Focus and learning in program design

Robert S. Rist

Cognitive Science Program
Department of Psychology,
Yale University.

ABSTRACT

The construction and retrieval of plan schemas was studied using a programming task. Novice programmers were asked to write programs and their problem solving strategies analysed from the manner in which they expanded a program goal. Four programs were used in the study, comprising two sets of problem isomorphs. These isomorphs required the same programming plans for solution, but differed in their cover stories. Plan development within a problem solving episode and across episodes was thus easily tracked.

Development of a plan to achieve a problem goal showed a focus of attention on the current goal to the exclusion of concerns about other parts of the program. A solution was not found by a process of reasoning and top-down design. Rather, one goal was selected for expansion and in the process of this expansion new goals were discovered and solved to the extent required by the current context. The program was built up from solving individual goals, not down from the problem specification. The development of plans to achieve three of these goals, the selection, simple (non-looping) sum and read loop goals, is discussed in some detail to show the relation between problem solving and schema formation. Evidence of top-down design using schemas appeared only late in the study, suggesting that its use in teaching should be delayed until the novice can 'speak the language' of schema retrieval.

INTRODUCTION

A generative model of program construction is presented that tracks the development of program plans in novice programmers. The outstanding feature is a focus on the current problem under consideration, the current goal. This forces a local view of the program, but creates the pieces that are required to implement the goal. It also creates many bugs (Spohrer, Soloway and Pope, 1985) when these pieces need to be later combined. Novice behavior is best described at this local level to capture the relative isolation of knowledge that excludes consideration of facts outside the current context. Systematic, top-down development is not useful for initial problem solving, since the pieces of mentioned knowledge do not yet exist. It is achieved by novice programmers only after prolonged and laborious effort.

The focus during initial problem solving is on how to create plans to achieve the program goals. One goal at a time is selected and a plan constructed within the context of this current goal. A focal idea is expanded and tested to see if it satisfies the goal (Kant, 1985; Sussman, 1975). In the process of expansion, new goals are discovered and implemented to the extent required by the current context; the program is grown from these seeds rather than designed. Once a plan has been formed, it can be modified or moved to satisfy other constraints, such as data flow, control flow or efficiency. Once many such plans have been constructed, the focus of attention can shift from local details to comparing plans and plan segments. Planning, which uses plans as stable conceptual chunks, replaces local problem solving. Schema retrieval and tailoring replaces schema construction.

The development of three plans is described in detail to show a local focus that is gradually expanded. The simple (non-looping) *sum* plan adds a set of values to find their total; its development illustrates the selection of objects and their combination. The *selection* plan shows the change from focus on a single goal at a time to a combination of related goals. This plan

divides the input into a set of categories so that the correct calculations can be made for each category. The *read* plan shows the creation of a sequence of goals from the focal goal. The need to read a set of data items creates the new goal of looping; the use of a WHILE statement for looping creates the need to deal with loop termination. Effects of focus apart from any particular plan schema were seen in general problem solving methods, where new information was integrated (incorrectly) in terms of the focal goal.

DESCRIPTION OF THE STUDY

Design.

A protocol study was used to analyze the program construction strategies of novice programmers. Subjects were students beginning an introductory course in Pascal at Yale University. The study began in week three of the course and subjects were tested as soon as possible after learning a programming construct, such as IF-THEN-ELSE or WHILE, before they had the opportunity to use the construct in assignments. Eight novices began the study, but one dropped out after the second week. Each remaining subject wrote four programs, two sets of program isomorphs that required the same plans but used different cover stories.

The first isomorph required selection and sum plans (Soloway and Ehrlich, 1984); it was presented in weeks three and five of the course, with a read loop added for selection in week five. The second isomorph was presented in weeks four and six. It required looping, running total, running count and maximum plans; the selection plan was used to count the input values in a set of categories. The same plans (read, sum, select) were used for all programs in the study, allowing any shift from plan construction to retrieval to be easily seen. The data base consists of 30 programs using the same plans in different contexts, about 50 hours of protocol data.

