Toward Authentic Measures of Program Comprehension

Judith Good* and Paul Brna**

* Organizational Learning and Instructional Technologies,
College of Education, University of New Mexico,
106 Education Office Building,
Albuquerque, NM 87131-1231

** School of Informatics
Pandon Building
Northumbria University
Newcastle upon Tyne NE1 8ST, England

This paper describes an analysis scheme which was developed to probe the comprehension of computer programming languages by students learning to program. The scheme operates on free-form program summaries, i.e. textual descriptions of a program which are produced in response to minimal instructions by the researcher/experimenter. The scheme has been applied to descriptions of programs written in various languages, and it is felt that the scheme has the potential to be applied to languages of markedly different types (e.g. procedural, object-oriented, event-driven). The paper first discusses the basis for the scheme, before describing the scheme in detail. It then presents examples of the scheme's application, and concludes with a discussion of some open issues.

1 Introduction

The comprehensibility of programming languages is a topic of interest in an era in which much effort is being expended to open up programming to an increasingly wider audience. Although much work has already been carried out in the area of program

comprehension (see Upchurch (2002) for an extensive bibliography), it is a complex topic which deserves further study.

One of the criticisms levelled at program comprehension studies, and indeed at many studies of programming in general, is that they do not capture the richness of the activities which occur in a naturalistic setting. Studies of programmers are often limited to experiments involving small numbers of undergraduate students individually studying short programs and answering multiple choice questions during a one-hour period. This is far removed from industrial settings where groups of programmers work for several months or more on programs which are thousands of lines long and which may have originally come from a number of sources.

At the same time, it is useful to distinguish between *learning to program* and *exercising the skill of programming*. The former case is of interest to many researchers because of the steep learning curve typically associated with the process, and because of a desire to address the issue of the bimodal distribution of scores which often occurs in programming courses .

In the case of novices learning to program, more naturalistic research does not necessarily imply longitudinal, large-scale group studies involving complex programs, as they may not be representative of novice programming tasks. On the other hand, it is difficult to envisage program comprehension activities which are naturalistic for novices, since most teaching of computer science is low on tasks that support learning to understand programs as their primary goal. Nonetheless, many of the tasks set for novice programmers do require them to understand chunks of code, and we would argue that by opting for more open-ended methods of measuring skills, and finer grained analyses of data, we can develop a deeper understanding of the processes involved in novice program comprehension.

We have been working on more authentic ways to measure program comprehension. One of the ways we are investigating this is by looking at how people relate their own understanding of a program when given few cues. Other researchers are also tackling this issue: for example, von Mayrhauser and Lang (1999) have developed a scheme for analysing verbal protocols of the actions which a programmer makes during a software maintenance task such as debugging. Similarly, O'Brien et al. (2001) have

focused on the processes involved in program comprehension, using verbal protocol analysis to investigate the use of different comprehension strategies.

In this paper, we present a scheme for measuring program comprehension which involves coding free-form program summary statements. Rather than examining the processes involved in program comprehension (the goal of the research described in the preceding paragraph), this scheme focuses on the structure of the information artefacts which are produced following a program comprehension phase. In a study of this type, participants are asked to examine programs, and to write a summary describing what the program does in their own words. We then code the statements in the summary according to the *types* of information which they represent. We feel that the scheme offers a promising way of investigating program comprehension in-depth and that, combined with quantitative measures, can give us more insight into the types of information that people glean from a program.

The rest of the paper is structured as follows: Section 2 discusses some of the issues involved in analysing program summaries. Section 3 describes the background to the program summary analysis scheme, and the origins of the idea of information types in programs. Section 4 describes the scheme itself, while Section 5 presents some examples of applying the scheme to different types of programming languages. Finally, Section 6 discusses open questions and ideas for further work.

2 Program Summaries: Issues of Analysis

Program summaries have played an important role in the information types methodology, and data of this type has been collected by Pennington, (1987a), Corritore and Wiedenbeck (1991) and Good (1999).

