Computer Science Education
Vol. 15, No. 1, March 2005, pp. 7–24

Routledge
Taylor & Francis Group

Towards a Framework for Characterising Concurrent Comprehension

Connor Hughes*, Jim Buckley, Chris Exton and Des O'Carroll (SVCR Group)
University of Limerick, Eire

This paper proposes an evaluation framework for assessing students' comprehension of concurrent programs. The need for such a framework is illustrated by a review of various Computer Science Education forums. This review suggests that there is little pedagogical research in the area of concurrent software, particularly with respect to assessing students' knowledge. In particular, the proposed framework attempts to categorize the types of information students seek and the types of information they obtain, as they learn about concurrent software. Thus the framework could be used to guide teaching methods in this area, to hone the representations used to portray concurrent software to students and subsequently, to assess students' performance.

1. INTRODUCTION

The term concurrent system can be used to refer to ''any type of environment allowing the execution of application code on multiple processors simultaneously'' (Erbacher & Grinstien, 1996). These systems can no longer be viewed as the domain of operating systems developers or real-time programmers exclusively, as the inclusion of threads as a first class entity in Java and C# has resulted in wide adoption of concurrent programming practices.

This has been acknowledged by the 'Association for Computing Machinery' (ACM) who initiated the drive towards concurrency-literate graduates as part of its Curriculum 91 recommendations. In this, the association advocated introducing distributed and parallel programming into the undergraduate study curriculum and since then, concurrent programming has become an essential part of many undergraduate curricula. (Feldman & Bachs, 1997; Yang & Jin, 1995; Cunha & Lourenco, 1998). However the Association also noted that many of the comprehension mechanisms that students had established for non-concurrent programming techniques did not migrate well to a concurrent equivalent, an

*Corresponding author. CSIS Dept., University of Limerick, Castletroy, Limerick, Eire.
[E-mail connor.hughes@ul.ie; des.ocarroll@lit.ie]

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/05/010007-18
DOI: 10.1080/08993400500056522

assertion supported by several researchers in the field (Pollock & Jocken, 2001; Bedy et al., 2000).

This assertion was echoed by Dijkstra, in his famous 1968 letter to the ACM. He stated that ''our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed''. This is a particular weakness when we consider concurrency as, for a student to understand concurrent software, it is essential that they must develop a competent cognitive model of how an executing process evolves in relation to time. In addition, concurrent programs express not only one executing computation, but also parallel sets of executing computations, and the interactions (communication and synchronization) among those computations that define the parallelism (Browne et al., 1995).

Another associated complexity with concurrent systems is that they typically introduce non-deterministic behaviour to student programmers. (Pancake, 1992) describes how ''concurrent behaviour is susceptible to subtle variations in processor speed, load balancing, memory latency, the sequence and timing of external interrupts and communications topology.'' She explains how these ''susceptibilities can result in an inherently non-reproducible, non-deterministic behaviour, which is difficult to monitor and even more difficult to analyse'' during errors of deadlock and livelock.

For these reasons, it is widely agreed that, learning about and understanding these systems can be more challenging than learning about and understanding the behaviour of sequential programs (Erbacher & Grinstien, 1996; Erbacher, 2000; Kreamer, 1998). Given this complexity, it is somewhat surprising to find that there are only a small number of the papers in pedagogy forums describing work in this area. In a review of the Computer Science Education journal (CSE) and the proceedings of the Special Interest Group in Computer Science Education (SIGCSE), as presented in Table 1, the authors identified only twelve articles (1.1% of the total) referring to the teaching of concurrency. These articles typically referred to strategic issues, such as the laboratory facilities required (see section 2 for more details). None of the articles refer to the evaluation of students' knowledge of concurrency.

One important aid when assessing the effectiveness of concurrency teaching would be a framework that informs teachers as to their students' knowledge of concurrent systems and the quality of that information. If such a framework was applied at various points along the learning experience, teachers could better direct their teaching effort. In addition, developers of visualization tools could identify weaknesses in their portrayal of concurrent systems.

Software comprehension research has identified several ways in which to characterize the information obtained by programmers when studying systems (Pennington, 1987; Good, 1999). Although these studies have concentrated on sequential code and on the information types exclusively (rather than information type and information accuracy), they can provide a basis for extension into the concurrent-code domain. This is the approach adopted in this paper.

Table 1. Papers on Concurrency Teaching / Portrayal and evaluation of Concurrent knowledge in Computer Science Education Forums.

