Chapter 3.1

Expert Programming Knowledge: A Schema-based Approach

Françoise Détienne

Projet de Psychologie Ergonomique pour l'Informatique, INRIA, Rocquencourt, BP 105, 78153, Le Chesnay Cedex, France

Abstract

The topic of this chapter is the role of expert programming knowledge in comprehension. In the 'schema-based approach', the role of semantic structures is emphasized whereas, in the 'control-flow approach', the role of syntactic structures is emphasized. Data that support schema-based models of understanding are presented. Data that are more consistent with the"control-flow approach' suggest limitations of the former kind of models.

1 Introduction

The structures of knowledge and its organization is an important characteristic of a model of the expert programmer. Having such a model enables the design of systems which support programming activities. Such a model is also necessary to understand how expertise is acquired in a domain. The topic developed in this chapter is the role of expert knowledge in program comprehension. This discussion is limited to comprehension, because first, as shown in Chapter 1.3, the same kind of knowledge

Psychology of Programming
ISBN 0-12-350772-3

is used by the processes of program composition and program comprehension, and second, the understanding activity is particularly interesting to analyse because it is involved in several maintenance tasks. Constructing a mental representation of a program is necessary for effectively debugging or modifying a program.

Many studies on programming knowledge have been conducted in the theoretical framework of schema theory (see Chapter 1.4 for details of this theory). This approach is developed here and is evaluated on the basis of empirical data.

In Section 2, studies on program comprehension are briefly reviewed from a historical perspective. In the 'schema-based approach' the role of semantic structures is emphasized whereas, in the 'control-flow approach' the role of syntactic structures is emphasized. In Section 3, a theory of the knowledge structures which programmers are supposed to possess is developed. Then, empirical support for these hypotheses on knowledge structures is presented. In Section 4, comprehension mechanisms are analysed through empirical results.

2 A Historical perspective on approaches to program comprehension

Different approaches to comprehension can be distinguished according to the knowledge and the cognitive mechanisms that are assumed to be used in experts' program comprehension activities. There is disagreement as to the kind of knowledge that is important when understanding a program.

In the schema-based approach experts are assumed to use mainly knowledge structures that represent semantic information in the programming domain. These structures group together information on those parts of programs that perform the same function. In the control-flow approach experts are assumed to use mainly knowledge structures that represent elements of the control structure. These structures group together those parts of programs occurring within the same syntactic structure. The former emphasizes the role of semantic knowledge, i.e. knowledge about what the program does whereas the latter emphasizes the role of syntactic knowledge, i.e. knowledge of how the program works.

From a theoretical viewpoint, proponents of the schema-based approach have used concepts developed in the schema theory. Knowledge structures in programming are formalized in terms of 'programming schemas' or 'programming plans'. Schema theory is not always referred to in the control-flow approach. However, the schema concept could also be used to model the syntactic knowledge structures that experts are assumed to possess.

The schema theory is a theory of knowledge organization in memory and of the processes involved in using knowledge. A schema is a data structure that represents generic concepts stored in memory. This theory has been developed in artificial intelligence (Abelson, 1981; Minsky, 1975; Schank and Abelson, 1977) and in psychological studies on text understanding (Bower *et al.*, 1979; Rumelhart, 1981).

The schema-based approach in programming studies began with Rich's work (1981) and has been mainly developed by the group headed by Soloway. It can account for problem solving and understanding activities in various programming tasks such as program design, debugging and enhancement. The studies started in the early 1980s and some of the most important papers on this topic are by Soloway *et al.* (1982a,b) and Soloway and Ehrlich (1984). For several years, other researchers from different institutions (Brooks, 1983; Détienne, 1989; Rist, 1986) have conducted studies on programming using the same theoretical framework. In the control-flow approach, studies have also been performed by both computer scientists and psychologists (Atwood and Ramsey, 1978; Curtis *et al.*, 1984; Linger *et al.*, 1979; Pennington, 1987; Shneiderman, 1980).

There is not only disagreement on the kind of knowledge that is important for comprehending a program, but also on the way knowledge is used in program comprehension. On one side, proponents of the schema-based approach assume that understanding a program consists of the evocation of a programming schema (or several schemas) stored in memory, instantiating that schema with values extracted from the text and inferring other values on the basis of the evoked schema. The mechanisms of schema activation can be either data driven or conceptually driven. In the first case the activation spreads from substructures to superstructures, and, in the second case, the activation spreads from the superstructures to the substructures. Therefore, semantic structures can be evoked directly by information extracted from the code or by other activated schemas.

On the other side, proponents of the control-flow approach assume that understanding consists in identifying control structures and then combining these structures into larger structures until all structures correspond to some function. A good example of this approach is the syntactic/semantic model of program comprehension, developed by Shneiderman and Mayer (described by Shneiderman, 1980). This assumes that comprehension involves two separate processes, with syntactic processing occurring first followed by semantic processing.

In the rest of this chapter, the schema-based approach is developed. Data that support schema-based models of understanding are presented. However, data that are more consistent with the control-flow approach are also presented, suggesting some limits to schema-based models.

