Program Structure and Design

ROBERT S. RIST
University of Technology, Sydney

Most models of computer programming explain the programmer's behaviour by a single design strategy. This article presents a cognitive architecture that uses cue-based search to model multiple design strategies including procedural, functional, means-end or focal, and opportunistic design. The model has been implemented in an artificial intelligence (AI) system that generates Pascal programs from English specifications.

Knowledge is represented as nodes that reside in internal or external memory, where a node encodes an action that may range from a line of code to a routine in size. A program is built by linking nodes through a search cue of the form <role, goal, object>. The cue is broadcast to memory, and any matching node is returned; the cue provides a question to ask, and the return provides the answer. A cue on the newly linked node is then selected as a new focus, and the search process repeated. Each design strategy defines a specific node visiting order that traverses the program structure through its links.

This article models program design as a process of search through the program structure. Some of this structure is captured in the final program code, but much of the fine structure is implicit in the design process and is not apparent in the final artifact. Viewed from a procedural perspective, a program consists of a set of roles connected by linear order; this is the way a serial computer views a program. Viewed from a functional perspective, a program consists of a set of actions connected by data and control flow dependencies, so they define a nonlinear plan that can be used by a parallel machine. A person sees a program as a set of nodes connected through their data flow, control flow, or linear order. Nodes occur at many levels of description, from the most basic concrete node that corresponds to a line of code, up to an abstract node that may consist of one or more routines of arbitrary size.

My thanks to Clayton Lewis for pieces of the puzzle, and to the reviewers for their excellent comments. The model owes a great debt to the work of Roger Schank, Donald Norman, and Bertrand Meyer. This research was supported by the Australian Research Council, Grant no. AC910654.

Correspondence and requests for reprints should be sent to Robert S. Rist, School of Computing Sciences, University of Technology, Sydney, P.O. Box 123 Broadway, Sydney, N.S.W. 2007, Australia. E-mail: <rist@socs.uts.edu.au>
A node is described or indexed by a tuple of the form \(<\text{role}, \text{goal}, \text{object}>\); this is the cue for the node. Program design occurs when a cue is selected, broadcast to memory, and the returned node is linked to the search cue. The search mechanism used to dynamically link nodes is based on Norman and Bobrow’s (1979) model of description-, cue-, or content-based search. Once the data and control flow links between a set of nodes have been defined, a minimal linear order can be defined by constraint propagation on the links. Nodes provide the static structure that is dynamically linked by cues, so program design is implemented by memory search, and a solution state is defined by the nodes used in the solution and the cues that link these nodes.

A design strategy is defined by a start cue, a level, a link type, and a direction to search. The abstract cue of \(<\text{find}, \text{average}, \text{rainfall}>\) and a direction of backward search through the data structure, for example, defines the question “Is there a plan that produces the average rainfall?” The search cue is broadcast to memory, and a matching target cue is returned. The search typically returns a node that is linked to the search cue, and provides a set of new cues, such as \(<\text{find}, \text{sum}, \text{rainfall}>\) and \(<\text{find}, \text{count}, \text{rainfall}>\). A new cue is then selected and used to search memory, so design proceeds by asking a series of questions in which the current question depends on the last answer. If the search fails, then each slot of the cue is used as a separate search cue to find a matching node structure, and the three returned structures are merged to define a program node. Knowledge is thus stored as plan schemas, objects, and roles, combined dynamically when needed, and the combination stored for later retrieval.

The choice of start cue, level, link type, and direction defines a design strategy and creates a specific visiting order that traverses the program structure. Each application of a design strategy creates a design episode that finds and connects pieces of the program structure until there are no places left to search from the initial point in the given direction in the given dimension; at that point, a new location, dimension, or direction has to be chosen to continue design. Procedural design is forward and role based, functional design is backward and goal based, and object-oriented design is object based. Top-down and bottom-up design traverse the levels of abstraction in the cues in different directions. This unifying framework is a gross simplification of the actual design process, due to the interaction between a search strategy and opportunistic design, plan creation and retrieval, working memory limitations, and the structure of the specification and the program.

The process of design can be captured by specifying the type of questions that are asked and the knowledge that supplies the answers. Knowledge can be stored in both internal and external memory. Internal memory consists of semantic and episodic memory, and external memory consists of the problem specification, scrap notes made during design, and the program itself. When a question is asked, these data sources are searched in turn until an
answer is found, and the answer provides a new piece that is linked to the existing program structure. The new piece may then be used to ask the next question, or some other part of the solution may be chosen as a focus for design. Search continues until there are no more questions to ask and the solution is complete, or until an answer cannot be found and the solution cannot be completed. There is no overall guidance or control; because a design choice is defined by the question "Where do I go from here?" all planning is local or situated at the current focus of attention, and the moving focus of attention defines a trajectory through the linked structure that is a program. This model of design is massively redundant, because design continues as needed until all the vacant slots in the program structure have been filled.

The model has been implemented in an artificial intelligence (AI) system called Zippy, that allows a user to specify the design strategies to be followed, the order in which the internal and external memories are searched, and the content of each memory. The system then generates code in the order defined by the strategy and knowledge, so the generation order implied by any particular strategy is completely defined. The code generation order seen in a human design protocol is easily matched to the system's generation order, and the system can be set to reproduce that order. The variability that is characteristic of human design means that each protocol requires a different set of parameters, so only a single protocol is described in this article.

This article presents a single cognitive architecture that gives operational definitions of and executable mechanisms for the main program design strategies described in the literature. The first three sections describe the structure of a program: first the procedural or linear structure, then the functional or plan structure, and then the cognitive or node structure of a program. The next three sections show how this structure is used to design a program: a general architecture for program design is presented, simulated, and applied to explain a human design protocol. The last section compares the model to related work in memory search and planning and explores the power of the model.

**PROCEDURAL STRUCTURE**

The procedural or linear structure of a program is defined by its control flow. More formally, the procedural structure is built from blocks of code, where each block or unit has a single entry and a single exit point in the control flow. A single control flow unit is called a *proper* (Hartman, 1991) and corresponds to a unit that encapsulates a sequence, a selection, or an iteration. The three unit types form the basis of structured programming, and any program may be built from these three types of blocks. Most formal analysis of program structure in computer science (Dijkstra, 1976; Dromey, 1989) has used the control flow to define a basic unit analysis.
The standard method for describing the structure of a software system is a top-down calling chart that shows the modules in the system and the control flow between modules. In such a chart, the control flow links define the order in which modules are called and executed, so the control flow is explicit in the calling graph. The modules are executed in a depth-first, left to right order in the chart, so data flow is implicitly left to right but is not explicitly shown. Top-down design has been the dominant prescribed design strategy in the literature for the last 20 years (Dijkstra, 1968; Naur, 1972; Sommerville, 1989; Wirth, 1971; Yourdon & Constantine, 1979) and has often been observed as a common expert design strategy (Adelson & Soloway, 1985; Jeffries, Turner, Polson, & Atwood, 1981). Top-down design starts with an abstract solution that is decomposed and refined until the basic actions can be implemented in code. Top-down design is reflected in the calling chart, where the top-level abstract solution corresponds to the top-level control module, and each level in the chart corresponds to a more specific version of the solution.

To illustrate the different types of structure used in this article, I use a variant of the classic Noah problem. The problem statement is: "Write a program that shows the average and maximum rainfall for a period. The end of the period is signaled by an input value of -99." A solution program is shown in Table 1.

The procedural structure of this program consists of three main units: the sequence before, within, and after the loop. Each of these three proper units is easily decomposed in turn, and the process can be continued until the level of program code is reached. A unit in a calling chart usually corresponds to a single routine, so a link corresponds to a routine call and the semantics of the chart are clean and simple. A chart for the Noah program is shown in Figure 1, where a link corresponds to a procedural decomposition; the bottom level of the chart has been omitted for simplicity. A procedure is easily wrapped around the code in each unit to transform the decomposition chart into a strict calling chart.

The order in which actions are executed is directly captured in the left to right order at any level in the chart. This order is retained by both the conceptual decomposition (abstract to concrete) and the physical decomposition (routine calls), so it is more general to speak of the order of actions instead of the calling order; in the code shown in Table 1, for example, there is a strict order and sequence of propers, but no routines are called. Each chunk in the chart is based on the role of the code within that chunk; the example reflects a common procedural decomposition into the three roles of input, calculate, and output. The procedural structure of a program is derived from the role structure of the code, where the roles provide the basic transform for each line of code. The set of concrete roles for Pascal is shown to the left of Table 2, and more abstract versions are shown to the right; there are many more abstract roles than are listed here. In the Programmer's Apprentice project (Rich & Waters, 1990), a role is a slot to be filled by some type
TABLE 1
A Program to Solve the Noah Problem

1. program Noah (input, output);
2. const endValue = -99;
3. var sum, rain, day, max: integer;
4. average: real;
5. begin
6. sum := 0;
7. max := 0;
8. day := 0;
9. write ('Enter rainfall for day ', (day + 1), ': ');
10. readln (rain);
11. while rain <> endValue do begin
12. day := day + 1;
13. sum := sum + rain;
14. if rain > max
15. then max := rain;
16. write ('Enter rainfall for day ', (day + 1), ': ');
17. readln (rain);
18. end;
19. writeln ('The largest rainfall was ', max);
20. average := 0.0;
21. if day > 0
22. then average := sum/day;
23. writeln ('The average rainfall was ', average);
24. end.
of data. The usage of the word role in this article is based more on the role in a play (Goffman, 1974; Schank & Abelson, 1977) and refers to a type of slot filler rather than the actual individual that fills the slot.

Examination of the code in Table 1 and the structure of Figure 1 shows that the structure does not directly correspond to the set of propers in the code. Looking at the code before the loop, for example, the proper decomposes into five unit sequences, not into the initialisations and the read. The abstract roles are useful descriptive terms for the decomposition, but they do not correspond exactly to an algorithmic analysis of the code into propers. A complete analysis into propers is easily done, but then it becomes very difficult to give labels to each box in the chart. The reason for this difficulty is that an analysis into propers does not capture the conceptual chunks in the program; it captures the executable chunks that reflect the way that the computer views the code.

The cognitive basis for top-down design is that it allows a problem to be broken down into parts, so that each of these subproblems can be solved in
relative isolation. Indeed, top-down design is often called top-down functional decomposition, to indicate that the problem is divided into functional units. In the Noah problem specification, for example, the problem clearly breaks down into two main subproblems: to find the average rainfall and to find the maximum rainfall. Each of these subproblems can be broken down in turn to define a top-down functional decomposition chart; to find the average, for example, we need to read in a stream of data, find the sum and count of the data values, and calculate the average. A partial top-down functional decomposition chart for the Noah problem is shown in Figure 2.

