An empirically-derived control structure for the process of program understanding

Françoise Détienne, E. Soloway

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 886

AN EMPIRICALLY-DERIVED CONTROL STRUCTURE FOR THE PROCESS OF PROGRAM UNDERSTANDING

Françoise DETIENNE
Elliot SOLOWAY

AOUT 1988

RR-0886

An Empirically-Derived Control Structure for the Process of Program Understanding[1]

Une structure de contrôle dérivée empiriquement pour le processus de la compréhension de programmes

Françoise DETIENNE
Projet de Psychologie Ergonomique pour l'Informatique
Institut National de Recherche en Informatique et Automatique
Domaine de Voluceau, Rocquencourt
B.P. 105, 78105 Le Chesnay (France)

Elliot SOLOWAY
Department of Computer Science
Cognition and Programming Project
Yale University
New Haven, CT 06520 (USA)

Thanks are due to David Littman for helpful comments on the manuscrit.

---

[1]Cet article a été soumis pour publication à la revue "International Journal of Man-Machine studies" en juin 1988. L'expérience qui y est décrite a été réalisée à l'occasion d'un stage de recherche fait à l'Université de Yale par le premier auteur cité. Ce stage était financé par une bourse postdoctorale accordée par l'INRIA.

Abstract

The contribution of this study is the identification of different mechanisms involved in program understanding by experts, specifically the mechanisms which cope with novelty. An experiment was conducted to identify and describe the expert's strategies involved in understanding usual (plan-like) and unusual (unplan-like) programs. While performing a fill-in-the-blank task subjects were asked to talk aloud. The analysis of verbal protocols allowed the identification of four different strategies of understanding. Under "normal" conditions the strategy of symbolic simulation is involved. Information extracted from the program allows the activation of schemas and evoked knowledge creates expectations on what information should be in the program. But when failures occur additional strategies are required. The authors identified three types of understanding failures the subject may experience (no expectation, expectations clashes, insufficient expectations) and the additional strategies invoked in those cases: (1) reasoning according to rules of discourse and principles of the task domain, (2) reasoning with plan constraints, (3) concrete simulation. The authors develop an operational description of these strategies and discuss the control structure of program understanding in the framework of schema theory.

Key words: Understanding, Programming, Processes and Knowledge.

Résumé

La contribution de cette étude est l'identification des différents mécanismes mis en oeuvre dans l'activité de compréhension de programmes non familiers par des experts. Une expérience a été conduite pour identifier et décrire les stratégies mis en oeuvre par des experts pour comprendre des programmes canoniques et non-canoniques. Les sujets devaient compléter des lignes effacées dans des programmes. Il leur était demandé de verbaliser. L'analyse des protocoles verbaux a permis d'identifier quatre stratégies de compréhension. Dans des conditions "normales", une stratégie de simulation symbolique est mise en oeuvre par les sujets. Les informations extraites du programme permettent aux sujets d'évoquer des schémas de connaissances. Les connaissances évoquées permettent en retour d'attendre certaines informations dans le programme lu. Dès que des difficultés de compréhension sont rencontrées, des stratégies supplémentaires sont mises en oeuvre. Les auteurs distinguent trois types de difficulté de compréhension: pas d'attente, attentes non satisfaites, attentes insuffisantes. Trois stratégies supplémentaires sont mises en oeuvre:(1) raisonner avec des règles du discours et des principes élémentaires du domaine du problème posé, (2) raisonner avec des contraintes associées aux schémas, (3) simulation concrète. Les auteurs développent une description opérationnelle de ces strategies et discutent de la structure de contrôle des processus de compréhension dans le cadre de la théorie des schémas.

Mots clés: Compréhension, Programmation, Processus et Connaissances.

1. Theoretical framework and goals

Various approaches to program understanding (Rich, 1981; Soloway, Ehrlich, Bonar & Greenspan, 1982; Soloway & Ehrlich, 1984) have been developed from Schema Theory. A Schema is a data structure that represents generic concepts stored in memory. The concept of a schema was developed in Artificial Intelligence (Schank & Abelson, 1977; Schank, 1980) to account for natural language processing and in Psychology (Bower, Black & Turner, 1979; Rumelhart, 1978, 1981) to explain results of studies on sentence memorization.

From the perspective of Schema Theory, understanding a program involves the evocation of schemas stored in memory. To date most authors have sought principally to identify the knowledge that programmers have and use in understanding programs. According to this work, experienced programmers possess "Programming plan", e.g. *program fragments that represent stereotypic action sequences in programming.*

Soloway et al. (1982) have formalized different programming plans[2] and rules of programming discourse which experts possess. These rules that are analogous to discourse rules in conversation specify conventions in programming, e.g., *the name of a variable should agree with its function.* Results of previous experiments have supported the hypothesis that expert programmers have and use these types of programming knowledge to support program understanding (Soloway & Ehrlich, 1984).

Knowledge, however, is only one aspect of program understanding, another being the cognitive mechanisms that use knowledge. In order to develop a more complete cognitively-based understanding of programming we need to develop a model that posits cognitive mechanisms that interact with experts' knowledge in order to produce the behavior of program understanding.

Some researchers (Brooks, 1983; Détienne, 1988a; Détienne 1988b, Letovsky, 1986a; Pennington, 1987) have modeled some of the cognitive mechanisms involved in program understanding. Détienne has collected and analyzed protocols which highlight the involvement of schema activation processes in either a data-driven (bottom-up) or conceptually-driven (top-down) manner in program understanding by experts.

