Evaluating Algorithm Animation for Concurrent Systems: A Comprehension-Based Approach

Connor Hughes & Jim Buckley
SVCR Group
CSIS Department
University of Limerick
{connor.hughes, jim.buckley}@ul.ie

Keywords: POP-II.A. Novices; POP-II.B. Program Comprehension; POP-III.B. Java; POP-IV.A. Concurrency; POP-V.B. Observation.

Abstract

For over 15 years visualization tools have attempted to present the complexity of concurrent programs in easily digestible formats. For example, visualization tools, that display an execution-based animation of concurrent algorithms, have been used extensively in educational contexts to illustrate the behavior of concurrent algorithms to students. However, there is little documented evidence that such tools significantly improve the users' comprehension of the concurrent code.

This paper proposes an evaluation method for determining programmers' comprehension of concurrent systems. It is based on a review of current algorithm animation tools and on existing measures of comprehension. The resulting method proposes a framework within which creators of algorithm animation tools (and of other tools that support the understanding of concurrent systems) can evaluate their products.

Introduction

The term concurrent system is used here to refer to "any type of environment allowing the execution of application code on multiple processors simultaneously" (Erbacher & Grinstien 1996). The use of concurrent systems is becoming "increasingly widespread in all types of fields and occupations" (Erbacher & Grinstien 1996). Erbacher (2000) states that, "with the advent of symmetric multiprocessors in desktop machines, a new generation of concurrent systems is coming to reign".

A number of different architectures for the construction of parallel computers have been proposed. The most widely used classification for such architectures is the one proposed in Flynn (1972), who has given a classification based on the way data and instruction streams are processed (Santos Nicolau 2002). Flynn defines four main categories; SISD (single instruction stream, single data stream), SIMD (single instruction streams, multiple data stream), MISD (multiple instruction streams, single data stream) and MIMD (multiple instruction streams, multiple data streams). Of these categories SIMD and MIMD are the most widely used in parallel computing (Akl 1989).

SIMD is a model where a single sequence of instructions is applied to multiple independent data streams; computers with this architecture will have each processor executing the same instruction, but each on it's own data. MIMD is a model, that describes machines that can execute different types of instructions on different sets of data; each processor executes a different instruction on different data (Santos Nicolau 2002).

Thus, shared memory systems are the most commonly found examples of concurrent systems, but whether the system is using shared memory or not, they typically are large, complex and produce vast amounts of data (Kreamer 1998).

Consequently, understanding the behaviour of these systems is more challenging than understanding the behaviour of sequential programs (Appelbe et al 1991, Cox & Roman 1991, Erbacher & Grinstien 1996, Erbacher 2000, Kreamer 1998). This difficulty is exacerbated because parallel programs must express not only sequential computations but also the interactions (communication and synchronization) among those computations that define the parallelism (Browne et al 1995). Concurrent behaviour is susceptible to subtle variations in processor speed, load balancing, memory latency, the sequence and timing of external interrupts, communications topology etc. These susceptibilities create an inherently non-reproducible, non-deterministic behaviour, which is difficult to monitor and even more difficult to analyse (Pancake 1992).

Given the additional complexity of concurrent systems visualisation of concurrent programs is a subject that has been probed by many computer scientists (Stasko 1990, Brown 1992, pancake 1995, Erbacher 2000). In these studies they try to use visualisation to reduce these systems' inherent complexity. In the authors' review of visualization tools, over seventy such concurrent visualisation tools were identified, and the number is growing. This review suggests none of these tools have been adopted by industry or been empirically validated as reducing the comprehension complexity.

One means by which the maturity of such systems could be elevated is by providing an evaluation framework that informs developers as to the information portrayed by their visualisation tools and the quality of that information. This paper moves towards such an evaluation framework by reviewing algorithm animation tools and software comprehension research. Given the non-deterministic behaviour of concurrent systems when executing, algorithm animation tools would intuitively seem to be an effective aid when comprehending 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 information type as opposed to information quality, they can provide a basis for extension into the concurrent-code domain.

This resultant framework was then assessed by means of a pilot study. This pilot study is described and the findings of the study, in terms of refining the schema, are discussed.