This chart shows the main conceptual chunks into which the problem decomposes, but the semantics of the chart are problematic. First, the lines in the chart no longer indicate control flow, but functional decomposition. Second, the order of the boxes does not reflect the linear order of the actions in the code. Third, the functional chunks do not correspond to the execution chunks (pros) in the structure chart; indeed, the elements of a functional chunk may be spread out in the program, as is visible for the sum chunk. Finally, the order of elements is in part arbitrary; the sum and count have to be calculated, but either the sum or the count may be calculated first. The advantages of the structure chart notation—simple control flow and linear order—are lost in the top-down functional decomposition. The semantics of the chart are a mix of functional decomposition and linear order, an extremely useful combination that has a complex semantics. The functional decomposition chart captures the rough linear order of the major pieces within each functional chunk, but loses the control flow and the total linear order.
Top-down functional decomposition may provide a basis for expert design, but it does not explain novice design. Anderson, Farrell, and Sauers (1984) based their GRAPES simulation of novice design on top-down, depth-first flow of control for a set of simple problems. For these problems, analogical mapping from retrieved solutions were successfully used to guide the design of a solution. Kessler and Anderson (1986) presented novice programmers with a solution to one problem, and asked them to modify it to solve a similar problem. The unsuccessful programmers used the linear structure of the solution as a guide and tried to modify each line of code in turn, from the first to the last, by analogical mapping from the surface structure of the code. Adelson (1984) noted that novices do not possess the abstract knowledge needed to build a top-down solution structure, so novice design in general cannot be explained as search through a (nonexistent) top-down chart. These results indicate that the procedural structure of the code, by itself, is not a good representation to support design.

2. PLAN STRUCTURE

There is considerable evidence in the empirical study of programming that the plan is the basic cognitive chunk used in program design and understanding. Exactly what is meant by a program plan, however, has varied considerably between authors. Two main usages observed in the literature may be called top-down and bottom-up plans. Early accounts of program design emphasized top-down planning, but attention has now shifted to more bottom-up components of design. The shift in emphasis and usage is similar to the shift from scripts (Schank & Abelson, 1977) to memory organization packets (Schank, 1982). Scripts are large, monolithic structures that are difficult to modify and index, whereas MOPs are much smaller pieces of knowledge that are flexibly combined to form the sequence of behaviours defined by a script. In the same way, simple plans are combined to form the long sequence of actions that make up a serial program.

Top-down plan analysis in programs was pioneered by Soloway, Bonar, and Ehrlich (1983), who showed that a small change in the code was enough to change a program from "plan-like" to not "plan-like." Soloway, Ehrlich, Bonar, and Greenspan (1982) defined three levels of planning called strategic, tactical, and implementation plans, where a strategic plan defines a global strategy, a tactical plan specifies a local strategy, and an implementation plan specifies the language-dependent techniques used to implement strategic and tactical plans. Spohrer, Soloway, and Pope (1985) analysed the structure of program code into a goal and plan (GAP) tree by defining a set of abstract sequential or top-down goals for their sample programs. Using this serial structure, they then showed that program variation could be described as the choice of different plans for the same goal.
Botton-up plan analysis separates the solution structure from the problem structure. Kant and Newell (1984; Kant, 1985) showed that design can occur in two main spaces, characterised as algorithm design space (the solution structure, reflected in the linear structure of the program) and application domain space (the problem structure, reflected in the nonlinear plan structure of the program). Design often began with a kernel idea in the problem space that was then developed by stepwise refinement in the solution space. Development of the initial idea was uneven when the designer could not retrieve a known solution and had to design that part in more detail. Rist (1986) used a code-sorting task to show that experts group "related" code on the basis of its nonlinear or plan structure, but novices focus more on the control structure and on syntactic cues. Letovsky (Letovsky & Soloway, 1986) showed that programs in which the plans were delocalised were difficult to understand; a delocalised plan is one in which a plan's actions are widely separated in the program structure.

Code is often added to a program one plan chunk at a time (Detienne, 1995; Rist, 1990, 1991). Novices are forced to do this because they code each line as soon as it is designed, so the actions in the plan appear one at a time, backward from the goal. Experts can retrieve a complete plan schema as a single chunk, or plan a solution mentally before writing any code; in these cases, the code appears in linear order from the first to the last action (Davies, 1991; Rist, 1989, 1991). For languages in which the plan structure is clear, such as Pascal, code tends to be added in plan chunks. This is not true for languages such as Basic, where the plan links are not apparent (Davies, 1990). When a programmer learns a new language, the plan structure of plans in the first language is transferred to the new language (Scholtz & Wiedenbeck, 1993), whether it is appropriate or not.

This section formally and algorithmically defines the plan structure of a program as "a set of nonlinear plans." Once the plan structure has been defined, the actions in the plans can be placed in a minimal linear order by simple constraint propagation, grouped into chunks that share a role, goal, or object (Rist, 1990), and enclosed in routines. The key to program design is thus the plan structure and not the linear order; plan structure is defined in this section, and the remainder of the article shows how plan structure is generated during design.

2.1 Representation
Plan structure shows the backward data and control flow links between lines of code that are often defined before the execution structure of the plan is known. These links define a set of piecewise dependencies between individual lines of code; when the links are traced backward from the goal, they define a backward slice through the program (Horwitz, Prins, & Reps, 1989; Weiser, 1982, 1984; Weiser & Lyle, 1986); the algorithm used in this article
use

obey

control

make

Figure 3. External structure of a node.

is given in Rist (1994). The backward dependency structure is a nonlinear plan (Sacerdoti, 1975). Given this plan structure, it is a simple exercise to order the lines of code by constraint propagation (Stefik, 1981), so the program can be executed on a serial computer.

A line of code may be linked to other lines of code in four ways; it may use data from one or more lines, make data for one or more lines, obey another line, or control one or more lines. The external structure of a node is shown in Figure 3.

A line of code is a concrete node in the plan structure, with up to four sets of links and an internal action that generates the program code. A node has four ports, called :use, :make, :obey, and :control; a port is labeled with the Common Lisp Object System (CLOS) slot notation (:use) to differentiate the port from the general concept (use). The content of each port is the set of nodes that are linked to the current node. Data is passed between nodes through variables, so the :use and :make ports have an entry for each variable used or made by the node. A line can use no, one, or more variables, and it can make no, one, or more data values. A line can obey no or one line; it can control no, one, or many other lines.

2.2 Algorithm

The goal of a program is given in the problem specification as an output, to the user, to a file, or to another program. The function of the program is to produce this output, so the functional structure of the program is defined by the dependency or plan structure backward from the goal. A complete plan structure for a program is a backward slice through the program code from an output. The Plan Analysis by Reverse Engineering (PARE) system produces plan structure from Pascal code (Rist, 1994; Rist & Bevemyr, 1991); it is described here to show that "program plan" has a clear and algorithmic definition that is easily related to the literature on plan schemas.

From each output, PARE traces back through first the data and then the control flow links for that output until it finds a set of terminals: constants,
read (rain)   endValue = -99
while rain <> endValue do
  read (rain)   sum := 0   day := 0
  sum := sum + rain   day := day + 1
if day > 0  average := sum/day  average := 0.0
write (average)

Figure 4. Plan structure for the average plan.

read statements, or an assignment that uses only literals. In this way, links are found and stored from each line of code to the lines that produce data for it, or control it, for every line that is used in the program. When a routine is found, both the plan structure in the calling module and the structure within the routine must be built. The plan structure in the calling module depends on the type of parameter passed. A call by value uses a value, but makes nothing; it has no output data flow. A call by reference uses a value, and makes a value; it has data flow both into and out of the calling parameters. The plan structure for a routine is built by tracing back from its outputs, then the routine is unfolded (Burstall & Darlington, 1977; Feather, 1982) and its plan structure is connected to the caller’s structure. Recursive routines are unfolded once.

The product of the tracing process is a directed graph. Joins are common when two lines obey the same line or use the same data value. When a join occurs, the tracing process terminates at that node, since all further links from that node are already defined, and recursively resumes at the last missing link. If a program has several outputs, the tracing process is repeated for each goal. The complete plan structure is a directed cyclic graph with multiple roots.

The Noah program introduced in the previous section has two goals, two values that must be output: the average and maximum rainfall values. The rest of the program is required purely to create and display these two values. The lines of code that define the plan for each goal are found by tracing back from the outputs along first the data, then the control flow links. The resulting plan structures are shown in Figures 4 (average) and 5 (maximum); in the figure, single arrows indicate data flow, and double arrows indicate control flow. Plan tracing begins with the last write statement, in this case for the average value, and continues back through first the data flow, and
The plan structure for the maximum plan is shown in Figure 5. The maximum plan accepts one data flow (a stream of rainfall) and produces one data flow (the maximum rainfall). Because much of the loop control and data structure was already built when tracing the average plan, the maximum plan structure does not include them; the joins from the data and control flow are indicated in the figure. The plan structure separates the plan cleanly and automatically from its context.

The PARE algorithm defines a specific order of link traversal (data then control, recursively), so the links shown in Figure 5 are precisely those created from the algorithm, minus the joins. Readers familiar with the program schema literature will recognise the count, sum, max, and read loop plans. Each plan is a single branch of the plan structure, and the plan actions are spread out or delocalised in the execution structure. The nonlinear nature of program plans and their difference from the execution structure of the program may be seen by examining the three initialisations (lines 6–8). These lines are separate in the plan structure, and even come from different plans, but are adjacent in the serial execution structure. Plan structure defines the essential dependencies and is used in design, but is hidden in the final execution structure.

### 2.3 Linear Structure

A plan is converted to executable code by placing the plan actions in a serial execution order by a process of constraint propagation on the dependency structure. Plan dependencies impose a necessary order on the plan actions, but they usually do not impose a single order; because the dependencies are piecewise, there are often many possible orders that obey the piecewise constraints.

There are two constraint rules for Pascal programs, each with a single exception. For data flow, a producer precedes the user of the data. The exception is a while loop with two producers, one before the loop and one in the loop; in that case, the producer inside the loop is placed after all its users.
For control flow, a line that obeys another is placed after and in the scope of its master. The exception is the *repeat* loop, where the control condition is at the bottom of the loop below the lines it controls. One general, additional rule is required when a data value is destroyed: The value must be used before it is destroyed. The number of ordering rules may increase as more Pascal constructs and plans are tested, but these three rules have been sufficient to order the code in programs built from input, output, assignment, if, for, while, repeat, simple data types (integer, real, character, and boolean), and arrays, records, and files. Sets and pointers are more complex data structures, but they obey the same data and control flow order constraints.

Plan structure defines the functional structure of a program. Given the nonlinear plan structure, the code can be ordered by constraint propagation to define a serial order. The exact serial order is determined by the necessary order of the code, and an additional set of decisions that impose an organisation on the “free” code. When code is added to an evolving solution, the plan structure is often generated first, and then the actions are ordered and executed. The process of code generation is complex, however, because many different design strategies may be used within a single solution attempt. A model that generates plan structure by linking nodes is described in the remainder of this article; design within a node is discussed in Rist (1989).