Our goal is to collect data on the mechanisms involved in program understanding by experts specifically the mechanisms which cope with novelty. We assume that understanding a program requires constructing a representation of a program which consists mainly of plans and goals. This structure of goals and plans used to achieve the goals represents an explanation of what the program does and how it is done.

In the theoretical framework of Schema Theory, we can predict in which types of situation the mechanisms involved in program understanding might differ from one another. We can distinguish different situations as a function of the match between the program characteristics and the characteristics of the experts'

---

[2] A plan can be formalized as composed of slots (or variables). It includes constraints on the form in which they can be implemented, i.e., constraints on the values by which the plan's slots can be instantiated.

knowledge.

In a situation which we will call the "plan-like situation", the plans implemented in the program are very similar to the mental plans we believe subjects possess. In this plan-like situation, the activation and instantiation processes might be sufficient to understand a program: some cues evoke plans and there is a match between evoked plans and information extracted from the code. In particular, because program readers can match their internal representation of the plans very directly with the code in the program, the program reader does not need to explicitly reconstruct the causal relationships between the pieces of code in the program. Nonetheless we can expect failures whenever the cues that allow experts to evoke plans are not available in the code.

Most of the time the plans implemented in programs are in various ways different from the mental plans possessed by experts, so the activation and instantiation processes are not sufficient to account for program understanding. In these types of situation which we will call the "unplan-like situations3", either the way the plans are composed is not prototypical (usual) or a plan is implemented with a value which is not prototypical. Other mechanisms might be involved in constructing a representation of program.

A way of constructing unplan-like programs is to construct programs that violate some rules of discourse. These rules set up expectations in the minds of the programmers about what should be in the program. While understanding programs which do not conform to rules of programming discourse,an expert should experience failures insofar as the expectations drawn from his/her knowledge are not effective. He/she will then need to carry out more mental processing on the program. In particular, he/she might regenerate the causal links between the pieces of code, e.g., the programmer might make explicit the data flow, control flow, and goal relationships between the pieces of unplan-like code. These types of situation can be described as "no plan situation", i.e. the expert has no mental plan already constructed to explain the code.

The goal of this study is to identify different strategies involved in program understanding. As the planliness appears to be a relevant property with regard to the type of mechanisms which are selected and involved, it is important to collect data on the mechanisms involved in understanding programs that are either plan-like or unplan-like.

Our experiment has been conducted to identify and describe the experts' strategies involved in understanding plan-like and unplan-like versions of programs. The task of understanding is the fill-in-the-blank task in which subjects have to infer a blank line in a program. In order to identify different strategies involved in program understanding we collected data on the real-time activity of the program understanding. Having subjects talk aloud as they actively engage in an understanding task provides this sort of window onto subjects' processing strategies, and, then provide us with a more explicit view of the processes by which subjects develop their answers. Nevertheless the result of subjects' mental processing, i.e., the correctness of the answer in the fill-in-the-blank-task might allow us to evaluate the strategies used by subjects, i.e.,

---

[3]All situations are probably more or less plan-like. The planliness of a program is matter of degree on a continuum from sheer plan-like to sheer unplan-like situations.

their effectiveness.

In the remainder of this paper, we present the method we used to conduct our experiment. The data collected, i.e., verbal protocols collected during the understanding activity and the answer given as the result of the activity, have enabled us to construct a computational description of how understanding processes work, i.e., identify the conditions for the selection of a strategy, the goal the strategy tries to achieve, different characteristics of the strategy as the characteristics of the constructed representation, and the type of knowledge used.

2. Methodology

2.1. Subjects

Twenty two experienced subjects[4] participated in this study. All of them had at least two years of experience in programming with Pascal. These subjects were paid $15 per hour for participating in the experiment.

2.2. Material

Four programs[5] were used: Average, Sqrt, Maxnumber, Maxsentence. For each program, two versions were constructed: a plan-like version and an unplan-like version (see the Appendix).

2.3. Procedure

The subjects performed a fill-in-the-blank task: one line of the code was erased and the subjects had to fill the blank line in with a line of code that in their opinion best completed the program. We did not tell the subjects what the program was supposed to do. However, since there is only a single blank line per program, a great deal of context remained.

Five programs were presented successively to each subject. The first one was to familiarize the subjects with the task. The four other programs consisted of two plan-like programs and two unplan-like programs. Each subject received the four different program types but never received the plan-like and unplan-like version of the same program type. The order of presentation was counter-balanced.

Subjects were told their task was to fill in the blank line with a line of code which in your opinion best completes the program. They were asked to talk aloud during the experiment, i.e., talk about what they were doing/thinking/reading/writing everything that went through their head during the experiment. They were given as much time to perform the task as they wanted; almost all finished within thirty minutes.

---

[4]Most of them were graduate students in the Department of Computer Science at Yale University.

[5]There are the same programs which were used in the previous experiment by Soloway et Ehrlich (1984).

Subjects were then asked to rate the program comprehensibility and to rate the confidence they had in their answer. The points on the scale of program's comprehensibility were: (1) impossible to understand, (2) somewhat hard to understand, (3) somewhat easy to understand, (4) very easy to understand. The points on the scale of confidence with one's answer were: (1) not at all, (2) somewhat unsure, (3) fairly sure, (4) positive.