A program summary is a free-form account of a program which an individual produces after studying the program. Instructions given to participants have tended to be relatively non-directive, leaving the content essentially up to them. The lack of explicit guidelines for the content of the summary allows for wide scope and variation in the responses.

By the same token, the open-endedness of the task provides a valuable source of rich, realistic data. Furthermore, the program summary methodology neatly circumvents

the problems of 'false positive' results often associated with binary choice questions, and the difficulties associated with developing sensitive and reliable multiple choice questions and corresponding distracter items. Program summaries allow participants to express their view of a program, using their own words, at their chosen level of abstraction, including as much (or as little) detail as they feel is necessary.

As always, the price to pay for rich data is the difficulty of analysing it: quantitative statistics are not always appropriate, and qualitative methods must be devised. There are numerous ways of analysing written texts of this type, however, it is not simply a case of identifying the 'correct' method in the same way one chooses the right statistical test, particularly when the semantic content of the text is of interest. Analysis schemes are rarely universal, given differences in research aims between studies. They are both content and context sensitive, and must be developed through what is often a lengthy, iterative process.

The complexity of the analysis is related to the issue of *replicability* . Rich, complex data may lead to complex analysis schemes: extra care must be taken to ensure that these schemes are in fact understandable and usable, and are no more complex than is necessary for the purpose of the analysis. Replicability can also be compromised by schemes which are ill-defined. Schemes which are not fully worked out and/or which are not accompanied by explicit instructions enabling them to be used by persons other than the original researcher are not of much use: it is impossible to compare results reliably.

In designing a scheme to analyse program summaries, it is necessary to ensure that it is both complete, in the sense that all of the statements in the summary can be classified in some way, and that it contains a number of distinct categories, so that it allows for the detection of different patterns of statements across program summaries. In reviewing the literature on program comprehension, Pennington's description of *information types* was felt to provide a useful starting point for developing the scheme. Information types are described in more detail in the following section.

3 Background to the Scheme

3.1 Information Types

Information types have been defined as "different kinds of information explicit 'in the text' that must be detected in order to fully understand the program" (Green *et al*., 1980; Green, 1980) in (Pennington, 1987a, p. 299). The concept of information types has been used extensively by Pennington (1987a, 1987b), in studies by Wiedenbeck, Corritore colleagues (see, for example, (Corritore and Wiedenbeck, 1991; Ramalingam and Wiedenbeck, 1997)) and more recently, by Romero *et al.* (2002).

Pennington described information types in terms of internal, rather than external, abstractions of a computer program. They are not meant to be mental representations of the program, but are "based on formal analyses of programs developed by computer scientists" (Pennington, 1987a, p. 298), and can be compared with the abstractions which are made with respect to natural language, such as referential or causal abstractions.

Pennington identified five types of information: *function*, *control flow*, *data flow*, *state* and *operations*, defined as follows:

Function: information about the overall goal of the program, essentially, "What is the purpose of the program? What does the program *do*?". Since function also includes program subgoals, goals and subgoals can be represented in a goal hierarchy. Some information about the order of events can be inferred, but not details of how the events are implemented.

Control flow: information about the temporal sequence of events occurring in the program, *e.g.* "What happens after X occurs? What has occurred just before X?" If the information is represented graphically, then the links will correspond to the direction of control, rather than to the movement of data. Data flow information can be inferred in a program by searching for repeated occurrences of data objects, but goal/subgoal information is harder to detect.

Data flow: essentially concerned with the transformations which data objects undergo during execution, including data dependencies and data structure information, *e.g.* "Does variable X contribute to the final value of Y?" Data flow and function are linked in the sense that function information can be partially reconstructed from a data flow abstraction. Similarly, control flow information can also be more readily inferred from the data flow abstraction than from the function abstraction.