Source	**C.S.E.**							**S.I.G.C.S.E.**									
Year	**04**	**03**	**02**	**01**	**00**	**99**	**98**	**04**	**03**	**02**	**01**	**00**	**99**	**98**	**97**	**96**	**95**
Total No. of Papers	7	19	17	18	17	17	12	121	99	102	103	98	91	91	93	78	97
Those on Concurrent Teaching	0	0	1	1	0	1	0	0	0	0	1	3	0	3	0	0	2
Those on Evaluating Concurrent Knowledge	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2. EVALUATION OF STUDENTS KNOWLEDGE OF CONCURRENT SOFTWARE

The authors' review of Computer Science Education forums suggests that there has been little research performed on evaluating students' learning of concurrent software systems. Of the articles that addressed teaching concurrency, many looked at strategic issues such as appropriate laboratory facilities for teaching parallel programming (Kessler, 2004), (Jin & Yang, 1995), the appropriate programming language for teaching concurrency (Ben-Ari, 1996) and the relevant issues when introducing concurrent programming into a syllabus for the first time (Hailperin et al., 2000).

A number of the papers did address the teaching methods appropriate for concurrency. Generally these were descriptive (McDonald & Kazemi, 2000; Shene, 1998), but some also included evaluation (Burkhart, 1997; Kessler, 2004; Pollock & Jocken, 2001), in the form of qualitative comments from students and lecturers. While this form of evaluation does provide some information with which to guide teaching practice, it should also be complemented by quantitative empirical evidence (Perry et al., 1997).

2.1. Empirical Studies on Algorithm Animation Tools

One of the teaching methods discussed in the above papers was the introduction of algorithm animating, visualisation systems (Bedy et al., 2000; Stasko et al., 1993). Algorithm Animation tools provide highly application-specific views of a program's data structures, the operations which update these data structures, abstract representation of the computation and its progress (Kreamer, 1998). It is believed that the dynamic, symbolic images in an algorithmic animation help provide a concrete appearance to the abstract notions of algorithmic processing, thus making them more explicit and clear (Kehoe et al., 1999). Consequently, many concurrent program comprehension tools are algorithm animators (Kreamer, 1998), two well-known examples being POLKA (Jerding et al., 1997) and PARADE (Stasko, 1995).

Because a limited number of quantitative studies have been performed in this area (Stasko et al., 1993; Kehoe et al., 1999; Byrne et al., 1996; Hübscher-Younger & Hari Narayanan, 2003) this literature was reviewed to see if it could provide relevant information with respect to the pedagogy of concurrent software. However, these studies provided very limited evidence to suggest that algorithm animations substantially assist in learning algorithms (Wilson & Aiken, 1996; McDonald & Ciesielski, 2002). For example, studies by (Stasko et al., 1993) and (Byrne et al., 1996) failed to find statistically significant improvements in performance, based on the presence of algorithm animation tools.

According to Kehoe (Kehoe et al., 1999) there may be several explanations for the lack of statistically significant findings from such studies:

- That there are no or only limited benefits from animation;
- That there are benefits, but the statistics in the experiments used in the studies are not sensitive to them;
- That something in the design of the experiment is preventing participants from receiving the benefits. In other words, the theory of *how* animations could help needs to be re-examined.

They carried out an empirical study to assess if the third reason could be responsible. Their hypothesis was that the pedagogical value of algorithm animations would become more apparent in open, interactive learning situations (such as a homework exercise) than in closed exam-style situations (Kehoe et al., 1999). Thus, their study measured the effectiveness of algorithm animation in an open homework-style learning environment, in which students were provided the questions at the beginning of the experiment and there was no time limit. To test the learning capacity of the students, questions were based on the operations, definitions, mathematical properties, and running times of the algorithm. The questions were taken from the post-test in (Byrne et al., 1996).

In this study, the students from both the animation and non-animation groups performed similarly on most of the exam questions. One notable difference occurred on questions about concrete instances of the insert, union and extract-min operations on specific examples of heaps. On those questions, the animation students clearly out-performed the non-animation students. Thus algorithm animations seem best suited to help convey the procedural step-by-step operations of an algorithm. They provide an explicit visual representation of an otherwise abstract process, thus raising the congruence of the representation (Kehoe et al., 1999; Good, 1999) for the tested task.

2.2. Open Issues

With respect to students' learning of concurrent systems, this review illustrates a number of salient issues. Firstly, it suggests there has been very limited evaluation of the effectiveness of knowledge portrayal in this domain, either through teaching or by means of visualization tools. Secondly, the empirical data that has been gathered has focussed on informal feedback or measuring the products of learning, rather than focusing on the learning process itself.

