff* is that of "local vs. global connections." She also knows that global connections will tend to be more area-expensive than local connections.

The designer makes use of tradeoff information to service her *goals.* She chooses the serial implementation of the stack because her highest priority goal is that the stack take up as little space as possible.

We now present a framework for design that accounts for scenarios like the above. The basic constituents of this framework are design descriptions, principles, tasks, and control descriptions.

2. Design descriptions

Design descriptions are the data of the design process. Our framework contains several basic kinds of circuit design descriptions: components, plans, goals, and tradeoffs. (We will use these terms as abbreviations for "component descriptions", "plan descriptions", etc.)

Components. Any problem-solving framework must represent answers to the problem; in circuit design, the "answers" are component descriptions. Because design is an incremental activity, incomplete component descriptions must also be represented. Associated with each description may be several refinement alternatives, component descriptions that are refined versions of the original description.

In practice, a component description is complete when it can be fabricated. Relative to what can currently be fabricated (e.g. TTL descriptions and chip layouts), circuit designers create many intermediate descriptions of their end product that cannot be fabricated; they describe its function (what it does), its behavior (how it does it), and its control (when it does it). When they finally fully specify their circuit they have transformed such descriptions into explicit structural descriptions, describing connections and "subcomponent" relationships. In our viewpoint, functional, behavioral, and control descriptions are *implicit* structural descriptions, embedded in structural descriptions; hence the designer sees only one network, a network of structure. This approach avoids the problems of managing multiple networks – functional, behavioral, and structural – that are interlaced in complex ways. (For a related and more detailed discussion, see [Brown].)

Functional, behavioral, and control descriptions can be embedded in structural descriptions by viewing components as *processors*; each component can have multiple functions, behaviors, controls, data structures, datatypes, inputs, and outputs. We have created a processor language that can accomodate descriptions as abstract as architecture, and as concrete as circuits. Architectural descriptions usually describe data structures and behavior, and represent control abstractly; in circuit descriptions, these have been converted into explicit structure: the network of connected processors.

Plans. Plans are descriptions of the design process; they are the designer's "Things To Do". They describe the activities of creating or modifying descriptions, even other plans. Associated with each plan may be several alternatives for executing it. Plans are a special kind of *task* (see section 5); while most tasks are not reified as descriptions, instead being "hardwired" into strategies (see section 6), plans are descriptions and hence can be manipulated; they permit planning of the design process (by separating plan creation time from plan execution time), and explanation of design updates as consequences of earlier plan execution.

Goals. Why choose one refinement alternative over another? Or one plan alternative over another? Goals express the motivations of the designer, providing reasons for making such decisions. *Implementation goals* capture the desire to design a complete circuit description, i.e. one that can be fabricated; *subjective goals* permit the designer to define her notion of a good circuit description, usually through stipulations about circuit resources (delay time, number of connections, etc.).

What our goals are, how we prioritize them, and how we go about realizing them

changes in response to the components we create or fail to create. (See, for example, **P4** and **P5** in section **4.**)

Tradeoffs. Tradeoffs permit goals to be used to help make choices (e.g. choice of refinement alternative, or choice of plan execution alternative). In our framework, we distinguish four kinds of tradeoffs: component tradeoffs, plan tradeoffs, goal tradeoffs, and pattern tradeoffs.

A *component tradeoff* is an association of specific component refinement alternatives with subjective goals they satisfy. A *plan tradeoff* associates specific plan execution alternatives with subjective goals they satisfy. Such associations facilitate comparison of alternatives.

A *goal tradeoff* goes one step beyond such specific associations by making a general statement about goals that applies to *all* alternatives for a particular component or plan; "the more global connections, the larger the area of the stack" is an example.