Analysis.

The basic structure of plan generation was shown by the changes a plan underwent in development. The plans were identified using the Bug Catalogue developed at Yale (Spohrer, Pope, Lipman, Sack, Freiman, Littman, Johnson and Soloway, 1985). The catalogue lists several hundred bugs taken from over 200 programs, grouped according to the goals and plans in which the bugs occurred. Within this goal/plan context, it classifies a bug in terms of what is wrong with the plan (malformed, missing, spurious, etc.), a descriptive taxonomy. The bugs have been described here in terms of a process model of their conceptual, developmental structure, to show how they arise during problem solving.

The different stages through which a plan evolved was seen within a single program. They were also seen across programs, where the plan was re-created or retrieved when needed. The same development strategies were seen in all subjects, but the first form of the emerging plan differed markedly across subjects. Thus all forms of the plan were not seen in all subjects. The analysis presented is a composite of stages based on partial overlap between subjects.

PLAN DEVELOPMENT

When a new problem is first encountered, there is no existing schema to guide solution or attention. The problem solving is guided by the problem information given (objects and values) and the simple operators (verbal, logical or mathematical) that can be used to combine this information. The initial focus is on identification of these objects and operators, then on their combination and finally on factors outside the locus of attention. The solution is driven from the most basic pieces outward. The current plan is tested to see if it achieves the goal and a solution accreted around it, with multiple cycles of identification, expansion and testing if necessary.

The problem fragment that required these pieces is shown below, in the form of the first

isomorph, the first problem in the study. The second isomorph presented it as a problem of calculating Welfare benefits for unwed mothers, where the amount of additional benefits decreased as the number of children increased:

> An electric company charges its customers by the kilowatt hour (kwh) for electricity used. The cost per kilowatt hour decreases as a customer uses more electricity according to the following rate schedule:
>
> ```
> 9 cents per kwh for the first 350 kwh
> 5 cents per kwh for the next 275 kwh
> 4 cents per kwh for the next 225 kwh
> 3 cents per kwh for all kwh over 850 kwh
> ```

The sum plan.

Two main variants of the sum plan were seen, based on development of the schema from a focus and then on the retrieval of this schema and the expansion of its slots. The initial problem solving effort is centered on the rate that must be calculated for *this* category, on the rate for the selected range. Subsequent expansion of the schema treats the new rate as a formula to be added to the existing framework. Six stages were seen in the development of the sum plan. The first four of these may be termed plan construction, since they involve a large amount of problem solving by the novice. Each stage reflects the problem solving that has occurred, from a very simple analysis to one that integrates all of the information required for a correct solution. The fifth may be viewed as a retrieved solution that still retains the form of its construction. The final (expert) form is the complete schema, divorced from its ontogeny.

1. *Straight rate*: the surface form of the description creates the final code. The focus is on calculating the charge for this category. At this stage, the object has been identified (kwh), the existence of separate rates and categories noticed and the rate has been attached to the object. The rate is then directly calculated as

   $$cost1 := kwh * .09$$
   $$cost2 := kwh * .05$$

2. *New + old simple*: the surface form of the sum is calculated. In this stage, the term 'next' has been parsed and it is realised that the different rates must be added to get the final charge. The focus is on the rate for the category and so the new rate is coded first, then the other pieces without further analysis as

   $$cost1 := kwh * .09$$
   $$cost2 := kwh * .05 + kwh * .09$$

3. *New + old local*: the correct form for the new rate is calculated. In this stage, the subject has realised that the rate changes over boundaries and only the amount of kwh over the boundary should be included in the new rate calculation. The focus is still on the new charge and so the insight is not extended to both charges:

   $$cost1 := kwh * .09$$
   $$cost2 := (kwh - 350) * .05 + kwh * .09$$

4. *New + old constructed*: the correct form of the schema is constructed. In this stage, the boundary insight is applied to all the individual rates, but the subject has explicitly coded one of the previous stages or has to devote considerable energy and time to constructing the solution. A particularly strong block to the discovery of the correct solution is that the old calculation does not involve the program object, kwh. The object has to be removed and a constant inserted in its place:

