Strategies in programming programmable controllers: A field study on a professional programmer

Willemien Visser

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

STRATEGIES IN PROGRAMMING PROGRAMMABLE CONTROLLERS: A FIELD STUDY ON A PROFESSIONAL PROGRAMMER

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

Abstract. One of the questions raised at the end of the First Workshop on Empirical Programmers (see 17), and which subsequently became the title of a Future Directions paper, was "By the way, did anyone study any real programmers?" (7). Our answer is "Yes." It is our wish in presenting this paper to contribute to the understanding of some aspects of "programming-in-the-large," in particular those concerning the specific strategies that the programmer uses.
A professional programmer constructing a program that was to control an automatic machine tool installation was observed full time for four weeks in his daily work.
In this paper, we chose to focus on the strategies used, under the hypothesis that they differ, at least partially, from those observed to date in most novice, student programmers working on artificial, limited problems.
We observed some strategies already known to be at work in "programming-in-the small": planning, top-down and bottom-up processing, schema-guided information processing. However, other strategies seem indeed to be characteristic of programming in a work context: the frequent use of example programs, the importance of analogical reasoning, and the search for homogeneity (for comprehension and maintenance reasons). Finally, the opportunistic nature of the activity we observed also seems to be a characteristic of real programming activity.

Keywords. Program design; Software design; Programming strategies; Design strategies; Opportunistic organisation; Real-time observational study; Field study; Protocol analysis; Industrial programmable controller; Automated machine-tool installation

Résumé. Cet article présente quelques aspects de la programmation "in-the-large", notammentles stratégies particulières utilisées par un programmeur professionnel. L'hypothèse qui sous-tend cette étude était que ces stratégies diffèrent, au moins en partie, de celles observées jusque là auprès de la plupart des programmeurs novices, généralement des étudiants travaillant sur des problèmes artificielllement restreints.
Un programmeur professionnel a été observé pendant quatre semaines tout au long de son travail qui consistait à concevoir un programme de commande pour une installation automatisée de machine-outil. Certaines stratégies dont on savait déjà qu'elles sont mises en oeuvre dans la programmation "in-

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

the-small" on été observées: de la planification, du traitement de-haut-en-bas et de-bas-en-haut et du traitement de l'information guidée par des schémas. Cependant, d'autres stratégies semblent en effet être propres à la conception de logiciel dans un contexte de travail professionnel: l'utilisation fréquente de programmes-exemples, l'importance du raisonnement analogique et la recherche de structures homogènes (pour des raisons de de compréhensibilité et de maintenance).
Le caractère opportuniste de l'activité observée semble également typique d'une activité de conception réelle.

Mots-clés. Conception de programme; Conception logicielle; Stratégies de programmation; Stratégies de conception; Organisation opportuniste; Observation en temps réel; Etude de terrain; Analyse de protocoles; Automate programmable; Installation automatisée de machine-outil

INTRODUCTION

Most studies on computer programming to date involve an activity:
- performed by more or less experienced novices with little or, more commonly, no professional experience, most often students involved in a learning rather than a work situation;
- starting with a rather simplified, ad hoc problem;
- observed in an experimental setting;
- involving a program from some ten to a few hundred lines at most;
- using a more or less "classical" programming language such as FORTRAN or PASCAL (see 17, 16).

Specifically, the task and subject characteristics in these studies (novices and/or students working on artificial, limited problems) probably place severe limitations on the validity of the results for "real" programming activity and therefore on their usefulness for the construction of programming aids.

There are some notable exceptions to this characterization (7). Hoc (13) has researched programming by subjects working on quite realistic problems. Valentin (19) studied professional programmers working on management problems in a real work context.

The study we present in this paper deals with:
- an experienced professional programmer,
- solving a real, complex industrial problem (control of a machine tool installation),
- observed during his daily activities, in his real working environment, that is, depending, for example, on information from colleagues (specifications and other information),
- constructing a program of about 1200 instructions (which, due to the type of programming language used, represent many more "lines" (see below Programmable Controller & Programming Language Used)),

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

- using a kind of programming language that has not yet been studied (a declarative boolean language designed for programmable controllers).

Since our hypothesis was that the aspect of programming behavior most likely to differ in a fairly large, workrelated project was the strategies used, we chose to present these strategies in this first paper on our study.

METHOD

An Observational Study

We conducted full time observations on a professional programmer in a machine tool factory for a period of four weeks. We observed his normal daily activities without intervening in any way, other than to ask him to verbalize as much as possible his thoughts about what he was doing (10, 15).

We collected notes concerning:
- all the programmer's comments and writings;
- the order in which he produced the different documents, and how he gradually built them up;
- the changes he made;
- the information sources consulted;
- events we judged to be indicators of the subject meeting with difficulties.