### 3. COGNITIVE STRUCTURE

A cognitive structure for program design is outlined in this section, based on the idea of description, content, or cue-based search (Norman, 1981; Norman & Bobrow, 1979; Williams & Hollan, 1981). In cue-based search, a description or cue is formed and used to search for information that matches the cue. An external source of information may be searched with the cue, such as the specification or the program, or an internal source may be searched, such as working or long-term memory. A description may be more or less precise. If the cue finds its target, that information is returned by the search and can be used immediately. If not, the cue is modified (Kolodner, 1984; Williams, 1984) and the new, refined cue is broadcast. The process is repeated until the target information is found.

An executable model of cue-based search through nodes has been implemented in an AI system called Zippy (Griffith, 1980). In the system, a search cue of the form \(<\text{role}, \text{goal}, \text{object}>\) is broadcast and matched to a target cue on the appropriate port of a node; the match is called the *return* from the search. A search cue defines a question to be asked and the return provides the answer. A node is returned from a search, linked to the search cue, and cues on the ports of the new node can be used to continue the search. The addition of a node to the solution creates a new state of the solution,
and the order in which nodes are added defines a path through a problem behaviour graph (Waterman & Newell, 1971). Design can thus be viewed as search through a state space, where a state is a partial solution composed from nodes and cues. Nodes define the static structure or knowledge used during design, and cues provide the dynamic mechanism used to link nodes and generate a complete program.

Because the first return for a cue is accepted and used immediately, there is no provision in the current model for generating and comparing different choices; a single solution is gradually built up during design by adding new nodes or new information on existing nodes. Variability in the current model is caused by different knowledge and design strategies between runs, not by choosing between several competing returns; Zippy is a satisficing (Simon, 1969), not an optimising model of design. Cues and nodes are implemented in Zippy as CLOS objects. The basic design process is slot filling with a three-valued logic: A slot value is initially unknown, then it is filled by an object or by the special value none. Design continues until all of the needed slots have been filled, so the basic design process is massively redundant; search continues in any order until the slots have been filled. The exact order in which slots are filled depends on the expertise and search strategy of the designer, plus the form and content of the program specification.

This section presents the cognitive structures that underlie and generate program code. The basic representation of cues and nodes is defined in the first part, then the goal, object, plan, and role structures are illustrated.

3.1 Cues and Nodes

A cue is a tuple of three keys of the form <role, goal, object> that forms an index to memory. A role cue has only the role slot defined, such as <read, -, ->>, <find, -, ->>, or <test, -, ->>. A goal cue has only the goal slot defined, such as <-, average, ->>, <-, maximum, ->>, or <-, volume, ->>; similarly, an object cue has only the object slot defined, such as <-, -, rain>, <-, -, password>, or <-, -, house>. A plan cue has both the goal and object slots filled, such as <-, average, rain>, or <-, volume, house>. When a cue is used as a variable in the program, it is given a name and a type. Zippy follows the usual heuristic used by programmers to derive the variable name from the goal key ("volume"), the object key ("prism"), or a combination of the two ("volPrism").

The cue and node representation is shown in Figure 6 for a single level of abstraction; the full CLOS class and slot structure used in Zippy is shown in Appendix A. A role cue is used to search for a role structure, a goal cue is used to search for a goal structure, and an object cue is used to find and link an object structure; the goal and object structures define general knowledge about goals and objects. A plan cue is used to find and link specific knowledge about a goal and object combined, such as the formula to find the
volume of a sphere. The program structure is built by linking the node returned from plan search, or by merging the nodes returned from goal and object search to define the plan node, and then adding role structure. Figure 6 shows three (data flow) cues and nodes in a generated program structure, but only the pointers to the single key structures (role, goal, and object) are shown. The pointers above the cue provide the definition for that cue, where the pointers below it serve to link the cue to the nodes where it is used.

Compound objects such as `<-, maximum, <-, count, rain>` are decomposed through their goal and object slots during design. The slots in a cue can be combined to define compound cues; the three slots define seven possible indices. Compound objects and compound cues can be easily decomposed into simpler parts, so problem decomposition is automatic and simple; it is solution recomposition that is hard. Each type of cue defines a dimension of search (Rist, 1994): search through the role, goal, and object structures. The goal and object keys generate the plan structure of a program and provide the basis for functional design. The role key is used to identify the mechanism used in part of a plan and provides the basis for procedural design.

A link consists of a set of six cues that point to the role, goal, object, plan, all, and episodic nodes. A cue is a multilevel structure, so each cue points to any cues that are more or less specific descriptions (:up and :down), any nodes on either side of it in the plan structure (:forward and :backward), and any nodes on either side of it in the linear structure (:before and :after). A multilevel link on a data port corresponds to a program variable.

A node has four ports that capture the data flow (:use, :make) and the control flow (:obey, :control) of that node, and a port contains one or more
The :make port contains the cue that indexes the node during plan retrieval. An abstract node has the cues on its ports linked to internal data and control flow, and a list of the nodes within its boundaries in linear order. A concrete node has its ports, an operator, and an internal expression defined. The operator and a numeric (data) or boolean (control) expression are used to produce the text of the program code, although several nodes may be merged into a single line of code. A node is most simply equated with a single line of code, but it can equally describe an arbitrarily large chunk of code. This flexibility is made possible by the explicit separation of the plan from the linear structure of code, and separating the external from the internal structure of a node. One powerful result of this chunking is that an abstract node can be used as a unit in planning even when the planner has little idea about how to implement its internal details.

There are four main types of search that connect cues and nodes. The simplest form of search traverses all the cues in a link to create a complete link at a single level. The second type of search examines one type of structure at a time, so the goal, object, or plan structure may be searched separately, or a single part of each structure may be combined through the links. The third type of search links nodes through their data flow, control flow, or linear order. These three varieties of search all create structure at a single level of abstraction; the fourth type of search changes the level of description up or down a level to use an abstract cue or add a new level of detail to the existing structure.

3.2 Goal
The goal structure stores knowledge about how to achieve the goal, abstracted over the set of possible objects. A plan schema is a plan in which the object has been abstracted, so it is a goal structure. More formally, a plan schema is an abstract node indexed by a goal cue, with an internal plan structure and a stored linear order. The terminals of the schema are the cues in the abstract enclosing node, so the schema can be retrieved by its goal, the :use cues located, and the schema actions coded in forward linear order.

The goal structure for the average plan is shown in Figure 7 as one concrete and two abstract levels of structure; because this is a goal structure, the object key in each cue is vacant. The first level of abstraction consists of the read loop and average nodes. For the average, the second level of abstraction consists of three nodes, the sum and count schemas, plus a schema for the two cases when the count equals zero and when it does not. The internal, concrete nodes for each of these abstract nodes are shown to the right of the figure, first the read loop, then the sum and count schemas, and then the guarded average calculation schema.

The focus of the sum plan is the node that adds a new data value to the existing sum. There are three inputs to this node, the object to add, the initial value of the sum, and the accumulated value of the sum. Only a single
Figure 7. Node structure for the average plan schema.
cues for the sum values is shown at the top of the sum node, because the two values have the same goal and object but different roles; the initialisation is \(<\text{init, sum},-\rangle\), and the accumulation is \(<\text{acc, sum, -}\rangle\); roles have been omitted from Figure 7 for simplicity. The use of the initial and accumulated values are internal to the sum plan, so those cues do not appear on the abstract node; a similar situation occurs for the count plan.

In this example, there are three levels of abstraction for the average goal structure. The object structure is trivial because the object is a real number such as the daily rainfall. When the complete average plan schema is returned and merged with the object structure, the plan structure is defined at all levels of abstraction, so no further search is required and the program code can immediately be written. If a plan schema is not known, then further search at a more concrete level is required to locate and link the complete plan structure.

3.3 Object

An object cue is broadcast and matched to a target cue, and the returned node defines the object structure. The goal and object structures are merged to generate the program plan structure, and the plan and role structures are merged to define the program structure. Two examples of object structure are discussed in this section, one to show how an object is defined at many levels of abstraction, and the other to show how an embedded object is handled in the model.

Consider a specification that asks the programmer to find the volume of a house, where the house consists of a single room with an attic on top; the room is a box shape and the attic is a triangular prism. The program has to read in the length, height, and depth of the objects, calculate the two volumes, and add them together. The volume calculations are simple. The object structure is quite complex, however, due to the fact that the house splits into two objects so that the length and depth of the house, room, and attic are the same, but their heights are different.

The goal, object, and plan structures to find the volume for a prism (the attic) are shown in Figure 8; the role key has been omitted for simplicity because all roles are “calculate.” Small circles represent a cue, and the number in each circle indicates the cue’s link. The goal structure provides a set of cues that are matched through their links to the corresponding cues in the object. At the abstract level, the plan requires a height, length, and depth that are exactly matched to the object cues. At the concrete level, the “area” goal requires a “front” surface, that is matched to the concrete object “triangle” in the object structure; the goal cues essentially filter the object cues to define the plan structure. The plan structure may be found by search if it is indexed by the plan cue, or it may be derived from the goal and object structures by unifying the cues and merging the nodes. If the area of
goal structure

\begin{align*}
<\text{height}, & \rightarrow <\text{length}, \rightarrow <\text{depth}, \rightarrow \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

volume, \rightarrow

object structure

\begin{align*}
<\text{height}, & \rightarrow <\text{length}, p> <\text{depth}, p> \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\begin{align*}
<\text{area, front}, & \rightarrow >*< >\rightarrow \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\begin{align*}
<\text{volume, } & \rightarrow \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

plan structure

\begin{align*}
<\text{height}, & \rightarrow <\text{length}, p> <\text{depth}, p> \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\begin{align*}
<\text{height, } & \rightarrow <\text{tri}, > * < > <\text{depth}, p> \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\begin{align*}
<\text{area, } & \rightarrow >*< > <\text{depth,p}> \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\begin{align*}
<\text{volume, } & \rightarrow \\
1 & \rightarrow 2 \rightarrow 3 \rightarrow 4
\end{align*}

\textit{Figure 8.} The goal, object, and plan structures for the cue \textit{<volume, prism>}. 
a triangle was not known, then it would be defined (incorrectly) by the general formula supplied in the goal structure.

The goal structure represents the general knowledge that a volume is calculated from the dimensions of the object, and more specifically that a volume is the product of the area and the depth, and the area is the product of the height and length. The object structure represents the general knowledge that a prism is a three-dimensional object, and more specifically it is a triangle with depth. The derived plan structure is created by merging the nodes and cues in the goal and object structures. In this case the derived structure can be used to generate code (that may or may not be correct), but in other cases a plan search is needed to find a formula.