Measuring the product of learning provides only a coarse grained measurement of the learning experience, thus limiting the potential to impact on the teaching process. To measure the learning process as it proceeds instead, echoes Kehoe's third explanation for the absence of empirical evidence on the benefits of algorithm animation tools: that a better 'theory of *how* animations could help' is required. To gather data on this, a fine-grained approach is needed and an empirical instrument that tracks comprehension during the learning process would be more useful in this regard. As a result, it was decided that talk-aloud data should be captured as students use various representations to study concurrent

algorithms. This has the potential to illustrate how various teaching representations could help the student and where they might be lacking. A framework for evaluating this data is then required.

3. IDENTIFYING A FRAMEWORK FROM PROGRAM COMPREHENSION STUDIES

Program comprehension, has been defined as ''the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to ones of the underlying application domain, for maintenance, evolution, and reengineering purposes'' (Muller, 1994).

Within software comprehension research, several empirical studies have been performed to evaluate programmers' mental models of software systems (Good, 1999; Pennington, 1987; Corritore & Weidenbeck, 1991; Burkhardt et al., 2002). These studies have employed schemas that characterize the information captured by programmers after they study a software system. However, it would also be interesting to study students' knowledge as they use external representations, to more accurately capture the value of these representations. Here we propose to extend Good's (Good, 1999) information-type schema for assessing programmers' mental models. In applying this framework to students' talk-aloud data, as they view external representations, we have the potential to evaluate the types of information that they obtain and miss during their comprehension process. In extending the schema to cover concurrency, our aim is to identify the impact of the representations on concurrent-system comprehension.

3.2. Good's Schema

Good (Good, 1999) suggests a content-analysis (Krippendorff, 1980) approach to analysing and measuring novice program comprehension. Her work is based on that of Pennington (Pennington, 1987) who carried out content analysis by dividing up program summaries into phrases and classifying each phrase in terms of the type of information it emphasized. This categorization schema was, in turn, based on a theoretical review of the information types available in programs and has been subsequently used in several other works (Corritore & Wiedenbeck, 1991; Bukhardt et al., 2002).

However, Good was concerned that Pennington's empirical study was 'opaque' (Good, 1999) and difficult to replicate because the mechanism used to translate talk-aloud utterances into categories was not explicit. Hence, in addition to expanding the schema, Good produced a coder's manual that described the categorisation process (from utterance to category) in detail. In this manual, the translation process was supported by descriptions of each category, examples of sample utterances for each category, and a decision tree. Her revised information type classification consisted of eleven categories:

- Function: the overall goal of the program – " *The program calculates the differences between the input distances*"
- Actions: goals occurring in the program, which are described at a lower level than function, and at a higher-level than operations – "*This sub-program checks each individual element of this list.*"
- Operations: Small-scale, single line of code actions, which occur in the program, such as assignment – "*Then the program sets the height to head (height)*"
- State-High: A high-level description of the conditions that specify the execution sequence of the code. State-high differs from state-low in terms of granularity. It refers to a condition in an aggregate form – "*if all the elements have been processed. . .*"
- State-Low: A lower-level description of the conditions that specify the execution sequence of the code. Typically state-low utterances refer to the condition in terms of specific elements – "*If the head is greater than 180*"
- Data: This category refers to inputs and outputs to / of the programs, data flow through the programs, and descriptions of data objects and data states – "*The program accepted a list of numbers indicating sun hours*"
- Control: Information having to do with program control structures and with sequencing including recursion, and calls to subprograms – "*And the sub-program is called recursively.*"
- Elaborate: Further information about a process/event/data object that has already been described – "*[If the current mark is above a certain pass level] 65 in this case. . .*"
- Meta: Statements about their own reasoning process – "*. . .I can't remember.*"
- Unclear: Statements, which cannot be coded because their meaning is ambiguous, interpretable – "[*. . .which is the initial class] if your looking at [so it calls QS. . .].*"
- Incomplete: statements, which cannot be coded because they are incomplete.

4. A REFINED COMPREHENSION SCHEMA

This research has refined, and extended, Good's methodology, to extract information that shows the information-types that novice programmers focus on when comprehending a concurrent program, *and* their confidence in their captured knowledge. In addition the analysis is targeted at the data produced by programmers *as* they study a system, providing a more direct measurement of their information-seeking process as they use external representations.

4.1. Alterations and Extensions to Good's Schema

A pilot study was employed to evaluate our prototype schema: that is, it was used to determine how valid the schema was for evaluating comprehension of concurrent systems. The pilot study was also used to refine the experiment method, and as such was carried out in a fairly informal manner where questions and comments by

participants were encouraged. Detailed information about this pilot study can be found in (Hughes & Buckley, 2004).