Algorithm Animation Tools

Most tools that have been created as concurrent program comprehension tools are algorithm animators (Kreamer 1998). Algorithm Animation systems 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). Alternatively, (Stasko 1990) defines algorithm animation as the process of abstracting a program's data, operations, and semantics, and creating dynamic graphical views of those abstractions. The video Sorting out Sorting, presented at SIGGRAPH '81, is generally credited with initiating the field of algorithm animation (Byrne et al 1996)

Algorithm animation systems provide facilities for users to view and interact with an animated display of an algorithm (Brown 1992). The main use and reason for the development of algorithm animations was to be used as teaching aids to help explain how specific algorithms work (Stasko et al 1993). The best-known algorithm animation system and the pioneer for algorithm animation is Balsa (Brown & Sedgewick 1985), and its descendant Balsa-II (Brown 1988). Balsa's approach to animating algorithms is for a programmer to annotate the algorithm with markers that identify the fundamental operations to be displayed.

This interactive, workstation-based system allows users to watch a spatial, high-level representation of data structures in a running Pascal program. The current statement(s) being executed is(are) highlighted in the source code and each data structure graphic reflects its current contents. The user can stop and start the program at any time, as well as control the program speed and run the program backwards (Price et al 1992). Balsa-II evolved from Balsa to support colour (to increase the distinction between data), to support rudimentary sound and to provide a scripting facility. Both tools allow the user to view concurrent algorithms. In 1998 Brown began to create Zeus, which provides support for watching and hearing a program in action, through several different views. Again, the programmer animating an application, provides a description of the application's fundamental operations, called (as in Balsa) "*interesting events.*" Zeus is one of the earliest systems to use sound or audio, typically to reinforce visuals. In a sorting algorithm, as the data is being sorted into its specific tables, a different sound is emitted depending on which table the data is sorted into.

Stasko (1990) contributed an animation model with precise semantics called the path/transition paradigm. The focus of the path/transition paradigm is creating smooth, continuous image movement. This is accomplished by conceptually viewing all types of animation as an image moving along a path of incremental changes. To implement the path/transition paradigm, Stasko developed a textual algorithm animation system called Tango. Algorithm animation construction using Tango is, like Zeus, based upon Balsa's concept of identifying interesting events in an algorithm (Carlson & Burnett 1996). Tango eventually evolved to XTANGO, which has additional features, one of the most important being its ability to animate concurrent algorithms

Empirical Studies on tools

The general belief is that algorithm animations succeed in explaining how algorithms and concurrent code works. It's believed that the dynamic, symbolic images in an algorithm animation help provide a concrete appearance to the abstract notions of algorithm processing, thus making them more explicit and clear (Kehoe et al 1999). However, very little empirical research has supported this (Stasko et al 1993, Kehoe et al 1999, Byrne et al 1996, Hübscher-Younger & Hari Narayanan 2003, Hundhausen et al 2002). Our research review suggests that few empirical studies have been carried out on such tools these showed little, if any advantage when using algorithm animations to assist in learning algorithms.

A study conducted by Stasko et al (1993) used an interactive animation to teach the pairing heap data-structure to computer science students. The results showed a "non-significant trend favouring the animation group" in scores on a post-test used to evaluate understanding. The authors attributed the poor result to the possibility that the visualization represented an expert's understanding of the algorithm but not a novice's.

Another study that found limited effects for undergraduates using interactive animations was conducted in 1996 (Byrne et al 1996). This study carried out two experiments to examine the general claim that animations can help students learn algorithms more effectively. In this study, half of the participants were asked to make predictions on the behaviour of the animation. This was implemented because the authors believed that algorithm animations might encourage predictions of what is going to happen at each step of the algorithm.

Experiment 1 started with the participants being shown a six minute videotaped lecture on depth first search given by one of the authors. After watching the videotape, participants were given a three-page text describing the depth first search algorithm. The 88 participants were split into 4 groups, one group per condition. The first factor consisted of animation vs. no animation. Participants who were assigned to the animation condition watched an animation of the depth first search. Those in the no-animation condition worked with static paper materials (pages with illustrations of the algorithm). The second factor consisted of prediction vs. no prediction; participants in the prediction condition made a series of explicit predictions during training about the behaviour of the depth first search algorithm, those in the no-prediction condition did not. This is illustrated in fig.1. Participants then had an unlimited amount of time to work on a post-test. Questions on the post-test were divided into two categories: difficult and easy. An example of an easy question would be that the participant was required to determine the next step in a search. Difficult question were those that involved complete searches of novel graphs. Questions on the post-test were scored stringently as being either completely correct or incorrect.