*cost1 := kwh * .09*
*cost2 := (kwh - 950) * .05 + 950 * .09*

5. *New + old retrieved*: the schema is retrieved, but still shows the new part focus. In this stage, the parts appear without apparent effort and the correct solution is mentioned as it is coded:

*cost1 := kwh * .09*
*cost2 := (kwh - 950) * .05 + 950 * .09*

6. *Old + new retrieved*: the *sum* schema is retrieved and expanded in the 'logical' order. This final stage shows the problem solving automated and attached to the components within the overall schema. The schema is retrieved and control given to the components in the order old plus new, where the old part repeats through the number of steps required:

*cost1 := kwh * .09*
*cost2 := 950 * .09 + (kwh - 950) * .05*

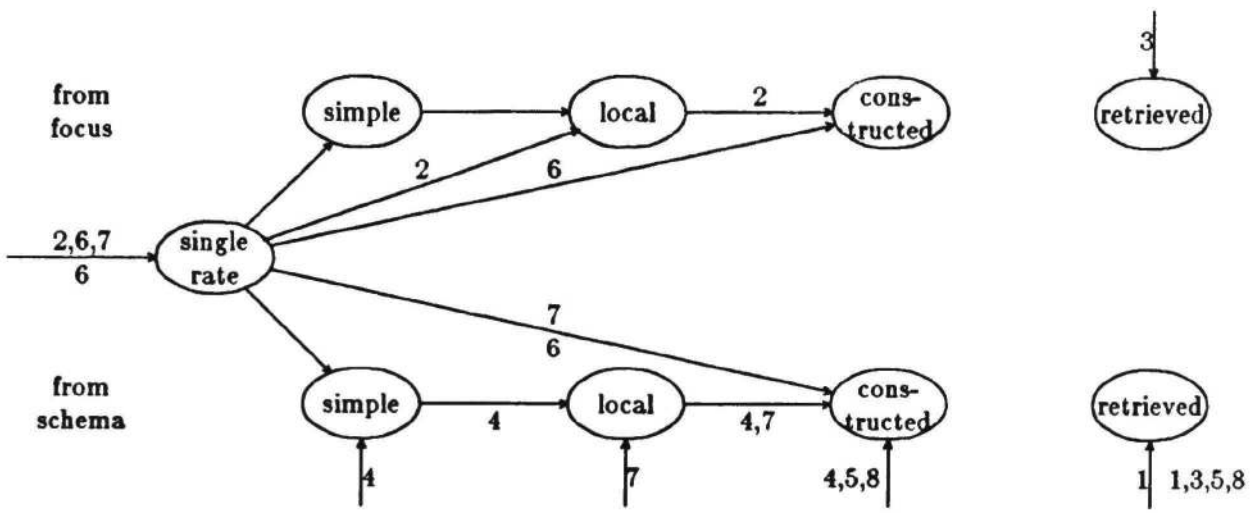

Figure 1: Development of the sum plan by subject

The expertise of the subjects in the study differed greatly. One subject retrieved the final *sum* plan without effort to solve the problem in the first isomorph. Another started at stage one both times. The transitions seen in the data are shown in Figure 1. Here, the development for the first presentation of the problem is shown above or to the left of the links. The behavior of the subject on the second isomorph is shown below or to the right of the link. Subject solutions within a problem solving episode developed according to the taxonomy described. Solutions across the episodes reflect the change from schema formation to schema retrieval and expansion of the slots.

The selection plan.