Internal cues within a returned node, such as <area, -> and <-, triangle>, may have their goal, object, and plan structures already stored with the cue, in which case no further search is needed within each structure, and the structures can be merged at each level. Alternately, the cues may not be linked to a node; in this case, further search is needed to fill the slots in the goal and object structures. The knowledge structures are easily tailored to define different knowledge and different degrees of cohesion (stored links) of this knowledge.

A larger view of the plan structure is shown in Figure 9, where the object structure splits the object "house" into two parts, the room and the attic; the concrete node that merges the volumes is shown at the bottom of node 1 in the figure. The room is a box, so search using that object key returns the object structure for a box, which is merged with the known goal structure for the volume; the detailed story for the prism was given previously. Each object has an internal structure, so all the boundary cues for the box, rectangle, prism, triangle, and house must be unified in the final structure. As always, this knowledge may be already stored in the node returned from semantic memory, or may be found by a process of search and unification.

The length is described at different levels of abstraction as the length of the house, the box, and the rectangle. It is also the length of the prism and the triangle. The room is a box, and the attic is a prism. These "part-of" and synonym relations are captured by the multiple levels of cue, where each level may have a different key, and synonyms are stored at each level of abstraction; each cue points to the node at that level of abstraction. The data flow is a single data value, when the program is run, that is described and used in planning at three levels of abstraction.

A compound cue may have embedded goals and objects, in which case a search through the object structure may be required. Consider the specification goal to find the longest sunny spell in a period of days, where a sunny day is a day with zero rainfall. The cue derived from the specification has the content <longest, sunny spell>. A sunny spell is a count of sunny days, so the object slot has to be decomposed into the cue <count, sunny day>.
Figure 9. Goal, object, and plan structure of \( \langle \text{length, house} \rangle \), at three levels of abstraction.
A sunny day is one with no rainfall, so this cue in turn has to be decomposed into the cue < sunny, day >. This cue can finally be matched to the definition given in the specification. This process of decomposition and search through the object structure is shown in Figure 10.

Decomposition through the object structure allows a programmer to identify the basic objects that are needed, with no analysis of the goal or plan structure. It identifies the basic data that is needed to achieve the goal of the problem, without requiring the designer to have a plan for how that data is used in the solution. Once the input data has been defined, it can be used as a focus for detailed, forward design.

3.4 Role

The role provides the operator for a program node, so it is most simply thought of as an attribute of the node. However, it can exist in a search cue and be used to search for the matching type of node, so the role key exists independent of and before a node has been linked. The role and its operator define the type of data or control transform for the node, and the transform in turn creates the behaviour for the node.

A role may be combined with a goal or object structure; a read loop, for example, is indexed by the cue < read, stream, ->. The code, node structure, and cue contents for a read loop are shown in Figure 11 at two levels of abstraction, one for the read loop schema and one for the concrete nodes. In the cues shown at the right of the figure, both abstract (first, next) and concrete (read) roles are shown. In this example, the role and goal form the index for the read loop schema, so the object slot is vacant in all the cues.

Roles cut across the plan structure, because the same role can occur in many plans. For this reason, they are very general; essentially, a role is a construct that is independent of, and can thus be applied to, any plan and object. The role structure seems to have been the main focus of interest in automatic programming, with its emphasis on mechanism, procedural
read (rain)
while (rain <> stop) do
  read (rain)

1. <first read, stream, ->
   <next read, stream, ->
2. <stop, stream, ->
3. <loop, stream, ->
4. <next read, stream, ->

Figure 11. Role structure for a read loop, at two levels of abstraction.

structure, and behaviour. The Programmer’s Apprentice project, for example, is concerned with the interaction between roles and objects and their effect on behaviour. In this article, search through the plan structure is the focus of interest; roles are terminals in the node structure, and can be considered after the plan structure has been built.

The complexity in a plan structure may come from the role, goal, or object structure. For an average, the object is simple but the goal structure is complex; for a volume, the goal structure is simple but the object is complex; for a count, the plan and object are simple but the roles are complex. Role, goal, and object keys index general knowledge that is retrieved and combined as needed. A plan cue combines the goal and object keys to locate a specific plan for this combination. If no specific plan is returned, then the general knowledge is used to decompose the task and create a plan.

4. COGNITIVE ARCHITECTURE

The basic mechanism of design is to start at a cue and link nodes from there in the order defined by the design strategy. A design strategy defines a starting cue, a direction, a level, and a link type to explore from the current focus that determines which slot to fill next and thereby defines the next state of the solution. Any decision about the next link to pursue is local to the current node; there is no remote knowledge and no supervising controller. Search continues until there is no next step from the current state, when a new focus or strategy has to be chosen. When all the desired slots are filled, the solution is complete. The design strategies currently implemented in Zippy are procedural, functional, focal, and opportunistic design.

4.1 Search

The three keys in a cue define eight possible search cues or link types, shown in Table 3. The node structure is built by linking or merging the returned nodes. The link structure of the cue fills up as more complex structures are built, from one to two to three keys. After a link is returned or built, it is merged with the existing program structure, so the program structure is the final repository of all the merged node structures; it is a single, multilevel
TABLE 3
The Eight Cue and Link Types

<table>
<thead>
<tr>
<th>Cue</th>
<th>Example</th>
<th>Name</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;, -, -&gt;</td>
<td>&lt;, -, -&gt;</td>
<td>role cue</td>
<td>node</td>
</tr>
<tr>
<td>&lt;r, -, -&gt;</td>
<td>&lt;read, -, -&gt;</td>
<td>role structure</td>
<td>role structure</td>
</tr>
<tr>
<td>&lt;-, g, -&gt;</td>
<td>&lt;-, average, -&gt;</td>
<td>goal cue</td>
<td>goal structure</td>
</tr>
<tr>
<td>&lt;-, o&gt;</td>
<td>&lt;-, -, house&gt;</td>
<td>object cue</td>
<td>object structure</td>
</tr>
<tr>
<td>&lt;r, g, o&gt;</td>
<td>&lt;+, vol, house&gt;</td>
<td>plan cue</td>
<td>plan structure</td>
</tr>
<tr>
<td>&lt;r, g, -&gt;</td>
<td>&lt;read, stream&gt;</td>
<td>plan structure</td>
<td>node</td>
</tr>
<tr>
<td>&lt;r, o&gt;</td>
<td>&lt;read, -, rain&gt;</td>
<td>node</td>
<td>node</td>
</tr>
<tr>
<td>&lt;r, g, o&gt;</td>
<td>&lt;init, count, rain&gt;</td>
<td>complete cue</td>
<td>node</td>
</tr>
</tbody>
</table>

structure. In the current implementation of the model, only the role, goal, object, plan, and complete links are used.

In the model, search is conceptually parallel, self-terminating, and based on activation levels (Anderson, 1983; Sternberg, 1966). This parallel search is approximated by linear search where each target cue is tested in turn until an exact match has been found or the memory has been searched. Human memory search can fail, in which case a great deal of time and energy has to be spent finding an index that will match; this is the exception, however, not the rule. In the model, backward search always succeeds eventually; if it fails, then a solution cannot be found. Forward and linear search can, and often does, fail.

4.2 Search Command
Search is controlled by the current cue and a search command. The current cue defines the current location in the node structure, and the search command is applied to the current cue to define the next current cue. This process repeats until there are no new nodes to add and search terminates. The same strategy or command may be used throughout design, or a series of different strategies may be used.

A search command is built from two parts that can be called the plan and act stages: The plan part defines what is meant by "next," and the act part defines what is done to the next node. A search command specifies a design strategy by listing the direction to search, the type of link to pursue, and the level of search. Search may proceed from a cue through the goal, object, or plan structure, or up or down the abstraction structure. Search may proceed from a node through the data flow, the control flow, or the linear order. The planning slots in a search command and their possible values are shown in Table 4, and the action slots are shown in Table 5.

Search is directed from the current focus through the choice of the next slot to fill. A port is selected by the search direction and flow, and the first vacant link at that level in a cue in that port becomes the next focus of the
search. The search command is then applied to find and link a new node and define a new focus. Search continues along the chain of cues until a set of filled cues or terminals are found; a chain of searches with the same start cue and command is called a search episode.

At the end of an episode, the designer has come to the end of a chain of thought and must start a new chain. The same command and focus can then be applied to generate another branch of the plan. For a single program goal and search command, the complete plan structure can be generated by a series of episodes. When there is nothing left to explore in the current plan, then a new beginning must be made to start a new search. The current focus may be used with a new command, the same command may be applied from a new focus, or both focus and command may be changed; the most common choice is to select a new focus and retain the same design strategy. To find a new focus, the data sources can be scanned in some order; when there is no new focus at all, the program is complete.

The overall behaviour seen during search can thus be broken into three stages. First and simplest, a chain of nodes are linked until the end of the chain is reached, so there is a cycle through the nodes in the chain. Second, another branch of the plan structure is chosen and chained, so there is a cycle through branches of the graph. Third, complete plan graphs within a node are added, then a new focus is selected and a new node added, so there is a cycle through goals in the program. The action is either a note written on scrap paper, or a code that is added to the program. The name of a cue can be written on scrap paper, or coded as a variable declaration; in the same way an expression in a concrete node may be written on paper, or coded.
TABLE 6
Memory Search and Scan Orders

| search order: make: | working, episodic, semantic, scrap, spec, program |
| search order: use:  | working, episodic, scrap, spec, program           |
| scan order:        | working, scrap, spec, program                    |

4.3 Memory

Both external and internal sources of data may be searched with a cue. External data sources are the specification, notes written on scrap paper, or the program itself; these are implemented in Zippy as specification, scrap, and program windows, where the specification is a read-only window. Internal data sources are working memory, episodic memory, and semantic memory. Internal memory consists of cues and nodes, and is indexed directly by a cue; external memory (Norman, 1986) provides a key, such as a word in the specification or a variable name in the program, that has to be matched to a cue before it can be used. Usually, the specification contains the major goals and the input values, and semantic memory provides plans for the cues mentioned in the specification.

Memory can be searched to find a target, or scanned to find a search cue; the search and scan orders are shown in Table 6. If there is a search cue, then the various memories are searched for a target; this occurs during the first two stages of design to generate a chain and a graph from an existing focus. Backward search returns a plan, so semantic memory is the major data source for initial planning. Forward search to find a use for the current cue does not examine semantic memory because there are lots of ways to use something, nearly all of which are irrelevant to the current goal. If there is no search cue, then the memories are scanned to find a search cue that is returned; this occurs when a node is complete and a new focus is sought.

A data source is scanned in linear order, either forward or backward, from the start, the end, or the current focus. The number of data sources can be easily extended to include program listings, so a programmer may recall that a solution exists in a listing, find the listing, and then use the coded solution as a plan for the current problem (Visser, 1987).