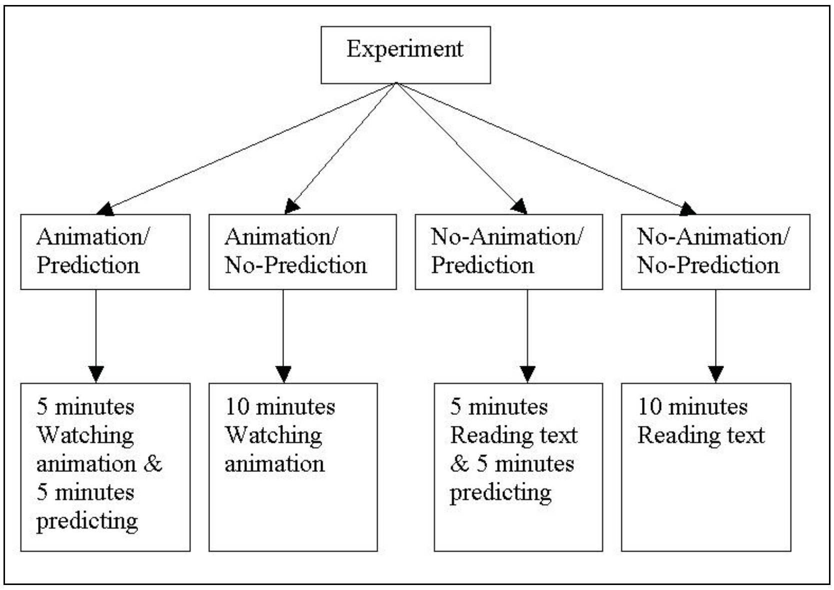
Experiment
Animation/ Prediction
Animation/ No-Prediction
No-Animation/ Prediction
No-Animation/ No-Prediction
5 minutes Watching animation & 5 minutes predicting
10 minutes Watching animation
5 minutes Reading text & 5 minutes predicting
10 minutes Reading text

Fig 1. – Breakdown of experiment 1. carried out in Byrne et el 1996

The second experiment was similar to experiment 1. Participants first watched a videotape of an 11-minute lecture on binomial heaps. After watching the videotape, participants were given a 12-page text describing binomial heaps. Participants were again broken into four groups as in experiment 1. Participants in the no-animation/no-prediction condition were given 35 minutes to read the text. The other 3 groups were given 20 minutes to read it. Those who received only 20 minutes with the text then either simply watched the algorithm animation (animation/no-prediction condition), made explicit predictions while watching the animation (animation/prediction condition), or made explicit predictions from printed graphs (no animation/prediction condition). All participants then received the post-test, on which they had 25 minutes to work.

The results of experiment 1 were not statistically reliable. However, the authors did state that the algorithm animation was beneficial to those who used it to answer the challenging questions in the post-test. Experiment 2 did not show that the animation benefited the participant when answering the post-test questions.

According to Kehoe there may be several explanations for the lack of statistically significant findings:

- 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, or
- 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.

In 1999, Kehoe et al carried out an empirical study to assess if the third reason could be responsible. There hypothesis was that the pedagogical value of algorithm animations would be more apparent in open, interactive learning situations (such as a homework exercise) than in closed exam-style situations (Kehoe et al 1999).

The 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. Twelve students participated in the study, all volunteers and all graduate students. The students were divided into two groups and provided with learning materials about binomial heaps, with one group having access to algorithm animations and the other with learning materials such as still figures of the operations' key points, but no algorithm animation. To test the learning capacity of the students, questions were asked on binomial heaps about operations, definitions, mathematical properties, and running times. The questions were actually taken from the post-test in Byrne et al (1996).

The animation that was used was the priority queue implemented with a binomial heap. 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, providing an explicit visual representation of an otherwise abstract process raising the congruence of the representation (Green 1977, Kehoe et al 1999 & Good 1999)

Lessons Learnt

Algorithm animations provide visualisations to data structures and operations, which do not have any pre-existing visual basis. So, animation is being used not only to explain a dynamic process but also to depict entities without existing visual representations (Kehoe et al 1996). To visualise these entities appropriately for novices is a large barrier in the creation of algorithm animations. As stated earlier, Stasko stated that the visualisations displayed represented an expert's view of the algorithm, not a novice (Stasko et al 1993). As experts define most of the algorithm animations, to inform novices, this suggests that evaluation is a core element in the definition of powerful algorithm animation tools.

Much can be learnt from Kehoe, Stasko and Taylor's study (1999). The study, in part, validated computer scientists who believe that animations improve the learning of algorithms. However our review suggests that this study is alone in providing evidence for the utility of algorithm animation tools. In addition, the questions posed did not assess the algorithm's concurrency per se and so no studies have been used in the domain where algorithm animation tools would seem to have high utility.

Program Comprehension Theories

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).