4.2. Information Types

In its original state, Good's schema concerned itself with characterizing comprehension of serial programs. It did not concern itself with concurrency issues or student learning with respect to the representations available. Thus we extended the schema in several ways. Firstly, our proposed information types classification consisted of thirteen categories. Six of the categories are the same as Good's information types (function, action, operation, control, state-high, state-low) with four other information types (Elaborate, Meta, Unclear and Incomplete) grouped into one, 'bucket' category. The reason for grouping these four categories into one is because they seem of lesser relevance when determining the information types portrayed to students. Other changes are discussed below:

- Due to the objected-oriented nature of Java, the 'data' utterances in the pilot study seemed diverse, and poorly served by one category, a finding also reported by (Burkhardt et al. 2002). This prompted us to refine the data category into three sub-categories:
  - **Simple types**: This information type refers to utterances that focus on primitive data-structures integers, characters etc. – "*so it is an integer*"
  - **Complex types**: This information type refers to utterances that focus on non-primitive data structures and classes – "*it's of class producer*"
  - **Instances**. This refers to instantiated variables of complex and simple types and the data-flow between them - "*there shouldn't be anything in that array yet*"
- The efficiency and safety of a program are major concerns when making programs concurrent. As concurrent programs are created to increase the performance of systems, whilst ensuring correctness, more analysis of these aspects is appropriate (Pollock & Jocken, 2001) and it is possible that student utterances will mention the performance / safety aspects of a concurrent program. Hence three additional information types were incorporated into the schema:
  - **Thread Evaluation**: This sub-category refers to utterances that discuss the safety and liveness of a program. - "*it calls p1 which could cause thread errors*"
  - **Thread Synchronisation**: This sub-category refers to utterances which show that the programmer is thinking about inter-thread synchronization. – "*it waits when it says stack.notifyall*"
  - **Thread Admin**: This sub-category is logically grouped with the other two as it refers to the creation, starting, stopping and destruction of threads. – "*New con.start calls run*"

4.3. Confidence Dimension

A new, orthogonal, confidence dimension had been added to the schema and it specifically relates to the goal of evaluating the representations used. It will serve to identify where students have difficulty in comprehending concurrent systems. Each category in this dimension refers to differing levels of confidence students' have in their utterances:

- Confusion: This category represents a complete lack of understanding and is rated as the lowest level of confidence. An example would be a statement like: "*I don't know what it's saying*".
- Questioning: This category demonstrates a high level of uncertainty in the student's understanding of the code. Typically, it can be identified by question-type phrasing (*are there. . .?, I wonder if. . .?)*.
- Hypotheses: At a slightly higher level of confidence, the programmer can state educated guesses about the system being viewed. Typically, these utterances are signalled by key-phrases such as '*I presume. . .', 'I assume. . .' or 'I think. . .'*.
- Certainty: Phrases in this category demonstrate a very high degree of confidence by the participant. They believe that what they are saying is a statement of fact. It is signalled by key-phrases such as '*it is. . .' and 'so, it does. . .'*.
- Bucket: All other utterances.

However, confidence is only a partial measure of the students' difficulty in comprehending concurrent systems. Consider the scenario where a student makes an incorrect statement about the code, but is certain about it. For example, one participant in the pilot study said that the program was in a continuous loop, and that "the threads seem to keep running, they don't suspend themselves at any time". Under our original coding scheme this would be classified as 2 phrases in the confidence dimension (separated by the comma). The former would be a 'hypothesis' and the latter would be a 'certain'. However, as they were both incorrect, they should not indicate elevated comprehension. In fact, given previous empirical findings that novice programmers tend to cling to incorrect assumptions much more so than experts (Boehm-Davis et al., 1992), such statements could be very damaging in terms of the students' comprehension and should lower their comprehension rating. Hence, the categorization process should only consider correct utterances.

Another concern is that it is easier for a student to be confident when discussing a single line of code (an 'operation' type utterance) than when discussing the 'function' of the entire program. This distinction is accommodated by incorporating the concept of deep understanding in the confidence dimension (Byckling et al., 2004). Deep understanding refers to data, function, action and state-high type utterances (Pennington, 1987; Burkhardt et al., 2000), as these reflect a semantic, non text-based, understanding of the code. To these deep categories we add the 'thread evaluation' and 'thread synchronization' categories of our expanded schema.

5. EMPIRICAL STUDY

An empirical study was performed to assess the schema in terms of its ability to inform on the quality of student comprehension. This study captured eight student programmers' talk-aloud data, which was subsequently categorised into the two coding dimensions described above. As one of the core aims of this work is to ensure that other researchers could replicate the analysis this coding process was based on a coder's manual that described each category, gave examples of utterances for each category and provided a decision tree for deciding the category associated with each utterances. Two independent coders used this manual to perform the analysis and the reliability was assessed to see if the approach could be replicated. The results are given in section 5.2.3. The coding manual is available from the first author.