The responses in the fill-in-the-blank task were scored by the following weighting scheme: (0) no reponse was made, (1) incorrect response, (2) acceptable response, (3) correct response. By correct response, we mean the line of code that in our judgement best fulfills the overall intent of the program; it corresponds to the same answer in the plan-like and unplan-like versions of the same program. However in some case even if the answer is not the one expected, we have considered it as "acceptable" inasmuch as it completes the program in a consistent way.

They were four dependent variables: accuracy of the response, time spent to complete the problem, score of program's comprehensibility, score of confidence with answer.

3. Main results

As expected, the programs' planliness had an effect on the subjects' program understanding activity. Comparing the behaviors of experienced programmers in a plan-like and an unplan-like situation they performed better in the plan-like situation. The subjects' responses on the plan-like versions were significantly more correct than their responses on the unplan-like versions (Mann-Whitney U test, $z=1.87$, $p=.03$).

The timing data gathered in the fill-in-the-blank task should be suggestive for a model of the processing strategies employed by programmers. It is found that the time spent to perform the task is significantly longer on the unplan-like versions than on the plan-like versions ($F(1/84) = 7.85$, $p<.01$). Furthermore, it took subjects 72% more time to respond correctly to the unplan-like versions than it did to respond correctly to the plan-like versions. This suggests that more processing is required to understand an unplan-like program than to understand a plan-like program[6] inasmuch as processing is time-consuming.

Furthermore, subjects judged the unplan-like programs less comprehensible than the plan-like programs. (Mann-Whitney U test, $z=4.05$, $p<.00003$) and they are less confident with their answer for unplan-like programs compared to plan-like programs (Mann-Whitney U test, $z=2.71$, $p<.0034$).

It appears that four different strategies account for the behavior of subjects. In Figure 1, the strategies are roughly described and illustrated by protocols' excerpts. These different strategies involved in program understanding have been identified by the analysis of the verbal protocols collected while subjects performed the task. For each subject in each condition we have characterized his/her activity of understanding by identifying the different strategies he/she used to perform the fill-in-the-blank task. These data are displayed in Figure 2 in

[6]The results presented above replicate previous findings of Soloway & Ehrlich (1984).

which we present for each condition how many subjects exhibited the use of a certain combination of strategies[7].

Figure 2 makes it clear that the strategy called "symbolic simulation" is involved in all conditions. In fact, we have observed that this is the first strategy applied in program understanding. What differs according to the condition is the type of the additional strategies used.

As expected, extra processing is involved when the programs are unplan-like, i.e., other strategies are involved in addition to the symbolic simulation. The strategy called "reasoning on plan characteristics" is only involved in unplan-like situations: for the Maxnumber program, this strategy is used significantly more often in the unplan-like condition than in the plan-like condition (Chi-squared test=15.23, $p<.001$). The strategy called "rules of discourse and goal plausibility" is mostly involved in unplan-like situations: for the SQRT program, this strategy is used significantly more often in the unplan-like condition than in the plan-like condition (Chi-squared test=8.25, $p<.01$). The use of these strategies may therefore be related to, and even possibly triggered by, the failures subjects experience when the code does not conform to their expectations.

Another finding is that the use of a particular strategy seems to be dependent not only on the program planliness. Some strategies are used only for certain program types. For example, the strategy "concrete simulation" is mostly used only to understand the program type called "Average". In fact it seems that even in what we have called "plan-like situations" subjects may experience failures. This suggests that a more precise characterization of situations which entail failures must be made to account for the selection of each strategy by the subjects. This account will be the focus of the next section.

4. An operational description of the strategies

Symbolic simulation is the strategy used as long as the subject does not experience any understanding failure. Our data suggest that, inasmuch as no failure occurs, this strategy leads to the integration of information extracted from the code into a goal/plan representation. As soon as understanding failures are experienced, additional strategies are invoked.

A kind of failure which we call "no expectation" may be experienced whenever no plan can be evoked or plans evoked are too generic to account for the code in a definite enough way. The subject must construct a new plan to account for the code. We will illustrate later on that the strategy which appears to be involved in this case is the one called "reasoning according to rules of discourse and principles of the task domain". The subject builds up a new plan from elementary principles in the programming domain and in the task domain and also from generic programming plans[8].

Another kind of failure which we call "insufficient expectations" may be experienced when an activated plan must be integrated into a representation

---

[7]The "not categorized" category of Figure 2 corresponds to four protocols in which the subject has not verbalized enough to allow us to analyze his/her actiuvity.

[8]A generic plan represents a very general structure of a programming object.

which is a composition of plans about which the subject judges he/she does not have enough expectations according to certain criteria. We will describe a situation in which one plan of this composition has non-prototypical values and another plan is a count plan. In this kind of situation, the subject seems to judge he does not possess enough information based on the goal/plan representation to check for unforeseen interactions between plans or, in other words, to evaluate the external coherence between plans. This triggers the use of a strategy called "concrete simulation".

The values implemented in a plan are dependent not only on the constraints internal to a plan but they must be, in some way, dependent on the context in which this plan is implemented. Evaluation the external coherence between plans 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.

A third kind of failure may be called "expectation clashes". This is experienced whenever a value of the code contradicts the constraints on the instantiation of a plan. The subject sees in the code a value which is not expected and sometimes even contradicts what he expects to see. In this case, we will illustrate that the kind of strategy which is exhibited is the one called "reasoning on plan constraints". In these circumstances the subjects spend more processing effort to evaluate the internal coherence of the activated plan which lead them either to keep this plan, with some modifications, or to activate an alternative plan to account for the code.