3 Knowledge organization in memory

3.1 Theoretical framework

In the schema-based approach to comprehension, hypotheses are made on the typology of schemas possessed by experts, the relationship between schemas, and the structure of representations constructed from programs. These points are developed in this section. Remarks are also made on the knowledge organization assumed in the control-flow approach.

3.1.1 Typology of schemas possessed by experts

Soloway *et al.* (1982a) have encoded experts' schema knowledge as frames. A schema is represented as a knowledge packet with a rich internal structure. A schema (or 'plan' in their terminology)[1] is composed of variables ('slot types') that can be instantiated with values ('slot fillers').

Program understanding involves the evocation of schemas of different domains and their articulation. Brooks (1983) assumes that program understanding involves a mapping between, at least, two domains, the programming domain and the problem

[1] I will use the terms 'plan' and 'schema' interchangeably throughout this chapter.

domain. Experts would possess schemas representing information on problems which are dependent on the problem domain (or task domain). Détienne (1986b) describes the general structure of those schemas as including slots on the data structure and on the possible functions in a particular problem domain. For the particular domain of stock management, the following values could be assigned to the slots:

Task domain: stock management

Data structure: record (name of file, descriptor of file.)

Functions: allocation (creation or insertion), destruction, search

Among the schemas relative to the programming domain, Soloway *et al.* distinguish variable plans and control-flow plans. Variable plans generate a result that is stored in a variable. For example, the Counter_Variable plan can be formalized as:

Description: counts occurrences of an action

Initialization: Counter := 0

How used? (or update): Counter := Counter + 1

Type: integer

Context: iteration

Control-flow plans do not generate results, rather they regulate the use and production of data by other schemas. For example, the Running_Total_Loop plan, which computes the sum of a set of numbers, initializes a total variable to zero, gets a number, adds that number to its total and loops until a stopping value is entered. It is described by Soloway as the following frame:

Description: build up a running total in a loop, optionally counting the numbers of iterations

Variables: Counter, Running_Total, and New_Value

Set up: initialize variables

Action in body: Read, Count, Total

In this schema, the Running_Total variable refers to a variable that builds up a value one step at a time, like, for example, a variable in which would be accumulated the sum of the numbers. The New_Value variable holds the values produced by the generator. This can be, for example, a Read_Variable plan that receives and holds a newly read variable or a Counter_Variable plan that counts occurrences of an action. Experts would also possess more complex schemas representing algorithms like search or sort algorithms or abstract types like tree structures or record structures.

Pennington (1987) describes different kinds of relations that formally compose a program: control-flow relations that reflect the execution sequence of a program, data-flow relations that reflect the series of transformations on data objects in a program, functional relations that concern the goal achieved in a program. To contrast the schema-based approach and the control-flow approach, Pennington remarks that the plan knowledge analysis is closely related to an analysis of program text in terms of data-flow and functional relations. Programming schemas represent information relative both to functions performed in a program (e.g. search for an item) and to data flow relations. In contrast, the control-flow knowledge analysis is more closely related to elements of the control structures such as sequences, iterations and conditional structures. These basic structures of programs are called 'primes structures' by Linger *et al.* (1979).

Soloway *et al.* assume that experts also possess rules of discourse that are programming conventions about how to use and compose programming plans. For example, a rule might express the convention that the name of a variable should reflect its function. The application of these rules produces prototypical values associated with plans.

3.1.2 Relationship between schemas

Soloway *et al.* (1982b) describe two kinds of relationship between plans: a relation of specialization ('a kind of') and a relation of implementation ('use').

The relation of specialization links together plans that are more or less abstract. A specialized plan forms a subcategory of the plan just above it in the hierarchy. For example a New_Value_Variable Plan which holds values produced by a generator can be specialized in two distinct plans: a Read_Variable plan and a Counter_Variable plan. In the same way, the Running_Total_Loop plan can be specialized as three distinct plans: Total_Controlled_Running_Total Loop plan in which the Running_Total is tested, a Counter_Controlled Running_Total Loop plan in which the Counter is tested and a New_Value_Controlled_Running_Total_Loop plan in which the New_Value is tested.

The relation of implementation links together plans that are language independent on one side, with plans that are language dependent on the other side. Implementation plans specify language-dependent techniques for realizing tactical and strategic plans. For example, the For_Loop plan is a technique for implementing the Counter_Controlled Running_Total_Loop plan in Pascal.