5.1. Research Questions

The experiment attempts to answer three questions:

- Will utterances reflecting deep understanding of information-types, as identified by (Byckling et al., 2004), correlate with normal pedagogical evaluation? By 'normal' pedagogical evaluation, we mean assessment by a lecturer of students' written summaries of concurrent software.
- Will utterances that reflect a programmer focusing on thread synchronization, live-ness and safety correlate with normal pedagogical evaluation?
- Will correct certainty of deep information-types correlate with normal pedagogical evaluation?

5.2. Experiment Outline

5.2.1. Participants Eight final year computer-systems degree students participated in the study (To be referred to as P1 – P8). All of them were heading towards a first class or 2.1 honours degree. The participants volunteered by replying to a soliciting email. All participants except for one mature student were between the ages 20 – 29 and had attended an algorithms or data structures module as part of their degree. Half of the students have programmed a concurrent Java program before and only three had any industrial experience programming Java (two participants had 3 months while one had 10 months). Finally on a scale of one to five, (five being the highest) six participants rated themselves with a level of expertise in Java of three; the remaining participants gave themselves four. In a generic Java test (available from the first author), after their comprehension sessions, the participants scored an average of 7.3 out of 10. Two participants achieved full marks: P1 and P4.

5.2.2. Source Code The source code used in the study was a multi-threaded Java program which was 86 lines long. It was a producer, consumer algorithm that used a stack as a repository. It had four threads: two consumer threads and two producers.

The producer threads pushed elements onto the stack while the two consumer threads concurrently popped elements off the stack.

5.2.3. Protocol The experiment session was around 45 minutes in duration and was broken into three tasks, a warm-up task, a main task and a generic Java test. An element of competition was introduced into the experiment by awarding the top two performers an mp3 player. Performance was evaluated by two lecturers who marked the participants' textual summaries of the code (the 'normal pedagogical evaluation' referred to in the research questions).

The warm-up task introduced the participants to the talk-aloud protocol. It involved the participants being asked to comprehend a multi-threaded Java program of just over 80 lines of code using their IDE of choice. Three used RealJ, three used notepad and two used Eclipse. The participants were asked to understand the program and comment the code for another programmer. They were requested to think-aloud for the duration of the warm-up experiment, so that they would be more comfortable thinking aloud during their main session. This section of the experiment lasted 10 minutes.

Between the warm-up task and the main task, the participants were given a distracter task, in which they were asked to attempt a crossword puzzle. The distracter task lasted three minutes, to allow the participant time to clear their mind of the warm-up task, and to give the experimenter time to set-up the main task.

The main experiment lasted 20 minutes and required the participants to comprehend and comment a concurrent Java program of 86 lines of code, again using their favoured IDE. Participants were subsequently asked to write a program summary and were informed that this summary would be the basis for evaluation of their performance. All participants were requested to think-aloud for the duration of the main experiment so that their comprehension could be captured. The instruction given was to ''say everything that comes into your head'', in line with the best-practice guidelines by (Ericsson & Simmons, 1980) for gathering talk-aloud data. If at any stage the participant began to work silently, they were prompted, ''what are you thinking now?'' by the experimenter. Each participant returned to the distracter task, for three minutes, after his or her summary was completed. The empirical study finished up with the generic Java test and the background questionnaire discussed earlier.

As one of the aims of this research is to generate an evaluation scheme that can be applied consistently, the reliability of the coding schema was evaluated using kappa tests. Two coders, both experienced in Java, used the coder's manual, to identify the information types and confidence of participants from their talk-aloud records. Both Kappa statistics reported a 'very good' strength of agreement (Landis & Koch, 1977):

- Information-Types: For samples taken off 3 different participants, the Kappa was 0.820.
- Confidence: For samples taken off 4 different participants, the Kappa was 0.825.

Two lecturers who worked independently of the coding process marked the textual summaries, produced by the participants. They were asked to rate the summaries in terms of which one they thought showed the highest level of understanding. The resultant rankings of the participants were averaged and are displayed in the next section.

5.3. Results

The results obtained from the study are presented in tables 2, 3 and 4. Table 2 reports on the confidence dimension, using columns three and four to break down the 'certain' utterances by correctness and 'deep'ness. The 'confusion' utterances noted in this table, in conjunction with several utterances that portray their context seem to be powerful in identifying where the representations of concurrent software systems lack the expressive power required by students.

Table 3 presents the utterances of the students, as categorized by information type. Of the three new 'Thread' categories, only 'Thread Evaluation' was infrequently used. However, as noted by other researchers (Pollock & Jocken, 2000), this may be because analysis of concerns like safety and live-ness reflects a high level of comprehension that procedural programmers would not be familiar with.