Component, plan, and goal tradeoffs are compiled associations between alternatives and goals. When such associations are not available, one alternative can sometimes be distinguished from another using pattern tradeoffs. A *pattern tradeoff* is a pattern that is matched by a set of alternatives (usually one or two) – thus, that two alternatives fundamentally differ in the way they encode a particular input (parallel vs. serial) is represented by a "parallel vs. serial encoding" tradeoff. We call these patterns tradeoffs because, beyond recognizing distinctions, they also usually link particular distinctions with particular goals; for instance, a parallel encoding of data is *faster* than a serial encoding. Thus, after alternatives are *distinguished*, they can be *judged* with respect to their distinguishing features and the current set of goals. (Note: Our framework contains pattern tradeoffs for component refinement alternatives, but not for plan execution alternatives.)

Each kind of tradeoff has a distinct use. Goal tradeoffs detect sets of goals that are not concurrently achievable; component tradeoffs rule in specific component refinement alternatives; plan tradeoffs rule in specific plan execution alternatives; pattern tradeoffs rule out specific refinement alternatives.

~~~~~

The remaining discussion appeals to two distinct visualizations of the design process: one in which the design process proceeds through a *sequence* of design states (consisting of design descriptions), and another in which a *single* state continually evolves. Each view is used to clarify a different kind of point: the localized single state view permits such intuitively clear terminology as "creating or revising descriptions"; sequences of design states provide a better and broader visualization of the process of design.

From the single state viewpoint, descriptions change over time. How we view a description at a given moment is reflected by its *status*. Each description has at least two possible statuses, one indicating that it still has transformations to undergo (unrefined, unevaluatable, unexecuted), the other indicating that has been completely transformed (refined, satisfied, executed). The status of a description (component, goal, plan) is independent of the status of its subdescriptions (subcomponents, subgoals, subplans). The rationale behind this is: the fewer descriptions we need
~~~~~

actively consider the better; if we can actively remove from consideration superdescriptions that have been completely decomposed into subdescriptions, we need only consider the subdescriptions.

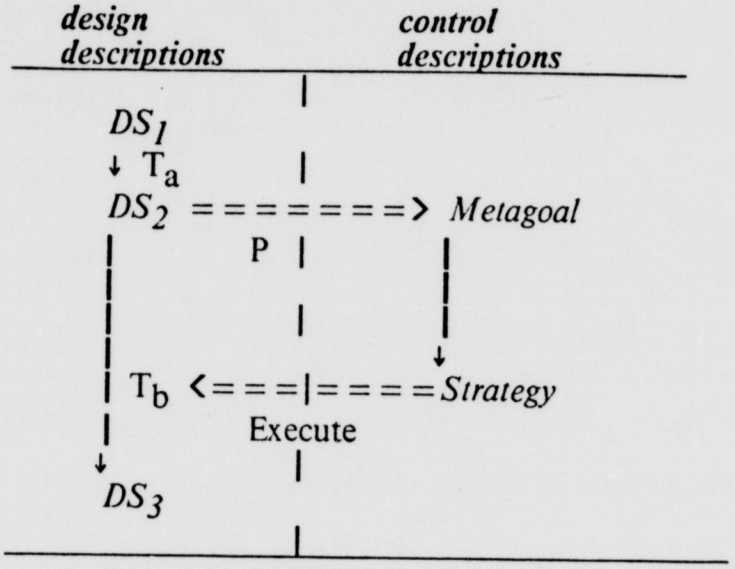
3. The big picture

An expert designer creates a circuit design. She conceptualizes her design process as a sequence of steps; as an expert, she can safely assume that, when necessary, she will be able to infer details missing in this picture, and can detect and revise those places where she has traded away total accuracy for simplicity and clarity in her account of the big steps. This manner of design allows her to rapidly explore many possibilities without having to fully detail them.

We do not claim that this story has universal psychological validity; it simply introduces our viewpoint that design centers around *transformation* of descriptions, and *maintainence* of basic design principles. *Tasks* transform by synthesizing or revising design descriptions. *Principles* describe relationships that should exist among descriptions. Principle maintainence involves detecting incompleteness in or violation of the relationship, and recording this information; unlike task execution, principle maintainence does not actually transform the design. A principle violation or incompleteness is sometimes easily fixed (e.g. adding a reverse pointer where there is currently only a forward pointer); remedying other violations can occupy much of the design process (e.g. making an unevaluatable goal evaluatable).