Theories and Models of Program Comprehension

Current research suggests that programmers attempt to understand code using predominantly two main strategies (Brooks 1997, 1983, Von Mayrhauser & Vans 1995, Pennington 1987).

The first of these strategies, commonly known as the 'top-down approach', suggests that the comprehension process is one of reconstructing knowledge about the domain of the program, and mapping that to the actual code itself. Initially high level domain goals are hypothesized, and these goals are broken down into sub-goals. These sub-goals will be searched for, and the programmers will try to identify their existence in the implementation (Brooks 1983).

The other strategy, known as the 'bottom-up approach', suggests that understanding a program is built from the individual lines of code, by reading the code and then mentally chunking or grouping these statements into higher-level abstractions (Pennington 1987). Initially this chunking is based on the source code representation and later on its mapping to the domain.

However, it is unlikely that any programmer exclusively relies on one of the mentioned strategies. It has been proposed that the programmer subconsciously chooses one of these to be there predominant strategy, based upon their knowledge of the domain under study (Von

Mayrhauser & Vans 1997, O'Brien & Buckley 2001) but free to switch as suitable cues become available to them (Letovsky 1986).

O'Brien (2003) mentions that although all software comprehension models differ significantly in their emphasis, they all consist of four common elements, a knowledge base, a mental model, external representation and some form of assimilation process. External representations are any external views, which assist the code comprehension. For example documentation, the source code, experts advice and comments in the code could all be considered external representations. In the context of this research, external representations are the representations of the concurrent systems that algorithm animation tools portray to the programmer. The mental model is a programmer's existing understanding of the system under study (O'Brien 2003). Hence, these 2 elements are the most relevant in our efforts to build an evaluation framework for animation algorithm tools.

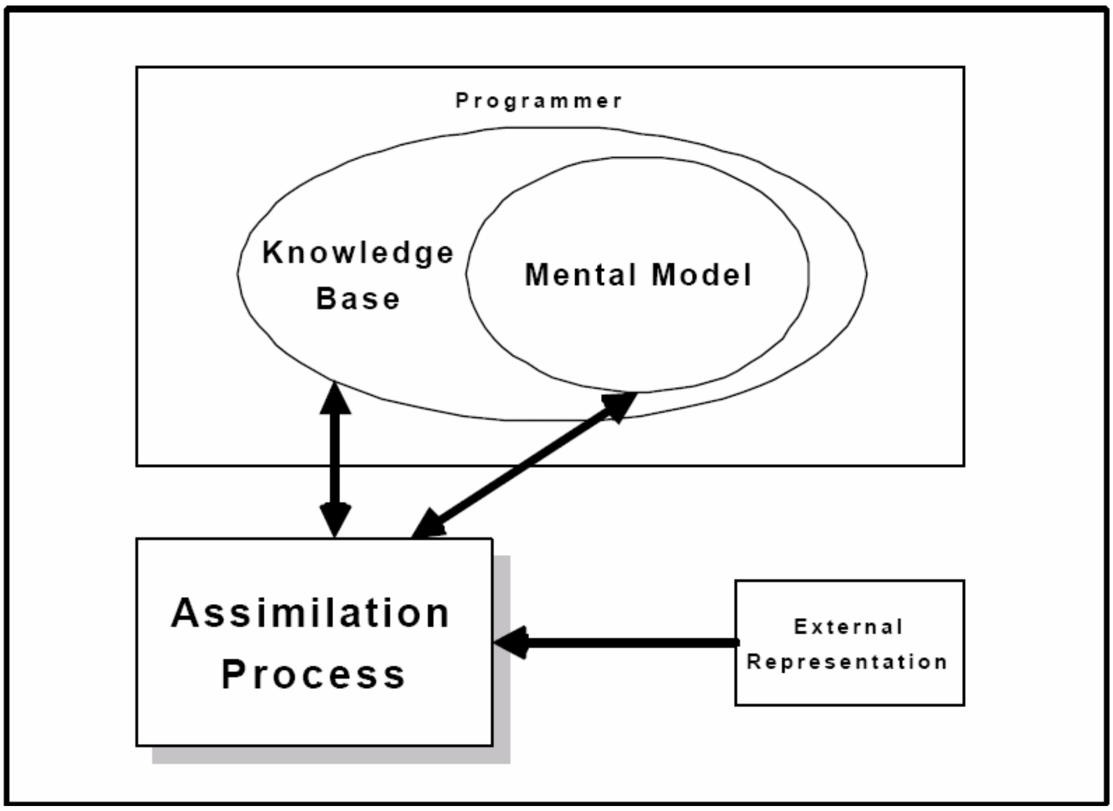

Fig 2. – The main components of a software comprehension model

Permission of O'Brien (2001)