As can be seen from the data in Table 3, 'State-high' and 'State-low' utterances were seldom made during these comprehension sessions, but this probably reflects the limited control-flow complexity of the program under study. Table 4 shows the ranking of each participant's summary, where a better ranking is presented as a lower number.

5.4. Result Analysis and Discussion

Our first research question asked if 'Deep understanding information-types would correlate with normal pedagogical evaluation'. To address this question with this

Table 2. Confidence Dimension Utterances in Ranked Order (% of all Utterances) [Figures are rounded to nearest whole number].

	Certanties						
	All	Deep	Deep & Correct	Hypothesis	Questioning	Confusion	Bucket
P1	76	72	68	5	3	6	11
P2	86	63	62	3	3	1	7
P3	79	72	67	5	2	8	5
P4	75	60	59	11	7	3	4
P5	80	61	41	6	8	5	1
P6	81	77	74	9	4	0	5
P7	86	82	76	4	3	1	7
P8	85	69	67	2	0	0	28

Table 3. Information-type Dimension Utterances in Ranked Order (% of all Utterances).

	Information Type												
Part No	Function	Actions	Opera-tions	Thread Admin	Thread Synch	Thread Eval	Complex Type	Simple Type	Instance	State High	State Low	Control	Bucket
P1	0	18	3	5	4	5	3	0	18	0	1	1	42
P2	0	12	24	1	5	0	1	0	19	1	0	5	33
P3	1	16	2	2	5	0	2	0	24	0	0	6	44
P4	0	16	19	2	1	3	4	1	17	1	0	3	33
P5	1	14	2	1	4	0	2	0	21	0	0	5	37
P6	0	21	3	4	1	3	3	1	21	0	0	7	28
P7	1	16	19	2	10	3	4	1	29	1	0	8	28
P8	0	5	11	3	1	0	1	2	26	0	1	10	41

Table 4. Ranking of Participants by Lecturers.

Participant	Lecturer 1 Rank	Lecturer 2 Rank	Overall Rank
P 1	1	2	1
P 2	4	1	2
P 3	3	4	3
P 4	5	3	4
P 5	2	7	5
P 6	5	5	6
P 7	5	6	7
P 8	5	8	8

data, we calculated the total percentage of deep phrases from each participant's talk-aloud summaries, ranked them and compared them to the rankings given by lecturers. Spearman's rank correlation measure was used, due to the ordinal nature of our data. However, no correlation between the deep understanding categories and the overall rank was identified. We feel that this may have been due to the 'concurrency' emphasis of our work. This emphasis may have been picked-up on by the lecturers involved, with the result that they may have looked for a more specific focus (concurrency) from the participant's summaries when assessing them. This is supported by several of the comments made by the lecturers ('Does not display understanding of threads in code', 'Understood some thread elements of code', 'Picked up on threads issue').

If this indeed is the case, then, this should support our second research question: 'Will utterances that reflect a programmer focusing on thread synchronization, liveness and safety correlate with normal pedagogical evaluation.' The Spearman rho for this analysis was 0.938 suggesting a strong correlation in the data. This correlation was significant at the 0.01 level.

Our third research question asked if 'correct assertions, that showed deep understanding (information-types) would correlate with normal pedagogical evaluation.' No significant correlation was established between these two variables, although again this could be related to the lecturers' emphasis on students' understanding of the concurrency aspect of the code.

6. CONCLUSIONS AND FUTURE WORK

In operating systems courses and concurrent programming courses students study various classical synchronization problems and parallel versions of sorting algorithms (Hartley, 1994; Tenenbaum, 1992; Quinn, 1994). Using various representations teachers and visualisation tool providers have attempted to reduce the high learning curve needed to comprehend these highly complex algorithms. However, as shown in our review of computer science education, there is very little empirical evaluation performed to assess these efforts.

This paper moves towards a more quantitative method of evaluation than is present in the relevant literature. In contrast to the post-learning measurement of other papers in the area, it provides a framework that attempts to identify the information types that students focus on, the information types they miss and their confidence of their assertions, *as* they understand concurrent software. It is felt that this framework, when established, will assist teachers in evaluating their ability to portray concurrent systems. It has the potential to inform on the information types that representations show effectively and the information types that they obscure. Visualisation tool developers may also use the framework as a common baseline to allowing comparisons across different tools, and thus evaluate their relative merits and disadvantages of their tools.