The selection plan provided the context for sum calculation by selecting when each of the new rates was applicable. The two plans may be treated in relative isolation, however, since all six stages of the sum plan occurred in the context of a successful selection. The declarative information used for the two plans has considerable overlap, but the use of boundaries in selection was not extended to the sum calculation. Generally, local focus mitigates against such

systematic comparisons, since the relevance of a particular item of data is determined by search from the focal goal. Schema application, with its description of required slots and slot fillers, is not yet available for use.

The select plan shows a development from separate goals each using IF to test the defined, exclusive ranges, to the complete ELSE-IF form where the structure of the ELSE-IF replaces the tests used in separate IFs. The change may be viewed as the progressive replacement of explicit boundary tests by the implicit testing of the ELSE-IF construct. Initially, each case is treated locally, in isolation from the other ranges. Over the course of development, the concept of exclusive, abutting boundary conditions and their use in the ELSE-IF statement emerges. This creates a different organisation of the problem information, from one based on separate goals to one based on filling the newly-defined slots of the ELSE-IF, embedding the boundary information and testing within the language construct. Several subsidiary plans are involved with the select plan and also show a developmental shift. These are the choice of what to use as a selector (one or two ranges), the form of the test within the selector and the combiner (OR or AND) when two tests are used. These will be discussed after the select plan.

Else-if development

Five stages of the select plan were seen for the problem described here, that of continuous, abutting ranges. The general form of the development may be seen as an optimization that requires less code for each stage. More informatively, it may be seen as the use of the structure of the ELSE statement to replace the explicit selection tests required by a local formulation, the embedding of the explicit goals into a structure where they are achieved implicitly.

1. *IF tests*: the selection is done using separate IF tests for each case. The case is then expanded within the context of the IF and treated in isolation from the other pieces. The condition in the test, the selector, may be one or both boundary tests.
2. *IF to ELSE-IF test*: the subject begins by using separate tests, but at some point, usually the second IF, links the separate tests using ELSE statements. The ELSE functions purely as a connector between IFs and has no other functionality.
3. *ELSE-IF with separate tests*: an ELSE-IF structure is used from the outset, but both boundary tests are still included as the selector. The decision to use the ELSE-IF as the operator was local and had no impact on subsequent (local) goal expansion.
4. *ELSE-IF with last test*: an ELSE-IF structure is used, but the subject realises that the structure can be used to test one of the conditions and only one boundary need be specified in each if statement. This insight is applied to all the tests, so the last test also has an explicit (unnecessary) boundary.
5. *ELSE-IF*: the final, expert form, with one boundary tested in all cases except the last, where it is implicitly true.

The initial goal of the subject is to select out the different ranges so that the charge can be computed for each category. A selection operator is retrieved (IF or ELSE-IF) and each of the categories is analyzed within the context of this operator. The definition of the category is inserted in the condition slot of the operator. This definition usually is conceived as a range with two boundaries, which is most directly coded as a test for more than the lower bound and less than the upper bound, creating the repeated IF structure of the first stage.

In the second stage, the IF schema has been formed and its use now becomes an issue. The set of exclusive alternatives cues the ELSE structure, which is now seen simply as an IF connector, and the separate IF statements are linked by this (syntactic) connector. In the third stage, the analysis is made before the code is written and the structure laid out for all the cases; each case is then expanded separately. In the fourth stage, the subject realises that only one test is required inside an IF, and so each IF test, including the last, is coded with a single test. In the last stage, the realization is fully applied and the optimized ELSE-IF structure created.

Selector development

In solving the selection problem, the first step is to discover the different categories. The simplest form of analysis focuses on the conditions that separate one category from another, such as 'under 350', 'over 350', 'over 625' and so on. These can then be directly coded using the known comparison operators. In the second stage, the subject realizes that both conditions are required and codes them both as the test condition.