Plans can be formalized as composed of slots (or variables). They are assumed to include constraints on the form in which they can be implemented, i.e., constraints on the values by which the plan's slot can be instantiated[9]. Evaluating the internal coherence into plans consists in checking whether or not the values instantiated in plan satisfy the constraints on the instantiation of the plan's slot.

In the following paragraphs we develop, for each strategy, an operational description which makes explicit the conditions for the selection of a strategy in terms of the kind of failure the subject experiences and the goal achieved by the strategy. This is summarized in Figure 3. The description of the strategies was given earlier and can be found in Figure 1. Each strategy will be illustrated through excerpts of verbal protocols. Based on our data we also attempt to evaluate the effectiveness of each strategy.

4.1. Symbolic simulation

4.1.1. Conditions for the selection of the strategy

This strategy is the first one used by subjects to understand a program and so is not triggered by understanding failures of all types. On the contrary, this strategy leads to different kinds of failures depending on the situation. As we have seen previously, this strategy is involved in the understanding activity whatever the program planliness and type are (see Figure 1), but these situations

[9] Instraslot constraints define a set of values allowed to instantiate a slot and interslot constraints define a subset of values which can instantiate a slot if another slot has been instantiated by another value. For instance, in a Count-plan, an intraslot constraint may bear on the form of the updating of the count-variable: "the updating must be an incrementation".

differ depending on whether or not this strategy is used alone. In the plan-like situation, the use of symbolic simulation represents 93% of the observations. In 77% of these observations, this strategy is the only one used. On the other hand, in the unplan-like situation, subjects use this strategy in 95% of the observations and this strategy is used alone in only 40% of these observations.

4.1.2. Goal

Our observations lead us to the conclusion that the goal of this strategy is to construct a goal/plan representation of the program. The fact that this strategy leads to the integration of information extracted from the code in a goal/plan representation is obvious in the following excerpt recorded after a subject has used this strategy in a plan-like situation:

(Maximum plan-like, subject 1)
...That looks like a maximum algorithm of some type, it looks like it loops ten times, reads a number, compares the number to maximum and sets Max to Num if something is true, then this must be, oh I guess "greater than" ....

After having used a symbolic simulation the subject is able to make explicit the different subgoals of the program ("loops ten times, read a number") and he evokes a programming plan for computing a maximum ("it looks like a maximum algorithm of some sort"). When the symbolic simulation is successful, i.e., has allowed the construction of a goal/plan representation of the program, then inferring the missing line is quite straightforward and rapid as in the excerpt above ("I guess greater than").

In some protocols, it is made explicit how the representation constructed as a result of the symbolic simulation allows the subject to infer the role of the missing line. This is explicit in the protocol of some subjects for the Average program as in the following excerpt.

(Average unplan-like, subject 1)
....OK, Sum is initialized, Num doesn't need to be because you get it then you add it...and Count, oh Count has not been initialized, OK, I was looking for something to be initialized, Count should be initialized...

After having performed a symbolic simulation of the program, the subject draws the inference that the role of the missing line is the initialization of a variable and then checks which variable has not been initialized yet. In this case the subject seems to use two types of knowledge: knowledge of the role structure of the program (input, calculate, output) and knowledge of specific programming plans (a count plan in this case). Then he relates them together: the role of the missing line is "input" which can be related to the slot "initialization" of a variable plan. One constraint of initialization for a Count-variable might be: the initialization must be an assignment by 0, +1 or -1, 0 being the most prototypical value. Then this constraint may be used in order to infer the missing initialization of a Count-variable.

4.1.3. Empirical evaluation

When used alone, this strategy is effective for inferring the missing line in the plan-like situation and leads to correct reponses in 94% of the observed cases.

However, when used alone in an unplan-like situation, it leads to correct responses only in 41% of the observed cases. This confirms that in these cases a symbolic simulation is not sufficient to construct a correct goal/plan representation of the program.

4.2. Reasoning according to rules of discourse and principles of the task domain

4.2.1. Conditions for the selection of the strategy

This strategy, involved in 34% of the observed situations, is used when subjects experience a failure which we have called "No expectation". It is used by subjects for the Maxsentence program whatever version it is and for the unplan-like version of the Sqrt program (see Fig 1).

In the Maxsentence program, even in its plan-like version, it seems that the plan implemented in the program (RESET to boundary condition PLAN) cannot be evoked by the subjects and even confuses the subjects[10]. In this situation, therefore, we can say that the condition for the selection of this strategy is that no mental plan and no goal account for the program Maxsentence. The fact that subjects cannot evoke the RESET PLAN can be explained in two ways: (1) the RESET PLAN is not familiar to subjects, which is unlikely for experienced subjects, or (2) the RESET PLAN cannot be evoked because the missing line is the most informative line in this plan of the program and thus, what Brooks (1983) would call a beacon. The fact that no plan accounts for the program is clear in the following excerpts, recorded before some subjects start using this strategy.

(Maxsentence unplan-like, subject 19)
...I don't know what they want to do with "while sentence greater than Maxsentence"...
(Maxsentence unplan-like, subject 15)
...What the program is for?...

For the unplan-like version of the Sqrt program, no goal accounts for the program. The subjects judge that the goal which is "to compute the square root of the same number ten times" is not plausible and, so, reject this goal.

(SQRT unplan-like, subject 1)
...If it would be doing this around ten times it would not doing anything...