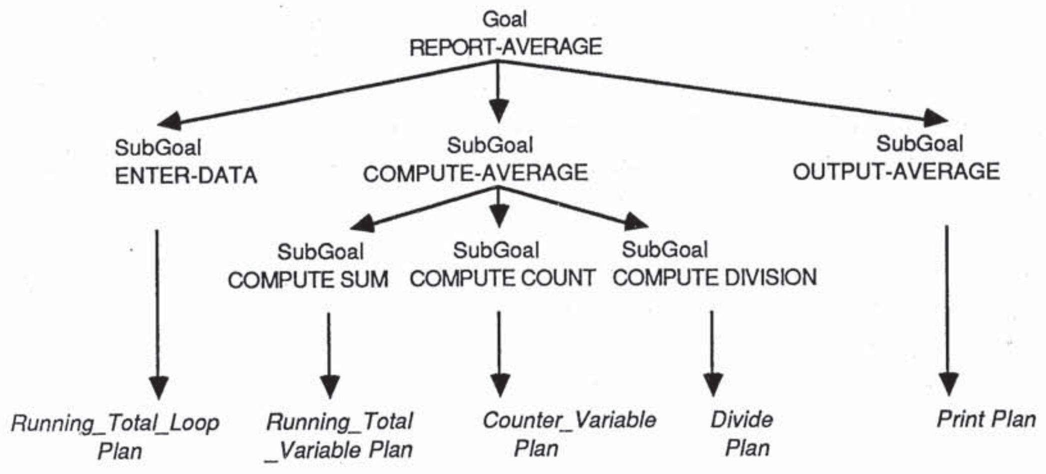
The relation of implementation refers also to the way some plans are composed of elementary plans. For example a Running_Total_Loop Plan is composed of different Variable plans which are associated with it by the use link.

3.1.3 Program representation

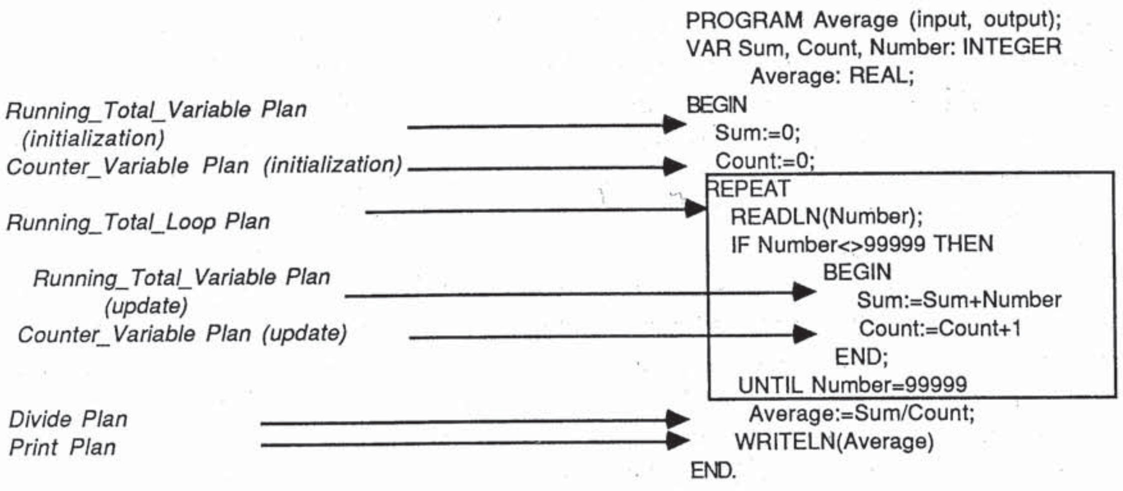
In the schema-based approach, a program may be represented hierarchically as goals and subgoals with potential programming schemas associated with each goal or subgoal. Figure 1a illustrates the notions of goals and subgoals for a program that computes the average of a set of numbers. Goals are decomposed into subgoals. For example, the goal 'report-average' is decomposed in three subgoals: 'enter-data', 'compute-average', 'output-average'. At each terminal subgoal in the tree there is an associated schema for coding that subgoal in Pascal.

Figure 1b illustrates the implementation of schemas in a program. It shows, for example, that the variable whose name is 'Count' implements a Counter_Variable

(a)



(b)


Running_Total_Variable Plan (initialization)
Counter_Variable Plan (initialization)
Running_Total_Loop Plan
Running_Total_Variable Plan (update)
Counter_Variable Plan (update)
Divide Plan
Print Plan

```
PROGRAM Average (input, output);
VAR Sum, Count, Number: INTEGER
     Average: REAL;
BEGIN
  Sum:=0;
  Count:=0;
  REPEAT
    READLN(Number);
    IF Number<>99999 THEN
            BEGIN
               Sum:=Sum+Number
               Count:=Count+1
            END;
    UNTIL Number=99999
    Average:=Sum/Count;
    WRITELN(Average)
END.
```

Figure 1: Representations of a program computing an average.
(a) Hierarchical representation of goals and schemas.
(b) Representation of the combination of schemas implemented in the program code.

plan. Its initialization corresponds to the line 'Count := 0' and its update corresponds to the line 'Count := Count + 1'.

In the control-flow approach, a program representation would be structured, at a detailed level, in terms of syntactic structures that could be combined, at a more abstract level, in terms of semantic structures.

3.2 Empirical support

Empirical data from a variety of tasks support the hypothesis that experts possess schemas of programming that represent semantic information. These include categorization tasks and using fill-in-the-blank tasks for plan-like and unplan-like programs.

3.2.1 Categorization of programs

The experts' knowledge organization gives them a capacity of processing that is superior to novices. Experts recall programs better than novices whenever the order of presentation is correct (Shneiderman, 1976) but this superiority disappears when programs are presented in random order so that meaningful structures are not visible.