Combiner development

When two selector tests are defined, they need to be connected with a boolean operator, AND or OR. The correct operator is AND, yet very often subjects chose to use OR and corrected themselves later. The OR operator is used correctly in the context of independent goals, such as enumeration (it's one or the other). The locality heuristic assumes that everything is independent until proven otherwise, when AND is substituted as the correct connector.

Test development

The initial concern in defining the boundaries for selection is to state which values enclose the category. A local focus on each of the categories created a definition in terms of equality, by specifying the lowest and highest possible values. In the electric bill, the equality focus identified the value pairs (0, 350), (351, 625), (626, 850) and over 850. These were then coded using >= and <= tests. The second stage saw the replacement of these 'add-one' values and >= tests with tests for >. This returned the boundary values to their form in the problem description and may indeed have created the initial insight that eventually led to the ELSE-IF optimization, that one plan's ceiling is another plan's floor.

The read plan.

The focal flavor of development is particularly noticeable in this plan. The goal of reading the data spawned the goal of looping, which was coded. The loop was then ignored, since the goal seemed to be achieved by writing the loop code. The rest of the goals were implemented and attention only returned to the loop code when the program was executed. The implications of using an input-controlled loop (Soloway and Ehrlich, 1984), with an end value of 99999, were discovered at this point. The removal of this value started a new round of problem solving. The difference between a bug and an extension of the design is the time at which the bug is found, not the process of discovery or solution. If the goal is currently focal, then an addition is part of the new, improved design. If the goal is past, and in the normal case if all the goals are past and the program considered complete, then the addition is a patch to solve a bug. The problem solving strategy and local focus is the same in either case.

Two patterns in the read plan development are especially interesting. The first is that novices would often come to the end of the WHILE loop and reconstruct the goal of terminating the loop at that point. The local focus means that attention is directed toward achieving the current goal and when it is achieved, or in this case when a sequence of goals was achieved, the context is reconstructed from the external memory provided by the program code. The reconstructed goal is then implemented in isolation from the program and indeed the problem, since substitute solutions such as keeping a count were generated as solutions. These contradicted the problem specification, which stated that 'end of input is signalled by a value of 99999'.

The second pattern emerged in debugging the WHILE loop. Two types of solutions were seen, arising from different conceptualizations of the bug. The first conceptualization was that the value *in the calculation* was wrong and the value had to be fixed, leading to a backout strategy. The second conceptualization was that the bug was *caused by* the read. This produced

two types of fixes. The first was to protect the calculation from the read *value* by adding code, such as a guard. The second was to protect the calculation from the *last* read by moving the read to the bottom of the loop and using the WHILE test as an implicit guard. These plans show increasing distance from the bug symptom to the final solution.

The stages in development seen to achieve the read goal were

1. *single read*: this is the simple prompt (WRITELN/READ) plan.
2. *read then loop*: the subject coded the input prompt and read and then mentioned the loop goal, inserting code before the read statement.
3. *loop then read*: the loop statement preceded the read. The need for a loop was noticed when reading the program and the read schema was coded.
4. *different plan*: local plans were spawned to solve the reconstructed termination goal. Two inappropriate plans were seen in the data. The first used an input value as a counter to control the loop. The second asked the operator to input a character variable, 'Y' or 'N', to control the loop. This plan had been taught in the course and was retrieved when a loop terminator was required.
5. *the backout plan*: subtract 99999 from the sum and 1 from the counters. The bug is that the variables contain the wrong value, so the simplest solution is to fix the values. The backout could be done directly where needed, after the loop and before the output, or be slightly improved and done when initializing the variables.
6. *selective backout*: if the value is 99999, make it zero. The bug is that the wrong value has been added in, so the solution is to make the value innocuous and convert it to zero. This fix saves the sum, but all the counters still need to be backed out.
7. *guard*: only add if the value is not 99999. The bug conceptualization is that the wrong value has been added and the patch is to prevent the end value from being included in the calculations. This saves all the running totals, but creates two tests for the end value, once in the WHILE and once in the guarding IF.
8. *read at end*: read the data at the end of the loop. The conceptualization of the bug is that one too many data have been read in, so the read at the end stops the extra read by using the control machinery to guard the calculations. It is an uncommon solution, because it requires a system viewpoint and then a selective backout.
9. *double read*: read before the loop and at the bottom of the loop. It is an extremely difficult solution because it requires a system view of the loop and two separate reads to achieve the single goal. The use of two pieces of code violates the implicit locality heuristic, that a goal is achieved by one piece of code. Subjects added code with abandon, but seldom changed existing code to achieve a new goal. This plan was commonly used for validation, to loop until a good value was read, but the different problem solving context seems to have prevented transfer.