Several empirical studies have been performed to evaluate programmers' resultant mental models of their systems (Good 1996, Pennington 1987 and Ramalinghan & Widenbeck 1998). These studies have generated schemas that characterize the information captured by programmers after they study a software system. However, it would be interesting to study programmers' knowledge as they view external representations, to more accurately capture the value of these representations. Here we propose to extend Good's (1999) information-type framework for assessing programmer's mental models. In applying this framework to programmers' talk-aloud data, as they view external representations, we can evaluate the types of information that programmers use when trying to understand systems. In extending the schema, our aim is to identify the impact of concurrency on system comprehension and to evaluate the quality of the information captured by programmers.

Good's Schema

Good (1999) suggests a content-analysis (Krippendorff 1980) approach to analysing novice program comprehension, which is based on lessons learned from previous approaches. Her methodology in assessing learning was to gather retrospective program summaries and analyze them. It was based on previous work carried out by Corritore and Wiedenbeck (1991) and more specifically, Pennington (1987).

Pennington carried out her analysis by dividing up program summaries into statements. She then performed two analyses on the program summaries, classifying each statement by both information type and level of detail. Pennington's information type classifications were:

- *Function*: The main goal of the program is described.
- *Control flow*: The execution sequence of a program, the order in which actions will occur.
- *Data flow*: The series of transformations that data objects (variables, files) undergo from there initial states to the final program outputs.
- *Operations*: The actions the program performs at the level of individual lines of code.
- *State*: The conditions that specify the execution sequence of the code.

In terms of level of detail analysis, four levels were defined:

- *Detailed* statements contained references to specific program operations and variables.
- *Program* level statements referred to a program's procedural blocks such as a search routine or to files as a whole.
- *Domain* level statements referred to real world objects such as cables and buildings.
- *Vague* statements did not have specific referents.

From analysing Pennington's analysis, Good's main criticism was a lack of clarity in how these categories could be obtained from programmer's summaries. Good stated, "This is not to say that the analysis scheme itself is necessarily inadequate in some ways, but the absence of detail does not allow this to be ascertained" (Good 1999). Good proposed two new schemes on the back of this information. The classification is similar to Pennington's but the information types classification is more finely grained. The object classification is essentially a more restricted version of Pennington's level of detail. The revised information type classification now consists of eleven categories as opposed to Pennington's five, as listed below:

- Function: the overall aim of the program – " The program calculates the differences between the input distances"
- Actions: Aims 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 events, which occur in the program, such as tests, assignment, etc. – "Then the program sets the height to head (height)"
- State-High: A high-level definition of the notion of state: a test condition being met. State-high differs from state-low in terms of granularity – if "all the elements have been processed…"
- State-Low: A lower-level version of the category stated above.– "If the head is greater than 180"
- Data: Inputs and outputs to 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 e.g. recursion, calls to subprograms, stopping conditions – "And the sub-program is called recursively."

- Elaborate: Further information about a process/event/data object which has already been described – “[If the current mark is above a certain pass level] (65 in this case)…”
- Meta: Statements about the participant’s 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, e.g. unfinished sentences – no example was given

Good’s aim for using this classification was to “look at the way in which objects are described”. The main distinction being made is between program objects and domain objects. Goods’ object classification comprises of seven categories, as opposed to Pennington’s three levels of detail. Pennington’s program and detailed classifications have been collapsed into one “program” classification, while other finer-grained distinctions were introduced.

Concurrency and Good’s Comprehension schema

This research aims to use, and extend, Good’s methodology, to extract information that would show the information-types that novice programmers focus on when comprehending a concurrent program, and their confidence in their captured knowledge. We believe that analysis of program summaries will not sufficiently aid us in discovering information on how effective representations are in facilitating novice programmers’ comprehension of a concurrent program. Our opinion is that, to gather (the most relevant) information with respect to tool evaluation, concurrent talk-aloud data must be employed. In other words, if we are trying to capture the programmers comprehension process the best way to do this is to gather information from the programmer while he/she is conceptualising/comprehending the program. Once we have gathered the talk aloud data, we then intend to apply our modified version of Good’s methodology to it.

In the sections below, alterations to Good’s schema are proposed and evaluated by means of an informal pilot experiment. Lessons learnt from the pilot, with respect to refining the schema are documented.