The categories formed by experts indicate the knowledge structures they use. The categories formed by novices should be different from those formed by experts inasmuch as novices do not possess the same knowledge structures. Using a recall procedure, Adelson (1981) showed that categorization of programs or parts of programs is different according to the expertise of subjects. Novices' categories depend on surface features of the program like syntactic structure, whilst experts' categories cluster around elements performing the same function and elements displaying a procedural similarity.

These results suggest that experts possess knowledge structures grouping together information relative to the same function. This supports the schema-based hypothesis for programming plans. However, Adelson's results also suggest that experts possess knowledge structures that reflect the control structure of the program. This last finding is more consistent with the control-flow approach.

3.2.2 Understanding plan-like and unplan-like programs

Soloway and Ehrlich's results (1984) support the hypothesis that experts possess and use programming plans and rules of discourse in comprehending programs. If this hypothesis is correct, then giving experts programs that have a disrupted plan structure should make them much more difficult to understand than programs which are normal. Novices, who do not yet possess programming plans and conventions, should not be sensitive to whether programs do or do not conform to programming plans.

To evaluate this hypothesis, Soloway and Ehrlich constructed two versions of each of a set of programs: plan-like and unplan-like. In unplan-like versions, either the composition of the plans is not prototypical or the plans are implemented with values that are not prototypical. A further way of constructing unplan-like programs is to violate some rules of programming discourse. Figure 2 presents an example of plan-like and unplan-like versions of a program that computes an average.

PLAN-LIKE VERSION

```
PROGRAM Grey (input,output)
VAR Sum, Count, Num: INTEGER ;
  Average: REAL ;
BEGIN
  Sum:=0;
  Count:=0; * line to fill in
  REPEAT
    READLN(Num);
    IF Num <> 99999 THEN
                                  BEGIN
                                    Sum:=Sum+Num;
                                    Count:=Count+1;
                                  END;
  UNTIL Num=99999;
  Average:=Sum/Count;
  WRITELN(Average)
END.
```

UNPLAN-LIKE VERSION

```
PROGRAM Orange(input,output)
VAR Sum, Count, Num: INTEGER
  Average: REAL ;
BEGIN
  Sum:=99999;
  Count:=-1; * line to fill in
  REPEAT
    READLN(Num);
    Sum:=Sum+Num;
    Count:=Count+1;
  UNTIL Num=99999;
  Average:=Sum/Count;
  WRITELN(Average)
END.
```

DESCRIPTION (extract from Soloway and Ehrlich, 1984)
This program calculates the average of some numbers that are read in; the stopping condition is the reading of the sentinel value, 99999.
The plan-like version accomplishes the task in a typical fashion: variables are initialized to 0, a read-a-value/process-a-value loop is used to accumulate the running total, and the average is calculated after the sentinel has been read.
The unplan-like version was generated from the plan-like version by violating a rule of discourse: *don't do double duty in a non-obvious way.* That is, in the unplan-like version, unlike in the plan-like version, the initialization actions of the COUNTER VARIABLE (Count) and RUNNING TOTAL VARIABLE PLANs (Sum) serve two purposes:
-Sum and Count are given initial values
-the values are chosen to compensate for the fact that the loop is poorly constructed and will result in an off-by-one bug: the final sentinel value (99999) will be incorrectly added into the RUNNING TOTAL VARIABLE, Sum, and the COUNTER VARIABLE, Count, will also be incorrectly updated.

Figure 2: Program AVERAGE.

The subjects performed a fill-in-the-blank task: one line of the code was deleted and the subjects had to fill in the blank line with a line of code that in their opinion best completed the program. They were not told what the program was supposed to do. The results were that the experts performed better than the novices, that all subjects were more likely to get the plan-like version correct more often than the unplan-like versions, but that this difference was much greater for experts than for novices. These results support the hypothesis on experts' plan knowledge structures.

Furthermore, the schema-based model predicts the kind of errors that experts might make in the unplan-like condition. If their understanding is based on programming plans and rules of discourse then they will try to infer the missing line on this basis. Thus, they should tend to give plan-like answers even for unplan-like versions. This was indeed Soloway and Ehrlich's observation.

4 Comprehension mechanisms

4.1 Theoretical framework

According to a schema-based approach of understanding, schemas representing semantic knowledge are evoked while reading a program. They may be evoked either in a bottom-up way or in a top-down way. The direction of activation is bottom-up when the extraction of cues from the code allows the activation of schemas or when an evoked schema causes the activation of schemas it is part of. The direction of activation is top-down when evoked schemas cause the activation of less-abstract schemas; activation spreads from superstructures toward substructures. In Brooks' model (1983), the activation process is assumed to be mostly conceptually driven.

Concerning the evocation of schemas, Brooks attributes an important role to 'beacons', that is, to features or details visible in the program or the documentation as typical indicators of the use of a particular operation or structure. They allow the activation or recognition of particular schemas.

Détienne (1989) stresses that as most schema-based models focus on the construction of goal/plan representation based on activation and instantiation processes, it seems important to analyse the processes that evaluate this representation. Détienne distinguishes two processes: the evaluation of the internal coherence between plans and the evaluation of the external coherence between plans. The former consists in checking whether or not the values instantiated in a plan satisfy the constraints on the instantiation of the plan's slots. The latter consists in checking whether or not there are interactions between plans and between goals and if they create any constraints on the implementation of plans.