The solutions tried by each subject are shown in Figure 2; links are labelled with the number of the subject who showed that particular transition. The plan was necessary in three of the problems; the first problem did not involve looping. The data from these three problems may be read from top to bottom of the subject numbers. The place marker '-' is used to indicate that no subject made a transition on a particular problem. Some subjects ran out of time on initial attempts and did not encounter the 99999 problem, so some early paths are not complete. Thus on the first occasion, subjects 3 through 7 coded locally from the read to the loop goal. Subjects 3 and 4 then used a guard, subject 5 ran out of time, subject 6 used a retrieved, inappropriate solution and subject 7 used backout then guard. It is interesting to note that none of the plan patches were schematised, being abandoned or redeveloped from the focus each time.

Program development.

The effect of focus has been demonstrated within individual plan segments. Two more general examples of local development will be presented here, showing the use of a local

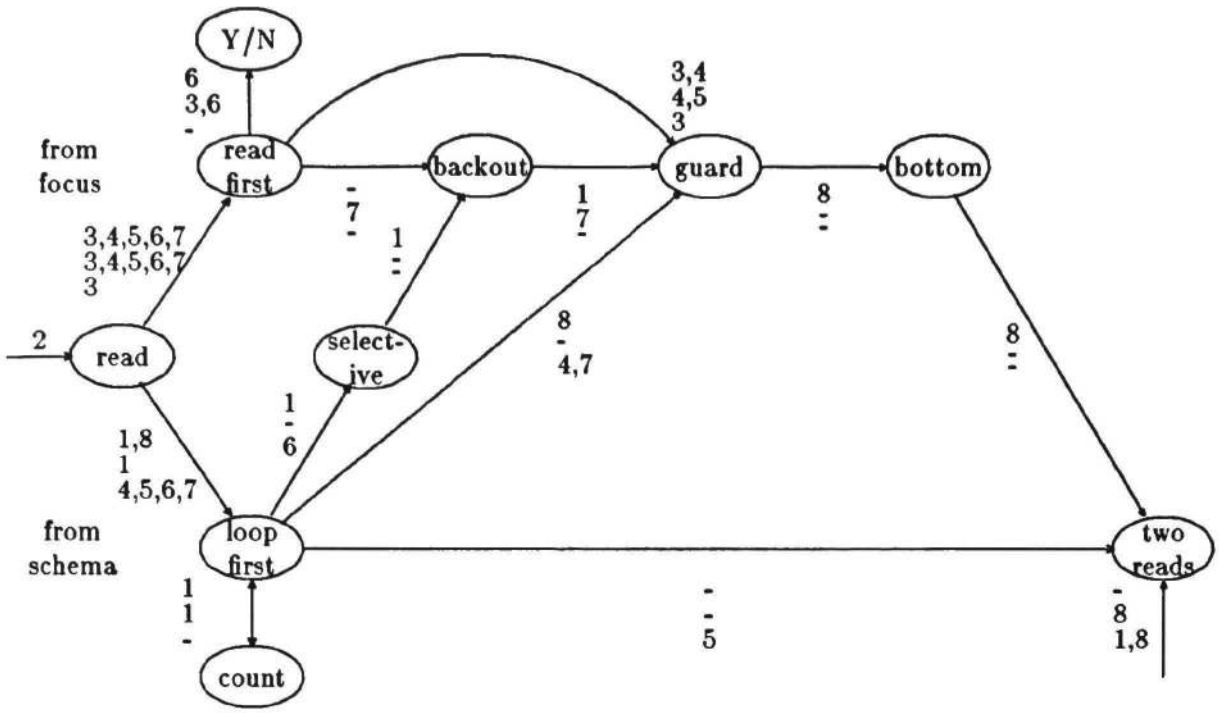

Figure 2: Development of the read plan by subject

expansion strategy divorced from any simple, identifiable schema. The first reflects the development of knowledge about the structure of plans in a program. The second reflects the integration of information by a focal goal as a general problem solving heuristic.

A plan may be viewed as consisting of several segments, each of which fulfills a required role. These roles are general plan pre- and post- requisite descriptions, such as input, process and output (IPO) or initialise, calculate and use. The theoretical value of such a description is demonstrated in Rich's (1981) plan calculus; evidence for their use by novices and experts is presented in Rist (1986). The development of a modular program structure may be viewed as the hierarchical organisation of plans and plan segments that implement the 'correct' ordering at the appropriate level. A simple loop program, for example, shows a goal centered structure (IPO) at the most abstract level. Plan segments are then organised by each of these roles (III, PPP, OOO), spreading elements of a plan across the program in a role centered structure.