Proposed Alterations and Extensions to Good’s Schema

In its current state, Good’s schema does not concern itself with concurrency issues or tool evaluation. Thus we have extended the schema in several ways. Firstly, our proposed information types classification now consists of twelve categories. Eleven of the categories are the same as Good’s information types with one added: Efficiency/Effectiveness: This information type is not explicit to concurrent programming but is a major concern when making programs concurrent. As concurrent programs are created to increase the performance of systems, it is possible that more utterances will mention the performance of a concurrent program. Hence the inclusion of this category – *“This simultaneous processing must reduce the speed of the program…”*
Two new, orthogonal, coding dimensions have been added. The first describes the level of detail with respect to the concurrency used in a program. Each category in this dimension illustrates how deeply novices delve into the concurrency of the programs, and the amount of time they spend at each level:

- Program Level: This is the master or the highest level in a concurrent program. This level is where a programmer creates and initializes variables, lists and arrays that are to be used in a program.
- Admin-Thread Level: This is the level where threads are defined, created, synchronized, and / or destroyed.

- Inter-Thread Level: This level includes all utterances that involve the subject mentioning cross thread actions. E.g. Comparisons of one thread to another and checks for deadlock etc.
- Intra-Thread Level: The lowest level is the intra-thread level. This level is where the (thread) client work is done. All utterances about simple calculations (variable swapping etc) occurring exclusively within a thread are included here.

The second orthogonal dimension is one of confidence and this specifically relates to the goal of evaluating the representations used. Each category in this dimension refers to differing levels of confidence in programmers' utterances:

- Questioning: This category demonstrates the high level of uncertainty in the participants understanding of the code. Typically, it can be identified by question-type phrasing (are there…?, I wonder if…?) or can be based on intonation of the utterance, where the participants tone elevates as they finish their phrase (although this analysis does require access to the spoken material).
- 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 in the participant. They believe that what they are saying is a statement of fact. It is signalled by key-phrases such as 'it is…., 'so, it does….'.

Pilot Study

The pilot study described in this section was primarily employed to evaluate the schema for our requirements: that is, it was used to determine how valid the schema was for evaluating comprehension of concurrent systems. However, it was also used to refine the experiment method, which may be used as a basis for future studies. As such was carried out in a fairly informal manner where questions and comments by participants were encouraged.
Two post-graduate students in computer science were used as participants for the study (here on in, referred to as P1 and P2). After they had signed a consent form, they were informed that they would be asked to study a small software system and summarize it, retrospectively. They were then presented with a hard-copy, source-code representation of this program, a multi-threaded quick-sort algorithm to try and understand, for 15 minutes. The code was in java and was 80 lines long.
The subjects were requested to think aloud for the duration of the experiment so that their utterances could be captured. If at any stage they began to work silently, they were simply prompted, "What are you thinking now?" by the experimenter. They were then asked to write a summary of the program and fill out a short profiling questionnaire. From this profile, both participants had studied java, but only one had programmed concurrent applications and done a 'Data-structures / Algorithm' course as part of their undergraduate degree program (P1). Neither of the programmers used the full fifteen minutes allocated. P1 took eleven minutes, and P2 took seven minutes. At this stage, they felt they had comprehended the code sufficiently to write a summary of the program. Unfortunately, a review of their summary indicated that the participants had only a very general understanding of the overall function of the algorithm. P2 stated in the summary that it was 'some sort of sorting' algorithm. P1, also alluded to this sorting function with statements like'…that sorts the array…'

Result Analysis

Due to the pilot nature of this experiment, the first author encoded the talk-aloud transcripts alone. However, in doing so, he adhered to the decision processes defined in a 'coder's manual' which was generated for the experiment. This manual describes how to translate utterances into the categories of the schema, and provides concrete examples to facilitate this encoding. This 'coder's manual' is available, on request from the first author, and while it is still in its prototype stages, it does provide a degree of transparency over the coding process.

Unfortunately, due to the fact that there was only one encoding of the data, the reliability of the findings cannot be guaranteed. The findings are presented in table 1.