When the specification is read, each phrase is translated by a simple process of keyword matching to identify the role, goal, object, or expression in the phrase, and create a node from the pieces. Because the output representation is simple and the specifications are written in simple, active, and declarative phrases, a sophisticated NLP system has not yet been required. The specification could be completely read and then one node selected as an initial focus for search (the current strategy), or a node from the specification can be used as a search cue as soon as it is created.
When a search cue is selected, it is placed at the front of episodic memory and any returned nodes are attached to that current search cue; working memory comprises the first seven nodes of episodic memory. As each new cue is added to working memory, an old cue drops off the end of working memory. If the intermediate results of an extended search are not dumped to scrap paper or the program code, they will be lost from working memory (Anderson & Jeffries, 1985). A node is a persistent object that is built in memory during design, so it can be indexed and retrieved from episodic memory at any time. Episodic memory is implemented as a link between the current and the next search cue, so there is no separate type of memory, just another way of linking nodes.

A cue is placed at the front of episodic memory when it is broadcast, and any matching target is placed at the front of semantic memory when it is matched (activated). A serial search of semantic memory thus roughly corresponds to a search based on the level of activation (Anderson, 1983). It is interesting to speculate that the apparent extended working memory shown by experts (Chase & Ericsson, 1981) can be explained by the larger schemas possessed by experts. Because working memory is used only for search and not for merging, the effective memory limit increases as the size of the returned nodes increases, with no change in the number of search cues stored on working memory.

Because nodes and cues can be stored in external memory, they have pointers to the relevant text in the specification, scrap, or code windows. These pointers provide a location in the window, so a search may be directed forward or backward from that location. If there is no current search cue, then the text can be used to generate or index a node in internal memory. This part of the model implements the parsing-gnisrap model of coding proposed by Green, Bellamy, and Parker (1987), where external memory is used to store the results of planning. Unlike that model, the text is used here as an index to the plan structure already built in memory, so the plan is retrieved instead of rebuilt.

A search strategy is defined by choosing the next slot in the next cue. The cue is broadcast, and the search returns a node that is linked to the search cue and provides a new set of cues on its ports. A new cue is chosen, and search continues until nothing is returned from a search, at which point a new focus has to be found. The links between cues and nodes are shown in Figure 12. Given the current cue, search may occur through the goal, object, or plan structure at that level, or a new level may be chosen. The next cue may be chosen in the current node within the same port, in the current node in another port, or in the next node in plan or linear order. The notion of location is fundamental to Zippy, because the location defines the knowledge that can be accessed from there. The location is defined by the current cue, and search from the current cue defines the behaviour of Zippy.
here, there are lots of places to go next, and the choice about where to go next defines the new location.

The search strategy defines the order in which nodes are added to the evolving solution. Three main search or design strategies are reported in the literature and captured in the model: procedural, functional, and focal design. Procedural design is based on the role and linear structure of the program, and design starts at the inputs and adds new nodes forward and down in linear order; the linear order is defined by the specification or by constraint propagation. Functional design is based on the plan structure of the program, and design starts with the goals and adds new nodes backward and up in plan structure order. Focal design is based on the separate goal and object structure of the program, and design starts with the most basic goal and object and adds nodes before and after this focus.

4.4 Control Structure

The power of the control structure comes from the ability to specify and change the values in the search command. There are two main components to this power. First, a single command slot can be changed and the others
retained, so a new strategy is easy to define. Second, a series of search commands can capture the distinction between batched and interleaved processing.

**Batch** processing means "do this one thing to all," hold the slot and vary over nodes. **Interleaved** processing means "do everything to this one," hold the node and vary over slots. These are actually the extremes of a continuum, because any number of slots can be attempted within a single node. In batch processing, design occurs in a series of passes over the nodes; an example is to declare all the variables, then write all the code. In interleaved processing, design occurs in a single pass in which all the slots are filled in each node; an example is to code one line and declare its variables, then process the next line, and so on. A slot in a search command may contain many slot values, and a design strategy may use many search commands. If multiple slot values are specified, then each slot is filled in turn for the current node; this is how interleaved processing is implemented. A series of search commands can be listed that differ in a single slot value; this is how batch processing is implemented. Switching to a new focus starts the list from the first command.

The key to understanding how this control structure generates behaviour during design is to look at the interaction between forward and backward search at different levels of abstraction. Given a command to do abstract backward search, Zippy moves the current focus backward through the abstract plan structure. Each node provides a new cue, so the current cue moves from the end to the front of the plan structure. Backward search always succeeds, so at some point a terminal will be reached, and design shifts to forward search down one level. From the current (terminal) cue, the cue moves down one level and then forward search occurs from the :use port within that node. If forward search succeeds, then that node can be coded and its :make port defines the new focus. If forward search fails, then backward search is resumed from the :make port of the abstract node.

Switching from forward to backward at the next level down creates a fractal pattern of expansion, as each level is traced backward, the first internal node is selected, and that node is then searched forward or expanded. If code is written when the concrete structure is generated, then the observable behaviour this strategy generates is overall forward coding in plan chunks, but backward coding within a chunk: Design is forward globally, but "everywhere backward locally!" If a schema is retrieved, then it can be coded forward through the stored linear structure, and design will be generally forward with some chunks showing forward and some backward in generation order; this was the pattern of coding reported in Rist (1989, 1991).

The level of search can be specified as the top level, the bottom level, or up or down from a current level. If a cue has no current level, then a command of "up" starts at the first up level, the concrete level; similarly, the
first command “down” from any empty cue starts search at the most abstract level. The nicest thing about this convention is that the basic search strategies are literally forward top down and backward bottom up. Several search strategies are outlined in Table 7, first with the fractal pattern discussed, then the typical novice and expert design strategies, and then focal design. In the fractal example, details about the link types and the action have been omitted so the main pattern is more easily seen. In Table 7, slot values that are interleaved are shown across the page, and batched slot values are shown down the page. Interleaved values are abbreviated in the table; the reader should examine Tables 4 and 5 for the full names. An initial empty slot in the table means that slot is not used, and a subsequent empty slot retains the previous value.

The typical novice design strategy is to code forward from the first action to the last by tracing through first the data and then the control links. Each variable is declared as soon as it is encountered in a line of code, then the line of code is added to the program. This strategy fails when a concrete node cannot be found that uses the current cue, so the first enclosed node in the plan structure is then traced backward from its :make port. Novices seem to focus on the role structure in two ways when designing. First, novices use the role as a search cue; they try to read everything, then calculate everything, and then output everything (Rist, 1986). Second, they try to fill the role slot in a cue immediately, so they search for the matching node that is indexed by all three keys. This is the smallest and most concrete node in the program structure, because the more general role, goal, object, and plan structures are not considered.

The most common expert design strategy described in the literature is forward, top-down, breadth-first design, followed by a single pass to code the complete solution. The linear structure allows the internal nodes to be listed in linear order and expanded in that order; for this to happen, the linear order must be available. There is evidence (Guindon, Krasner, &
TABLE 8
The Key Line of Code, From Various Definitions of "Key"

<table>
<thead>
<tr>
<th>Plan</th>
<th>Action</th>
<th>Object</th>
<th>Focus</th>
<th>Result</th>
<th>Beacon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>/</td>
<td>sum</td>
<td>sum/count</td>
<td>sum/count</td>
<td>sum/count</td>
</tr>
<tr>
<td>Sum</td>
<td>+</td>
<td>sum</td>
<td>sum+num</td>
<td>sum+num</td>
<td>sum+num</td>
</tr>
<tr>
<td>Maximum</td>
<td>&lt;</td>
<td>max</td>
<td>if this &gt; max</td>
<td>max := this</td>
<td>?</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>&lt;</td>
<td>next</td>
<td>if this &gt; next</td>
<td>next := this</td>
<td>swap</td>
</tr>
<tr>
<td>Insert sort</td>
<td>&lt;</td>
<td>new</td>
<td>if this &gt; new</td>
<td>here := this</td>
<td>move down</td>
</tr>
<tr>
<td>Selection sort</td>
<td>&lt;</td>
<td>max</td>
<td>if this &gt; max</td>
<td>max := this</td>
<td>?</td>
</tr>
</tbody>
</table>

Curtis, 1987; Parnas & Clements, 1986) that this strategy is reported more often than it occurs. An interesting result of the model is that the expert search commands will create markedly different behaviors for novice and expert programmers. Experts will show forward, breadth-first design because their retrieved knowledge (node) has an internal structure, but forward design often fails for novices because they cannot retrieve schemas.

The idea of a program focus recurs in programming. In program design, it is called a key (Kant & Newell, 1984) or focus (Rist, 1989); in program understanding it is called a beacon (Brooks, 1977) or key (Harandi & Ning, 1990). Expertise is needed to define the focus of a plan a priori, but experts select the same line as the focus of a program (Wiedenbeck, 1986) when asked to choose an action. Focal design (Rist, 1989) occurs when a problem is decomposed into the simplest and most basic action and object that defines the focus of the solution, and then the rest of the solution is built around the focus. Essentially, the focus is where you break out of theory into action, out of the abstract into the concrete level of design. This strategy is implemented by search through the goal structure to find the operator that achieves the simplest goal, followed by search through the object structure to define the objects for that operator. The focal line of a program is the line that contains the focal operator and objects. For the average plan, the focal operator is divide; for the sum and count plans, it is add; for the maximum plan, it is the greater-than operator. Focal design is similar to means-ends analysis (Newell & Simon, 1972) in which the most important operator is chosen first during planning; both methods begin design with a key operator that is located in the middle of the linear plan, and build the plan outward from the focus.

Care must be taken to separate four concepts that define the “most important” line in a plan: the basic action, the basic object, the basic result, and the beacon. The lines that correspond to each of these definitions of important are shown in Table 8 for a set of examples; in the table, the focal line contains the focal action and object.

The focal line is the line that contains the focal operator and object. The focal or basic operator implements the goal and uses the focal or basic
object. A sort routine places the elements in a list in sorted order where each element is greater than (or equal) to the previous element, so for a sort the focal operator is the test. The objects used by this operator define different types of sort routine. If the objects are adjacent elements, then a bubble sort is defined. If the objects are the largest element and an element of the list, then a selection sort is defined. If the objects are a new element and an element of the list, then an insertion sort is defined. Looking at the insertion sort, the line of code that places the new number in its correct place is the key line in the final plan structure, because all the other lines are added to support it; however, it is not the focal line in the design of the plan.

A beacon is used to identify a plan, such as the swap in a bubble sort. The beacon is thus a "most differentiating" line that separates out one plan from its competitors and is thus central in recognition. In a particular solution, of course, these four concepts may all map onto the same line, but this happy coincidence is not necessary.