Categorizing design knowledge as transformation or maintainence knowledge distinguishes knowledge about synthesis and revision from knowledge about analysis. This distinction permits designers to take steps that are not necessarily complete or consistent, but whose incompleteness or inconsistency is noted and catalogued; thus, tasks can be honed for clarity and use in exploring alternatives instead of completeness or consistency. For example, one task might add a new component, but only crudely delineate its connections to its circuit environment. The designer is not forced to immediately rectify non-volatile violations (e.g. wires with bits coming in one end and signals going out the other); these simply remain in the background as "problems to be fixed."

When do we execute a task or check a principle? Which task do we execute? Which principle do we check? For which descriptions? The answers to these questions define the *control* of the design process implied by our framework. We now describe an architecture for control that accounts for control descriptions as well as the design descriptions discussed earlier. In the diagram below, DS_1-DS_3 are design states comprised of component, plan, goal, and tradeoff descriptions:



Tasks T_a and T_b transform one state into another. After a task is executed, all relevant principles P are checked.

When a principle is checked, and a violation is discovered or missing details are detected, a *metagoal* is created whose satisfaction should resolve the violation (or at least, tend toward its resolution), or fill in the missing details.

Our framework contains two kinds of metagoals: principle-driven metagoals and resource metagoals; principle-driven metagoals represent the need to resolve a design principle violation or incompleteness, while resource metagoals constrain or prioritize the resources of the design process (e.g. design time, quality, and complexity).

Earlier, we mentioned that the time at which a designer deals with a non-volatile violation or incompleteness should not be forced. As records of violations or incompleteness, principle-driven metagoals provide that flexibility.

To satisfy a metagoal, an associated *strategy* is created and executed. A strategy provides a control structure in which particular design tasks are executed on particular design descriptions. Strategy execution should partially or completely satisfy the associated metagoal.

We now highlight a few of the ideas behind our architecture for control:

In design, "what to do next" depends largely upon the current design state, and only minimally upon the sequence of steps that led to that state.

Control is the sequencing of task execution and principle maintainence.

Control is knowledge-intensive; designers make use of many strategies.

Control strategies can be phrased as nested sequences of simple task types.

Maintainence of a principle should be triggered by changes in those descriptions relevant to the principle.

A primitive task should be fully executed before principle maintainence takes place.

Maintainence of design principles can be used to motivate the execution of tasks.

Examples of strategies and metagoals will be provided in section **6.**

4. Design principles

We now describe some of the basic principles that constitute our circuit design theory.

P1. Description information should be preserved. The correlation between description and transformed description must be preserved. For example, an integer data structure should be refined into a number of bit data structures. Note that this principle is not to be identified with a task that performs the transformation. Maintainence of this principle provides a means for checking the legality of a task execution.

P2. Descriptions should obey composition rules. **P1** accounts for disappearing information, ensuring that it reappears in the appropriately transformed form. Description composition accounts for information whose appearance or modification impacts other information (in the same description or a different description). This principle deals with both *hierarchical* and *connective* composition of descriptions. As an illustration of hierarchical composition maintainence, when a component is made to point to a subcomponent, the reverse pointer should be installed (if it hasn't been already). As an example of connective composition maintainence, given a connection between two ports, if the datatype of one port (e.g. bit) is more refined than the datatype of the other (e.g. integer), that other port description must eventually be refined.

P3. The status of a description should be current. When the status of one description changes, the status of related descriptions may change. For example, after a plan has been executed, checking the status of the goal that brought the plan into existence (see **T4** in the next section) is a good idea.

P4. Goals should be satisfied. Goals are used to help make choices; when multiple component refinement alternatives exist, goals (in conjunction with component tradeoffs) can help prune alternatives; when multiple options for executing a plan exist, goals (in conjunction with plan tradeoffs) can help prune alternatives. Goals can also be used to locate "bottleneck" components.