The framework is based on Content Analysis and, as a research method Content Analysis has the advantages of spanning both quantitative and qualitative domains. As such it has the possibility of providing a rich insight into the complexities of the programmers thought process. As with all methods it however does not prove to be a silver bullet and suffers from a number of disadvantages, both theoretical and procedural. In particular, content analysis can be extremely time consuming, especially when it is applied in a fine-grained fashion to assess learning as it is performed. Also, the approach is inherently reductive, particularly when dealing with complex utterances. It can be extremely difficult to automate it without losing richness of data and simply finishing up with a word count algorithm that disregards the context in which the word was produced. Once these dangers are understood, Content Analysis does provide an additional method which when used in conjunction with others, may assist in providing a further more complete model of how students learn.

The empirical study reported in this paper refined a previous framework and compared the result to human evaluation, as a benchmark. The refinement process was successful in that the categories in the extended schema were reliably populated by independent coders from the talk-aloud. In addition, there were few utterances that seemed to reside outside the schema space.

The comparison process was partially successful in that there was a correlation between thread synchronization / evaluation utterances and lecturer assessment. However a correlation between deep-understanding utterances and normal pedagogical assessment could not be found, possibly because of the implicit emphasis the researchers may have placed on understanding concurrency, to the ranking lecturers.

Another flaw, hinted at by a brief inspection of the student's summaries, was that the categories emphasized in their talk-aloud data did not particularly correlate with the categories emphasized in their summaries. For example, as you would expect, there were no 'confusion' or 'question' utterances in their summaries. In comparing the process (talk-aloud data) to the product (their textual summaries) there is a level of indirection involved and future work will assess this by performing content analysis on the students' summaries.

Other limitations of the study include the small amount of participants and the small size of the code used in the studies. These factors reduce the representative-ness

of the study and lessen its ability to reflect students' behaviour generally. Another confounding factor to consider is the wide range of individuality displayed in programming domains (Prechelt, 1999). Given that several learning styles have been proposed in the literature, this individuality is likely to be evident in learning concurrency, and other studies should build on the initial evidence provided here to address this issue.

A final limitation of the study was that the ''correctness'' categorization of utterances, discussed at the start of section 5.3, was carried out by only one of the authors, and therefore the reliability of the analysis could not be assessed. This limitation raises questions about the degree to which this particular analysis could be replicated by another researcher.

ACKNOWLEDGMENTS

The authors thank all the participants in the study. This work is funded by IRCSET.

REFERENCES

Bedy, M., Carr, S., Huang, X., & Shene, C.-K. (2000). A Visualization System for Multithreaded Programming. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer Science Education, March 7–12* (pp. 1–5). Austin, Texas, USA.

Ben-Ari, M. (1996). Using inheritance to implement Concurrency. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education, February 15–17* (pp. 180–184). Philadelphia, Pennsylvania, USA.

Boehm-Davis, D.A., Holt, R.W., & Schultz, A.C. (1992). The Role of Program Structure in Software Maintenance. *International Journal of Man-Machine Studies*, *36*, 21–63.

Browne, J., Hyder, K.M.S., Dongarra, J., & Newton, P. (1995). Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, *3*(1), 1995.

Buckling, P., Kuittinen, M., Nevalainen, S., & Sajaniemi, J. (2004). An inter-rater reliability analysis of Good's program summary analysis scheme. In *Proceedings of the 16th Annual workshop of the Psychology of Programming Interest Group, 5–7th April* (pp. 170–184). Carlow, Ireland.

Byrne, M.D., Catrambone, R., & Stasko, J.T. (1996). *Do Algorithm Animations Aid Learning?* Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GITGVU -96-18, August 1996.

Burkhardt, J.-M., Detienne, F., & Wiedenbeck, S., (2002). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, *7*, 115–156.

Corritore, C.L., & Wiedenbeck, S. (1991). What Do Novices Learn During Program Comprehension? *International Journal of Human-Computer Interaction*, *3*(2), 199–222.

Cunha, J.C., & Lourenco, J. (1998). An integrated course on parallel and distributed processing. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer Science Education, February 26–March 4* (pp. 217–221). Atlanta, Georgia.

Erbacher, R.F., & Grinstein, G.C. (1996). *Visualization of data for the debugging of concurrent systems*, SPIE Conference on Visual Data Exploration and Analysis 96.

Erbacher, R.F. (2000). *Visual assistance for concurrent processing*, University of Massachusetts at Lowell, Doctoral Dissertation (1998 CS-3), Lowell, MA 01854.

Ericsson, K., & Simmons, H. (1980). Verbal Reports as Data. *Psychological Review*, *89*(3), 1.

Feldman, M.B., & Bachns, B.D. (1997). Concurrent Programming CAN be introduced into the Lower-Level Undergraduate Curriculum. *SIGCSE Bulletin*, 29(3), 1.

Good, J. (1999). *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. Ph.D. Thesis, University of Edinburgh.