Opportunistic design is implemented in the model as a deviation from the dominant search strategy (Hayes-Roth & Hayes-Roth, 1979) caused by an unexpected vacant slot when the vacant slot becomes a new focus for design. A common opportunistic pattern is forward design through the plan or linear structure that is interrupted when a vacant :use slot is encountered. At that point, forward design is replaced by backward design to find a plan that :makes the vacant cue. The missing slot is often a plan link, but opportunistic design can also occur within the goal and object links.

Situated design is inherent in the structure of the model, because the notions of "current" and "next" create the model’s behaviour. Search from the current location is seen at five levels of detail, each further from the current cue. Within a cue, the levels of abstraction provide a set of nodes to explore. Within a link, a goal structure may supply an abstract cue (such as "front") that is used to search the linked object (and find "triangle"). Within a node, a use port examines the make port of its node for an object definition before it searches further afield. Within a data source, search proceeds from the current location in the source until an answer is found to the current question (a target is returned that matches the search cue). Finally, a data source can be scanned to find a new question to ask and thereby start a design episode. Data from internal and external memory are combined to form a plan, with little distinction between the sources of the data: a plan is built up from data of various types as they are encountered. This mechanism captures the opportunistic, exploratory, and flexible behaviour required both of Simon’s (1969) and of Suchman’s (1987, 1993) situated planners. For Suchman, design is situated within the external environment and cues from the environment can suggest steps to add to a plan. Zippy is situated at a location within the head and then within the environment, thereby extending the notion of situated design.
A designer chooses where to start search and how to search, and the model executes the details of this choice. The basic model says nothing about informed choice, because that is the hallmark of an expert. Informed choice is used to decide on the focus of a plan and start design from that point; once this focus has been set as the initial goal, the rest of the design can be executed in the model. Expert design can be implemented as an informed decision about the next cue to choose at each node boundary, but the expertise required for this task is beyond the scope of this article. Observed expert design strategies have been based on linear, plan, and focal strategies; depth-first and breadth-first expansion; and easiest and hardest first (Ball, 1990) as well as the cognitive cost of pursuing a design choice (Visser, 1990).

Procedural, functional, focal, opportunistic, and situated design in Zippy have been described in this section. The model can be used to explore many other design strategies; essentially, the control structure defines a language that is used to program the model.

5. MODEL SIMULATION

An extension of the Noah program is used in this section to illustrate the model of design. The initial state of the system is given by the specification and the content of memory. The specification is “Write a program that shows the average rainfall for a period. Read in the rainfall each day. The end of the period is signaled by an input value of -99. Also find the maximum rainfall and the longest sunny spell in the period. A sunny day is one with no rainfall.” Memory is loaded with the count and sum plan schemas, and isolated nodes for the other knowledge. The strategy is forward, concrete design with instant coding, as shown in Table 7 for the typical novice strategy.

The initial state of the solution is the cues and nodes created when the specification is read. The declarative representation created by reading the specification is shown at the top of Figure 13; for simplicity, only the read role has been shown in the figure. The roles, goals, and objects in the specification are translated into cues, and the values are stored as expressions. The nodes are linked when they are created to define a linear order for the program, shown in the bottom part of the figure.

One node makes a single rainfall value, and two other nodes use this value, so these three nodes are easily connected. Two nodes (average and maximum) use a stream of rainfall, so a new node that makes a stream of rainfall has to be created. The read node makes a single rainfall, so it can be used to make a stream by adding a loop to control it; the loop uses the sentinel in its test. The longest sunny spell is translated from <longest, sunny spell> to <maximum, count sunny day>; this cue is then used to retrieve the maximum schema, and the new node is linked to the sunny day producer.
Figure 13. Declarative representation of the Noah specification in specification order (top) and program order (bottom).
For simplicity, only the generation of the average plan is simulated here; the order in which cues are linked for the average plan is shown in Figure 14. In the scenario, abstract backward search from the cue <average, rain> is used to identify the input data, then the plan to get this data is generated and the read loop is coded. Attention then returns to the average cue, and the abstract plan structure is expanded. Each part of the retrieved plan is then coded in forward order.

The first two cues in the figure are generated by finding (cue [1] in Figure 14) and then doing backward search on the average schema. The schema uses a stream of data, so this becomes the current goal [2]. The specification states that the rainfall must be read [3] over a period, so the designer adds a loop [4] to control the read node. This loop node in turn requires an initial data input [5] and a flag to stop the loop [6]; the stop value was given in the specification. No read loop schema is known in this scenario, so the abstract read loop node shown at the left of the figure has no filled cues.

Forward search selects the links below the cue <read, stream, rain> to pursue, and looks for a concrete node that uses that cue. None is found, so control transfers to backward search from the <average, rain> cue [1], down a level of detail [7]. Because there is no guarded average schema, the cue points to a concrete node that uses the sum and count. The first :use cue is selected [8] and broadcast. The cue matches the sum schema, so the schema is returned, bound to the object [9, 10], and its internal nodes are filled in linear order [11-14]. The next open cue is the count [15], and the count schema is returned, bound [16], and filled [17-19]. At this point, the essential plan structure to produce the average is complete because all the cues are linked and there is a concrete node for each cue.

In this scenario, the need to guard the average is realised after the rest of the solution has been designed and coded. The guard controls the average calculation [20], and consists of a test that the count is positive [21]. The guard is defined here as an if-else construct, so a node is required to deal with the else case when the count is zero; when the count is zero [22, 23], then the average is assigned to zero [24]. The complete plan is now linked and coded.

The generation orders of the cues, nodes, and code in this scenario are shown in Table 9. The first column of the table shows the order in which each cue is defined, so this is also the order of the declarations. Only concrete nodes are shown. The begin-end block is omitted from the program ontology because it is not a node, it is a procedural syntactic marker translated from the node control structure. The code for the average plan was completed in four design episodes: the read loop backward, the sum schema forward, the count schema forward, and the average calculation both backward and forward. Code generation in this scenario was generally forward, based on abstract backward search and forward schema instantiation.
Figure 14. A novice design strategy applied to the average plan: the numbers show the cue filling order.
The order of abstract and concrete node generation is shown in Figure 15 in the form of a top-down functional decomposition tree. In the tree, a number refers to a node, not a cue as in Figure 14. All nodes are abstract, except for those at the bottom of the tree. Child nodes are listed across the page in linear order within the parent; a total linear order for the program is obtained by constraint propagation.
The numbers in the top-down tree indicate the order in which nodes are generated; nodes without a number indicate the hierarchy, but do not exist in the design scenario. The generation order can be roughly described as forward and top-down, but there are many deviations from this strategy in terms of both direction (backward generation) and level (missing levels). For a top-down account, these deviations are problematic. For a model of design based on alternating forward and backward search at different levels, these deviations from the forward and top-down design are expected.

6. PROTOCOL ANALYSIS

The theoretical model of design generates a characteristic appearance order of program code as each plan is generated and added to the program. It can be used to model human design by setting the knowledge, focus, and search commands used by an individual. The protocol analysis in this section shows both the power and the limitations of the model to explain human behaviour. The claim of the model is that a program can be built from roles, goals, objects, and their expressions; the support for this claim is demonstrated in the analysis. However, most of the difficulty encountered by the participant was in the definition of the roles, goals, and objects, particularly in the roles. This level of problem solving is below the grain size of the model, though it is evident in the protocols. Once the role, goal, and object keys are defined, the model can build a program from the filled cues in the same way the human does.

The problem and protocol are taken from a study of student programmers described in Rist (1989). The problem specification is "Write a program that shows the number of sunny days (days in which no rain fell), the longest sunny spell, and the average rainfall for a month. Read in the amount of rainfall each day, measured in centimetres." The problems in the study were presented as soon as they could be solved by the student; this problem introduced the program loop. Participants talked aloud and wrote their notes and code on paper.

One participant read through the specification, then started to organise the problem. The protocol for this first cut at solving the problem is given here, followed by a translation into cues and nodes. The growing node structure is shown in Figure 16. The numbers in the protocol and the figure indicate the order in which the cues are added to the solution.

OK, sunny days [1], longest sunny spell [2], average [3]. See what I'm doing...sunny days [4], OK, when rain is zero [5]...longest sunny spell [6]...how many days it was sunny [7] and...average rainfall [8]. What is the period [9]? One month, so make it 30 days [9]. How to read [10] 30 days? Input 30 times. Make some kind of loop [11], do a loop to make the input. So now I need a constant [12] and some variables.
The problem goals were listed in specification order [1-3]. The first goal was selected for expansion [4], and the data for this goal was located [5]. The next problem goal was then mentioned [6], but attention returned to the first goal and the fact that the goal was a count of the sunny days [7]. The third problem goal was then mentioned [8]. Attention again returned to the first goal, and that goal was explored. The count of sunny days requires a stream, so the participant tried to define this and assumed a period of 30 days [9]. A plan to read a single day was known, but no plan to read a stream of data was retrieved, so attention moved to the control for the read [10]. The loop role was then defined [11] and a constant was defined for the end test [12].

The next part of the verbal protocol echoed the code protocol, so the verbal protocol is omitted and the generation order of the program code is shown in Table 10. The read loop was coded as a chunk, then the sunny day count was added as a chunk. The code for each of these plans is spread out in the execution order, as shown in Table 10.

The constant and the rainfall were declared [1-2], followed by the loop, its read, and the prompt for the read [4-6]. The sunny day count was then coded in schema or linear order [7-10], not in plan structure order. The node to test for a sunny day is shown as a data node in Figure 16, and was used as a data node during planning; its role was not mentioned. Its role is a test, so the node controls the count in the final program structure and in the code, therefore the addition of the role to the cue changed it from a :use port to an :obey port. From this and the following protocol fragments, it appears that the role is the last part of the cue to be defined and used. This is a good
<table>
<thead>
<tr>
<th>Code Generation Order for the Program Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>program Noah (input, output);</td>
</tr>
<tr>
<td>1    const n = 30;</td>
</tr>
<tr>
<td>2    var rain: integer</td>
</tr>
<tr>
<td>7    sunDay: integer;</td>
</tr>
<tr>
<td>11   sunSpell: integer;</td>
</tr>
<tr>
<td>13   sum: integer</td>
</tr>
<tr>
<td>3    begin</td>
</tr>
<tr>
<td>8    sunDay := 0;</td>
</tr>
<tr>
<td>12   sunSpell := 0;</td>
</tr>
<tr>
<td>(14) sum := 0;</td>
</tr>
<tr>
<td>4    for i := 1 to n do</td>
</tr>
<tr>
<td>17   begin</td>
</tr>
<tr>
<td>6    writeln ('Enter amount of rain:');</td>
</tr>
<tr>
<td>5    readln (rain);</td>
</tr>
<tr>
<td>9    if rain &gt; 0 then</td>
</tr>
<tr>
<td>10   sunDay := sunDay + 1;</td>
</tr>
<tr>
<td>15   sum := sum + rain</td>
</tr>
<tr>
<td>16   end;</td>
</tr>
<tr>
<td>18   average := sum / n;</td>
</tr>
<tr>
<td>19   writeln ('Average rainfall is ', average);</td>
</tr>
<tr>
<td>20   writeln ('Number of sunny days is ', sunDay);</td>
</tr>
</tbody>
</table>

heuristic for design because most of the plan structure can be generated without roles, then roles can be added later to define the detailed control structure.