A conflict situation exists when a choice point is reached and *no* alternative is consistent with every goal; resolution may come by relaxing or prioritizing goals. Similarly, a conflict situation exists if, at some point, a component description is discovered to be inconsistent with some goal; resolution can come by changing either the goal or the component description.

P5. Goal tradeoffs should be observed. Maintaining a goal tradeoff like "the more

global connections, the larger the area of the stack" requires making appropriate consequences occur when goals about both area and global connections are associated with a stack description (e.g. unpost or relax one of the goals).

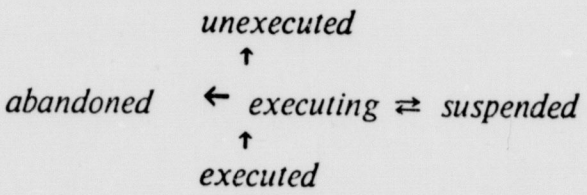
~~~~~

Violations of principles **P1** through **P5** can be resolved fairly quickly. We now list some "violations" whose detection or correction is inherently a long-term process.

P6. Goals should be evaluatable. Circuit designers often start out juxtaposing abstract component descriptions such as "a calculator with basic arithmetic functions" with concrete goals like "Make the delay time in performing the × operation small"; we cannot even *evaluate* the goal before our component description has structurally reified the "×" operation. Because such evaluation is necessary, we need to transform our component and goal descriptions so that they are at the same level of abstraction. Bringing this about can occupy much of the design process.

P7. Descriptions should be reduced. As tasks are executed, is it possible for task loops to occur? Can the same task be repeatedly executed on the same description? We will say descriptions are being *reduced* when they become more primitive or concrete; for component descriptions this means that embedded, implicit structural descriptions are being transformed into explicit structural descriptions. This principle tries to ensure that descriptions are continually reduced.

In general, detection of loops, deadlocks, or stagnation cannot be localized to observing a single task execution, but must involve sequences of tasks. (For example, in the "meet in the middle" strategy of section 6, a goal is actually *abstracted*, a move that locally appears to be non-reductive.) However, certain progressions can be "locally" observed and constrained. For example, the status of a description has a well-defined progressive direction; for example, here is the directed graph defining progress in plan status:



Together, principles **P1** and **P7** constitute a simple notion of *progress*: information is not lost, and information is continually being reduced to its lowest terms.
~~~~~

5. Design tasks

Principle maintainence does nothing but detect and report violations and incompleteness; tasks are needed to actually transform the design.

AI researchers have often viewed design as transformation of descriptions. In the simplest view, an incomplete description of the solution is continually refined by application of appropriate refinement rules (e.g. [Barstow]). In our view, not only solutions, but goals and tasks can also be continually transformed.

Earlier we mentioned that the design process could be viewed as transforming a single evolving design state. With respect to this viewpoint, we distinguish three overlapping categories of tasks: additive, substitutive, and choice-creating. Additive tasks (denoted by $\rightarrow a$) add new information to the design state; substitutive tasks ($\rightarrow s$) replace old information; finally, choice-creating tasks ($\rightarrow c$) combine the properties of additive and substitutive tasks, *adding* new descriptions (e.g. refinement alternatives), one of which can potentially *replace* an existing description.

We now present some of the basic kinds of tasks in our framework. We use a notation that displays the transformational nature of these tasks. In this notation, "*descr1*(*descr2*)" means "*descr1*, associated with *descr2*."

T1. Create refinement alternatives. In the simplest version of component refinement, one or more refinement alternatives for a component description can be created in a single step:

Comp, Goals(Comp), Tr(Comp) $\rightarrow c$ $\{Comp_i\}$

to refine

Comp: a component;
Goals: the subjective goals associated with Comp;
Tr: a component or goal tradeoff associated with Comp;
$\{Comp_i\}$: a subset of the refinement alternatives for Comp

---

Only those alternatives that are consistent with the goals (according to Tr) are generated.