Confidence Dimension

Participant	Questioning	Hypothesis	Certainty	Total
P1	1	18	124	143
P2	2	7	60	69

Information Type

Part No	Function	Actions	Operations	State-High	State-Low	Data	Control	Efficiency/ Effictiveness	Elaborate	Meta	Unclear	Incomplete	Total
P1	0	9	24	1	7	48	28	0	19	8	3	0	147
P2	0	3	13	6	5	18	8	0	9	9	1	0	72

Utterance Focus

Participant	Program Level	Admin thread Level	Inter thread level	Intra thread level	Total
P1	79	16	2	45	142
P2	53	13	2	1	69

Table 1: Results from the Data Analysis

The reason that the analyses totals are different across rows is that there are no ‘bucket’ (Good 1999) categories associated with the ‘Confidence’ dimension and the ‘Utterance’ dimension. However, on first viewing, it seems that all the new categories are relevant, bar perhaps the ‘efficiency/effectiveness’ category. This is probably due to the fact that the participants had still only achieved a basic understanding of the algorithm and were not at a stage where they could start evaluating it.
One surprising finding is the high level of certainty associated with participants’ utterances, especially given the general nature of their summaries. On closer inspection, certainty seems to be associated with lower level descriptions of the code, such as ‘operation’ type statements. Participants often described single lines of code with certainty, even when referring to ‘data’, ‘control’ and state-low’ type utterances. This suggests that the confidence dimension should be analyzed with respect to only certain (more aggregate) information-type utterances.
Another surprising finding was the low level of inter-thread focus. Again, this seems to be related to the general understanding that the participants developed of the code. Specifically, while they had identified the overall functioning of the system, they were not very aware of the mechanics of the algorithm used or its threaded-ness.

Pilot-Based Refinements

The pilot experiment suggests some obvious refinements to make before the next experiment. The first refinement is based on the representation given to participants. Giving a hard copy of the code to the participants was too limiting, particularly given the non-deterministic, complex, execution behaviour of concurrent systems. A soft copy in an executable environment would be more familiar to the participants and would increase their ability to evaluate what the code was doing. The static nature of a hard copy representation serves to obscure the complex execution behaviour of concurrent programs (Kehoe et al. 1999).
In terms of experiment mechanics, a number of points were observed. P2 suggested that there were no comments in the code, which they found frustrating. This will not be amended for a future experiment as they were omitted purposely to reduce the ease in which the participants could discover the goal of the program. Also, as both participants used only seven and eleven minutes of the allocated time while getting a general understanding of the system, we believe

that 15 minutes is satisfactory, even when seeking more comprehensive summaries from participants. However, a follow-on point is that achievement (in terms of comprehension) must be stressed to future participants. Both participants in this study failed to gather a comprehensive understanding of the code, and while this can inform us as to the difficulties of comprehending such systems, we also need information generated by participants who did manage a comprehensive understanding. This will allow comparisons across the reasoning of both groups. Possible solutions include having a comprehension competition where the summaries are independently evaluated and prizes are given for succinct algorithm descriptions.

Another reason for the study was to refine the coding schema. The first and most obvious refinement is the fact that there is no correctness category in the confidence dimensions. For example, P1 uttered 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 under the confidence category as 'certainty' but, as it is an incorrect statement, it should not indicate elevated comprehension. Hence, we need to combine our confidence dimension with a correctness measure.

As mentioned above, in this pilot, a high amount of certainty utterances were produced. This was because participants simply reiterated lines of code. In future a refinement of this dimension is that it should only be applied when statements are of an information-type higher than operational (or more generally 'Line of Code') level.

Conclusion

In operating systems courses and concurrent programming courses students study various classical synchronization problems and parallel versions of sorting algorithms (Hartley 1994b, Tenenbaum 1992, Quinn 1994). Using algorithm animators, several computer scientists have attempted to reduce the high learning curve needed to comprehend these highly complex algorithms. There is insufficient empirical evidence supporting the assertion that these tools actually reduce the difficulty of the comprehension process.

This paper moves towards an evaluation method for determining programmer's comprehension of concurrent systems. It attempts to identify the information types they focus on, the confidence of their assertions and their focus with respect to concurrency. It is felt that this framework, when established, will assist algorithm animation creators in evaluating the ability of their tool to facilitate comprehension of concurrent systems. Future developers may also use the framework as a common baseline when extracting relevant information from their studies of programmers, thus allowing comparisons across studies.

References

Akl, Selim G. (1989). *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, NJ: Prentice Hall.

Applebe, W.F. Stasko, J.T. Kraemer, E. (1991). Applying program visualisation techniques to aid parallel and distributed program development, (work-in-progress) Technical Report GIT-GVU-91-08, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6), 543-554.

Brooks, R (1997). User interface design activities. *The Computer Science and Engineering Handbook* 1997: 1461-1473

Brown, M. H. & Sedgewick, R. (1985), Techniques for algorithm animation, *IEEE Software*, Vol. 2, No. 1, Jan 1985, pp28-39

Brown. M. H. (1988). Exploring algorithms using Balsa-II, *Computer* 21, 5 (May 1998), 14-36

Brown, M. H. (1992). Zeus: a system for algorithm animation and multi-view editing, in *Proceedings of the 1991 IEEE Workshop on Visual Languages*, IEEE Press, Los Alamitos CA, 1991, pp. 4-9

J. Browne, K. M. S. Hyder, J. Dongarra, and P. Newton, (1995). Visual programming and debugging for parallel computing, *IEEE Parallel and Distributed Technology*, vol. 3, no. 1, 1995.