In the schema-based approach, the programmer is assumed to construct a representation of the functions performed in the program, making more or less explicit the data-flow relations in the program.

In the control-flow approach, syntactic constructs are assumed to be evoked first. This means that the programmer will construct representations of the control-flow relations in the program before the performed functions. Syntactic constructs are assumed to be combined in more and more abstract constructs until reaching a level of functional representation.

4.2 Empirical support

Data on the inferences a programmer makes in comprehension and recall tasks reflect the kind of knowledge used during comprehension and provide information on the way that knowledge is used. 'Chunks' of program code perceived by a programmer group together information belonging to the same mental constructs. Empirical data supply information on the kinds of chunks constructed by experts. The kind of evoked knowledge and comprehension mechanisms may vary according to the task performed. Some observations support the schema-based approach whereas others support the control-based approach.

4.2.1 Inferences collected in understanding tasks

According to the schema-based approach, schemas representing semantic knowledge are evoked in program comprehension. These schemas permit the drawing of inferences. The inferences collected during comprehension reflect the kind of knowledge that has been evoked in memory. To study this process of schema activation, Détienne (1986a, 1988) designed an experimental setting in which experienced programmers had to verbalize while reading programs (written in Pascal) presented one instruction at a time. At each instruction newly presented, the subjects had to express the information provided by it and elaborate any hypotheses concerning other instructions or the function of the program.

The results supported the hypothesis that experts possess programming plans like those formalized by Soloway. The verbal and behavioural protocols were coded in the form of production rules: 'IF A, THEN B' representing various mechanisms for using knowledge when drawing inferences. Evoked knowledge structures have been formalized as frames 'SCHEMAn' composed of variables 'VARn'. Two hundred and fifty-nine rules have been identified that have been classified as examples of forty-seven general rules. In its general form, a rule describes the types of variables (slot types) composing a schema. In its instantiated form, it describes possible values (slot fillers) that can instantiate a variable. In this section, several examples of these rules are presented in their instantiated form, with the slot types in parentheses.

Some identified rules describe activation processes that are data driven (the direction of activation is bottom up); they are expressed as 'IF VAR1a, THEN SCHEMA1' (VAR1a is a variable composing SCHEMA1) or as 'IF SCHEMA1a, THEN SCHEMA1' (SCHEMA1a is linked to a variable composing SCHEMA1). As instantiation begins as soon as a schema is evoked, some rules also express instantiation processes; expressed as 'IF VAR1a, THEN VAR1b' (VAR1a and VAR1b are two variables that are part of the same schema).

Détienne observed that a Counter_Variable plan, for example, can be evoked by the extraction of cues such as the variable's name or its form of initialization. This is illustrated by the following rules:

```
IF VAR1a (variable's name): I
IF VAR1b (variable's type): integer
THEN SCHEMA1 (schema of variable): Counter_Variable plan
THEN VAR1c (context): iteration
```

```
IF VAR1a (variable's initialization form): I := 1
THEN SCHEMA1 (schema of variable): Counter_Variable plan
THEN VAR1b (variable's update): I := I + 1
```

These rules make it clear that the experts infer other values associated to a Counter_Variable plan when it is activated. According to the first rule, the reading of the declaration of a variable whose name is 'I' and type is 'integer' evokes a Counter-variable plan. This activation allows the programmer to infer the value associated with another slot of this schema which is the context in which the variable 'I' is used. Thus, the subject will expect to see some kind of iteration in the program code. According to the second rule, the activation of a Counter-variable plan allows the subject to expect a particular form of update which is an incrementation.

Détienne observed that activation of elementary schemas such as variable schemas can allow the evocation of schemas representing algorithms that are partly composed of those elementary schemas. This is illustrated in the following rule:

```
IF SCHEMA1a (schema of variable): Counter_Variable plan: initialization I := Z,
    name I
IF SCHEMA1b (schema of loop): while A <> B
THEN SCHEMA1 (algorithmic schema): Linear_Search plan
THEN SCHEMA1a (schema of variable): Counter_Variable plan: update: I := I + 1
```

In this example, the initialization, the Counter variable and the while loop evoke an algorithmic schema for linear search. The elementary schemas that permit this activation are a Counter variable for which subjects infer the update and a schema of loop.

Other rules describe activation processes that are conceptually driven (the direction of activation is top-down); this is expressed as 'IF SCHEMA1 and..., THEN SCHEMA2' (SCHEMA 2 is a subcategory compared to SCHEMA1) or as 'IF SCHEMA1, THEN SCHEMA1a, THEN SCHEMA1c, ..., THEN SCHEMA1n'. Détienne (1988) gives an example of this process that allows the expectation of a complex combination of several schemas implemented in the code.

Norcio (1982) showed that semantic constructs can be evoked and inferences drawn from expectations based on comments in the program code. He asked subjects to fill in a blank line either at the beginning of a chunk or in the middle of a chunk. In one experimental condition, comments were inserted in the program at the beginning of each chunk. Results show a positive effect of comments compared to no comments in a fill-in-the-blank task when the line to fill in is at the beginning of a chunk. In that case, the comment provides the subjects with a cue for schema evocation.