T2. Prune refinement alternatives. When multiple refinement alternatives have been created (T1), tradeoffs can be used to distinguish, compare, and prune alternatives, relative to the current set of goals:

$\{Comp_i\}$, Goals(Comp), Tr $\rightarrow s$ $\{Comp_j\}$

to prune

Comp: a component;
$Comp_i$: a refinement alternative for Comp;
Goals: the goals associated with Comp;
Tr: a pattern tradeoff matching $\{Comp_i\}$;
$\{Comp_j\} \subset \{Comp_i\}$

~~~~~

T1 provides a direct way of refining a component, based on cached refinement alternatives. When such alternatives are not available, a more indirect route to refinement can be pursued, involving creation of an implementation goal (**T3**), creation of a plan to satisfy that goal (**T4** and **T5**), and execution of that plan (**T6**), to construct one or more refinement alternatives.

T3. Create an implementation goal. The desire to refine a component is represented explicitly as an implementation goal:

Comp $\rightarrow_a$ ImplGoal(Comp)
to goalify

Comp: a component;
ImplGoal: an implementation goal

T4. Plan how to satisfy a goal. A goal is either primitive, or composite, i.e. decomposable into subgoals. Primitive goals are considered satisfied when associated satisfaction-checking procedures say so; composite goals are considered satisfied after decomposition into subgoals.

One way to potentially satisfy a goal is to create an associated plan (whose execution may or may not be sufficient to satisfy the goal):

Goal(Comp) $\rightarrow_a$ Plan(Comp)
to plan

Goal: a goal; Comp: a component; Plan: a plan

T5. Create plan alternatives. Plans can usually be executed in a number of ways:

P(Comp), Goals(Comp), Trs(P) $\rightarrow_a$ {PlanAlt$_i$(P)}
to constrain

P: a plan;
Goals: the subjective goals for Comp;
Trs: the plan tradeoffs associated with P;
PlanAlt: an alternative for executing P

Plan tradeoffs help select execution alternatives that are consistent with the designer's goals.

T6. Execute a plan. As mentioned earlier, a plan is a special kind of task that is explicitly represented as a description. Plan execution usually modifies the set of component descriptions by creating new components, or by fleshing out or revising old ones; plans can also create more plans:
~~~~~

PlanAlt(Plan(Comp)) $\rightarrow_{s,a}$ ΔComponents $+\{\text{Subplan}_i(\text{Comp})\}$

to execute

Comp: a component; Plan: a plan;
PlanAlt: an alternative for executing Plan;
Components: the existing components;
Subplan_i: a subplan of Plan

For example, executing a plan often involves creating new subcomponents and subplans whose execution will connect the subcomponents.

T7. Decompose a description into non-redundant descriptions. Components and goals can be decomposed into subcomponents and subgoals, respectively:

Description $\rightarrow_{s,a}$ $\{\text{Descr}_i\}$

to decompose

Description: a description;
Descr_i: a subdescription of Description

Intelligent decomposition should only add subdescriptions that are not redundant, e.g. a redundant component or an already satisfied subgoal. For example, after creating a plan for a goal (T4 and T5), and executing that plan (T6), the original goal is decomposed only into subgoals that were not satisfied by the plan execution.

T8. Hierarchically compose descriptions. Descriptions can be hierarchically composed by making one description a subdescription of another description:

Descr_2 $\rightarrow_s$ Subdescr(Descr_1)

to compose

Descr_1, Descr_2: descriptions;
Subdescr: a subdescription of Descr_1

T9. Abstract a subjective goal. Because a very concrete goal is often impossible to evaluate against any current component description, it is useful to generate a related but more abstract goal:

SubjGoal_1(Comp) $\rightarrow_a$ SubjGoal_2(Comp)

to abstract

SubjGoal_1, SubjGoal_2: subjective goals

The abstracted goal is closer to being evaluatable against the current component description; satisfying the abstracted goal should help satisfy the original goal. For example, a subjective goal like "Make the number of interconnections in the final circuit small" might be abstracted to "Make the number of interconnections in the current processor description small." The abstractions of a goal and its subgoals are two different things; subgoals are at the same level of abstraction as the original goal, while abstracted goals are not.