The longest sunny spell goal was the next goal in specification order. The verbal protocol and the translation into cues and nodes is shown in Figure
"Now I have to ... longest sunny spell [1], how to count [2] it? Let me try sunny spell [codes line 10], will be zero at the beginning [3; lines 11 and 12], how to count it? Ah, two things. If rain was zero, then add one to sunny days ... if sunny days I have to count it ... if zero then ...

Now, let's see for average ..."
loop code (lines 16, 17). The average was calculated (line 18), and the output of the average (line 19) and sunny day count (line 20) plans were coded. The node structure is easily inferred from the average plan schema described earlier.

The longest sunny spell goal was then resumed and generated a long chain of problem solving. The difficult part of the design of the maximum plan is what to do with a larger value; the operator " > " requires two values to be compared, but the test does not indicate what should then be done with the larger value. The decision to store the largest value so far is the key to the design of the plan; this corresponds to finding the correct role for the larger value. The verbal protocol and the cognitive representation for the first attempt at the longest sunny spell plan are shown in Figure 19.

The focus of attention was the plan to find the longest sunny spell. The goal supplied an operator [" >"] that takes two objects, and the rest of the problem solving was spent trying to define the correct roles for the two objects. The objects were first defined as "a value" [1], then as the first [2] and second [3] values. The participant realised that only two values were needed [4, 5] for the comparison [6], because the second value could be stored if it was larger [7]. At this point, the comparison and store operations on the sunny spell values have been defined, but only for the two specific values that are first and second.

The remainder of the verbal protocol and the code generation order is shown in Table 11. The participant first coded the input to the plan, the sunny spell; the coding order was the initialisation (line 20), the count (line 21), and the test (line 22). The input to the sunny spell plan (the rainfall) was
"Now I think about sunny spell ... sunny spell is a number [1, writes sunSpell = k1], after this there will be two [2, 3, writes sunSpell = k2], but how can I compare this? There are some numbers I have to keep, ah, maybe only two numbers, sunny spell 1 and 2 [4, 5]... If sunny spell 2 is bigger [6], then sunny spell 1 goes out, [7] OK. So sunny spell is zero (codes line 20), then sunny spell (codes line 21) ...

1  <-, -, ss>
2  <first, -, ss>
3  <second, -, ss>

Figure 19. First attempt at maximum plan for longest sunny spell.

TABLE 11

<table>
<thead>
<tr>
<th>Code</th>
<th>Node Generation Order and Interleaved Verbal Protocol for the Longest Sunny Spell Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>8    sunSpell := 0;</td>
</tr>
<tr>
<td>22</td>
<td>10   if rain = 0 then</td>
</tr>
<tr>
<td>21</td>
<td>9    sunSpell := sunSpell + 1;</td>
</tr>
<tr>
<td>23</td>
<td>12   if rain &gt; 0 then</td>
</tr>
<tr>
<td>24</td>
<td>13   store := sunSpell;</td>
</tr>
<tr>
<td>25</td>
<td>begin</td>
</tr>
<tr>
<td>26</td>
<td>14   if sunSpell &gt; store then</td>
</tr>
<tr>
<td>27</td>
<td>15   store := sunSpell;</td>
</tr>
<tr>
<td>28</td>
<td>16   sunSpell := 0;</td>
</tr>
<tr>
<td>29</td>
<td>end;</td>
</tr>
<tr>
<td>30</td>
<td>17   write ('Longest sunny spell is ', store);</td>
</tr>
<tr>
<td>31</td>
<td>end.</td>
</tr>
</tbody>
</table>

549
then considered, but no additional code was needed because the plan already existed in the program.

The corresponding cognitive structure for the protocol is shown in Figure 20. The node at the top left of the figure shows the node structure for the first part of the sunny spell plan that was coded by tracing the data and control links, not the linear order (8, 9, 10, 11). The remaining two sets of nodes show the two attempts at coding the maximum plan. The first attempt (lines 23, 24) was based on the control flow, because it specified that the sunny spell count should be stored whenever it rained (rain > 0). This located the maximum plan in the procedural structure, but the data flow was incorrect; the participant realised that storing the count by itself was not sufficient. The second attempt (lines 25-31) expanded the maximum plan by adding the test to compare the stored to the new value, under the control of the rainfall test. The count was then reset to zero after being used in the maximum plan, then an output for the maximum plan was coded.

Code generation for this solution was generally forward, starting with the input, then the sunny day count, the average, the sunny spell count, and the maximum. Within three of the four plans, however, backward verbal and code design was visible in the protocols; only the average plan showed complete forward generation. The code was mainly written by tracing through the plan structure instead of the linear structure, so lines of code appeared for adjacent nodes in the plan structure, creating both forward and backward code generation. Focal design was visible in the decomposition of the longest sunny spell cue into its goal and object, and the attempt to define the object first. When the plan was attempted a second time, the maximum plan was partially developed verbally, then its input was coded, then the plan was coded in two attempts. Lines of code far from the focus of the plan, such as initialisations and declarations, were never coded.
The role, goal, and object representation was able to explain the protocol at a very fine level of detail. The problems in the protocol can be described as problems about how to define the slots in a cue: Goals were the simplest slot to define, then objects, then roles. The plan structure was built first through the goal and object links, then roles were added to define the code at the concrete level. The model does not consider design within the cue, so the model's behaviour is simpler and more regular than that seen in the protocol.

7. DISCUSSION

Cue-based search provides a powerful and general search mechanism. Search using the role, goal, and object keys in a cue can retrieve more or less specific forms of knowledge. As the returned structures are merged, new and more specific knowledge is created and stored, so learning is automatic. Because the current location defines the questions that can be asked from there and the answers are returned and linked to that cue, all knowledge is local. There is no global supervisor: The focus of attention moves from one cue to another during design. Partial knowledge within a cue is captured by the three-valued logic in which a slot is unknown, known to contain nothing, or known to be filled with a useful value. If a link is vacant, then it can be filled by searching for the desired knowledge, or by building the knowledge from simpler pieces. Partial knowledge within a node or plan is simply the current, incomplete state of the solution.

The process of design is explained as a choice about how to traverse the program structure. A design strategy is defined as a choice of direction, link, and level. The main design strategies are forward and top-down, and backward and bottom-up, through the plan or linear structure. Focal design links cues through their object and goal structures, then the plan and program structure are added later. Opportunistic design to fill an unexpected vacant slot causes a deviation from the current search command. The design process is massively redundant and scale invariant, because design continues until all the needed slots have been defined and filled at any level of description.

A new behaviour has been reported and simulated in this article: the distinction between batch and interleaved processing. Batch processing holds the slot constant and cycles over nodes; interleaved processing holds the node constant and cycles over slots. At a larger scale, traditional waterfall design can be seen as batch processing, and prototyping can be seen as interleaved processing.

Problem decomposition is simple and basic to the model. People have no problem decomposing a solution into parts; the difficult part of design is recomposing the pieces to find some correct way to put the pieces together.
Because all planning is local in the model, there is no lookahead to find the best way to decompose the problem; decomposition occurs simply through the goal and object cues. Although it is difficult to cleanly separate analysis from design (Swartout & Balzer, 1982), the model suggests one key distinction: Analysis makes no decisions. In terms of the model, analysis proceeds through the goal and object links to define the high-level plan structure and the data flow in the detailed plan structure. The objects and goals can be composed in many ways, so a choice of how to combine parts is considered to be part of design. Any detailed mechanism is then added by specifying the role in a cue and by choosing a specific linear order. Applying this distinction, Zippy is more a model of analysis than design, because it says little about how to constrain the form of the final solution.

Variability in design is explained as a choice of which question to ask next and the answer that is returned from the search. Variation within a solution can be explained as different choices about the order to search and to combine the role, goal, and object structure. This order is important, because different approaches to the problem can generate radically different solutions (Ratcliff & Siddiqui, 1985; Rist, 1989). A choice made during generation determines the appearance order and organisation of the code, and this order is retained in the final solution, unless the code is reorganised. Because human working memory is quite small, the results of design must be dumped to external memory or lost; when a part of the solution is placed in external memory, then the design is “frozen” and difficult to change (Green, 1989). An existing design thus constrains future choices, because the existing pieces are often used in a part of the solution, even if they are not the best pieces to use.

Different programmers will generate a wide range of solutions due to their different knowledge and their different search strategies; the model captures variability within a population, not within an individual. Plan selection is not an important issue in this model, because this is a satisficing (Simon, 1969) and not an optimizing model. The basic approach may be called the “grab and run” model of design, in which a programmer chooses a cue as a focus, searches for a node that matches the cue, and repeats the process until a solution is complete. Variability in choosing a focus for design applies the same mechanism used by Schank (1986) to generate explanation questions: “Part of the variability in the explanation process comes from the great deal of choice we have in picking features to focus on... that will determine the questions that are posed and which, in turn, will determine... possible answers” (p. 87).

Expertise in programming should reduce variability in three ways: by defining the best way to approach the design task, by supplying a standard set of schemas to answer a question, and by constraining the choices about execution structure to the “best” solutions. Expert design occurs through
the more abstract levels of search, particularly the goal, object, and plan structures. Both top-down and bottom-up design occur mainly through the plan links, and focal design occurs through the goal and object links separately. The plan structure of a program can be built without roles, so the structure and dependencies can be mapped out before committing to a specific mechanism. In the protocol of design, the plan structure was usually derived first, then the roles were added as a final step. Novice design focuses on the mechanism so it emphasizes the role structure; this makes novice design both concrete and complex, because roles do not link nodes in the way that plan structure does.

The major omission in this model is informed choice. The comparison of alternative plans for the same goal is a characteristic of experts and not of novices, so it is not part of a basic model of design. A novice does not choose between solutions; each novice generates a different solution. The architecture can provide the data to be used in an informed choice about what to do next and a location for the choice to be made, but the definition of the expertise required to evaluate the choices is a large project. Such a decision involves an evaluation of plan complexity and difficulty, the comparison of alternate choices, and a knowledge about the resources available to the designer.