4.2.2. Goal

The goals of this strategy are to construct a new plan and to evaluate the internal coherence of this plan. The subjects construct a new plan by the evocation of rules of discourse, principles of the task domain and also the evocation of generic programming plans.

For the sqrt unplan-like version the construction of a new plan is made on the basis of a generic plan that has been modified (specialized). Most of subjects

[10] Some subjects try to activate familiar plan as for example a Maximum-search-loop-plan but this misleads them and they realize this is wrong because a value in the code is inconsistent with this plan, i.e., the fact that Max is a constant and not a variable.

seem to have evoked a very generic plan "input data and compute data". This generic plan is sufficient in the plan-like situation to infer the role of the missing line. As the variable Num is computed in some way, then it must be input somewhere and a protypical way to do this is via a read statement. But in the unplan-like version, both of the slots of this generic plan are filled in as the Num variable is initialized via an assignment. The subjects realize that it makes the goal of the program implausible inasmuch as this would compute the square root of the same number ten times.

The subjects try to infer the subgoal (or role) achieved in the program by using rules of discourse ("if there is a loop then something must change at each execution" or " if there is a test for a condition then the condition must have the potential of being true"). This allows the subject to infer a modification of the goal and then a subgoal which would be more plausible, which is "modify data". In those situations, the programmer generates the causal links between the pieces of code, e.g., he makes explicit the goal relationship between the pieces of code.

4.2.3. Empirical evaluation

This strategy is relatively effective. It leads to errors in only 20% of the observed situations.

4.3. Concrete simulation

4.3.1. Conditions for the selection of the strategy

The strategy of concrete simulation is used whenever subjects experience the kind of failure which we have called "insufficient expectations": an activated plan must be integrated in a composition of a plan about which the subject judges that he does not have enough expectations according to certain criteria.

The strategy of concrete simulation is used for the Average plan-like and Average unplan-like versions. What is specific to those programs is that a loop is used with a counter. This leads us to say that the presence of a count plan can trigger this strategy. In this kind of situation, the subject seems to judge that he does not possess enough information based on his goal/plan representation to evaluate the external coherence between plans. This is shown in the following excerpt in which the subject justifies why he has used a concrete simulation.

(Average plan-like, subject 8)
...The problem is making sure that you're not off by one, the option is setting by zero, -1, +1 and you have to make sure...

In the unplan-like condition, the simulation is run with the sentinel value by 6 subjects whereas this is done only by 2 subjects in the plan-like condition (this difference approaches conventional levels of significance by Fischer's exact test ($p=.07$). The unplanliness of the program affects the initialization value of the sum plan (running-total-variable plan), which is not prototypical. This value also represents the sentinel value of the loop plan. In this case, this value is used for the mental execution of the program, so the unplanliness of the program triggers a more specific strategy which is mental execution of a sentinel value. That means that the strategy of concrete execution which is unspecified when the

program is plan-like, is used in a more specific way when the program is unplan-like.

4.3.2. Goal

Our observations lead us to say that the goal of this strategy is to evaluate the external coherence between plans. i.e., to check for unforeseen interactions. The kind of representation which supports this strategy is dynamic. More exactly, the subject seems to shift from a static representation in terms of plan to a dynamic representation in terms of data flow.

The strategy of concrete simulation is used regardless of the planliness of the program. This suggests that the experts know by experience that the use of a count plan in a program often causes unforeseen interactions and that the best way to check for these interactions is to execute the part of the program with the count.

4.3.3. Empirical evaluation

The strategy of executing sentinel values is effective in the plan-like situation; it leads to correct responses in 100% of the observed cases. Surprisingly, it is less effective in the unplan-like situation, in which it leads to correct responses in only 57% of the observed cases. It does not permit the detection of an initialization of the count variable with a prototypical value, which is zero, when that value is wrong. This result underlines some limits of expertise inasmuch as the experts seem to trigger a specific strategy in a specific situation, a strategy which is often thought as particularly useful, but which strategy is not effective in this situation.

4.4. Reasoning with plan constraints

4.4.1. Conditions for the selection of the strategy

This strategy of reasoning with plan constraints seems to be used after the subjects have experienced an understanding failure which we have called "expectation clashes". This strategy is used in approximately 20% (9/44) of the observed situations.

This strategy is used only to understand the unplan-like version of the Maxnumber program. What characterizes this situation is that the program uses a very familiar plan (Max plan) but the plan is implemented with a non-prototypical value for the initialization of the variable (99999 instead of 0) and that this value discriminates between a Max plan and a Min plan. Thus, a value of the code contradicts the constraints on the instantiation of the Max-plan so the subject sees in the code a value which is not expected and even contradicts what he expects to see. This kind of failure is shown in the following excerpts.

(MaxNumber unplan-like, subject 4)
...the line Max:=99999 is a little obscure... I don't see why the maximum is initialized like that I would have initialized it to zero...
(MaxNumber unplan-like, subject 16)
...Usually this kind of initialization would be a very small number to try to find a larger number...

4.4.2. Goal

The goal of this strategy is to evaluate the internal coherence of an activated plan. A Maximum-search-loop-plan is activated but the subjects notice that the value of initialization is unusual and could even activate an alternative plan, the Minimum-search-loop-plan. In this situation they either try to keep the activated plan (Maximum-search-loop-plan) and to customize the goal achieved by this plan so it is consistent with the value of initialization (in this case, this particular Maximum-search-loop-plan computes the max of numbers over 99999) or to activate the alternative plan (see excerpt displayed in figure 1). Some subjects find a way to explain why the discourse rule "a variable name should reflect its function" has been violated (a programmer has modified this program and forgotten to change the variable's name, for example).