Subjective goals and their abstractions tend to be only heuristically related; this introduces a degree of uncertainty into the results of an evaluation based on abstracted goals. Thus we only abstract *unconstrained* subjective goals; these take the form "Make the circuit's delay time small" or "Minimize the circuit's delay time", rather than "The delay time of the circuit must be less than 10 msecs."

T10. Budget a goal. When a subjective goal is associated with a component, and that component is decomposed into subcomponents, the goal can sometimes be partitioned in a corresponding manner:

$SubjGoal(Comp), \{Subcomp_i\} \rightarrow_a \{SubjGoal_i(Subcomp_i)\}$
to budget

Comp: a component; SubjGoal: a subjective goal;
$Subcomp_i$: a subcomponent of Comp;
$SubjGoal_i$: a subjective goal for $Subcomp_i$

Thus, a goal like "keep the number of circuit interconnections below 1000" might decompose into an interconnection-constraining goal for each subcomponent.

T11. Relax a goal. Sometimes the designer can find no way to meet a particular subjective goal. Rather than abandoning it altogether, she *relaxes* her original goal:

$SubjGoal_1(Comp) \rightarrow_s SubjGoal_2(Comp)$
to relax

$SubjGoal_1$, $SubjGoal_2$: subjective goals

Thus an upper bound on the number of interconnections might be increased; or it might be removed altogether, leaving an unconstrained goal like "Make the number of interconnections as small as possible, giving priority to satisfying goals G1 and G2."

6. Control descriptions

A strategy provides a control structure for sequencing task execution. We now illustrate strategies with examples.

Component refinement. Our framework accomodates component refinement

strategies that vary along two dimensions: refinement step size and number of refinement alternatives pursued. (By "step" we mean "task execution".)

Different strategies have different step sizes: that is, one might create a new refinement alternative in a single step, while another might take several steps. The size of the step is largely dependent on the knowledge available; usually we take the biggest step we can with the available knowledge. Typically, the smaller the size of the step, the more steps required, and the smaller the descriptions being pieced together to build the new refinement alternative.

The second dimension, number of refinement alternatives, depends on the importance of the component. For *bottleneck* components, that is, components whose design has a large impact on the total design, designers may want to explore multiple refinement alternatives, postponing a choice until more details are available.

We now present three strategies for component refinement, in order of decreasing step size.

Single step refinement strategy

For an unrefined component Comp **do**

Comp, Goals(Comp), Tr(Comp) $\rightarrow_{a}$ $\{Comp_i\}$ (T1)

to refine

For each pattern tradeoff PTr matching $\{Comp_i\}$ **do**

$\{Comp_i\}$, Goals(Comp), PTr $\rightarrow_{s}$ $\{Comp_j\}$ (T2)

to prune

Goals: the subjective goals associated with Comp;
Tr: a component or goal tradeoff associated with Comp;
$\{Comp_i\}$: a subset of the refinemennt alternatives for Comp;
$\{Comp_j\} \subset \{Comp_i\}$

Designers seem to pursue a single step refinement strategy for components they repeatedly use (like a FIFO stack); they remember respectable refinement alternatives *in toto*, and follow up on those that best match their goals. This approach is quick when possible; it is often impossible to pursue because no *in toto* refinement alternative is available (none were ever created, none were ever recorded, or none were small enough to record).

Single step refinement is a *generate-test* strategy. Component tradeoffs prune alternatives before generation, while pattern tradeoffs prune after generation.