In addition, we collected all documents produced by the programmer during his work:
- the diagrams and schemas he constructed for himself during analysis and problem solving;
- the different versions of these documents and of the program;
- the rough drafts of (parts of) them.

The Observed Programmer

The observed programmer received two years specialized university training in electronics.

At the time of observation, he had been working for four years as a professional programmer with programmable controllers and the mechanical installations they control.

The Observed Task

The programmer was to construct a program for a programmable controller controlling an automatic machine tool installation, based on the specifications given.

He wrote this program in three main stages. We observed the entire first stage, during which he constructed the first part of the program, which was to control the installation.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Programmable Controller & Programmming Language Used

A programmable controller is a computer specialized in controlling industrial processes, in this case a machine tool installation. Greatly simplified, it receives input on the state of the process, mostly from detectors on the installation it controls; the processing of this information (together with internal information) produces output, that is, commands governing installation functions, e.g., shaping operations.

The language used for programming the programmable controller is a declarative boolean language. The most frequently used instructions, called "sequences," control the installation's sequential functioning. They assign logical values to variables.

The logical value results from the evaluation of a logical expression consisting of a boolean combination of bits. The variable which holds the resulting value corresponds to (a) a programmable controller output, that is, a function or a display on the controlled installation or (b) an intermediate variable (see below).

The general semantics of a sequence is the following:
IF the conditions are satisfied
THEN set the result variable to 1
ELSE set the result variable to 0

The following is an example of such an instruction:
(E234 + /E56) B35 = A67
That is, bit A67 will be assigned the value resulting from the evaluation of the following logical expression:

((the value of bit E234)
OR NOT (the value of bit E56))
AND the value of bit B35

The E bits correspond to the programmable controller's physical inputs and the A bits to its physical outputs. They constitute its basic elements of information. The B bits are derived. They correspond to intermediate variables, that is, combinations of Es and/or As (and/or other intermediate variables).

In the instruction presented above, the bit A67 corresponds, e.g., to the jack for the Advance movement of a cutting tool; E56 to the component detecting the end of this Advance movement; E234 to the detector of the end of its preceding operation; B35 to a combination of conditons that must been satisfied for the operation to occur in mechanical safety.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

The program, composed of a series of instructions, is continuously read ("scanned") by the programmable controller. At the end of each scan (< 50 ms), all the E bits and physical output are updated. In other words, the electrical state of each physical input is translated into a logical value for the corresponding E bit, and the logical value of each A bit is translated into an electrical state of the corresponding physical output. Thus, each instruction assigning a value to an A leads, after a maximum of 50 ms, to the activation or inhibition of an operation or display on the installation, according to the logical state of A.

The language can be written in two different ways: in boolean expressions or in relay diagram equations. Programs or program instructions written in one format can be read in the other on the screen of the programming terminal.

Figure 1 presents the relay diagram equation for the example written above in boolean expression.

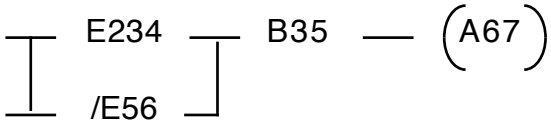

Figure 1. Relay diagram equation for the boolean expression
(E234 + /E56) B35 = A67

Parallel instructions or groups of instructions, such as E234 and /E56 here, are said to constitute different instruction "branches." The example we have given shows a very simple kind of instruction. Most instructions are composed of different branches, each one containing from about three to eight bits (see Figure 2).

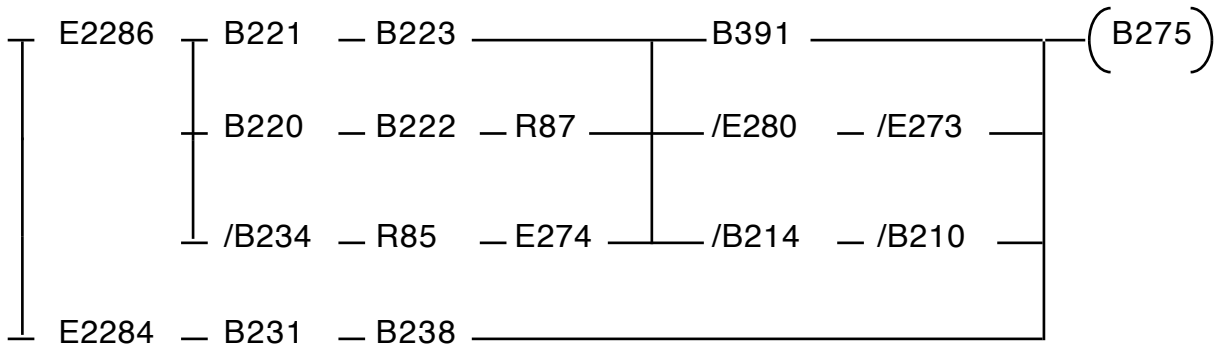

Figure 2. An example of an instruction from the studied program[1]

[1] R corresponds to a bit similar to B.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Comparison of Programmable Controller and "Classical" Programming

Most studies on "classical" programming deal with numerical computations or symbolical data processing resolved in procedural languages (such as Pascal).