4.4.3. Empirical evaluation

This strategy is effective. It leads to correct responses in 78% of the observed cases (7/8) and acceptable responses[11] in 22% of the observed cases (2/9).

4.5. Remarks

We have observed another strategy which could be called "reasoning driven by parallelism between plans". Although we have only one sample of this strategy which has been used only by one subject in the average plan-like condition, it seems noteworthy. This strategy can be characterized by a reasoning process which is supported by a parallelism drawn between two plans. In the observed case, the parallelism is drawn between the function of the initialization in two variable plans, a count plan and a sum plan (running-total-variable plan) implemented in the Average program. This is illustrated by the following excerpt:

(Average unplan-like, subject 1)
...Ok, Count should be zero. However they add Num, they add Num into the Sum and then substract it (he points out the initialization of Sum), so Count should be one off. I want to make count one bigger, I mean one smaller, so at the end it's less. So I assume it should be negative one...

This strategy is used in the same situation as the concrete simulation strategy and seems to achieve the same goal, which is to evaluate the unforeseen interactions between plans, e.g., to evaluate the external coherence between plans. In contrast with the strategy of concrete simulation which uses a dynamic representation of the program, this strategy uses a static representation and is driven by parallelism between plans. The subject draws an analogy between the function of the initialization of the Sum plan and the function of the initialization of the Count plan. However, he uses a concrete simulation afterward, so as to check the external coherence in another way.

---

[11] Here, we consider the answer > as acceptable whenever the subject explains at the same time that the program computes the maximum of numbers over 99999. In contrast, we consider the same answer as an error whenever the subject explains that the program computes the maximum of numbers.

5. Discussion

This experiment gives us information for modeling the goals experts have in understanding programs. The structure of these goals guides the selection and use of strategies. According to our data, several goals are prevalent in the understanding activity: the construction of a goal/plan representation, the construction of new plan, the evaluation of the representation, i.e., the evaluation of the internal coherence into plans and the evaluation of the external coherence between plans.

The "construct or expand a goal/plan representation" goal refers to the construction of the current representation of the program. As far as possible the construction of the representation is based on the evocation of plans already constructed. This is based on the extraction of cues in the code that evoke some plans. For example, the name of a variable called Max is likely to evoke a Maximum-search-loop-plan. Whenever no plan acounts for a part of the program, the subject constructs a new plan.

When a plan is already in memory and evoked to account for the code, the "evaluate internal coherence into plans" goal consists in checking whether or not the values instantiated in plan satisfy the constraints on the instantiation of the plan's slots.

Constraints must exist on the way plans may be implemented. For instance, one of these constraints for a Maximum-result-variable-plan might be: the initialization must be an assignment to a small number which is prototypically zero. Then reading the code "Max:=0" satisfies this constraint; on the contrary, reading the code "Max:=99999" contradicts it. When the plan which is being evaluated has just been created, the subjects reasoning might not bear on constraints which do not exist yet but might bear on rules of discourse.

The "evaluate external coherence between plans" goal 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.

The values implemented in a plan are dependent not only on the constraints internal to a plan but they must also be in some way dependent on the context in which this plan is implemented. For instance, the way a count-variable must be initialized varies according to the relationship between the count-plan and the loop plan in which it appears, as well as according to the kind of loop plan. With a Repeat-loop-plan, the count-variable counts the last number entered in the loop which represents the sentinel value (except if we add a valid-data-entry plan in the loop). With a while-loop-plan, the count-variable does not count the last number tested which represents the sentinel value. So if we want the count-variable not to count the sentinel-value, the Count-variable must be initialized to -1 in a repeat-loop-plan context and to 0 in a while-loop-plan context.

Our data suggest that the achievement of each goal involves a different strategy and that, to cope with novelty, more than one strategy is used. To construct a model of program understanding based on achieving program understanding goals, this research should be extended in the following directions.

First, it is likely that strategies different from the ones described in this paper

are involved in the activity of program understanding. In particular, it is possible that different strategies can be used to achieve the same goal. For example, we have noticed that two strategies may be involved to achieve the goal "evaluate external coherence": concrete simulation and reasoning driven by parallelism between plans. These strategies differ according to the type of representation on which they work, which is either dynamic or static.

Second, it is likely that some strategies are easier to use and more efficient in certain contexts. For instance, in function of a characteristic of the program, like its length and of tools available, one of the strategies involved to achieve the goal "evaluate external coherence" may be easier to use than other one. In large programs, there are potentially more problems of interactions between plans. There are also interactions between parts of the code which can be very far from one another, as in "delocalized plans" (Letovsky, 1986b). Using a strategy of concrete simulation in this case seems very difficult unless some tools support this activity. Using a strategy of reasoning driven by parallelism between plans seems a priori more useful in this case inasmuch as information on delocalized plans and interactions between parts of code are made explicit in the documentation.

It is also possible that one strategy is more useful than another according to characteristics of the task inasmuch as a particular task might require the subject to achieve one particular understanding goal of understanding. For example, a debugging task requires careful evaluation of the representation constructed from a program. In such a situation a strategy more likely to be useful might be a concrete simulation. Littman and al's empirical data (Littman, Pinto, Letovsky & Soloway, 1986) show that a strategy of simulation is usefull in an enhancement task and that there is a strong relationship between using this strategy to acquire knowledge about the program and modifying it successfully.