Widowski and Eyferth (1986) have collected data on reading strategies used by programmers for programs varying along a dimension of stereotypeness. They remark that the strategies used by experts are different for usual and unusual programs. When reading usual programs, the activity seems to involve a conceptually driven processing. When reading unusual programs, they describe a more bottom-up oriented processing.

It should be noted too that in some cases, the evocation of schemas may create negative effects on performance inasmuch as the presence of inferred values is not confirmed in the code. Détienne (1984) observed this kind of negative effect in an experiment in which experts had to debug a program written by somebody else and where they had difficulties detecting errors when they strongly expected a particular value in the code (the actual value being different and incorrect). The experts failed to verify whether or not the value was actually present. This kind of negative effect is more likely to happen when the program has been written by the programmer, since this will create stronger expectations of what should be in the program.

In summary, data support the hypothesis that programming plans are evoked in program comprehension. The evocation of schemas allows inferences to be drawn. However, it may happen that there are no cues in the code to evoke schemas. Mental execution has been shown to be used to infer the goal of a part of code when the programmer has no expectation about it. By acquiring knowledge about the intermediate values of the variables during execution, the programmer can infer the goal of the process, and so, the goal of the part of code they have executed. Experts debugging programs have been observed to mentally simulate while not having enough textual cues about the goal of a part of code: no name, no familiar structure, no documentation (Détienne, 1984). This result suggests that schema evocation may be based either on static information like 'beacons' or 'dynamic information' which change with the program execution. This last kind of information is similar to the data-flow relations in a program.

Détienne and Soloway (1988) noted that mental simulation can also be used to infer information about the interactions between plans. In an experiment in which experts had to perform a fill-in-the-blank task on programs and had to verbalize while performing that task, they show that whatever the 'planliness' of programs is, the experts use simulation when they want to check for unforeseen interactions. This is typically used to understand programs in which a loop is used with a counter, suggesting that experts know by experience that the use of a count plan in a program can cause unforeseen interactions and that the best way to check for these interactions is to execute the part of the program with the count.

4.2.2 Inferences collected in recall tasks

Inferences collected in recall tasks subsequent to comprehension reflect the kind of knowledge that has been evoked in memory. Results of some studies support the hypothesis of semantic structures whereas other results support the hypothesis of syntactic structures.

In two experiments conducted by Détienne (1986b), experts were asked to recall a program following a debugging task. During the debugging phase, the subjects had to comprehend an unknown program and to evaluate its correctness. The programs were written in Pascal.

Distortions of the form of the program were observed in the recall protocols. For example, distortions concerned the name of a variable used as a counter. The subjects recalled I instead of J. This suggests that the variables are memorized in a category, a Counter_Variable plan, and that the lexical form is not kept in memory. In the recall process, the subjects use another possible value of the slot 'name of variable'.

Other observations suggest the existence of prototypical values in the slots of a schema. Each slot is associated with a set of possible values, as seen before. Those values have not the same status, some values being more representative of a slot for a schema than the others. When there is a prototypical value in a category, this value comes to mind first when the category is activated.

For example, in a schema for a flag, the slot 'context' is an iteration that can take the values 'repeat' or 'while'. In the program used, the variable, V, appears in a 'repeat' iteration. During the debugging phase, the value of the iteration expected by most of the subjects was 'while not V do'. In the recall protocols, a distortion was observed: a subject has recalled the instruction as 'while' instead of 'repeat'. This subject has reported the prototypical value instead of the adequate value. These observations give support to the hypothesis of prototypical values being associated to the slots of the schemas.

Distortions of the content of the program were also observed in the recall protocols. When a schema is activated, information associated to this schema is inferred. So information typical of a schema may be recalled while not included in the program text. Détienne reports a thematic insertion that was observed. For the particular domain of stock management, the following values could be assigned to the slots of the problem schema:

Task domain: stock management

Data structure: record(name of file, descriptor of file)...

Functions: allocation (creation or insertion), destruction, search

The function of creation was not isolated in a subprogram and the subjects did not read any information concerning this function in the program. Nevertheless, a subject has reported this function as if it was a subprogram. This suggests that schematic knowledge dependent on the task domain has participated in the elaboration of the representation and in the process of recovering of stored information at recall.

Thus, the data suggest that the knowledge structures evoked in comprehension are organized according to the semantics of that information. However, other data suggest that:

(1) experts also use knowledge structures that are organized in function of syntactic information (procedural);

(2) a representation based on this kind of knowledge may be constructed before one based on semantic knowledge (functional). This is compatible with the control-flow approach.

In Pennington's study (1987), subjects were asked to read a short program before answering questions. Results show that questions about control-flow relations are answered faster and more correctly than questions about data flow and functional relations. Furthermore, in a recognition memory test, subjects recognized a statement faster when it was preceded by another segment of the same control structure than when it was preceded by a segment of the same functional structure. This suggests that knowledge structures representing control flow have an important role in program understanding. Results also suggest that the understanding of the program control structures may precede the understanding of program functions.