The programming of controllers mainly involves defining the activation of physical actions on a controlled process according to the state of this process.

The activation of an action is not defined in a procedural but in a declarative way as a boolean combination of its conditions.

The syntax of the language permits each action to have only one definition in the program, that is, each action may figure only once as the result of a sequence. Most actions however may take place in several states of the process. So this programming amounts to building, for each action, a combination of the different configurations of conditions under which it may occur.

As different actions may share conditions, groups of conditions are gathered into intermediate variables. Many of these variables are themselves combined into other, higher level, intermediate variables (see the example given in figure 2). So, rather than expressing the conditions for an action to occur in terms of (the E bits corresponding to) physical conditions, the programmer combines higher level units whose correspondence to the physical state of the installation is not immediate.

Physical Layout of the Installation Controlled by the Program Written by the Observed Programmer

The installation is composed of four "stations." A central turntable rotates to bring and position the pieces in front of each of them. The pieces are conveyed onto and from the loading-unloading station by a system of conveyors (see Figure 3).

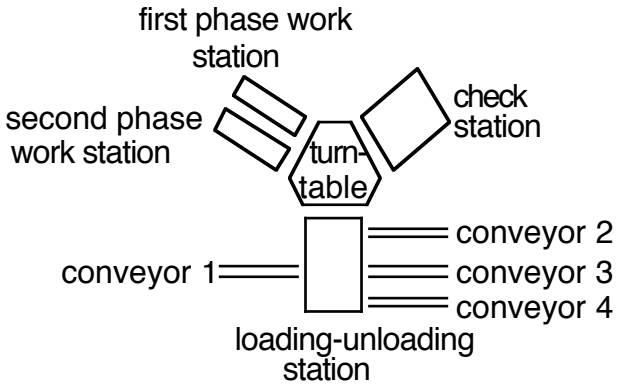

Figure 3. Spatial organization of the machine tool installation

For reasons not relevant here, the second phase work station precedes the first phase work station in the rotational direction of the turntable.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

RESULTS

Before presenting the results of the different strategies used by the programmer, we will (a) describe briefly the main specifications documents, so that we can refer to them later, and (b) present some methodological results concerning the verbal protocol method we employed.

Specifications

The programmer received two kinds of specifications:

A program specifying the structure of the program to be written. This "example program" had previously been written by the programmer himself to control an analogous installation. The programmer's manager judged some specific points of this program's structure to be "good" from the point of view of maintenance, and the programmer was to adopt these features.

Five documents specifying the operation and the physical layout of the installation. We will present the three that were most frequently used by the programmer.

The functional representation of the installation (the "Grafcet"). The Grafcet is a process control specification tool. It produces a graphical representation of the sequential progress of the process, showing the alternation of its actions and their enabling conditions. For each function, the graphical illustration is accompanied by documents providing a written description of (a) the safety conditions for each action; (b) the physical starting point of the cycle; (c) the procedure to be followed in the event of a mechanical or process control problem (14).

The physical representation of the installation. The installation and each of its functions are represented in a figurative, schematic form. Each different data and control element is labelled with its mnemonic (e.g., CAOU, "Contrôle Avance Outil," i.e., "Check Advance Tool," next to the electrical detector providing this information).

A document showing programmable controller input and output (the "I/O document"). All electrical data sources on the installation provide input to the programmable controller; the programmable controller's output governs the controlled functions on the installation and its display panels. The I/O document provides the corresponding programmable controller variable for each physical entity in the installation.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Verbalization Problems

When we asked him to verbalize his thoughts before starting our observation, the programmer was quite willing. However, though we repeated our request during observation, the programmer verbalized rather little while writing the program.

In general, as long as he was performing an overt activity (mainly writing), we noticed that he found it very difficult to "think aloud." When we insisted during these periods, his verbalizations consisted of stating (the names of) the elements with which he was currently working: "B67" (Intermediate Variable Bit 67), "Not-E34" (Input Bit 34, connected with the logical operator NOT), or, more rarely, the mnemonics corresponding to the current bits: "AAV" ("Auxiliaire Avance," i.e., "Advance Auxiliary").

In general, he only verbalized his thoughts when he was faced with a problem, when he was not sure what to do, or when he had noticed an error or an omission.