Operations: information about specific actions which take place in the code, generally corresponding to a single line of code or less, such as "does a variable become instantiated with a particular value?" Although Pennington doesn't describe these in great detail, operations seem most related to control flow information, in the sense that describing the control flow of a program would lead to "stringing together" a series of operations.

State: time-slice descriptions of the state of objects and events in the program, *e.g.* "When the program is in state X, is event Y taking place, or has object Z been created/modified?" This abstraction is quite distinct in the sense that other types of information are hard to infer from it, and vice versa.

The categories are orthogonal in terms of information coverage. Although Pennington doesn't address the issue of granularity explicitly, the categories vary: *function* can often cover the entire program, while *operations* will concern only a single line (or node, in the case of a visual programming language).

Pennington was interested not so much in information types per se, but in the relationship between information types and what she called *programming knowledge structures*. She looked at two competing knowledge structures: *text structure* knowledge, which is organised around control structure primes, and *plan* knowledge, which, according to her, is primarily functionally oriented. Pennington carried out two experiments to investigate these ideas, one of which involved the analysis of participants' written summaries of the programs they were asked to examine, as described in the next section.

3.2 Pennington's Methodology for Program Summary Analysis

Analysing program summaries as a way of measuring program comprehension can be traced to an experiment carried out by Pennington (1987a). In addition to answering binary choice questions about a program of moderate length, participants were also asked to write a summary of the program at two points during the experiment: firstly after a 45 minute study period, and again after having carried out a modification to the program. Although the exact wording of the request is not given, it is likely that the instructions were brief and non-directive with respect to the type of information the summary should contain.

Pennington performed two analyses on the program summaries, classifying each statement by both *information type* and *level of detail*. The methods used in the analysis are described in the following two sections.

3.2.1 Information Type Analysis

Pennington states that the information types investigated *included* procedural, data flow, and function statements. The other categories used in the program comprehension tests, namely operations and state, do not seem to have been used: no results were reported for them in any case. Why they were omitted from the analysis is not discussed. Pennington's definition of each category is very brief, and expressed primarily through examples. She defines the three categories as follows:

- "**procedural statements** include statements of process, ordering, and conditional program actions." (Pennington, 1987a, p. 332);
- "**data flow statements** also include statements about data structures" (Pennington, 1987a, p. 332);
- **functional statements** are not defined by Pennington, but illustrated with an example.

The following summary excerpts are provided to illustrate each type of statement, all from (Pennington, 1987a, p. 332):

Procedural: “after this, the program will read in the cable file, comparing against the previous point of cable file, then on equal condition compares against the internal table ... if found, will read the tray-area-point file for matching point-area. In this read if found, will create a type-point-index record. If not found, will read another cable record.”

Data flow: “the tray-point file and the tray-area file are combined to create a tray-area-point file in Phase 1 of the program. Phase 2 tables information from the type-code file in working storage. The parameter file, cables file, and the tray-area-point file are then used to create a temporary-exceed-index file and a point-index file.”

Functional: “the program is computing area for cable accesses throughout a building. The amount of area per hold is first determined and then a table for cables and diameters is loaded. Next a cable file is read to accumulate the sum of the cables’ diameters going through each hole.”

3.2.2 Level of Detail Analysis

Pennington defined four levels of detail for program summaries:

- **detailed**: references to a program’s operations and variables;
- **program**: references to a program’s ``procedural blocks'';
- **domain**: references to real world objects;
- **vague**: statements with no specific referents.

Pennington uses the example summary segments above as illustrations of the level of detail: the procedural summary is the most *detailed*, the data flow summary is described at the *program* level, the functional summary is described at the *domain* level, and an example of a vague statement is, “this program reads and writes a lot of files.” (Pennington, 1987a, p. 333).

4 Description of the Program Summary Analysis Scheme

The brevity of the description of the summary analysis schemes in (Pennington, 1987a) meant that they were not possible to replicate. However, the schemes served as a useful basis for the development of new schemes, which are proposed below. The classification is similar to Pennington's in that it depends on two passes through the summaries: one based on *information types* and the other based on *object descriptions*.