However, results also suggest that understanding may be schema based or control based according to the understanding situation. Pennington remarks that there was an effect of the language. Two languages were used in the experiment: Fortran and Cobol. Cobol programmers were better at questions about data flow than were Fortran programmers, whereas control flow relations were less easily inferred by Cobol programmers. Pennington also remarks that there was an effect due to the comprehension task, i.e. the goal the subjects had while understanding the program. A modification task produced a shift toward increased comprehension of function and data flow at the expense of control-flow information.

4.2.3 Chunks constructed in program understanding

A chunk is the result of the identification of units belonging to a same mental construct. So a chunking task enables the study of the cognitive units on which comprehension is based. Several studies highlight the fact that the chunks constructed by experts and novices in understanding are different and that experts' chunks reflect semantic structures. This is compatible with the schema-based approach.

As programmers analyse programs on the basis of units that correspond to cognitive constructs, it is likely that highlighting those units in programs would facilitate the understanding process. Norcio (1982) has shown this type of effect by indenting programs on the basis of functional chunks defined by programmers and asking subjects to fill in a blank line either at the beginning of chunks or in the middle. The results indicate that subjects with indented programs supply significantly more correct statements compared to the non-indented group.

Black *et al.* (1986) and Curtis *et al.* (1984) remark that elements of code that are parts of the same plan are often dispersed in a program and that this characteristic of interleaving (see Chapter 1.2) is language dependent. This dispersion makes these parts difficult to integrate into a functional whole. This characteristic is part of what makes program comprehension hard. Letovsky and Soloway (1986) illustrate how difficult a program is to understand when using what they call a 'delocalized plan', i.e. a plan whose parts are dispersed in the program.

Rist (1986) provided experts and novices with programs to describe in terms of groups of lines of code that 'did the same thing'. In describing a program, novices used a mixture of syntactic and plan-based chunks. Experts used almost only plan-based groupings. Rist notices that when programs are complex, plan use decreases. In that case, construction of a mental representation of the program cannot be done from plan structure only, thus subjects use control-based program understanding. This last finding suggests that understanding may be schema based or control based according to the comprehension situation.

4.2.4 Comprehension mechanisms in different tasks

Comprehension is involved in different tasks of maintenance. Two tasks have been studied: enhancement (or modification) and debugging. In both of these tasks, studies show that experts may evoke programming plans so as to construct a representation of the program. However, studies also stress the importance of simulation mechanisms. This highlights the role of information on data-flow relations and control-flow relations in these tasks. This is compatible both with the schema-based approach and the control-flow approach.

In the task of modification, mental simulation has been observed to be used by experts. As simulation allows the inference of information on connections and interactions between functional components, it is likely to be used in tasks like enhancement in which processing this kind of information is particularly useful. Results from an experiment by Littman *et al.* (1986) show that some experienced programmers use this symbolic execution so as to acquire causal knowledge which permits them to reason about how the program's functional components interact during reading. Symbolic simulation is used to understand data flow and control flow.

This kind of strategy is the most effective inasmuch as it prevents subjects from introducing errors by modifying a part without taking into account the relationship between that part and other parts in the program. Simulation is a way to evaluate the external coherence between plans, i.e. to check whether or not there are interactions between plans and between goals (Détienne, 1989).

As reported before, Pennington (1987) noticed a marked shift toward increased comprehension of program function and data flow at the apparent expense of control-flow information following a modification task.

Simulation (i.e. mentally executing the program with values) is particularly useful in tasks like debugging in which it is important to judge whether the values produced at the execution are the expected values. In the debugging task, experts have been observed to mentally simulate the program so as to have information on the values taken by variables during the program execution (Détienne, 1984; Vessey, 1985).

It is noteworthy that mental simulation has also been observed in design tasks (see Chapter 3.3). It is not surprising, however, as the design task involves some comprehension. It is used so as to predict potential interactions between elements of the design and to check parts of the programs.

5 Discussion

Empirical data support the schema-based approach of program understanding. However, data also support the control-based approach. Studies show that according to the understanding situation, knowledge used may be related to different kinds of information: data-flow relations, functional relations or control-flow relations. Programming plans formalize information on data flow and functions whereas syntactic constructs reflect more the structure of the program as described with control-flow relations.

However, the two approaches of understanding are not contradictory. First, as said before, syntactic constructs may also be formalized as schemas. Secondly, a model of experts' knowledge may integrate those different kinds of knowledge. Thus, it should characterize the understanding situation so as to account for what kind of knowledge is used, when it is used and how it is used.

The understanding situation may be described by the characteristics of the language, the task, the environment and the subject. Concerning the language, the presence of cues in the code which allow the activation of schemas representing semantic knowledge may be dependent on the notational structure. Green (see Chapter 1.2) assumes that languages vary along a dimension of 'role-expressiveness'. With a 'role-expressive' language, the programmer can easily perceive the purpose, or role, of each program statement and, thus, the schemas evoked may be based on static cues whereas, with non-role-expressive languages, they may be based on the extraction of dynamic information like control-flow information.