Constructive refinement strategy

For an unrefined component Comp **do**

Comp $\rightarrow_a$ ImplGoal(Comp) (T3)
to goalify

ImplGoal(Comp) $\rightarrow_a$ Plan(Comp) (T4)
to plan

Plan(Comp), Goals(Comp), Trs(Plan) $\rightarrow_a$ $\{PlanAlt_i(Plan)\}$ (T5)
to constrain

While an unexecuted plan P(Comp) **do**

PlanAlt(P) $\rightarrow_{s,a}$ ΔComps $+\{Subplan_i(Comp)\}$ (T6)
to execute

For each pattern tradeoff PTr matching $\{Comp_i\}$ **do**

$\{Comp_i\}$, Goals(Comp), PTr $\rightarrow_s$ $\{Comp_j\}$ (T2)
to prune

ImplGoal: an implementation goal; Plan: a plan;
SGoals: the subjective goals for Comp;
Trs: the plan tradeoffs associated with P;
Comps: the existing components; $Subplan_i$: a subplan of P; $\{Comp_j\} \subset \{Comp_i\}$

Constructive refinement can be viewed as a procedural version of single step refinement. While single step refinement makes use of component descriptions recorded *in toto*, constructive refinement can be used to build a refinement alternative out of smaller component descriptions. Such a strategy is necessary if the component knowledge base is to be modular and of manageable size. Plan(Comp) will typically have multiple alternatives for execution, each leading to a different refinement alternative; the designer's goals are used to constrain the alternatives that are actually executed.

Like single step refinement, constructive refinement is also a generate-test strategy. Plan tradeoffs prune alternatives before generating them, while pattern tradeoffs prune after generation.

Attribute refinement strategy

For an unrefined component Comp **do**
Comp $\rightarrow_a$ ImplGoal(Comp) (T3)
to goalify
ImplGoal(Comp) $\rightarrow_{s,a}$ {Subgoal$_i$} (T7)
to decompose
While an unsatisfied goal G(Comp) **do**
G(Comp) $\rightarrow_a$ Plan(Comp) (T4)
to plan
Plan(Comp), Goals(Comp), Trs(Plan) $\rightarrow_a$ {PlanAlt$_i$(Plan)} (T5)
to constrain

While an unexecuted plan P(Comp) **do**
PlanAlt(P) $\rightarrow_{s,a}$ ΔComps + {Subplan$_i$} (T6)
to execute

For each pattern tradeoff PTr matching {Comp$_i$} **do**
{Comp$_i$}, Goals(Comp), PTr $\rightarrow_s$ {Comp$_j$} (T2)
to prune

ImplGoal: an implementation goal; Subgoal$_i$: a subgoal of ImplGoal;
Plan: a plan; Goals: the subjective goals associated with Comp;
Trs: the plan tradeoffs associated with Plan;
PlanAlt$_i$: an execution alternative for Plan;
Comps: the existing components; Subplan$_i$: a subplan of P; {Comp$_j$} $\subset$ {Comp$_i$}

Attribute refinement refines each component attribute individually (e.g. behavior, control, i/o); each Subgoal$_i$ corresponds to refinement of a different attribute. This kind of refinement makes use of remembered refinement alternatives for the primitive descriptions out of which component descriptions are built: functions, data structures, control, inputs, and outputs.

"Meet in the middle" strategy. Component refinement is one way to maintain principle **P6** – keep refining the component descriptions until they are at the same abstraction level as the goals. A more sophisticated approach transforms both goal and component descriptions; it abstracts the goals (**T9**), and refines the components (using any of the previously described strategies), attempting to "meet in the middle" to evaluate the abstracted goals against the refined components.

Resource metagoals. The strategies in the previous examples could all have been pursued to resolve violations of principles (e.g. **P6**). However, awareness of such violations is not sufficient to completely determine the flow of control. For example, in the strategies presented, we used the notation:

"**For** an unrefined component Comp **do**. . ."

but we provided no means for selecting Comp.

To solve such *focus of attention* problems, we solicit additional information from the designer. We ascertain and rank her *resource metagoals.* As delimiters of design process resources, these metagoals define the designer's notion of a *good* design

process. Below we present a table that describes strategies associated with the various resource metagoals:

resource metagoal	strategy description
quick design	bigger refinement step sizes prefered
	attention focused on highly interacting components
	goals relaxed sooner
	single refinement alternative pursued
good design	subjective goals solicited
	"meet in the middle" strategy is pursued
	attention focused on "bottleneck" components
	goals relaxed later
	multiple refinement alternatives pursued for "bottleneck" components
simple design	economic descriptions prefered
simple design process	simple staging strategy pursued
	modification prefered over backtracking

Table 1

For related work on strategies and metagoals, see [Stefik] and [Wilensky].