In understanding the problem, novices identified the goals and then the input objects for these goals. The goals and their associated inputs create the set of transforms (IPO) that is used to build the program. In the electricity program there were two goals, to find the cost (C) from the number of kilowatt hours used and the billing area (A) from the customer number. These goals are independent and their role organisations thus show the reasoning process unconstrained by issues of data flow. The correct, systematic placement of roles and objects must be learned through experience. The types of role patterns for the two isomorphs are shown in Table 1.

The initial problem solving organisation comes from the role structure. As was seen for the loop, the first heuristic is to read in the data, then process it. The input role is especially useful in problem solving in programming, since the input defines the objects required, is simple, always independent and presents few problems. At each role transition (I -> P, P -> O), an object must be selected for the new role. The simplest selection is to use the goal that was last used, producing the object order ACCAAC shown on the first line. The first stage of learning and planning is to link the process and output roles for the same object, creating an implicit goal link

Table 1: Development of role organisation

occurrence				pattern		
subjects		numbers				
elec	welfare	elec	welfare	independent		loop
3	4	1	1	I_aI_c P_cP_a O_aO_c		
6		1		I_aI_c P_cP_a O_cO_a	-->	I_cI_a P_cP_a O_cO_a
2,4,5,7,8	3	5	1	I_a $I_cP_cO_c$ P_aO_a		
1	1,5,6,7,8	1	5	$I_cP_cO_c$ $I_aP_aO_a$		

(CACA). This link is made explicit in the the next stage, where process and output are organised underneath the objects (CCAA). The goal organisation is then extended to all the roles and the goals are treated independently, producing the order CCCAAA used in the second isomorph. A similar development was seen for the validate (V) role. It was initially considered at the end of the program for all input objects (role centered, IIPPOOVV) and coded in the program at various positions. It was then considered at the end of each independent goal (IPOV, IPOV) and finally the role was embedded in the normal plan structure (IVPO, IVPO).

The first role reorganisation (line 2) creates the concept of systematic nesting, here nesting under roles. Complete role nesting is derived from this pattern by extending it to the input role, with the transition ACCACA -> CACACA. This organisation is correct for plans inside loops, where the main processing is delimited by the loop boundaries. The expert selects which pattern is needed from the type of plan; the novice begins by using local expansion and develops these patterns and selection rules through debugging.