Studies of programming have been developed on the basis of experiments conducted with a relatively narrow sample of programming languages: mostly Pascal, rarely Lisp, more rarely Basic, and more recently Prolog. It seems important now to conduct studies with other languages (for example, object-oriented languages) so as to take into account the effect of languages' characteristics in the understanding activity.

A model of the expert should be extended to take into account task variations. As Gilmore and Green (1984) remark, the information needed to be extracted from the code is different according to the task in which the programmer is involved. Furthermore, a particular language may make explicit in the code certain information or a particular environment may emphasize certain information important to achieve a particular task. So knowledge used by the expert may depend on the goal of his/her activity and on the availability of information in a particular situation.

References

Abelson, R. P. (1981). Psychological status of the script concept. *American Psychologist,* **36(7),** 715-729.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition,* **9(4),** 422-433.

Atwood, M. E. and Ramsey, H. R. (1978). Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging. US Army Research Institute for the Behavioral and Social Sciences. Technical report (TR-78-A21), Va: Alexandra.

Black, J. B., Kay, D. S. and Soloway, E. (1986). Goal and plan knowledge representations: from stories to text editors and programs. *In* J.M. Carroll (Ed.), *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* Cambridge, MA: MIT Press.

Bower, G. H., Black, J. B. and Turner, T. J. (1979). Scripts in memory for text. *Cognitive Psychology,* **11,** 177-220.

Brooks, R. (1983). Towards a Theory of the comprehension of computer programs. *International Journal of Man-Machine Studies,* **18,** 543-554.

Curtis, B., Forman. I., Brooks. R., Soloway. E. and Ehrlich. K. (1984). Psychological perspectives for software science. *Information Processing and Management,* **20(12),** 81-96.

Détienne, F. (1984). Analyse exploratoire de l'activité de compréhension de programmes informatiques. *Proceeding AFCET, séminaire 'Approches Quantitatives en Génie Logiciel'.* 7-8 June 1984, Sophia-Antipolis, France.

Détienne, F. (1986a). La compréhension de programmes informatiques par l'expert: un modèle en termes de schémas. Thèse de doctorat. Université Paris V. Sciences humaines, Sorbonne, 1986.

Détienne, F. (1986b). Program understanding and knowledge organization: the influence of acquired schemata. *Proceedings of the third European Conference on Cognitive Ergonomics*, Paris, September 15-20, 1986. (To appear in Falzon, P. (Ed). (1990). *Psychological Foundation of Human-Computer Interaction.* London: Academic Press.)

Détienne, F. (1988). Une Application de la Théorie des Schémas à la Compréhension de Programmes. *Le Travail Humain*, numéro spécial *Psychologie Ergonomique de la Programmation*, 1988, **51(4)**, 335-350.

Détienne, F. (1989). A schema-based model of program understanding. Eighth interdisciplinary workshop on 'Informatics and Psychology'. Schaärding (Austria), May 16-19 (to appear in *Mental Models in Human-Computer Interaction.* Amsterdam: North-Holland).

Détienne, F. and Soloway, S. (1988). An empirically-derived control structure for the process of program understanding. Research report 886, INRIA (to appear in International Journal of Man-Machine Studies).

Gilmore, D.J. and Green, T.R.G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine studies,* **21**, 31-48.

Letovsky, S. and Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software,* **3(3)**, 41-49.

Linger, R.C., Mills, H. D. and Witt, B. I. (1979). *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley.

Littman, D.C., Pinto, J., Letovsky, S. and Soloway, E. (1986). Mental models and software maintenance. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers: First Workshop.* Norwood, NJ: Ablex.

Minsky, M. (1975). A framework for representing knowledge. *In* P. Winston (Ed.), *The Psychology of Computer Vision.* New York: McGraw-Hill. pp. 211-277.

Norcio, A.F. (1982). Indentation, documentation and program comprehension. *Human Factors in Computing Systems,* **15-17**, 118-120.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology,* **19**, 295-341.

Rich, C. (1981). Inspection methods in programming. MIT AI Lab, Cambridge, MA., Technical report TR-604, 1981.

Rist, R. (1986). Plans in programming: definition, demonstration, and development. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers: First Workshop.* Norwood, NJ: Ablex.

Rumelhart, D. E. (1981). Understanding understanding. Report, Center for Human Information Processing, University of California, San Diego, California.

Schank, R. and Abelson, R. (1977). *Scripts-Plans-Goals and Understanding.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Man-Machine Studies,* **5(2)**, 123-143.

Shneiderman, B. (1980). *Software Psychology.* Cambridge, MA: Winthrop.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **10(5)**, 595-609.

Soloway, E., Ehrlich, K. and Bonar, J. (1982a). Tapping into Tacit Programming Knowledge. *Human Factors in Computer System,* **15-17**, 52-57.

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J. (1982b). What do novices know about programming? *In* A. Badre and B. Shneiderman (Eds), *Directions in Human Computer Interaction.* Norwood, NJ: Ablex.

Vessey, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies,* **23**, 459-494.

Widowski, D. and Eyferth, K. (1986). Representation of computer programs in memory. *Proceeding of the Third European Conference on Cognitive Ergonomics.* Paris, September 15-19.