Our hypothesis is that many of the programmer's actions during program construction are automatized (as a consequence of his experience in the field) and that a verbalization other than stating the variables processed would require a "decompilation" of these automatized procedures (2). Encouraging the programmer to verbalize more might lead him to make the knowledge sources underlying these procedures explicit, but such verbalization would not express the real activity the programmer performs in writing his program.

Global Strategy: Organization of the Activity

We have broken the programmer's activities down into three consecutive stages. They are not quantitatively comparable, since the first and the second stages took relatively little time. However, they differ qualitatively, having rather different functions in the total activity.

Studying the specifications (one day). The programmer skims rather rapidly through the documents that give the specifications for the installation. He studies the three documents presented above in some detail, especially the graphical portion of the Grafcet.

Program planning (one hour). The programmer plans his writing along two lines:

- Global breakdown of the task, according to the relative urgency of different parts of the program. While the programmer is writing his program, the installation is being built and tested in the manufacturer's workshop. In order to test installation operation, they need the portion of the program controlling the installation process first. The programmer plans to start with this part. He will then write the part controlling the supply of pieces. Last, he will write the user help portion of the program (a screen with messages about functions that should have been carried out, tools to be changed, etc.).

- Breakdown of the program into programming modules. Referring back to the "example program," the programmer plans the modules for the target program. Only if an example installation

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

function exists in the target installation, its title is copied from the example program. At this stage, only functions existing in the example are planned (although others will appear later during writing).

Program writing. It took the programmer four weeks to write the part of the program controlling the machine tool installation. It contained about 700 instructions (the final program was some 1200 instructions long in all).

Although this third and last stage took a considerable amount of time compared to the others, no qualitatively different stage or sub-stage can be discerned. Program writing was interrupted for (a) functional analysis and problem solving and (b) program checking, but these interruptions were not systematic.

Programming activity organization can be compared to that of the writing (i.e., text composing) activity, as analyzed and modeled by Hayes and Flower (11).

The remainder of this paper presents the strategies used mainly in the last stage.

General Program Construction Strategies

Use of different information sources. The "example program," previously written modules of the target program, and the I/O document were the programmer's most frequently used information sources. A second example program was used for the construction of one target module carrying out a function which had more resemblance with a function on the installation for which this example program was written. The importance of existing modules for the construction of a target program was also shown by the Valentin study (see above) on programmers in a real work situation.

The Grafcet and the physical representation of the installation were seldom used after the first stage of the specifications study.

Inferring installation operation from a representation of its components. Rather than consulting the Grafcet, i.e., the functional representation of the installation, to find out its operation, the programmer often infers it from

- the I/O document

or, to a lesser extent, from

- the physical representation of the installation.

Examples of inference errors. This procedure sometimes leads him to make erroneous inferences.

Example 1. Attributing an erroneous role to a component. The programmer attributes sometimes an erroneous role to components whose correct role is not understood.

Example 2. Attributing an erroneous operation to the installation. At times a function whose components may perform different operations is programmed in a way that does not meet the

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

specifications. For example, on a machine tool, the grip function can be implemented on each separate station or on the installation as a whole. On the current installation, there is only one grip function for both work stations. However, the programmer programs two grip modules, one for each station.

Example 3. Omitting components erroneously judged to be superfluous. When the programmer comes across two series of components that he thinks serve identical functions, he judges one of them to be superfluous and omits it. By doing so, he changes the specified operation.

The example program and the previously written modules played an important role in several other strategies described below.

Following the order and the structure of the example program. To construct the target program, the programmer follows the order of the example program.

The example program has three main parts:
- about ten modules involve General Information retrieval and Command execution (see below);
- about six modules involve each of the installation stations individually, as well as some other functions;
- the last module involves the display.

The first part, the so-called "General Information," includes information concerning the whole installation, rather than just one of its specific functions. It contains the "General" Safety Conditions, Starting Conditions, Commands, and Checks.

An example of a "General" Check is KRUN ("Contrôle Retour Unités," i.e., "Units Return Check"). This variable accumulates individual $KRUN_X$ variables containing information about the position of each of the installation units, i.e., stations. For KRUN to be set to 1, $KRUN_X$ on each station must have been set to 1. Then the central turntable can rotate safely, without damaging the station tools.

The first "General" Command in the program is AATCY ("Auxiliaire Arrêt Cycle," i.e., "Cycle Stop Auxiliary"). This variable is made up of seven parallel branches. If any one of them is set to 1, installation operation stops immediately. The variables composing the branches in question will be defined only later in the program.

Following the order of the example program leads the programmer to adopt a modular top-down strategy at the level of the overall program structure. However, within the modules, he abandons this top-down strategy. At these lower levels, to follow the order of the example program involves adopting a rather bottom-up strategy.