The incorrect placement of plan roles inside the loop is shown in Table 2. Programs two and four used several plans, such as *max* and *average* that are not discussed in this paper; the table indicates the total number of incorrect placements for all four plans. Only those roles that provided a choice for placement are included in the table, such as final calculations for average and percentage and program output; these could be placed inside or after the loop. Their location is a sensitive indicator that demonstrates the lack of developed discourse rules.

Table 2: Local plan expansion

Program	Subject number						
	1	3	4	5	6	7	8
	IPO	IPO	IPO	IPO	IPO	IPO	IPO
2		14				22	
3		1			1	1	1
4		24				34	

The Welfare problem (problem 3) demonstrates local problem solving in the interpretation and organisation of problem data. It was the isomorph of the electric bill problem, requiring subjects to calculate the benefits an unwed mother would receive in a year, given the number of children in the family. At this point, three weeks into the study, it was expected that the selection and sum plans would have been schematised and could be retrieved and used. This was supported; subjects applied the plans and inserted values in the plan schema slots without apparent effort. The interesting point was that over half (four out of seven) of the subjects retrieved the plan, expanded it and coded benefits for each of the categories including the category of no children. A value of children less than zero was an error, a value of one or greater received benefits according to the rate schedule given and a value of zero received zero benefits. From a global perspective it is difficult to see how an unwed mother could have zero children.

IMPLICATIONS

The shift from a local and concrete to a system-oriented and abstract view in design marks and defines expertise in a domain. That shift has been explained here as arising from the problems encountered by a novice, the problem solving techniques used to overcome them and the use of the products, plan schemas, in solving further problems. The change from goal expansion to modular planning has been traced to the interaction between plan flow and control and data flow via the creation and automation of plans and plan segments.

It is interesting to compare the results of this study to the theoretical models of skill acquisition discussed by Sussman (1975), Brown and VanLehn (1980) and Hayes-Roth (1983). The general method of Sussman, that of attempting a solution and creating new goals and patches as required, was the design strategy used by novice subjects. The solution of impasses has been explained by a method of adopting a focal object, operator or schema and developing a solution around it. The problem solving heuristics used by Hayes-Roth can now be placed in a developmental sequence, from the method of noticing a bug and applying a local patch to examining the underlying causality of the bug and fixing the cause instead of the symptom.

The detailed analysis of the information involved in coding a computer program offers the hope of a realistic simulation of human problem solving behavior. The use of schema as knowledge organisers echoes Schank's (1982) MOP organisation and extends it by demonstrating the ontogeny of such theoretical constructs in an experimental task. The use of these MOPs in understanding involves many of the same issues as the use of plan schema in program design. Study of their development provides a strong answer to the question of what is inside a schema.

I wish to express my appreciation to Jim Spohrer and Dana Kay for their trenchant criticisms of early versions of the paper. The study was funded by grant number 66430 from IBM.

REFERENCES

Brown, J. S. & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, **4**, 379-426.

Hayes-Roth, F. (1983). Using proofs and refutations to learn from experience. In R. S. Michalski, J. C. Carbonell and T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Palo Alto, CA: Tioga.

Kant, E. (1985). Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, **SE-11**, 1361-1374.

Rich, C. (1981). *Inspection methods in programming.* (Tech. Rep. AI-TR-604). Boston: MIT, MIT AI Lab.

Rist, R. S. (1986). Plans in programming: Definition, demonstration and development. In E. Soloway and S. S. Iyengar (Eds.), *Empirical studies of programmers*. New York: Ablex.

Schank, R. C. (1982). *Dynamic memory*. Cambridge, MA: Cambridge University Press.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10**, 595-609.

Spohrer, J. C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L. & Soloway, E. (1985). *Bug catalogue: II, III, IV.* (Tech. Rep. 386). New Haven: Yale University, Department of Computer Science.

Spohrer, J. C., Soloway, E. and Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, **1**, 163-207.

Sussman, G. J. (1975). *A computer model of skill acquisition*. New York: American Elsevier.