### 7.1 Related Work

Zippy combines memory search and planning, so it has antecedents in both areas. In the area of information retrieval, it uses a description as a search cue; that is the basis for the RABBIT system (Williams, 1984). In the area of memory search, it is probably closest to Kolodner's (1984) model of memory retrieval, CYRUS. In that model, a discrimination net was traversed to answer queries and to reformulate a query when the answer could not be found. When data was added to the model, it was added at the location that was closest to the data so that a minimal change in the net was made. These two processes are similar to the search of a plan or chain, and the construction and addition of a new node at the end of a chain in Zippy. The construction of nodes from the words in the English specifications resembles the process of text understanding and direct memory-access parsing seen in Schank's work (Schank & Riesbeck, 1981; see also, Riesbeck & Martin, 1985).

The links between cues and nodes define a form of semantic net in which the cues on each node define a set of attributes or predicates for that node, so the underlying architecture may be seen as a type of semantic net. A node is a production that can be traversed in either direction, at any level of detail, so it should support the production mechanisms discussed by Anderson (1983): composition, proceduralisation, activation, strength, and so on. I think of a node as a neuron and the cues as axons and dendrites that link
neurons, so search is parallel; the current focus of attention is the most active part of the semantic net, and attention moves from one location to another in the net as nodes are linked.

Within the planning area, Zippy is a nonlinear planner that can do linear planning if the linear order is stored in a node or generated by constraint propagation. By using the multilevel structure of a cue, planning may occur at any level of abstraction in the nonlinear plan structure or in the linear structure, so Zippy overcomes the tyranny of detail seen in many nonlinear planners. Kambhampati's (1989) PRIAR system makes a similar attempt to chunk behaviour at several levels of description, but his system uses predicate logic in the block's world to do hierarchical planning, so the basic mechanism is very different from cue-based search. Zippy may also be seen as a form of belief maintenance system in which the activation of a cue in episodic memory corresponds to ruling a fact "in"; semantic memory contains far more knowledge than is used in any single planning episode.

As an automatic programmer, Zippy uses an extension of the producer-consumer relation (Barstow, 1982) and is perhaps closest to the DAROS system built by Kant (1991). In her system, a set of modules were indexed by various features, and planning occurred backward at the module level when a module asked to be given a certain feature. Kant's model does not have the scale invariance supplied by Zippy, and it was very much concerned with the efficiency of the implementation, so the concerns of the two models are very different. Most models of automatic programming are based on some form of predicate calculus and are focused on deriving provably correct programs, so they are not directly comparable to a model of human program design; people in general first design a program, then worry about proving correctness.

The definition of an external interface separate from any internal implementation provides much of the power seen in information hiding and formal specification (Dijkstra, 1976; Meyer, 1988; Parnas, 1972). A long-term goal of automatic programming has been the definition of a wide spectrum language that can be used to represent programs over a wide range of levels (Barstow, 1991). Node structure is more than wide spectrum, it is scale invariant because cues and nodes are treated the same at each level of abstraction right down to the base level. The semantics of specific goals and objects have been defined only for the set of examples used in Zippy, however, and the extension of these semantics to larger problems and different domains is an open problem.

Most research on the psychology of program design has concentrated on the mechanism of programming, so roles and behaviour have been of central concern. The Programmer's Apprentice project (Rich & Waters, 1990), for example, builds code forward and top-down by connecting roles such as "loop," "accumulate," and "filter" to complex data structures. The
emphasis on role, object, and behaviour fits into a view of design as forward, top-down, and state-based, where a state is defined by the value of the program variables. Most research on automatic programming also has a top-down, forward bias based on variable state, where the representation is some form of predicate calculus that is used for forward theorem proving. This article emphasises backward, nonlinear, and bottom-up planning where a state is a partial solution. The nonlinear data and control links can be traced forward or backward, so program slices and plan structure are formally isomorphic. However, the way that knowledge is chunked and combined is very different, depending on whether the plan or the mechanism is seen as the basic unit of planning.

Behaviour can be ignored if all the plans in a program are "correctness preserving." If the plans are correct, then simulation or execution of the code simply confirms the correctness of the plan. If a plan is correct in isolation but incorrect when applied, then the cues on the ports can be used to simulate or execute the chunk of code, and the expected output can be checked against the actual output. A formal program semantics based on roles can be used to define the behaviour of a node, and the behaviour can be tested against the assertions on each node to find any expectation violations. If there is an expectation violation, then the plan structure can be traced backward from the location of the incorrect output to find the bug location. Once found and identified, the code can then be extended or altered. In Norman’s (1986) theory of action, the model bridges the gulf of execution from goal to action, but it does not yet bridge the gulf of evaluation from system behaviour back to goal, because the behaviour of the plan is not used.

Experts develop and simulate a partial solution at each level of detail in top-down design (Adelson & Soloway, 1985). In the model presented here, this is done by an abstract backward pass followed by a forward pass that adds the procedural structure at the next level of detail. Forward simulation tests the behaviour of the artifact, so simulation based on behaviour may be seen as a later step than the linking of nodes discussed here. Experts follow a balanced, breadth-first strategy so their solution state shows an orderly development. For novices, or for experts working on a hard problem, the solution state is characterised by the retrieval of partial solutions, at different levels of detail or abstraction, without a previous procedural decomposition. Such serendipitous design (Guindon et al., 1987) neatly describes the pattern of development seen in the model simulation and the protocol, and seems characteristic of new design by the novice or by the expert.

7.2 Applications

An executable model of object-oriented design has not yet been written, but the theory is clear. Plans and objects are orthogonal, because one plan can
use many objects and one object can take part in many plans. The code that executes a plan will thus be spread over several classes. A node is a routine in an Eiffel system, because it contains the part of the plan that resides in that class; more formally, a node is indexed by the pair <goal, object>. A class is used to group the parts of each plan that use the same object, so a class is defined by a set of cues that hold the object constant but vary the goal. Pre- and post-conditions (Meyer, 1992) are easily stored with the cues on a node, so the model notation matches the structure of an Eiffel system. The executable system structure is generated by the interaction between the goals and objects in the system (Rist, 1994; Rist & Terwilliger, 1995).

Plan structure has been applied in a program maintenance tool called Tracer (Rist & McDermott, 1993) that takes the plan structure of a Pacal program produced by PARE, and allows the user to navigate through the data and control structure. The user clicks on a line of code to define the focus and selects the direction and type of link to pursue, then Tracer finds the connected code and highlights it. Routines may be expanded, and the amount of highlighted code can be controlled by setting terminals to stop exploration of a branch. The first task of a maintenance programmer is to find the relevant code to be understood when a change is made (Koenemann & Robertson, 1991). This is a difficult, time-consuming, and error-prone task that is often affected by leafing through pages of program code, looking for the link from one line of code to another. In Tracer, the user sets the focus, link, and direction, and the system finds and shows the relevant code; the person supplies the intelligence and the machine supplies the accuracy. The text is linked to the nodes and cues in the program, so finding the relevant code from the specification (requirements tracking) is trivial.

The representation has also been applied to automated program understanding. Plan structure abstracts over code order, code organisation, and role merging, but not over plan merging. The Plunder (Plan Underst ender) system (O'Keeffe & Rist, 1994) takes a "canonical" node structure and a target program and tries to match them. The target is converted to node structure by PARE, and the two structures are then traversed through their data and control flow links. If there is a mismatch, nodes are merged or split, and comparison of the two solutions continues until the canon matches the target. The node split and merge operations are correctness preserving, so at termination the two solutions are shown to be equivalent. Most of the initial matching information comes from the variable names that are usually built from the goal, object, or plan keys and are thus easily matched to their cues. Analysis of the program structure for plan recognition has tended to focus on the abstract syntactic, procedural, or logical structure of the code, so the plan dependencies and variable names are often ignored. From the analysis presented here, the words used in the specification and the program are the keys to design and understanding.
7.3 Conclusion

This article presents a new and powerful cognitive architecture that has strong links to human cognition. The basic mechanism in the model is to select a cue, broadcast it, and link the returned node to the cue, so a solution is constructed by linking parts of memory and design is implemented as a search through internal and external memory. By linking one node to the next during design, learning is automatic and the solution is a by-product of linking nodes through cues. The words that are used to define the problem are the keys to solving it because they define the role, goal, and object keys that index stored knowledge and are mapped directly onto the internal representation.

This article contributes four new pieces to the study of human program design. First, the cognitive representation based on cues and nodes is scale invariant, so it allows search at any level of abstraction. Second, plan structure is built by merging the goal and object structures in the model, and linear structure is derived from plan structure, so the model explicitly separates and searches the goal, object, plan, and linear structure of a program. Third, the cognitive architecture can be programmed to capture the common design strategies described in the literature; essentially, the representation defines a language for program design, and the architecture shows how to program in that language. Fourth, the model explicitly separates the two main components of programming expertise: knowledge of plan schemas and knowledge about how to design.

The model provides four new pieces for the field of automatic programming. First, it provides a wide spectrum language that can specify a node at any level of abstraction. Second, it shows how to represent and use knowledge that has an irregular structure as well as the regular structures that have received most attention in previous automatic programmers. Third, it uses the nonlinear plan structure of the code as the basic representation instead of the procedural or syntactic structure of the program. Fourth, it shows how the mechanism of program execution, defined by the role, is used within the nonlinear planning framework defined by the goal and object structures.

The architecture supports four programming paradigms: declarative, procedural, functional, and object-oriented. The representation of a problem is declarative, and search proceeds by a series of unifications and failures; the model bears a strong resemblance to Prolog, in fact. Procedural design is based on the role and linear structure of the program, and a procedure is used to group code that transforms a cue. Functional design is based on the goal structure of the program, and a function is used to group code that makes a new cue. Object-oriented design is based on the object structure of a system, and a class is used to group code that uses the same object.
From a single cue, there are many directions to search and many data sources to search. The program grows as a cue is chosen to define a question and a data source is chosen to provide an answer. A design strategy defines a trajectory through the program structure and defines the context within which later searches are made. A program grows as nodes are connected, with little overall control. The only planning mechanism is the shift of attention from one focus to another in internal and external memory. Cue-based search through the node structure provides a massively redundant model of design and simulates human design by supporting multiple design strategies, opportunistic design, and situated cognition within a single architecture.

REFERENCES


APPENDIX A: CLOS SLOT STRUCTURE USED IN ZIPPY

The CLOS class structure used in ZIPPY is shown, first the client structure and then the inheritance structure.

A1: Client Structure

Nodes and cues are the main classes in the model, but other classes are needed to deal with cues at multiple levels of abstraction, and multiple data flows from a single cue.

class slots

node
:use (control)
:make :obey :before
:goal :control :after

port
:cues

link
:role
:goal
:object

(cue)
:key
:forward :up :link
:backward :down :same

(key)
:role
:goal
:object
A2: Inheritance Structure

Each node and cue is linked to the specification, to scrap notes, and to the program. Two slots are shown inside each piece; these slots are conceptually similar, but have been kept separate for clarity.

class | slots
--- | ---
piece | :spec
 | :scrap
 | :code
node (piece) | :operator
 | :expression
cue (piece) | :type
 | :name