Buckley, J. System Monitoring: A Tool for Capturing Software Engineers' Information-Seeking Behaviour, PhD Thesis, University of Limerick, 2002

Byrne, M. D, Catrambone, R. and 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.

Carlson, P. Burnett, M.M. (1996), A seamless integration of algorithm animation into a visual programming language with one-way constraints, *International Workshop on Constraints for Graphics and Visualization*, Cassis, France, September 1995.

Corritore, C. L. & Wiedenbeck S. (1991). What do novices learn during program comprehension*? International Journal of Human-Computer Interaction*, vol. 3, no. 2, pp. 199-222.

Cox, K.C. Roman, G-C. (1991). Visualising concurrent computations, *Proceedings of the 1991 IEEE Workshop on Visual Languages*, IEEE Press, 1991, pp. 18-24.

Crescenzi, P. Demetrescu, C. Finnocchi, I. Petreschi, R. (2000). Reversible execution and visualization of programs with Leonardo, *Journal of Visual Languages and Computing (JVLC*), 11(2): 125{150, April 2000.

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

Flynn, M.J.(1972). Some computer organizations ad their effectiveness. *IEEE Transactions on Computers*, C-21(9), 948-960

Gilmore D.J and Green T.R.G (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*. Vol 21, pp 31-48.

Green, T.R.G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*. Vol: 50. pp 93-109.

Good, J. (1996). The right tool for the task: an investigation of external representations, program abstractions and task requirements. *Empirical Studies of Programmers*, Sixth Workshop. pp 77-98.

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

Hartley, S.J. (1994). Integrating XTANGO's animator into the SR concurrent programming language. *ACM SIGGRAPH Computer Graphics*, Vol. 29, No. 4, Nov. 1995, pp. 67-68

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

Hundhausen, C.D., Douglas, S. A. & Stasko, J. T. (2002). A meta-study of algorithm visualisation effectiveness. *Journal of Visual Languages and Computing* 13(3), 259-290.

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.

Kraemer E. (1998). Concurrency. In *Software Visualization* Eds: Stasko, Domingue, Brown & Rice, MIT Press, 1998, Chapter 17,pp.237-256

Krippendorff, Klaus. (1980). *Notes from Content Analysis - an introduction to its methodology*, Sage Publications.

Letovsky S. (1996). Cognitive processes in program comprehension. *Empirical Studies of Programmers*. Eds: Soloway and Iyengar. Ablex Publishing Corporation. ISBN 0-89391388-X pp 28-45.

Muller, H., (1994), Understanding software systems using reverse engineering technology, http://www.rigi.csc.uvic.ca

O'Brien, M. P., Buckley, J., (2001), Inference-based and expectation-based processing in program comprehension, *Proceedings of the 9th International Workshop on Program Comprehension*, Toronto, Canada

O'Brien, M. P. (2003). Software comprehension – a review & research direction. Technical Report UL-CSIS-03-3, University of Limerick

C. M. Pancake (1992). Debugger visualization techniques for parallel architectures. In *Proceedings of COMPCON*, pages 276--284, San Francisco, February 1992.

Pennington N. (1987). Comprehension strategies in programming. *Empirical Studies of Programmers: Second Workshop*. Eds. Olson, Sheppard and Soloway. Ablex Publishing Corporation. Pp 100-114

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

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

Santos Nicolau, M. N. F. (2002). A Fortran Parsing Tool to Extract Parallelising Information. MSc Thesis, University of Limerick.

Stasko. J. T. (1990). Tango: A framework and system for algorithm animation, *Computer* 23: 27-39

Stasko, J.T. Badre, A. Lewis, C. (1993). Do animations assist learning? An empirical study and analysis, *INTERCHI* '93, 24-29 April 1993, pp 61- 66

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

Von Mayhauser A. and Vans A. (1995). Program comprehension during software maintenance and evolution. *Computer*. August : pp 44-55.

Von Mayhauser A. and Vans A. (1997). Program understanding of large scale software. *Empirical Studies of Programmers 7th Workshop*. Eds: Weidenbeck and Scholtz. The A.C.M. pp 157-179

Wiedenbeck, S. Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Human.-Computer Studies* 51(1): 71-87 (1999)