The information types classification is a more finely-grained and fully specified refinement of Pennington's scheme, while the object classification is essentially a more restricted version of Pennington's levels of detail. It was decided to focus solely on data objects within the program, as describing program events in terms of level of detail was felt to entail an unwanted overlap with the information types classification. For example, if one is describing a program action, it is difficult to differentiate between describing that action in procedural terms (information types) and program terms (levels of description). Furthermore, given that the same data object can be described in very different ways (*e.g.* a basketball team, a list of heights, or a list of numbers), focusing on object descriptions provides much insight into how programmers choose to describe program objects.

4.1 Information Types Classification

The information types classification is used to code summary statements on the basis of the information types they contain. The categories which make up the classification are described below, followed by a short discussion of the relationships between categories, and the way in which they fit together to form a program summary.

4.1.1 Information Types Categories: Descriptions and Examples

The information types classification comprises eleven categories, described below with examples of each. In some cases, segments preceding or following the segment of interest have been included to provide context and aid understanding (shown in square brackets).

function: the overall aim of the program, described succinctly.

- The program is selecting all players over a certain height and allowing them to join the team.
- The program calculates the differences between the input distances…

actions: events occurring in the program which are described at a lower level than function (*i.e.* they refer to events within the program), but at a higher level than operations (described below). An action may involve a small group of nodes rather than one node only. Alternatively, it may be described as operating over a series of inputs, or describe actions in non-specific ways, *e.g.* describing tests in general, rather than the exact tests being carried out.

- This sub-program checks each individual element of this list...
- `Sun Span' is then worked out.
- The program makes two checks ...

operations: small-scale events which occur in the program, such as tests, assignment, etc. Operations usually correspond to one node in a VPL, or one line of textual code.

- ... then the program sets the height to head(height) ...
- … then it increments the counter by 1 ...
- A selector checks to see if the set is equal to [ ] ie 0 ...

state-high: a high-level definition of the notion of state. Describes the current state of a program when a condition has been met (and upon which an action is dependent). State-high differs from state-low in terms of granularity: the former describes an event at a more abstract level than the latter (which usually describes the direct result of a test on a single data object). The relationship between the two is akin to the relationship between actions and operations.

- Once all the elements have been processed…
- [The program continues] until there are no player left unchecked in the list

state-low: a lower-level version of state-high. State-low usually relates to a test condition being met, or not met, and upon which an operation depends.

- If the head is greater than 180 ...

- ... when the test is empty is true ...
- ... if empty distances (eg []) ...

data: inputs and outputs to programs, data flow through programs, and descriptions of data objects and data states.

- The program accepted a list of numbers indicating sunhours.
- ... it then passes a list of heights to a sub-program ...
- ... the heights over the height are sent to the team ...

control: information having to do with program control structures and with sequencing, *e.g.* recursion, calls to subprograms, stopping conditions.

- ... the nested recursions begin to unwind.
- It exits the program and goes back to the main program...

elaborate: further information about a process/event/data object which has already been described. This also includes examples.

- [If the current mark is above a certain pass level] (65 in this case)...
- [The head(numbers) is assigned to one variable] (which I'll call mark)...

meta: statements about the participant's own reasoning processes, *e.g.* "I'm not sure what this does".

- Dhoo! forgot where that route went!!!
- ... [and then joins it to the other value it would have created if it had done what i just said] (complicated).

unclear: statements which cannot be coded because their meaning is ambiguous or uninterpretable.

- [If the height is greater than 180, 1 is added to the counter] and the height is recorded. *It is not clear here whether 'recorded' means 'printed', 'added to a list', 'assigned to a variable' ...*
- The program is listing how many hours of sun there was only when the sun was High.

incomplete: statements which cannot be coded because they are incomplete. Statements which fall into this category tend to be unfinished sentences.