7. Implementation and conclusions

Depending on the amount of designer interaction required, a design program can be anything from a "computer environment" to an "expert system". The current version of our program lies somewhere between these extremes.

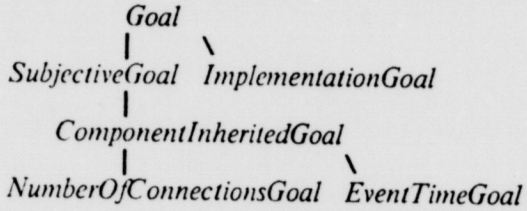
For each type of description, we have created a knowledge base. We are still in the process of extending these knowledge bases; however, we have proceeded far enough to demonstrate the framework's tenability for a few small-scale experiments (e.g. the design of FIFO stacks and barrel shifters).

We chose LOOPS as our implementation language mainly for its object-oriented programming paradigm (see [Bobrow]). In LOOPS, each object is an *instance* of another kind of object called a *class*. A class contain information shared by all its instances: a set of *variables* and default values for those variables; and a set of *methods*, procedures that can be invoked by sending a *message* to an instance of the class. Classes are useful for representing knowledge that change little with time;

instances of classes are created to represent the details of a particular problem.

Descriptions are implemented as objects; tasks are implemented as methods associated with the objects they transform. Plan and strategy descriptions have associated methods implementing their execution. Associations between descriptions appear as pointer values in description variables.

Description classes are organized into inheritance hierarchies; for example, a portion of the goal class hierarchy looks like:



A class inherits the variables and methods of the classes above it in the hierarchy. The ability to organize decription classes hierarchically has greatly simplified the representation of description features and tasks, as well as the incremental construction of description knowledge bases.

To implement principle maintainence, we built a "relationship" language, using LOOPS' data-access-oriented paradigm. Relationships are implemented as objects, while relationship maintainence is performed by methods associated with those objects. Relationship classes are associated with particular variables of a description class; whenever a new instance of the description class is created, a new instance of the relationship class is created and "attached" to the description instance variable. Whenever the value of an instance variable changes, the attached relationships are activated, causing the effects we associated earlier with principle maintainence to occur (e.g. creation of a principle-driven metagoal).

A number of features enhance the interface with the designer. Description editors and knowledge bases permit entering and saving design descriptions; description browsers (implemented using LOOPS' browser facilities) graphically display different kinds of associations among descriptions; tasks can be selected and executed using menus.

~~~~~

We have created an epistemology for circuit design. We have presented our framework in sufficiently general terms that its potential relevance for other design domains can easily be perceived and tested.

Our work is consistent with a major trend in current AI research: consolidation (see, e.g. [Bobrow]). To this effort we contribute a framework that integrates synthesis and analysis knowledge, and an architecture that exploits their differences.

Special thanks to Harold Brown and Bruce Buchanan, who went over earlier drafts of this paper with high-powered vacuum cleaners.
~~~~~

References

[Barstow]
Barstow, D. An experiment in knowledge-based automatic programming. *Artificial Intelligence* **12**, 1979, 73-119.

[Bobrow]
Bobrow, D. and Stefik, M. *The LOOPS manual.* Memo KB-VLSI-81-13, Xerox Palo Alto Research Center Aug. 1981, revised Jan. 1983.

[Brown]
Brown, H., Tong, C. and Foyster, G. Palladio: An exploratory environment for circuit design. *Computer Magazine*, December 1983.

[Stefik]
Stefik, M. Planning and meta-planning. *Artificial Intelligence* **16**, 1981, 141-170.

[Wilensky]
Wilensky, R. *Planning and understanding.* Addison-Wesley, Mass. 1983.