Hailperin, M., Arnow, D., Bishop, J., Chestere, L., & Stein, L.A. (2000). Concurrency the First Year: Experience Reports. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer Science Education, March 7–12* (pp. 407–408). Austin, Texas, USA.

Hartley, S.J. (1994). Integrating XTANGO's Animator into the SR Concurrent Programming Language. *ACM SIGGRAPH Computer Graphics*, *29*(4), 67–68.

Hübscher-Younger, T., & Hari Narayanan, N. (2003). Constructive and Collaborative Learning of Algorithms. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education*, 35(1), 6–10.

Hughes, C., & Buckley, J. (2004). Evaluating Algorithm Animation for Concurrent systems: a comprehension based approach. In *Proceedings of the sixteenth annual workshop of the Psychology of Programming Interest Group, April 5–7* (pp. 193–205). Carlow Institute of Technology, Ireland.

Jerding, D.F., Stasko, J.T., & Ball, T. (1997). Visualising interactions in program executions. In *Proceedings of the 19th international conference on software engineering, May 17–23* (pp. 360–370). Boston, MA.

Jin, L., & Yang, L. (1995). A laboratory for teaching parallel computing on parallel structures. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer Science Education, March 2–4* (pp. 71–75). Nashville, Tennessee.

Kehoe, C., Stasko, J.T., Taylor, A. (1999). *Rethinking the evaluations of algorithm animations as learning aids: An observational study*, Graphics, Visualization, and usability Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, 0280.

Kessler, C. (2004). A Practical Access to the Theory of Parallel Algorithms. In *Proceedings of the thirty-fifth SIGCSE technical symposium on Computer Science Education, March 3–7* (pp. 397–401). Norfolk, Virginia, USA.

Krippendorff, K. (1980). *Notes from Content Analysis - an introduction to its methodology*. Thousand Oaks, CA: Sage.

Landis, J.R., & Koch, G.G. (1977). The Measurement of Observer Agreement for Categorical Data. *Biometrics*, *33*, 159–174.

Letovsky, S. (1996). *Cognitive Processes in Program Comprehension*. In Soloway, R. & Iyengar, I. (Eds.), *Empirical Studies of programmers*. (pp 28–45). Norwood, New Jersey: Ablex Publishing Corporation.it>

McDonald, P., & Ciesielski V. (2002). *Design and Evaluation of an Algorithm Animation of State Space Search Methods*. Computer Science Education Journal, 2002, Vol. 12, No. 4, pp. 301–324. 0899-3408/02/1204-301.

Muller, H. (1994). *Understanding Software Systems Using Reverse Engineering Technology*. (http://www.rigi.csc.uvic.ca).

Pancake, C.M. (1992). *Debugger Visualization Techniques for Parallel Architectures*. In *Proceedings of COMPCON* (pp. 276–284). San Francisco, USA.

Pennington, N. (1987). Comprehension Strategies in Programming. In Olson, Sheppard, & Soloway (Eds), *Emperical studies of Programmers: Second Workshop* (pp. 100–114). Norwood, New Jersey: Ablex Publishing Corporation.

Perry, D., Porter, A., & Votta, L. (1997). *A Primer on Empirical Studies*. Tutorial Presented at the International Conference on Software Maintenance.

Pollock, L., & Jocken, M. (2001). Making Parallel Programming Accessible to Inexperienced Programmers through Cooperative Learning. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education* (pp. 224–228). Charlotte, North Carolina, USA.

Price, B.A., Small, I.S., & Baecker, R.M. (1992). A taxonomy of software visualization. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*.

Quinn, M.J. (1994). *Parallel Computing: Theory and Practice*. New York: McGraw-Hill.

Shene, C.K. (1998). Multithreaded Programming in an Introduction to Operating Systems Course. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer Science Education, Feb 26–March 1* (pp. 242–246). Atlanta, Georgia, USA.

Stasko, J.T., Badre, A., & Lewis, C. (1993). Do animations assist learning? An empirical study and analysis, *Interchi*, 61–66.

Stasko, J.T. (1995). *The PARADE Environment for Visualizing Parallel Program Executions: A progress report*. Technical Report GIT-GVU-95-03, College of Computing, Georgia Institute o Technology.

Tenenbaum, A.S. (1992). *Modern Operating Systems*, New York, Prentice Hall.

Wilson, J., & Aiken, R. (1996). Review of animation systems for algorithm understanding. In *Proceedings of the 1st conference on Integrating technology into computer science education* (pp. 75–77). Barcelona, Spain.

Yang, L., & Jin, L. (1995). Integrating parallel algorithm design with parallel machine models. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer Science Education, March 2–4* (pp. 131–135). Nashville, Tennessee.