To conclude, our experiment supplies important information for developping tools to support the understanding activity: the structure and the content of the representation on which the experts work while understanding programs. Two types of representation have been identified: one representation is in terms of goals and plans and another representation is in terms of data flow. This means that it could be useful to supply the subject with information structured in those ways: abstraction of the program text structured in terms of goals in the programs and associated plans and abstraction of the program structured in terms of the multiple transformations of the data objects. In the same way, it would be important to find a format for a representation which allows to go easily from one abstraction to the other, i.e. to shift representations, so as to facilitate linking multiple representations.

References

Bower, H. H., Black, J. b. & Turner, T. (1979) Scripts in Memory for Text. Cognitive Psychology, 11, p 177-220.

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

Détienne, F. (1988a to appear) Program Understanding and Knowledge Organization: The Influence of Acquired Schemas. In P. Falzon, J-M. Hoc, N. Streitz & Y. Waern (Eds): Psychological foundations of Human-Computer Interaction, Springer, Cognitive sciences series, NY.

Détienne, F. (1988b to appear) L'application de la théorie des Schémas à la Compréhension de programmes, Le travail Humain, special issue "Psychologie ergonomique de la programmation".

Letovsky, S. (1986a) Cognitive Processes in Program Understanding. In E. Soloway & S. Iyengar (Eds): Empirical Studies of Programmers. Proceeding of the first workshop. Norwood, N.J.: Ablex Publishing corporation.

Letovsky, S. & Soloway, E. (1986b) Delocalized Plans and Program Comprehension, IEEE Software, 3 (3), 41-49.

Littman, D. C., Pinto, J., Letovsky, S. & Soloway, E. (1986) Mental models and Software Maintenance. In E. Soloway & S. Iyengar (Eds): Empirical Studies of Programmers. Proceeding of the first workshop. Norwood, N.J.: Ablex Publishing corporation.

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

Pennington, N. (1987) Comprehension Strategies in programming. In G. Olson, S. Sheppard & E. Soloway (Eds): Empirical Studies of Programmers. Proceeding of the second workshop on empirical Studies of Programmers. Norwood, N.J.: Ablex Publishing corporation.

Rumelhart, D. E. (1978) Schemata: the Building Blocks of Cognition. University of California, Center for Human Information Processing, San Diego, California.

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

Schank, R. (1980) Language and Memory. Cognitive Science, 4, 243-284.

Schank, R. & Abelson, R. (1977) Scripts-Plans-Goals and Understanding. Laurence Erlbaum Associates, Hillsdale, NJ.

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

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J, (1982) What do novices know about programming? in A. Badre & B. Schneiderman (Eds): Direction in human computer interaction. Norwood, N.J: Ablex Publishing Corporation.

Symbolic simulation:
The subject simulates the different steps in which the program is executed.
(Max plan-like, subject 3)
...Max equal zero, for I equal 1 to 10 do read Num...
(Average plan-like, subject 10)
....read a number, if number different from 99999 add to the sum, count....
Reasoning on rules of discourse and principles of task domain:
The subject's reasoning is supported by the evocation of rules of discourse and/or principles of the task domain concerning the goal plausibility.
(If/While unplan-like, subject 2)
"..This is a constant (he points out Maxsentence) and this is a variable (he points out Sentence again), so I suppose, if you check if this thing (Sentence) is greater than this one (Maxsentence), you just want to decrease the sentence..."
(If/While unplan-like subject 17)
"...We're gonna make sure that the sentence is not bigger than the maximum sentence because we don't want to give somebody a sentence longer than he's able to live..."
Concrete simulation:
The subject mentally runs the program with values, e.g., boundary values.
(Average unplan-like, subject 1)
Count:=-1, Ok, now I check if it works.
Count is negative one, I set the number with one number which is five so the number should be five, so the average should be five, I put five in Sum, add 1, Num is not 99999, enter 99999 to get out, Sum is now five, Count is one...
(Average unplan-like, subject 15)
Ok, 99999 (he points out the initialization of Sum) is the sentinel value...(he mutters)...presumably...initialization is zero...plus one, then Sum equal zero...
Reasoning on plan constraints:
The subject evokes some plan constraints in order to keep or reject this plan as a part of his/her program representation.
(Maxnumber unplan-like, subject 14)
Except why the Max is initialized to be 99999, it's a pretty big number... so I can think of two possibilities, one is that, this is less than. it's actually computing the minimum and someone called the minimum variable Max, or it's greater than and it's computing the maximum of numbers and you know that the numbers will to be greater than 99999 for some reasons.
(Maxnumber unplan-like, subject 10)
I assume that this computes the maximum of len numbers. Oh, no! If you're computing the maximum, you must initialize the maximum to zero, not to 99999....