Information categories are related to each other in terms of level of granularity, which can be envisaged as follows: at the top level, the program can be described in terms of a small number of functions (in some cases, just one, if the programs are small). At a finer level of granularity, these *functions* are accomplished by a series of *actions*. The actions may be dependent on certain conditions, represented by *state-high* nodes. At an even finer level of granularity, the actions themselves are implemented in the program by *operations* which usually correspond to a single line of code, or one node in a VPL. Likewise, *state-low* nodes describe the state of a single data object, usually just after a test.

4.2 Object Descriptions

The object classification looks at the way in which objects are described. The basic question being asked is, "How do participants, when not constrained by specific instructions, choose to describe objects present in the program?".

Some objects cannot be classified at more than one level. For example, a program is, by definition, a program specific object. Similarly, objects introduced within the program (*i.e.* not inputs or outputs), and which have a *raison d'être* only within a program, cannot be classified in domain terms (*e.g.* a counter). However, the most interesting cases arise when there is a choice of levels at which the object can be described. For example, an input to the program could be described as *a list of numbers*, or alternatively, as *a series of basketball player's heights*.

4.2.1 Object Categories: Descriptions and Examples

The object classification comprises seven categories, described below with examples.

program only: refers to items which occur only in the program domain, and which would not have a meaning in another context, for example, a counter.

- This program initially sets *a counter* to zero…

program: an object, which could be described at various levels, described in program terms. Program terms refer to the use of any program specific data structure (*e.g.* a list) or variable (indicated by the lack of an article, the word in quotes, capitalized, etc.).

- ...checking first whether *the list* is empty or not ...
- If *'Height'* is then equal to or less than 180 'Sub Team' is run again.
- If *the current height variable* is above ...

program - real-world: object descriptions using terminology which is valid in both real-world and program domains, *e.g.* results, numbers. This category contrasts with the *domain* category in that the latter is specific to the problem domain described in the program, *e.g.* basketball players' heights, exam marks, distances between cities, while the former refers to terminology which is more abstract, and would be shared across problem domains. For example, a reference to *numbers* would be classified as a program - real-world description, while *exam marks* would be classified as a domain description.

- The program takes *2 numbers*...
- The program gives out *the 5 highest values* that were input to the program.

program - domain: object descriptions which contain a mixture of program and problem domain references, *e.g. a list of marks* (note that care must be taken to ensure that domain references are not in fact being used as variable names), or a reference which is equally valid in the program and the problem domains (*e.g.* differences).

- This is processing *a list of marks*...
- ...it then passes *a list of heights* to a sub-program.

domain: an object which is described in domain terms, *e.g.* a mark, a distance, sunny days rather than by its representation within the program.

- This program checks *a basketball players height* from [the list given].
- This program calculate *the number of students* who passed...

indirect reference: an anaphoric reference to a data object.

- ...*they* are stripped in turn out of [the list].
- ...if *it* is then the program returns to the main program.

unclear: statements which are ambiguous and cannot be coded, either because the statement itself is unclear, or because the object which is being referred to cannot be identified.

- ...is sent to *the pass marker*...
- ...[the head goes into] *a folder*.

Some of the categories above have links with other categories. Program and domain categories could be referred to as 'pure' categories in the sense that they refer to one level of description only. Program - real-world and program-domain are amalgamates of pure categories. Program only is a special case: unlike the categories just mentioned, it is used for objects which are inherently linked to the program domain and hence cannot be described at other levels. Finally, indirect reference and unclear statements are not linked to the others in any obvious way.

Additionally, the categories, as listed above, can be viewed as occurring on a continuum in terms of degree of ambiguity, starting with specific 'program' references having a low degree of ambiguity, through to 'unclear" statements at the other end of the ambiguity spectrum.

5 Examples of Applying the Scheme

The program summary analysis scheme described above has been used in an experiment which examined program comprehension in Prolog (Good, Brna and Cox, 1997), and also in a comparative study which looked at simple mock-ups of visual

programming languages based on the data flow and control flow paradigms. Program summary data was analysed in conjunction with multiple choice measures of program comprehension.

This section describes some of the results of the latter study. The aim is to show how the analysis scheme can be used, and its particular utility in comparatives studies of programming languages. Full details of these studies, and an in-depth consideration of results can be found in (Good, 1999) and (Good and Oberlander, 2002).

Figure 1 shows, on the left, a simple visual control flow version of a program designed to count the number of passing marks in a list, and, on the right, the corresponding data flow version.

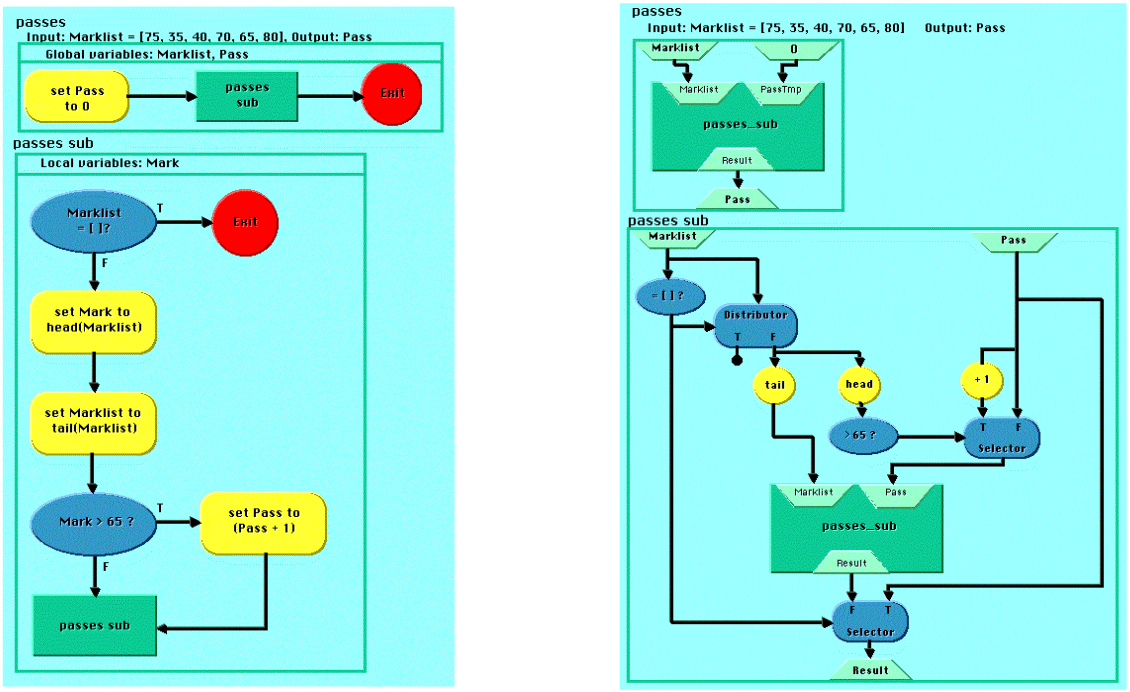

Figure 1: The `passes` program. Left: control flow version. Right: data flow version.

Tables 1 and 2 show the application of the information types classification to two program summaries of the `passes` program. The first summary is from a participant who studied the control flow version of the program, while the second is from a participant in the data flow condition.

Statement	Code
it first checks to see if the mark list is empty	operation
if it is	state – lo
then the program exits.	control
if that check is false	state – lo
it then sets mark to the first number in the list,	operation
and sets the rest of the list to some variable,	operation
it then checks to see if mark is greater than 65	operation
if it is	state – lo
it then adds 1 to pass	operation
and exit	control
if it is not	state – lo
it recurses	control

Table 1: Detailed description of the passes program, with statement types

Statement	Code
This is processing a list of marks	action
and finding which are greater than 65	action
and adding 1 to the counter	operation
to give a result of how many passed	data

Table 2: Higher level description of the passes program, with statement types

In the study which compared data flow and control flow visual program representations, differences between the two groups was obvious in the length of the program summaries: 70.91 words for the control flow group, and 48.85 words for the data flow group. However, the program summary analysis scheme allows for a finer grained analysis of the differences, as shown in Table 3. This shows the mean proportion of information types category statements for the control and data flow groups. Summaries from the data flow group contain higher proportions of function, action, state-high, and data flow information types than do the control flow group. The control flow group's summaries contain higher proportions of operation, state-low, and control flow statements. In terms of level of granularity, it emerges that the control flow group's summaries contain many more low-level statements than the data flow group.

Category	Control Flow	Data Flow
	Mean %	Mean %
function	11.62	20.93
data	13.10	24.68
state-high	6.22	8.23
action	7.10	9.10
operation	30.22	15.67
state-low	12.93	10.04
control	14.10	5.33
elaborate	.49	3.61
meta	.15	1.05
unclear	4.07	1.36

Table 3: Mean Proportion of Information Types Statements per Group

A comparison of object description categories yielded the following results:

Category	Control Flow	Data Flow
	Mean %	Mean %
program only	4.07	3.81
program	46.93	33.84
program - real-world	11.21	18.55
program - domain	4.52	5.29
domain	22.78	20.46
indirect	10.09	17.02
unclear	.39	1.03

Table 4: Mean Proportion of Object Description Statements per Group

Summaries from the data flow group contain higher proportions of program - real-world, program - domain and indirect statements than do the control flow group. The control flow group's summaries contain higher proportions of program only, program and domain statements. Finally, data flow subjects made more references to objects which were judged to be unclear than did control flow subjects.

The results from the information types analysis are compelling, and point to differences between the two groups which have been studied in detail (Good and Oberlander, 2002). This suggests that the information types scheme is of value in capturing fine-grained differences in program summaries. On the other hand, results

from applying the object description analysis are less clear-cut, and point to the need for further investigation of the scheme's utility.

6 Discussion and Conclusions

This paper presented a coding scheme for analysing program summaries. The scheme aims to provide more authentic measures of program comprehension, by allowing programmers to express their understanding in their own words. To date, the scheme has been applied to textual descriptions of programs, but it is hypothesised that it could be equally useful for verbal protocols gathered during comprehension tasks. The scheme has been used on descriptions of programs written in diverse languages, and seems to capture subtleties in participant's reporting of comprehension resulting from differences in program representation.

In addition, the scheme has also been used to look at *levels of abstraction* in program summaries. By mapping information types onto levels of abstraction (with function, elaborate and meta statements at the highest level of abstraction; action, state-high and data statements at an intermediate level, and operation, state-low and control statements at a low level of abstraction), we found that participants in the control flow condition tended not to change level of abstraction between statements: almost 80% of consecutive statements in their summaries represent cases where the statements may be of differing information type, but are at the same level of abstraction. By contrast, in the data flow group, less than half of their consecutive statements were of this kind, showing that that were as likely to change level of abstraction as to stay within the same level.

Furthermore, by using the scheme in conjunction with an analysis of errors in program summaries, where errors were classified at the highest level as either errors of *commission*, or errors of *omission*, we discovered that although control flow participants produced long summaries at consistently low levels of abstraction, they tended to exhibit more errors of omission than did data flow participants, who produced shorter summaries at varied levels of abstraction. This research is report in full in (Good and Oberlander, 2002), but is mentioned here in order to demonstrate varying ways in which the scheme may be put to use.

In order to further refine and generalise the scheme, the following issues are of interest:

- *Inter-rater reliability*: to date, the scheme has been applied by only one researcher. Therefore, although coding instructions and coding tools have been developed, we do not yet have information as to the relative difficulty of applying the scheme, or on issues of inter-rater reliability.
- *Universality*: as mentioned above, the scheme has been applied to diverse languages, however, two of these languages were not full-scale executable languages. In order to judge the general applicability of the scheme, it would be useful to test it out on summaries of programs written in other languages.
- *Scalability*: the scheme has so far only been tried on short programs studied by novice programmers. Program summaries of longer programs will necessarily have different characteristics: it remains to be seen whether the scheme allows useful comparisons of these types of summary.
- *Summary modality*: the scheme has been applied to written program summaries. It is an open question whether we would find the same patterns of information types if participants "self explained" verbally during the comprehension process.

The Cognitive Dimensions of Notations framework has been described as a potentially extensible framework which provides "a vocabulary that can be used by designers when investigating the cognitive implications of their design decisions" (Blackwell et al., 2001, p. 326). Similarly, we feel that the coding scheme described in this paper is a work in progress which may be of use to other researchers who are interested in describing the structure of program comprehension artefacts, and we welcome comments, suggestions and refinements.

Acknowledgements

This work was carried out at the Human Communication Research Centre, at the University of Edinburgh, and was supported by the UK Engineering and Physical

Sciences Research Council, through grant GR/L36987. Our grateful thanks to Jon Oberlander and Richard Cox, our collaborators on the project.

References

Blackwell, A.F., Britton, C., Cox, A., Green, T.R.G., Gurr, C.A., Kadoda, G.F., Kutar, M., Loomes, M., Nehaniv, C.L., Petre, M., Roast, C., Roes, C., Wong, A. and Young, R.M. (2001). Cognitive Dimensions of Notations: Design tools for cognitive technology. In M. Beynon, C.L. Nehaniv, and K. Dautenhahn (Eds.) *Cognitive Technology 2001*. Berlin: Springer-Verlag, 325-341.

Corritore, C. L., and Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, *3*: 199-222.

Good, J. (1999). *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

Good, J., Brna, P., and Cox, R. (1997). Novices and program comprehension: Does language make a difference?, In *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, LEA, 936. A longer version appeared as Technical Report 97/10, Computer Based Learning Unit, University of Leeds.

Good, J., and Oberlander, J. (2002). Verbal effects of visual programs: information type, structure and error in program summaries. *Document Design*, *3*: 120-134.

Green, T.R.G. (1980). Programming as a cognitive activity. In H. Smith and T.R.G. Green (Eds.), *Human Interaction with Computers,* Academic Press, 271-320.

Green, T.R.G., Sime, M.E., and Fitter, M.J. (1980). The problems the programmer faces. *Ergonomics*, *23*: 893-907.

O'Brien, M.P., Shaft, T.M., and Buckley, J. (2001). An Open-Source Analysis Scheme for Identifying Software Comprehension Processes. In G. Kadoda (Ed.) Proceedings of PPIG-13: 13th Annual Meeting of the Psychology of Programming Interest Group, 129-146.

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

Pennington, N. (1987b). Comprehension strategies in programming. In G.M. Olson, S. Sheppard, and E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop,* New Jersey: Ablex Publishing Corporation, 100-113.

Ramalingam, V. and Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Proceedings of 7th Workshop on Empirical Studies of Programmers,* New York: ACM Press, 124-139.

Romero, P., Cox, R., du Boulay, B., and Lutz, R. (2002). Visual attention and representation switching during Java program debugging: A study using the Restricted Focus Viewer. In Proceedings of the Second International Conference on the Theory and Application of Diagrams (Diagrams 2002), 221-235.

Upchurch, R. (2002). Code Reading and Program Comprehension: Annotated Bibliography. Retrieved January 6, 2003 from http://www2.umassd.edu/SWPI/ProcessBibliography/bib-codereading2.html.

von Mayrhauser, A., and Lang, S. (1999). A Coding Scheme to Support Systematic Analysis of Software Comprehension. *IEEE Transactions on Software Engineering*, *25*: 526-540.