This characterization of the programmer's activity in terms of top-down and bottom-up strategies is only applicable at a very general description level. We noted a great number of deviations from these general lines, some of which will be presented below (see also 20).

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Reasoning based on semantic or functional relationships between components. In order to write his program, the programmer makes use of relationships between operations or functions previously written and those that are to be written. The elements involved in such a relationship may be

- (a module of) the example program and (one from) the target program;
- a previously written module of the target program and a module to be written.

Occasionally, this relationship is one of opposition. Very often, the programmer uses a relationship of analogy he feels exists between programs or modules.

Example 1. Opposite functions on the target installation. To write the instructions for the Return of a movement, the programmer often adapts the movement's corresponding Advance instructions (or vice versa).

Example 2. Analogy between the structures of two programs. As we have already noted, the programmer follows the structure of the example program in establishing that of the target program.

Example 3. Analogy between functions of parts of the example and target programs. To determine whether there is an analogous example module for a particular target module, the programmer first decides whether the target installation function that the target module must address sufficiently resembles to an example installation function for a transformation to be worthwhile.

This leads us to distinguish three types of modules in the target program:

- those for which the programmer follows the example module structure down to the instruction component level and for which almost all changes made are at the level of bit and variable numbering (e.g., the "General Information" modules);
- those for which he follows the example module structure down to the level of the instruction's output variables and makes changes in the instruction components, in addition to those mentioned above (e.g., the work and check stations);
- those for which the example program does not offer an analogy and for which he will have to construct the module entirely from the specifications and his knowledge (e.g., the loading-unloading station).

Example 4. Analogy between functions of two parts of the installation. The installation has two work stations, First Phase and Second Phase. Having started with the module of the Second Phase station, the programmer refers back to this module in constructing the First Phase module.

Also, by doing so, he discovers errors in the Second Phase module.

Top-down processing. We have already noted the importance of this strategy. Starting program writing with the general portion and taking advantage of analogical or other relationships between installation operations or local functions were two instances of this strategy. Both require the programmer to use a top-down strategy to analyze the problem of controlling installation operation.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Creation of intermediate variables. By writing "General Information" before the specific parts, the programmer creates and handles variables in which data he has not yet defined are accumulated. Writing, e.g., the instruction defining KRUN before defining the individual $KRUN_X$ variables (see above) requires the programmer to know about the individual functions involved in the installation's general operation.

The programmer is able to adopt this strategy because he has an example program and, perhaps more importantly, because of his expertise in the field of machine tools and machine tool programming.

The use of intermediate variables in which more elementary information is accumulated before it has been processed is frequent.

There are several reasons for this:

- The intermediate variable is used to accumulate pieces of information which are often used together in that specific combination (e.g., the individual station KRUNs we mentioned above, which are made up of several input bits providing information about positions of the stations in question, are each used in about ten instructions).
- The programmer may judge the combination meaningful for him and, he thinks, for future workers on the installation. For example, AAV and ART, the Auxiliaries for Advance and Return, are intermediate variables accumulating along several branches the conditions for the different operation modes (e.g., automatic and manual) of one or more station operations.
- The third reason is pragmatic and involves the capacity of the programming terminal screen. This screen can only display instructions composed of a limited number of bits connected in series or in parallel. Therefore, to keep within this limit, the programmer may have to break up an instruction and construct one or more intermediate variables.

Several reasons are often associated with the construction of an intermediate variable, even though one would be enough.

Economical use of resources. As we noted above, the programmer follows a writing plan inspired by the order of the modules and instructions in the example program. However, he will abandon this order if another is more economical from the point of view of processing available information sources or using his cognitive resources.

Postponing the processing of information not yet available. The programmer skips parts of modules when he needs information from colleagues or other information sources not at hand to construct them. He lists his questions and waits to ask them (or to look up the answers) until he has gathered enough to make it worthwhile to interrupt his writing.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

He also skips parts of instructions, often leaving blanks in particular places. This means that he knows the number and type of variables required, which he sometimes even writes above the blank in a comment. (This procedure also reveals top-down processing.)

Information processing guided by the consulted information source. The programmer uses a lot of information from the I/O document. Often, when he looks up which E or A bits correspond to a particular function he is programming, he also retrieves other Es or As mentioned that are close to the one he wants. Taking them into account generally requires interrupting the construction of the current instruction and starting another.

This retrieval of bits other than the one he is looking for can be due to a conscious decision or to a drifting of attention. This last phenomenon sometimes leads the programmer to discover forgotten or unknown variables which he then takes into account.

These and other phenomena contribute to characterizing the programming activity as being guided by circumstances (the information encountered, the processing currently judged to be the least costly, drifting of attention) rather than by a hierarchical plan. Elsewhere (20), we suggested accounting for it in terms of opportunistic planning (12).

Search for homogeneity. Mostly for comprehension and maintenance reasons, the programmer makes the program as homogeneous as possible. He tries to create uniform structures at several levels of the program:

- *Instruction order in the modules*.
  Examples. Process state check instructions precede operation control instructions in the station modules. Advance instructions precede return instructions.
- *Branch order in the instructions*. For instructions controlling operations taking place with different installation configurations, different instruction branches define the conditions for these different operational occurrences.
  Example. The branch defining the automatic mode of an operation precedes its manual mode definition branch.
- *Bit order in instruction branches*. Instructions are scanned continuously (see above Programmable Controller & Programming Language Used). Inspection of an instruction by the programmable controller supervisor is stopped as soon as a bit set to 0 is encountered in a serial connection branch. The programmer takes advantage of this feature of the programmable controller by putting the most "important" conditions at the beginning of this type of branch.
- *Intermediate variable numbering*. Numbering of intermediate variables with counterparts elsewhere in the program is structured.
  Examples. The $KRUN_X$ we described above has analogous numbers on the different stations: B601 on one work station, B701 on the other, and B801 on the check station.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

Some variables nearly always found in a machine tool program are the same from one program to another (in the factory where the programmer is working). Thus, B0, B1, and B2 have the same content in all these programs.

Program changes. The wish to homogenize sometimes leads the programmer to reconsider previousy written instructions. In doing so, he will either

-have the current instruction follow their structure;

or

- copy their structure from the current one.

Schema-guided information processing. Some errors made by the programmer can be explained by the hypothesis that he is using schemata with prototypical values instantiated for particular slots. If he reads the specifications providing other values for these slots, his expectations based on these prototypical values are probably so strong that he does not take the values which are given into account (see also 8). If he writes part of the program without reading the specifications, the schema he instantiates may provide him preferentially with these prototypical values.

Example. In machine tool installations, work operations generally take place during the "Advance" movement of the station. In the target installation however, there is one such operation during "Return." By defining the occurrence of this operation under the same conditions as the others, the programmer modifies the specifications by creating a process in which the operation takes place during "Advance" instead of during "Return" (This error is subsequently corrected by the tester we observed afterwards).

Simulation. Two types of simulation have been observed: simulation of program operation and simulation of installation operation.

The first is sometimes used by the programmer to verify (modules of) the part of the program he has already written. These simulations were among the rare moments the programmer verbalized his thoughts spontaneously. For the most part, he followed one of two basic procedures:

- He attributed hypothetical values to the bits in the logical expressions of the instructions and evaluated the consequences for the output variables.

Verbalization example: "So, ... if E13 is set to 1 AND NOT-B67 to 1, yes, alright, then A356 fires."

- He evaluated which conditions had to be satisfied for an output variable to be activated.

Verbalization example: "So, for A526 to fire, B80 and AAV have to be set to 1."

He reasoned in terms of numbered bits or, more rarely, mnemonics, as he did when we urged him to verbalize during writing. In other words, for the programmer, there is a direct correspondence

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

between the variable names and variable content. This is probably due in part to the use of numbering rules created by the programmer (see Search for homogeneity).

Simulation of installation operation was mainly used by the programmer when studying the specifications, specifically in order to understand the Grafcet (i.e., the functional representation of the installation).

Jumping ahead of his plan. The programmer sometimes jumps ahead of his program construction plan (i.e., writing modules in the order of the example program). He interrupts the writing of the current instruction and deals with functions he thinks of and is afraid he will forget if he does not handle immediately.

He also thinks ahead to the possible problems the person who will transcribe his program may encounter (see the next paragraph).

Writing style. The programmer writes a paper and pencil version of the program. Another person will enter this program on the programming terminal from where it can be transmitted to the programmable controller.

The programmer writes his program in relay diagrams, judging it easier to read and to evaluate the functioning of the program in this form, especially in the case of parallel branches.

The person entering the program on the terminal uses the boolean format. Although the program can be entered under both formalisms, the relay diagrams require much more typing. Transcribing the paper version of the program from relay to boolean is apparently worth the effort it requires.

The programmer who uses relay diagrams for writing his program creates a mixture of relay and boolean format for some types of instructions. This takes place in instructions in which the elements are arranged in a different order depending on the format (e.g., the "counting instruction"). In this case, the programmer writes down the elements in relay format, but in the order in which they have to be entered in boolean, so that the person entering them at the terminal can follow this order (see Figure 4).

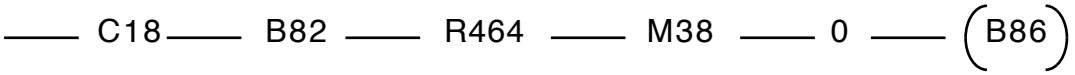

Figure 4. Example of the format used by the programmer to write a counting instruction

The following is the correct relay diagram format representation of this instruction, resulting from entering the elements in the above order (see Figure 5):

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

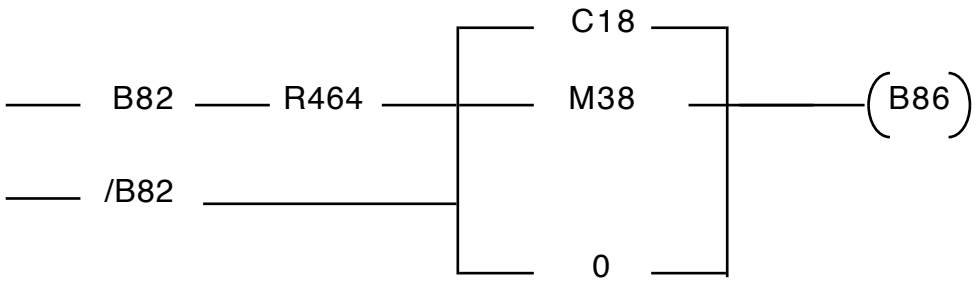

Figure 5. Example of the relay diagram representation for the counting instruction shown above (see Figure 4)

The correct boolean format representation of this instruction is:
C18 . B82 . R464 . M38 . 0 = B86

Thus, the programmer's approach could be interpreted as not only preparing the work of his colleague, but also anticipating the problems this person may encounter.

CONCLUSION

In our introduction, we put forward the hypothesis that the strategies used in "programming-in-the-large" would be different from those that have been observed to date in empirical programming studies which have tended to focus on student programmers working on problems of a somewhat limited nature. Among the strategies we observed, some were the same as those already shown to be at work in other contexts (planning, top-down and bottom-up processing, simulation of the program's execution, schema-guided information processing). This result is encouraging as it may well be hoped that the research conducted on "programming-in-the-small" provides knowledge which is applicable to programming as a whole (16).

However, we did find some hitherto unobserved strategies. Thus, the frequent use of examples and analogy, the search for homogeneity, and an opportunistic organization of the activity.

The frequent use of example programs seems to be characteristic for programming in a work context. The importance of "examples" is not particular to programming. We observed the mechanic who made the specifications for the programmer, drawing the functional representation of the installation. He also used example functional representations (20).

Computer scientists observe from introspection that they seldom create programs ex nihilo and often make use of analogy to debug incorrect programs, modify existing ones, and abstract programming schemata. They even propose to implement analogy as a tool in automatic programming systems (9).

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

The desire to ensure homogeneity which strongly guided the programmer's activity stemmed from a preoccupation with the future use of the program, and thus, by definition, from a source specific to programming in a real work context.

Another non-trivial result, presented elsewhere in greater detail (20), is the opportunistic nature of the programming activity. This also seems to be a characteristic of real activity as opposed to the hierarchical plan-guided activity mostly observed in problem solving involving artificial, limited problem statements.

The question remains as to whether it is possible to use these results to make generalizations to other programmers and/or programming other problems in other languages?

As with all case-studies conducted on one or two subjects (1, 3, 4), the one presented in this article can of course only lead to a hypothetical model requiring further empirical research in order to test it. But, as Anzai and Simon (3) claim, "[If] one swallow does not make a summer, … one swallow does prove the existence of swallows. And careful dissection of even one swallow may provide a great deal of reliable information about swallow anatomy" (3, p. 136).

If the possibility of applying our conclusions to other programming contexts is determined by the similarity of languages used and problems resolved, then little case can be made for it. We think however that, to a great extent, the observed strategies do not depend on language or problem characteristics. On this point, we only make a special case for the top-down processing as it was observed, e.g., in the writing of the general portion before the rest of the program and in the construction of intermediate variables before the definition of their components. The possibility of using this strategy in the way the programmer did surely depends, leaving aside the programmer's programming knowledge and experience in the field, on the type of programming language used. This strategy is especially appropriate with languages developed for structured programming.

Two sorts of arguments plead in favor of the plausibility of our conclusions concerning the other strategies being applicable to other programming, or even other problem solving, activities.

Firstly, the duration of our observations allowed us to observe some strategies that were recurrent and general, that is not triggered in only one limited context: the use of examples (programs or parts of programs) as well as analogical reasoning. If the specific examples observed here were particular to machine tools and their control programs, there is nothing in the nature of these strategies that makes them unique to this kind of programming. We have already mentioned the frequent use of examples by professional programmers working in a completely other problem domain, i.e., management (19). Moreover, these programmers used classical, procedural languages (GAP and COBOL).

Secondly, we may support our argument with concurrent results obtained in research on other types of problem solving. In experimental psychological studies, analogical reasoning has been considered, for some ten years (18), as an important mechanism in problem solving (6). In a study conducted on a mechanic making specifications, we observed similar strategies to those used by the

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

programmer (20). Other researchers working on design activity noted that an incremental approach is typical: a designer rarely starts from scratch (5, 11). The opportunistic organization of a problem solving activity has been observed in studies on planification (12).

We are therefore reasonably confident that our conclusions are not simply restricted to programming programmable controllers, even though of course they need other empirical confirmation.

In conclusion, we hope that our results encourage more research on "programming-in-the-large" conducted on professional programmers working on complex problems in a real work context.

REFERENCES

1. Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. IEEE Transactions on Software Engineering, SE-11, 1351-1360.
2. Anderson, J. R. (1986). Knowledge compilation: the general learning mechanism. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), Machine learning. An artificial intelligence approach (Vol. II). Los Altos, Calif.: Morgan Kaufmann Publishers.
3. Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. Psychological Review, 86, 124-140.
4. Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9, 737-751.
5. Carroll, J. M. & Rosson, M. B. (1985). Usability specifications as a tool in iterative development. In H. Rex Hartson (Ed.), Advances in human-computer interaction (Vol. 1). Norwood, N.J.: Ablex.
6. Cauzinille-Marmèche, E., Mathieu, J., & Weil-Barais, A. (1985). Raisonnement analogique et résolution de problèmes. L'Année Psychologique, 85, 49-72.
7. Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers. Papers presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, D.C.. Norwood, N.J.: Ablex.
8. Détienne, F. (1986). Program understanding and knowledge organization: the influence of acquired schemata. Preprints of the Proceedings of the Third European Conference on Cognitive Ergonomics, Paris, France, September 15-19, 1986. Rocquencourt: INRIA.
9. Dershowitz, N. (1986). Programming by analogy. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), Machine learning. An artificial intelligence approach (Vol. II). Los Altos, Calif.: Morgan Kaufmann Publishers.
10. Ericsson, K. A., & Simon, H. A. (1984). Protocol analysis. Verbal reports as data. Cambridge, Mass.: MIT Press.

This text is a preprint of Visser, W. Strategies in programming programmable controllers: a field study on a professional programmer. In G. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, N.J.: Ablex, 1987.

11. Hayes, J. R., & Flower, L. S. (1980). Identifying the organization of writing processes. In L. W. Gregg & E. R. Steinberg (Eds.), Cognitive processes in writing. Hillsdale, N.J.: Erlbaum.
12. Hayes-Roth, B., & Hayes-Roth, F. (1979). A cognitive model of planning. Cognitive Science, 3, 275-310.
13. Hoc, J. M. (in preparation). Towards effective computer aids to planning in computer programming. In T. R. G. Green, J. M. Hoc, D. Murray, & G. C. van der Veer (Eds.), Working with computers: theory versus outcomes. London: Academic Press.
14. Morais, A. (1985). Hierarchical planification in programming. In I. D. Brown, R. Goldsmith, K. Coombes & M. A. Sinclair (Eds.), Ergonomics International 85. Proceedings of the Ninth Congress of the International Ergonomics Association, Bournemouth, England, 2-6 September 1985. London: Taylor & Francis.
15. Newell, A., & Simon, H. A. (1972). Human problem solving. Englewood Cliffs, N.J.: Prentice-Hall.
16. Soloway, E. (1986). What to do next: meeting the challenge of programming-in-the-large. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers. Papers presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, D.C.. Norwood, N.J.: Ablex.
17. Soloway, E. & Iyengar, S. (Eds.) (1986). Empirical studies of programmers. Papers presented at the First Workshop on Empirical Studies of Programmers, June 5-6, 1986, Washington, D.C.. Norwood, N.J.: Ablex.
18. Sternberg, R. J. (1977). Intelligence, information processing, and analogical reasoning. Hillsdale, N.J.: Erlbaum.
19. Valentin, A. (1987). Etude ergonomique exploratoire du poste de travail de l'atelier logiciel Concerto. Observations d'analystes-programmeurs en situations réelles de travail "classiques" (hors atelier) (Rapport de fin de contrat EPHE-CNET). Paris: Laboratoire de Psychologie du Travail de l'EPHE.
20. Visser, W. (1987). Abandon d'un plan hiérarchique dans une activité de conception Giving up a hierarchical plan in a design activity. Actes du colloque scientifique COGNITIVA 87 (Tome 1). Paris: Cesta.