Fig. 1.
Description and illustration of different strategies involved in program understanding

PROGRAMS / STRATEGIES	AVERAGE		MaxNumber		SQRT		Maxsentence	
	Plan like	Unplan like	Plan like	Unplan like	Plan like	Unplan like	Plan like	Unplan like
SYMBOLIC SIMULATION (SS) alone	6	4	10	2	10	5	7	6
SS + CONCRETE SIMULATION	4	6	0	0	0	0	0	0
SS+ REASONING ON PLAN CHARACTERISTICS	0	0	0	8	0	0	0	0
SS + RULES OF DISCOURSE and PRINCIPLES OF TASK DOMAIN	0	0	0	0	0	5	4	5
SS + RULES of DISCOURSE ... + CONCRETE SIMULATION	0	0	0	0	0	1	0	0
REASONING ON PLAN CHARATERISTICS alone	0	0	0	1	0	0	0	0
NON IDENTIFIED	1	1	1	0	1	0	0	0

Figure 2.
Numbers of subjects using
a certain combination of strategies per condition

STRATEGY INVOLVED	GOAL OF THE STRATEGY	FAILURE CHARACTERIZATION
Symbolic simulation	CONSTRUCT GOAL/PLAN REPRESENTATION	No failure
Reasoning according to rules of discourse and principles of the task domain	CONSTRUCT NEW PLAN	No expectations, i.e., either no plans or no goals
Concrete simulation	EVALUATE EXTERNAL COHERENCE	Insufficient expectations
Reasoning on plan constraints	EVALUATE INTERNAL COHERENCE	Expectation clashes

Figure 3:
Characterization of different strategies involved in program understanding.

APPENDIX
Program 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 et 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 initialisation 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.

APPENDIX
Program MAXNUMBER

PLAN-LIKE VERSION

```
PROGRAM Magenta(input, output);
VAR Max, I, Num: INTEGER;
BEGIN
    Max:=0;
    FOR I:=1 TO 10 DO
        BEGIN
            READLN(Num);
            IF Num>*Max THEN Max:=Num    line to fill in
        END;
    WRITELN(Max);
END.
```

UNPLAN-LIKE VERSION

```
PROGRAM Purple(input, output);
VAR Max, I, Num: INTEGER;
BEGIN
    Max:=99999;
    FOR I:=1 TO 10 DO
        BEGIN
            READLN(Num);
            IF Num<*Max THEN Max:=Num    line to fill in
        END;
    WRITELN(Max);
END.
```

DESCRIPTION (extract from Soloway et Ehrlich, 1984)

In the plan-like version, the program finds the maximum of some numbers. It uses the MAXIMUM SEARCH LOOP PLAN which in turn uses a RESULT VARIABLE PLAN. The RESULT VARIABLE is appropriety named Max.

In the unplan-like version, the program finds the minimum of some numbers. It uses the MINIMUM SEARCH LOOP PLAN in the RESULT VARIABLE is inconsistent with the plan's function: the program computes the minimum of some numbers using a variable name Max. A rule of discourse is violated: *a variable's name should reflect its function.*

APPENDIX
Program MAXSENTENCE

PLAN-LIKE VERSION

```
PROGRAM Gold(input, output);
CONST
   MaxSentence=99;
   NumOfConvicts=5;
VAR
   ConvictID, I, Sentence: INTEGER;
BEGIN
   FOR I:=1 to NumOfConvicts DO
      BEGIN
         READLN(ConvictID, Sentence);
         IF Sentence>Maxsentence
            THEN Sentence:=MaxSentence;*   line to fill in
         WRITELN(ConvictID, Sentence);
      END;
END.
```

UNPLAN-LIKE VERSION

```
PROGRAM silver(input, output);
CONST
   MaxSentence=99;
   NumOfConvicts=5;
VAR
   ConvictID, I, Sentence: INTEGER;
BEGIN
   FOR I:=1 to NumOfConvicts DO
      BEGIN
         READLN(ConvictID, Sentence);
         WHILE Sentence>Maxsentence
            DO Sentence:=MaxSentence;*   line to fill in
         WRITELN(ConvictID, Sentence);
      END;
END.
```

DESCRIPTION (extract from Soloway et Ehrlich, 1984)

The program tests to see if some variable contains a number that is greater than a maximum, and if so, the variable is reset to the maximum (RESET PLAN). The plan-like version uses an IF statement, the unplan-like version uses a WHILE statement.

The unplan-like version was generated from the plan-like version by violating the following discourse rule: *An IF should be used when a statement body is guaranteed to be executed only once, and a WHILE used when a statement body may need to be repeatedly executed.*

APENDIX
Program SQRT

PLAN-LIKE VERSION

```
PROGRAM Beige (input, output);
VAR  Num: REAL;
     I    : INTEGER;
BEGIN
    FOR I:=1 TO 10 DO
        BEGIN
        READ(Num);*    line to fill in
        IF Num<0 THEN Num:=-Num;
        Writeln (Num, Sqrt(Num)),
            (*Sqrt is a built-up function which returns the square root of its argument*)
        END;
END.
```

UNPLAN-LIKE VERSION

```
PROGRAM Violet (input, output);
VAR  Num: REAL;
     I    : INTEGER;
BEGIN
    Num:=0;
    FOR I:=1 TO 10 DO
        BEGIN
        READ(Num);*    line to fill in
        IF Num<0 THEN Num:=-Num;
        Writeln (Num, Sqrt(Num));
            (*Sqrt is a built-up function which returns the square root of its argument*)
        END;
END.
```

DESCRIPTION (EXTRACT from Soloway et Ehrlich, 1984)

This program produces the square root of Num. Since N is in a loop which will repeat 10 times, 10 values will be printed out. A DATA GUARD PLAN is used protects the Sqrt function from trying to take the sqrt of a negative number.

In the unplan-like version, the VARIABLE PLAN for Num starts off with an assignment type of initialization (Num:=0). This is due to the violation of several rules: (1) *don't include code that won't be used* (the line of code "Num:=0" is not usefull in the program), (2) *a variable that is initialized via an assignment statement should be updated via an assignment statement* (as the Num variable is initialized via an assignment statement, it should be updated via an assignment statement).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique