Trends in the use of verbal protocol analysis in software engineering research

JANET HUGHES and STEVE PARKES

Department of Applied Computing, University of Dundee, Dundee, Scotland; DD1 4HN, UK; e-mail: jhughes@computing.dundee.ac.uk and e-mail: sparkes@computing.dundee.ac.uk.

Abstract. This article reviews the technique of verbal protocol analysis and gives a profile of its use within software engineering research over the last two decades. An overview is given of the procedures used in verbal protocol analysis, and commonly-found difficulties in the application of the technique by researchers are described. The article reports on published efforts to develop tools to automate the procedures. A review of the literature shows trends in the use of the verbal protocol analysis in software engineering research from the 1980s to the present. Recurring themes of its purpose within software engineering research are identified, including the comparison of the behaviours of subjects with differing levels of expertise and the identification of effective software comprehension strategies. Advances and problems with the development of a general-purpose encoding scheme for verbal protocol analysis are described.

1. Introduction

The technique of protocol analysis of verbal data has been prominent in cognitive psychology research for over 25 years. The underlying principle of the technique is that any verbalisation produced by a subject whilst problem solving—known as concurrent ‘think aloud’—will directly represent the contents of the subject’s working memory (Ericsson and Simon 1993). At the simplest level, the technique provides a bias-free method of revealing what a person is thinking when performing a task. Its importance stems from the value of obtaining a real-time insight into the knowledge that a subject uses and the mental processes applied while performing a process of interest. Detienne and Soloway (1990) describe the insight as a ‘sort of window onto subjects’ processing strategies’. The popularity of verbal protocol analysis comes in part from its versatility: any research which features a subject’s problem-solving behaviour, and which questions the underlying model the subject has of a task, may obtain valuable information from the use of verbal protocol analysis (e.g. Sen 1997). Using the technique, a researcher can obtain an insight into the subject’s cognitive processes and use that to address a research question, for example, to investigate a subject’s understanding of the problem space. There may be interest in the approach used to frame the problem space (e.g. Kim et al. 1997), or the strategy used to search the problem space to move towards the goal (e.g. Detienne and Soloway 1990), or tactics adopted to select between alternative sub-goals, or how the current state of the problem solution is evaluated. The requirements of the problem-solving task will direct what information is processed by the subject. Concurrent verbalisation articulates that information processing (Ericsson and Simon 1993: 167). As an example, verbalisations made during software debugging could be expected to reveal in real-time the various factors considered by the subject as potentially contributing to a defect. Similarly, verbalisations made during software maintenance could disclose the hypotheses considered and rejected, or images and mental models created in the process of formulating an understanding of the software.

Verbal protocol analysis is a simple technique to apply, requiring few special arrangements, and (if applied as described in the next section) it should not interfere with the software development process itself. Recorded verbalisations are transcribed, coded, and the codes analysed in accordance with the relevant research question. Section two of this paper presents an overview of the technique of verbal protocol analysis.

Alternative techniques, such as interviews and questionnaires, can be used to gain information about aspects of the software design process that cannot be directly observed. However such retrospective techniques can be criticised as providing, at best, an incomplete report, due to forgetfulness or selective
reporting of thoughts and mental processes (Ericsson and Simon 1993: xlix, Bainbridge 1999). At worst, interviews and questionnaires can produce inaccurate reports, due to erroneous or distorted reporting: a person’s recollection of his or her thinking may be significantly different from the actual thoughts made at the time (Durkin 1994: 560–566). In contrast, concurrent verbalisation should capture the latter reliably.

Verbal protocol analysis has been applied to a range of problem types, and data obtained has been used to create, confirm, or refute hypotheses across a range of research domains, including various fields of education (e.g. Chi et al. 1989, Trudel and Payne 1995), cognitive science and psychology (e.g. Narayanan et al. 1995, Renkl 1997), artificial intelligence (e.g. Conati and VanLehn 2000), human-computer interaction (e.g. Howard 1997), and computing (e.g. Herbsleb et al. 1995, Hale et al. 1999). For this article, a review of literature was performed by the authors to appraise the application of the verbal protocol analysis technique to software engineering in particular. Twenty-six studies, published in a range of 15 journals and the proceedings of six different conferences, were examined to establish how the technique had been applied within the field of software development. These studies show that researchers’ usage of the technique has ranged from the analysis of software engineers’ behaviours, strategies, intentions, and understanding to the application of methodologies and languages, and latterly to group interactions and dynamics in review and inspection meetings. Section three of this paper presents a summary of the recent use made of the technique in software development research and identifies themes in vogue over two decades.

The remaining sections describe current developments in the use of verbal protocol analysis in software engineering, in terms of coding schemes and protocol analysis tools which may facilitate its use, and anticipate aspects of software engineering for future application.

2. An overview of the verbal protocol analysis technique

Ericsson and Simon’s 1993 text on protocol analysis has been the initial guidance for most recent researchers in their use of the technique. The main phases are verbalisation, encoding of the transcribed verbalisation, and analysis of the codes. Figure 1 summarises the main aspects of the entire technique.

Some researchers deviate from that recommended procedure because the underlying theoretical basis of their analysis differs from that of a purely problem-solving model. This is exemplified in the work of Chi into the role of self-explaining in learning, which aims to uncover the knowledge that a learner has and determine how it influences subsequent reasoning and problem-solving, rather than focus on problem-solving processes per se. Chi (1997) gives an explanation of her version of the method (‘verbal analysis’), which postpones the process of devising an encoding scheme (stage 0) until after a protocol has been segmented and aggregated into episodes (stages 3 and 4).

2.1. Verbalisation

There are three types of verbalisation:

- level one is where simple vocalisation is possible without further processes, because the information is reproduced directly (e.g. Pennington et al. 1995);
- level two is where there may be a need to describe or recode information into a verbal form, for instance if the original format is graphical (e.g. Narayanan et al. 1995);
- level three verbalisation requires further processing, such as might result from a researcher prompting an explanation or interpretation by the subject (e.g. van der Veer 1993, Chi et al. 1989).

Think-aloud verbalisations of the level one type and level two type should not change the level or structure of a subject’s thought processes (Ericsson and Simon 1993: 106). The subjects are given detailed instructions, such as: ‘Try to think aloud.’; ‘tell me everything that passes through your head during your work’ … ‘no matter how irrelevant it may seem’. One implication of excluding level three type verbalisations is that the experimenter should not become involved other than to prompt to break a silent period. Usability researchers tend not to conform to the Ericsson and Simon model in this respect: Boren and Ramey (2000) suggest that the commonplace contingencies in usability testing require regular/extensive researcher interaction with the subject.
2.1. Stage 1: recording the verbalisation: Some subjects find difficulty giving complete verbalisations. It can be helpful, therefore, to give subjects practice with similar tasks before the main study, and it may be necessary to give a short reminder to keep verbalising if there is a period of silence (Ericsson and Simon 1993: 256). A reminder can take the form of a prompt such as ‘Please, think aloud’ or ‘Keep talking’. Sonnentag (1998) prompted subjects after 15 s of silence. Regrettably many other researchers do not provide specific details about the frequency of prompts to subjects to keep talking, or the prompt is said to be offered after an indeterminate amount of time, such as after a ‘prolonged period of silence.

The number of subjects utilised in software engineering studies tends to be small, with fewer than 30 protocols being collected. Usually a sub-set of these are only used in the next stages of the research, often with 10 or fewer being prepared for encoding (e.g. Kim and Hahn 1997). Large-sized groups have been more frequently used in the last 5 years than in the 1980s, when groups were normally of three to six people (e.g. compare Adelson and Soloway 1985 or Pennington 1987 with Sonnentag 1998 or Hale et al. 1999). Experiments normally take between 1 and 2 h to be completed, although shorter recordings have been reported (Pennington 1987, Sutcliffe and Maiden 1992, Burkhardt and Detienne 1995). In most cases the experiment has not been time-limited (e.g. Detienne and Soloway 1990).

The majority of researchers report recording verbalisations with video-tape, particularly in circumstances where the subject’s actions within the environment could contribute further data of interest. An example of this is the analysis of videotapes of design meetings by Herbsleb et al. (1995), which suggested that despite design being a highly collaborative active, no tools were used by teams (other than paper and whiteboard) to facilitate such collaboration by recording the design rationale. Where notes, diagrams, or other materials are produced during the course of the experiment, these can be retrieved for correlation with verbalisations made (e.g. Pennington et al. 1995). Bowdidge and Griswold (1997) describe the assembly of a range of data in support of a verbal protocol, including logs of keystroke and mouse actions, notes made by participants, and videotape evidence of gestures made and the focus of participants’ attention. Unfortunately few researchers have reported either the procedures or the results of processing such visual data, and/or how it relates to the verbalisation, in such detail.

2.2. Protocol encoding

The transcribed verbalisations, or protocol, are then transcribed and encoded.

2.2.1. Stages 2–4: transcribe, and segment the verbalisation and aggregate segments: The actions of transcribing and segmenting the verbalisations usually are performed without any difficulty. The protocol is subdivided into a number of segments (statements or phrases), each of which is given an identification number. Although there is no standard definition of a segment, most segmentation schemes are broadly similar, being based upon an identifiable single unit such as a reference or assertion (e.g. Vessey 1985, Pennington 1987, d’Astous et al. 2000), a distinct or pause-bounded utterance or phrase (e.g. Letovsky et al. 1987, Bielaczyc et al. 1995, Sonnentag 1998), or a complete thought or sentence (e.g. Arunchalem and Sasso 1996, Schenk et al. 1998). Segment aggregation is relatively straightforward, although encoders’ judgements are required if inferences are to be made from the given verbalisation. For example, the use of short statements such as: ‘right, OK’ may be judged simply to be habitual phrases without consequence, and therefore are not treated as episodes in their own right. Conversely, the context and intonation used by the subject may indicate that such a phrase indicates the confirmation of the quality of an object under examination, and hence it is regarded as an independent episode.

More than one consecutive segment may be aggregated as an episode, according to the encoder’s judgement of the context of the statements made. Chi (1997) identifies three issues worthy of consideration for the segmentation process: (i) grain size of the segment, such as sentence, paragraph, or dialogue interchange; (ii) correspondence of the grain size to the research questions being asked; and (iii) characteristics of the data. With respect to grain size, Chi notes that the sensitivity of the data depends upon the grain size of the segment, and that separate analyses of a protocol may result in re-coding with a coarser or finer grain size (Chi 1997). Characteristics of the protocol data may influence the features used for segmentation. Non-content segmentation can be based upon syntax or grammar, or events such as a change of activity or speaker, whereas semantic segmentation is based upon features such as ideas or threads of discussion. Although semantic feature segmentation is more demanding and time-consuming to perform than non-content segmentation, the context of the research may necessitate its use (e.g. Chi et al. 1989). Figure 2 presents an example of segments aggregated according to semantic feature.
2.2.2. Stage 4a: sample the episodes: The most commonly repeated observation about verbal protocol analysis is how extremely time-consuming the process is, with analysis: sequence time ratios ranging from 5:1 to 100:1 (Fisher and Sanderson 1996). To alleviate this, verbalisation recordings can be sampled or reduced (Chi 1997). Protocols can be sampled either at random or by using a non-content criterion—such as a given amount of time or speech—to choose a subset of the data. However few researchers appear to reduce the data for processing, possibly because this method introduces the question of how the subset should be selected, and raises obvious doubts that the subset is a valid representation of the remainder of the data. A more common way of reducing data volume is to process only a sub-set of all the subjects’ protocols (e.g. Sutcliffe and Maiden 1992, Bielaczyc et al. 1995, Kim and Hahn 1997, Irwin and Wasalathantry 1998, Sonmentag 1996, 1998, Hahn and Kim 1999).

2.2.3. Stage 5: encode the episodes: Episodes are encoded according to a coding scheme created either before or after the verbalisation was captured. Ericksson and Simon (1993: 276) are clear that the code alternatives ‘should be explicated prior to accepting input for encoding’, to avoid data contamination by ad hoc theory. Encoding decisions should be based upon the task model, the theory under test, and/or the problem space. Protocols are then scanned to identify the particular vocabulary used by subjects that corresponds to these objects and operations. Ultimately the vocabulary of codes should represent mutually exclusive cognitive categories for analysis. Examples of studies that generally conform to these guidelines include those of Olson et al. (1996), and Arunachalam and Sasso (1996). Table 1 presents a sample of a protocol encoded using a pre-constructed vocabulary of codes. The context for the protocol was the design of a system simulating a lunar surface.

However, the process of devising a coding scheme can be quite challenging for a researcher. Although an initial coding scheme may be constructed from the literature, published research is unlikely to provide a perfect model which can be re-applied: the choice of codes will depend upon the subject domain, the hypotheses being tested, the research questions being asked, and the ‘theoretical orientation’ of the researcher (Chi 1997). Table 2 provides an illustration of the wide range of coding schemes applied in investigations, in this case in the context of software maintenance.

It is perhaps unsurprising then that the preliminary coding schemes are often found, in their first application, to be too crude, and to merit subsequent revision. As Ericsson and Simon (1993: 309) acknowledged, protocols are even collected in the absence of a cognitive model defining the coding categories. This is known to take place in software engineering research: fewer than half of the research papers reviewed here used a coding scheme which evidently had been prepared a priori (although it should be noted that precise details of the encoding procedure are sometimes not clear from the presented research paper). Rather, examination of the protocols is used to construct an encoding scheme (e.g. Lee and Pennington 1994). Guided by the content of a small number of protocols examined, a researcher may construct a coding scheme, then attempt to apply the coding the coding scheme, and then revise it again, and so on. This type of iterative development of a coding scheme contributes to the time-consuming nature of the process of verbal protocol analysis. Less time-consuming is the approach taken by Schenk et al. (1998), whereby a coding scheme prepared in advance of data collection is adjusted after trial in a pre-test.

There is a trend from the use of purely numerical identification of codes towards a more meaningful format. D’Astous et al. (1998) introduced a formal hierarchical coding scheme with a Backus-Naur form (BNF) notation (figure 3). Appropriate to technical review meetings, codes were defined as activities, such as ‘request’ or ‘evaluate’, which could be performed on certain entities, such as ‘solution’ or ‘result of previous
activity’, with optional criteria relating to document format or content. Reporting on further work in this area in 2000, the group labelled activities, entities, and criteria using mnemonics. An episode requesting justification of an item of information identified as INTRO51, for example, would be encoded as ‘REQ/ JUST/INTRO51’ (D’Astous et al. 2000).

Vans, von Mayrhauser and Somlo (1999) used simple mnemonics with numerics to distinguish between different action types, for instance SIT7 representing generate hypothesis in a situation model, SIT10 representing a failed hypothesis in a situation model, whereas generate hypothesis in a program model is represented by SYS8. Developing this further, von Mayrhauser and Lang (1999) described a more generic coding scheme which they called ‘A Flexible Expandable Coding Scheme’ (AFECS). The hierarchical coding scheme with mnemonics and a dot notation (table 3) was developed to provide a system that could: (a) be applied to different research studies; and (b) be extended to accommodate newly-identified codes within the software comprehension domain, or indeed alternative codes for alternative domains.

2.2.4. Stage 6: check coding reliability: Whatever the origin of the coding scheme, there should be confidence in the reliability of a scheme’s application to a set of protocols. Sources of possible error include the need to make inferences from segments and the difficulty of judging coding episodes independently of other episodes. Therefore encoding consistency and reliability should be assessed, typically by means of a second encoder. The original encoding can be compared with that of the second coder to obtain either the percentage agreement or the coefficient of agreement, Cohen’s K. Accepted percentage figures are typically above 75% (e.g. Sutcliffe and Maiden 1992, Bielaczyc et al. 1995, Irwin and Wasalathantry 1998), or above 0.8 K (e.g. Arunachalam and Sasso 1996: 0.84, Schenk et al. 1998: 0.89, Hale et al. 1999: 0.85).

2.3. Protocol analysis

2.3.1. Stage 7: analyse the code patterns: Finally the encoded protocol is analysed to answer the research questions. The number and type of analyses used typically reflects the number and type of research question being asked. Examples include:- enumeration of specific categories (e.g. Bielaczyc et al. 1995); relative distribution of different activity categories identified, either in terms of protocol segment (e.g.
Vitalari and Dickson 1983, Lee and Pennington 1994) or in terms of actual time spent (e.g. Sonnentag 1996, Olson et al. 1996); analysis of the pattern of activities, for example switching between two types of activity (Arunachalam and Sasso 1996, Olson et al. 1996) and even verbalisation rates (Vessey 1985). The sequence of transitions between different types of activity may be explored using lag sequential analysis (e.g. Robillard et al. 1998, D’Astous et al. 2000), whereby the frequency of pairs of activities is calculated. The analysis determines if the first activity of each pair, called the criterion activity, is followed after a specified lag interval by the target activity. The test is repeated for each possible target activity and each specified lag interval, to determine if the occurrence of a pair is more frequent than might be expected by chance. Hale et al. (1999) provide a summary of the lag sequential analysis technique. Transition behaviour results are usually depicted graphically (e.g. Pennington et al. 1995, Kim and Hahn 1997); or figuratively (e.g. Olson et al. 1996, D’Astous et al. 2000).

Vans et al. (1999) report an example of how several different analyses can be applied to a single protocol:

(i) frequencies of action types were enumerated and used to investigate the level of abstraction at which programmers preferred to work;
(ii) analysis of the sequence of action types was used to examine programmers’ use of different comprehension models, such as top-down or bottom-up;
(iii) patterns of switching between comprehension models were used to explore the relationship between expertise/domain knowledge and the abstraction level used during corrective maintenance.

3. Use of verbal protocol analysis in software development research

Section two presented an overview of the mechanics of verbal protocol analysis. This section will consider key features and the role of the technique in software development research. Human-computer interaction and user interface studies are excluded from the studies reviewed, because, as described in section two, the ‘think aloud’ procedure is applied differently in these areas to its application in software engineering.

In Brooks’ (1987) assessment of the software development industry, there was no silver bullet: no single development which would provide ‘an order of magnitude improvement in productivity, in reliability, in simplicity’. His forecast then of the way towards improvement was in the systematic identification and nurturing of ‘great designers’. Similarly, Yourdon (1992) included ‘better people’ as one of 11 important areas for improvement of the American software industry. The 1980s and 1990s motivation to identify best practices in the software development process and incorporate these into education and training was understandable, given the predictions of a global need for software developments and an apparent paucity of great designers.

3.1. 1980s research

Verbalisation and protocol analysis was used throughout the 1980s to identify behaviours associated with expertise or high quality performance. Vitalari and Dickson (1983) employed the technique to compare the thought processes of high- and low-rated systems analysts, and thereby identified a set of basic skills of systems analysis. As well as identifying the better hypothesis management of high performers, the study flagged up the probable importance of requirements analysis and a good quality analyst-client relationship. Adelson and Soloway (1985) considered six different behaviours observed during the software design process, showing the influence of domain expertise upon the choice of behaviours by novices and experts.

The focus of attention at that time was in the domain of programming rather more than systems analysis, particularly featuring program comprehension studies in the late-1980s. Vessey (1985) compared expert and novice strategies to debug a COBOL program, and concluded that the expert group adopted a high-level systems thinking approach to the problem rather more than the novices. Letovsky (1986) used results from a protocol analysis study of professional programmers to develop a preliminary cognitive model of program understanding. Pennington (1987) suggested from a study of programmers’ protocols produced while modifying a COBOL or a FORTRAN program that the high-level performers were able to utilise different views of the program, in particular cross-referencing a program-level comprehension with comprehension of the relevant domain; their poorer-comprehending peers tended to use only one view.

This period also saw some initial explorations of team working activities. One small study (Walz et al. 1987) analysed the protocols from a series of requirements and design meetings of one project team, noting that conflict is normal in such group work and suggesting that its identification and resolution is more important than its minimisation. An analysis of the video-taped protocol of a code inspection meeting (Letovsky et al. 1987)
concluded that the major effort of an inspection was spent striving for clarity, correctness and consistency—and noted that a significant amount of the team effort was expended cross-checking information redundantly recorded in different formats in different documents.

3.2. Early–mid-1990s research

In the first part of the 1990s, LISP featured in several studies of knowledge and skill acquisition. A protocol analysis of the evaluations by experts of other programmers' code (Mastaglio and Riemann 1991) was performed to obtain information that would support the development of a knowledge-based programming language critic. Think-aloud protocols were used to investigate the transfer of knowledge between cognitive subtasks—the evaluation and generation of LISP instructions (Pennington et al. 1995)—concluding that declarative knowledge elaborations do contribute to the acquisition of procedural skills. Knowledge acquisition activities in LISP problem solving situations (Bielaczyc et al. 1995) provided early evidence that self-explanation and self-regulation of comprehension success could contribute to learning and problem-solving performance.

Not surprisingly during a shortage period of skilled systems analysts, the relation of research to tools which might support software engineering was highlighted. Research was reported which might inform the development of intelligent CASE tools which would assist novice analysts rather than experts (Sutcliffe and Maiden 1992). The verbal protocols collected from novice analysts were examined to identify any underlying basis of good or poor performance. No single factor was found, although the ability to reason well with a conceptual model and critical testing of hypotheses again is reported as relating to high levels of performance. Detienne and Soloway (1990) analysed protocols of experienced designers during a program understanding task. Information about the range of different strategies identified in program understanding, it was mooted, might be useful in developing tools to support program understanding. A study of software design team meetings (Herbsleb and Kuwana 1993) reflected on tools which might contribute to the software design process. The preponderance of comments found relating to how a high-level function or behaviour was to be realised by a lower-level function or behaviour led to the suggestion that design tools should optimise for traceability, retrieval, and display of functions which would demonstrate this realise relation.

3.3. End-of-decade research

In late 1990s research, several themes are evident again: team working, high and low performances, and software comprehension. Olson et al. (1996) analysed design meetings, finding that apparently very informal meetings that seemed 'quite chaotic' in organisation were in reality quite structured and ordered. Analysis of verbal protocols of technical review meetings have been reported (D'Astous et al. 1998, 2000, Robillard et al. 1998) that identified cognitive synchronisation activities—where participants ensure that they share a common understanding of design solutions or evaluation criteria—as major components of the review team's discussions. Consequently these authors have suggested that technical reviews may be improved by a revision of the methodology for such meetings to require an initial synchronisation phase which would assure a shared understanding of the review objects before the review proceeds. The most recent work by D'Astous et al. (2000) again uses evidence from protocol analyses to suggest a more efficient arrangement of peer review meetings: an initial separation of document format review from the group effort could be followed by a cognitive synchronisation meeting. Finally a defect detection meeting could be held which, by dint of the two previous meetings, would require a smaller number of participants.

Attempts to identify behaviours particular to high performers or experts have continued. Further evidence of the reluctance of novice systems analysts performing requirements analysis to perform hypothesis testing and rejection was found (Schenk et al. 1998). They also observed that compared to the experts, the novices had less domain knowledge, had a more superficial concern with user involvement, reacted less to specific problem-solving triggers and generated fewer goals. These conclusions lead the authors to suggest a role for creativity-enhancement tools, and more real-world experiences in education. Other studies (Sonntag 1996, 1998) gathered protocols from professional software designers during a software design task: subsequent analyses identified the strategies used, and a range of differences in the activities of the high- and low-performers, including more use of feedback processing and local planning by the high-performers.

The use of verbal protocol analysis in software comprehension and maintenance studies featured regularly. Shaft and Vessey (1996) highlighted the importance of application domain knowledge, whereas others attempted to identify activity patterns in programmer debugging (Hale et al. 1999), corrective maintenance (Vans et al. 1999) and software re-engineering (Arunchalam and Sasso 1996).
A relatively recent development has been the use of verbal protocol analysis in studies of object-oriented design; however the themes remain similar: identification of novice-expert differences and enhancements to tools in support of software development. Burkhardt and Detienne (1995) made an investigation of the design processes, mental processes, and mental representations involved when contemplating software reuse in the context of object-oriented design. Their conclusion reinforced the belief that design environments should be flexible, providing designers with support for their preferred design approach (top-down or bottom-up) to, or mental view (dynamic or static) of a developing solution. The cognitive demands of processing a multiplicity of diagrams, for instance as found in modern object-oriented methodologies, have been investigated (Kim and Hahn 1997, Hahn and Kim 1999). The authors suggest that guidelines could be developed for the design of diagrams that are ‘cognitively compelling’, and which might be used in different methodologies and CASE tools. Comparisons of the strategies of experts with those of novices in object-oriented design indicated that novices delayed the definition of goals (Lee and Pennington 1994) and performed fewer early monitoring activities (Irwin and Wasalathantry 1998). The former study reinforced earlier work (Pennington 1987) that novices found difficulty in linking different views, in this case to relate the solution domain to the problem domain.

3.4. Current research

Currently the quest continues to identify and disseminate best practices in software engineering. Zendler (2001), having constructed a preliminary theory of software engineering, suggested that experiments should be conducted to investigate the effects of various software engineering techniques on ‘efficiency and effectiveness during software development’. It is important that empirical data from studies of software engineering techniques as applied in practice in industry are collected in an ecologically valid way. This suggests the use of observational and protocol techniques, as for example used by d’Astous et al. (2000, 2001) in investigations of technical review meetings. Verbal protocol analysis also continues to feature in research relating to software engineering education, for example by Gama (2001) when investigating reflective activities in problem solving, and Kehoe et al. (2001) when investigating the efficacy of algorithm animations as learning aids.

3.5. Themes pursued over two decades

The preceding review of a number of studies involving verbal protocol analysis evidences the application of the technique across a broad range of software engineering activities. Table 4 summarises the pattern of application across a range of different software engineering activities. It appears that the history of verbal protocol analysis in software systems development partly reflects the evolution of software engineering itself. Early work supported efforts to devise a cognitive model of programming behaviour, and gave some attention to different strategies used with languages such as COBOL or FORTRAN. Consideration moved then to scrutiny of the software design process and use of tools to support it. Later studies utilising verbal protocol analysis considered alternative methodologies, including analyses of the object-oriented approach to software development, and the use of diagrams in the modelling process. Two themes have remained in vogue across the period: (a) an investigation of the design process; and (b) a comparison of behaviours attributed to differing levels of expertise. Themes which have recur periodically are the identification of: (a) effective team work practices; and (b) strategies for program comprehension. Table 4 reveals the absence of the technique in research into certain activities which might have been expected to feature it, such as test planning, technical authoring, cost estimation, use of formal methods, real-time systems design, and project management (including configuration and risk management).

4. Future use and developments in verbal protocol analysis

The preceding section considered the history of verbal protocol analysis in software engineering. This section will describe recent developments in coding schemes and in software to facilitate analysis, and anticipate other aspects of software engineering likely to be investigated.

4.1. General-purpose encoding schemes

Although specific themes have recurred throughout the past two decades in research applying the verbal protocol analysis technique, no common coding scheme has been developed that can be applied in a range of different research circumstances. The existence of such an encoding scheme would help avert accusations that encoding schemes which were not developed a priori were invalid, and would facilitate comparisons of different research studies. Von Mayrhauser and Lang (1999) introduced AFECS as a scheme which could be
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Area of Study</th>
<th>Activities Investigated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>Vitalari</td>
<td>(levels of expertise)</td>
<td>Requirements analysis, Design &amp; design meeting, Program coding &amp; debugging, Software re-engineering &amp; reuse, Team reviews &amp; inspections</td>
</tr>
<tr>
<td>1984</td>
<td>Adelson, Vessey</td>
<td>(novice vs. expert)</td>
<td>Letovsky (comprehension strategies)</td>
</tr>
<tr>
<td>1986</td>
<td>Walz</td>
<td>(team work)</td>
<td>Pennington (levels of expertise), Letovsky (team work)</td>
</tr>
<tr>
<td>1987</td>
<td>Sutcliffe</td>
<td>(novice vs. expert)</td>
<td>Letovsky (levels of expertise)</td>
</tr>
<tr>
<td>1988</td>
<td>Herbsleb</td>
<td>(team work)</td>
<td>Detienne (comprehension strategies)</td>
</tr>
<tr>
<td>1990</td>
<td>Lee</td>
<td>(novice vs. expert)</td>
<td>Mastaglio (levels of expertise)</td>
</tr>
<tr>
<td>1991</td>
<td>Pennington</td>
<td>(novice vs. expert)</td>
<td>Mastaglio (levels of expertise)</td>
</tr>
<tr>
<td>1992</td>
<td>Sutcliffe</td>
<td>(novice vs. expert)</td>
<td>Pennington (levels of expertise)</td>
</tr>
<tr>
<td>1993</td>
<td>Herbsleb</td>
<td>(team work)</td>
<td>Letovsky (team work)</td>
</tr>
<tr>
<td>1994</td>
<td>Lee</td>
<td>(novice vs. expert)</td>
<td>Pennington (novice vs. expert)</td>
</tr>
<tr>
<td>1995</td>
<td>Pennington</td>
<td>(novice vs. expert)</td>
<td>Burkhardt (re-use strategies)</td>
</tr>
<tr>
<td>1996</td>
<td>Olson</td>
<td>(team work)</td>
<td>Shaft (domain knowledge), Arunachalam (comprehension strategies)</td>
</tr>
<tr>
<td>1997</td>
<td>Kim</td>
<td>(use of diagrams)</td>
<td>Bielaczyc (problem solving strategies)</td>
</tr>
<tr>
<td>1998</td>
<td>Schenk, Irwin</td>
<td>(novice vs. expert)</td>
<td>D’Astous (team work)</td>
</tr>
<tr>
<td>1999</td>
<td>Hahn</td>
<td>(use of diagrams)</td>
<td>Roibillard (team work)</td>
</tr>
<tr>
<td>2000</td>
<td>Hale</td>
<td>(debugging strategies)</td>
<td>Vans (comprehension strategies)</td>
</tr>
<tr>
<td>2001</td>
<td>Vans</td>
<td>(comprehension strategies)</td>
<td>D’Astous (team work)</td>
</tr>
</tbody>
</table>
tailored to various research studies, suggesting that objects and operators to be included at appropriate nodes in the original AFECS scheme could be identified from the problem space of the particular software engineering domain being investigated.

A review by the authors of different software engineering coding schemes reported in the literature over 20 years has been used to identify categories of activity and object common to a spectrum of software development activities (table 5), several of which are potential additions to the original AFECS scheme. Code pairs which do not exist in the original AFECS scheme include, for example:

Develop or Explain solution; Justify solution/goal/intention; Test hypothesis; Reformulate problem/goal/hypothesis; Elaborate hypothesis/solution/goal/mental model; Manage goal/meeting/schedule.

Some activity codes do exist, such as Generate and State, but have a limited set of objects. Generate, for instance, cannot presently be associated with criteria.

Work is in progress by the authors to use this data to update the AFECS scheme, and to evaluate the revised version of the coding scheme in the circumstance of an object-oriented software design analysis. Figure 4 presents an example section of a revised AFECS scheme, with additions by the authors given in italics.

Results to date indicate that a general-purpose encoding scheme is helpful as a template to determine the basic structure of a coding scheme. It is a straightforward process to adapt such a scheme with the use of pilot study protocols. During the application of the resulting scheme, however, attention focuses entirely upon those peripheral nodes and leaves of the coding tree that are relevant to the specific research questions being asked or theories being tested. Other components of the encoding scheme effectively are excluded as irrelevant to that study. The actual encoding scheme applied thereby is a set of activity and object

Table 5. Activities and objects found in software engineering encoding schemes.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate</td>
<td>Hypothesis/question/answer/mental model/notes/criteria</td>
</tr>
<tr>
<td>Test</td>
<td>Hypothesis</td>
</tr>
<tr>
<td>Confirm</td>
<td>Hypothesis</td>
</tr>
<tr>
<td>Reject</td>
<td>Hypothesis</td>
</tr>
<tr>
<td>Abandon</td>
<td>Hypothesis</td>
</tr>
<tr>
<td>Read</td>
<td>Document/design/code/etc</td>
</tr>
<tr>
<td>Search</td>
<td>Document/design/code/etc</td>
</tr>
<tr>
<td>Understand</td>
<td>Document/design/code/etc</td>
</tr>
<tr>
<td>Request</td>
<td>Information</td>
</tr>
<tr>
<td>Justify</td>
<td>Solution/goal/intention</td>
</tr>
<tr>
<td>Develop</td>
<td>Solution</td>
</tr>
<tr>
<td>Evaluate</td>
<td>Solution/strategy/</td>
</tr>
<tr>
<td>Explain</td>
<td>Solution</td>
</tr>
<tr>
<td>Review</td>
<td>Solution</td>
</tr>
<tr>
<td>State</td>
<td>Solution/constraint/goal/intention/irrelevant comment/inference/conclusion</td>
</tr>
<tr>
<td>Manage</td>
<td>Goal/meeting/schedule</td>
</tr>
<tr>
<td>Plan</td>
<td>Goal/local activity</td>
</tr>
<tr>
<td>Simulate</td>
<td>Mental model/dynamic action</td>
</tr>
<tr>
<td>Identify</td>
<td>Error/omission/misunderstanding</td>
</tr>
<tr>
<td>Elaborate</td>
<td>Hypothesis/solution/goal/mental model</td>
</tr>
<tr>
<td>Modify</td>
<td>Solution/strategy</td>
</tr>
<tr>
<td>Reformulate</td>
<td>Problem/goal/hypothesis</td>
</tr>
</tbody>
</table>

Figure 4. Sub-set of AFECS (von Mayrhauser and Lang 1999) updated by the authors (entries in italics).
pairs that constitute a fraction of the customised general-purpose encoding scheme.

Von Mayrhauser and Lang (1999) argued that a general-purpose scheme could act as source data for (yet to be fully developed) tools that would automate the ensuing analysis process. For example, in a research project to investigate the test hypothesis and discard hypothesis behaviour of novice analysts, a tool could determine ‘which was the most common action to follow work upon a hypothesis?’ by searching for all actions which follow those of type *.hyp.*.* and calculating the most common type.

4.2. Analysis tools

The potential value of a tool to automate the protocol analysis procedure has long been recognised. The development of a Protocol Analyst’s Workbench (PAW) was reported (Fisher 1987) as a ‘multipurpose computer-aided protocol analysis system’. It included a user-defined vocabulary to aid in the reliability of encoding. Additionally it provided analysis assistance by generating a process model from the protocol data, for example allowing the comparison of flow models to consider the sequencing of operators.

Fisher and Sanderson (1996) provided a review of Exploratory Sequential Data Analysis (ESDA) tools developed to make data analysis easier. (The authors use the term ESDA in consideration of various analysis techniques, including verbal protocol analysis.) A number of these tools feature video data analysis, but few appear to be in common use. One exception appeared to be macSHAPA, developed by Sanderson at the University of Illinois. Coding schemes could be developed with the software and then used to tag data files or video records, before subsequent statistical analysis. MacSHAPA has been used in a small range of studies, for example investigating use of memory by expert programmers during navigation of large information displays (Altmann et al. 1995), and investigating the problem solving of novice engineering students (Atman et al. 1996, Adams and Atman, 2000). However no further developments of the software have been reported since 1997.

Evidently there is scope for developing software to assist in the verbal protocol analysis procedure, in particular to apply a general-purpose coding scheme suitable for software engineering if such can be validated. A note of caution has been sounded however that although many people are ‘driven to try to develop a software environment to make their task easier’, many such tools are too specific for general use and many fall into disuse (Fisher and Sanderson 1996). It may be unsurprising, therefore, that text analysis software in regular use is general-purpose software well-known in qualitative research, such as QSR N6 (previously marketed as NU-D*IST), ATLAS.ti, and Text Analyst. N6, by QSR International Pty. Ltd. (http://www.qsrinternational.com), provides the means to segment and code textual data, and then analyse the text associated with the codes, for example by searching for text units associated with the intersection of, or union of, or ‘just-one’ of associated codes. It can also be used to search for a particular text pattern, determine its frequency of occurrence, and to code the text units thus found. ATLAS.ti, produced by Scientific Software Development, (http://www.atlasti.de), provides similar facilities for graphical and audio data as well as textual data, and a networking feature to give a diagrammatic representation of semantic relationships between elements such as codes. TextAnalyst by MicroSystems Co. Ltd. (http://www.megaputer.com/) processes text via neural networks to produce a semantic network of concepts from the text and the relationships between them. Simple text searches and natural language queries can be performed, and summaries created. A large number of software packages are available as alternative choices to these; qualitative research WWW sites can be consulted to obtain lists of such content analysis resources.

4.3. Future directions

A primary motive of many of the studies examined has been to contribute towards a demonstrable theoretical underpinning for software engineering in order to advance the maturity of software engineering as a discipline. Whenever new methodologies, tools, and programming languages are devised to facilitate the software development process, it seems likely that verbal protocol analysis will feature as a method used to investigate cognitive factors associated with their use. For instance, advocates of eXtreme Programming believe that paired programmers can create higher quality software products in less time than individuals can (Williams et al. 2000). Verbal protocol analysis may be considered an appropriate technique to investigate the problem-solving behaviour of a pair of designers working side-by-side at one computer, in order that the cognitive processes underlying productivity and quality gains can be formally mapped rather than speculated about. Early research investigated the efficacy of object-oriented analysis and
design as a methodology compared to structured methodologies. More advanced aspects of the object-oriented methodology, such as the use of design patterns and reuse, are candidates for cognitive process investigation. Given the acceptance of the technique as a means of providing insights into the cognitive processes occurring during the performance of problem-solving type tasks, it is anticipated that certain aspects of the software process which have not featured to date (as indicated by table 4), such as validation planning, cost estimation, risk analysis and project management, will begin to feature. Given the on-going importance of improving the software development process, a research technique that can give insights into the difficulties of influences upon, and decisions made in software management should prove valuable.

5. Conclusion

Verbal protocol analysis is a technique used to obtain an accurate insight into a person’s thinking whilst performing a problem-solving task. An analysis of the verbalisations made while ‘thinking out loud’ can contribute understanding of the problem-solving process, and of knowledge and external factors influencing it. This paper has presented a review of the verbal protocol analysis technique itself, and summarised the peculiarities of its application in software engineering research. Data describing the behaviour and cognitive processes of software developers can provide information to assist in the identification of process improvements, and in the development of effective training, teaching, and learning methodologies.

It is evident that the technique has contributed towards the development and testing of models of the information processing that takes place during the software engineering process, particularly those relating to software design and comprehension. However some difficulties associated with the technique are: (i) the effort of devising a valid and reliable encoding scheme; (ii) the time-consuming nature of the encoding process; and (iii) the problem of comparing results from researchers who have applied different encoding schemes. Current research by the authors is investigating the development of a revised version of a common encoding scheme applicable to the software engineering process and which might alleviate these difficulties. With such an encoding scheme the use of verbal protocol analysis will gain in importance as a technique in empirical software engineering research.

References


