Characterizing Comprehension of Concurrency Concepts

Zhen Li Zhe Zhao Eileen Kraemer

Computer Science Department
University of Georgia
{zhen, zhe, eileen}@cs.uga.edu

Keywords: Concurrency, Software Visualization, Empirical study, Misconceptions

Abstract

A comprehensive understanding of students' common difficulties in understanding synchronization and concurrency is a prerequisite for developing tools and educational materials to alleviate these difficulties. In this paper we briefly present a study through which we identified students' misconceptions about concurrency and synchronization, categorized their misunderstandings into a misconception pyramid, and built subject profiles through which we were able to discover the nature and frequency of the misconceptions exhibited by the students in this study. Based on these findings, we developed metrics to capture the breadth and severity of individual subject's misconceptions. We describe these metrics and show how they correlate with other measurements of understanding of concurrency and synchronization.

1. Introduction

As early as 1986, researchers worried that "the complexity of (concurrent) programming--all those processes active at once, all those bits zinging around in every direction--is simply too great for the average programmer to bear" [Gelertner1986]. Today, it is generally agreed that multi-threaded programs are difficult to design and comprehend, and that concurrency and synchronization concepts are difficult for students to master [1, 2, 3]. We believe that the development and use of appropriate external representations has the potential to help students better comprehend the dynamic and non-deterministic nature of these programs. However, to properly design and evaluate such representations, we must develop a detailed understanding of what aspects of these concepts students find difficult and what misconceptions they harbor. Prior work by our group [4, 9] and by others [3, 7, 10, 11] provides some insight.

We conducted a new study that sought to obtain detailed information about the reasoning processes that students engage in when dealing with concurrent software. We analyzed student responses, identified misconceptions, and then categorized these into a "misconception pyramid." We then constructed per-subject profiles that captured the nature and frequency of misconceptions exhibited by each student, and developed metrics that we believe capture the breadth and severity of misconceptions held by a particular subject. In this paper, we briefly describe our study, our analysis, and the misconception pyramid and define the metrics for breadth and severity of misconceptions. We present the most common misconceptions in the sample group and explore the correlation of our proposed metrics with other measures of comprehension of concurrency and synchronization concepts. Finally, we propose new diagrams to aid in the comprehension of concurrent program executions, and future studies to further evaluate and refine this work.

2. Related Work

In the early 1990s, Resnick[7] recognized that realizing the potential benefit of concurrent programming would depend on the ability of people to effectively learn, use and understand concurrent programming constructs and languages. He developed a concurrent extension to Logo (MultiLogo) and conducted an experiment with a group of elementary school students who used MultiLogo to control simple robots built from LEGO bricks. He then evaluated their work and found three types of bugs: problem-decomposition bugs, synchronization bugs, and object-oriented bugs.

While he believed that object-oriented bugs might have been due to aspects of Multi-Logo, he suggested that difficulties inherent to thinking about concurrency were at the root of the problem-decomposition and synchronization bugs.

Kolikant performed empirical studies of students learning about concurrency [3]. Her results show that students develop pattern-based techniques to solve synchronization problems and then have trouble in solving non-familiar synchronization problems, perhaps as a result of their reliance on those pattern-based approaches. She found that student misconceptions were often the source of their difficulties, writing "we were able to uncover reasonable, yet faulty connections that many students had made ... these connections were the source of their difficulties."

Fleming, et al. [9] performed a think-aloud study of students in a graduate-level computer science class to study the strategies that students apply in corrective maintenance of concurrent software. He collected think-aloud and action protocols, and annotated the protocols for certain behaviors and maintenance strategies. He looked at whether study participants performed diagnostic executions of the program and whether they engaged in failure trace modeling (modeling how the system transits among various internal states, at least one of which is a clear error state, up to the point of failure). He found two key attributes of the most successful participants: they detected a violation of a concurrent-programming idiom and they constructed detailed behavioral models of execution scenarios.

Xie, *et al*. [4] performed an instructor survey and observational study and identified a core set of difficulties that students encounter in learning about concurrency. Common problems he identified included: 1) Thread inter-leavings are difficult for students to comprehend.; 2) Students often forget that context switches can happen when the thread is in a monitor or critical section and have trouble correctly applying that knowledge when they do remember; and 3) Students have trouble reasoning about why the implementations of synchronization primitives lead to correct synchronization behavior.

Recently, Armoni and Ben-Ari [11] performed an in-depth survey of the concurrency-related concept of non-determinism, how it is defined and used, and how it has been taught. They present a taxonomy of the ways that non-determinism can be defined and used, the categories of which are domain, nature, implementation, consistency, execution and semantics. Their survey of educational materials and practices on this topic leads them to the conclusion that "the treatment of non-determinism is generally fragmentary and unsystematic," and they go on to suggest various strategies for teaching non-determinism in the CS curriculum.

Lu, *et al*. [10] studied real-world concurrency bugs rather than student behavior or reasoning. They looked at four open-source applications and randomly selected 105 real world concurrency bugs. They found that one-third of the non-deadlock bugs involved violations of the programmer's intended order of operation, and that another one-third of the non-deadlock concurrency bugs involved multiple variables. In examining the bug-tracking records, they also found that many of the fixes to the bugs they studied were not correct at the first try, providing further support for the idea that reasoning about concurrent executions is difficult.

Each of the above studies attempts to gain insight into the question of what students and programmers find difficult in learning about and in managing concurrency and synchronization. Our study is most similar to that of Kolikant, in that we attempt to identify both the difficulties that students encounter and the reason for those difficulties. We get at this information by not only asking study participants to answer questions that evaluate their comprehension of the potential behaviors of a concurrent program execution, but by also asking them to explain their reasoning. It is in these explanations that we gain insight into their understanding of the meanings of concurrency-related terms, their mental models of the relationships among the objects and constructs by which concurrency and synchronization are achieved, and their comprehension of the consequences of thread activities and interactions.

Another element of our work is the evaluation of diagrams designed to support comprehension of multi-threaded program executions. Related work describes concurrency-related aspects of UML diagrams, proposes variations on UML diagrams to better support concurrency, or evaluates UML diagrams or their variations. For example, Schader and Korthaus described features of UML that support the representation of concurrency [12]. Mehner and Wagner [13, 14] added shading conventions on activations to indicate when, and within which activation, threads are ready or running. Xie, *et al.* [4, 15] developed an extension to sequence diagrams that uses colored activations to indicate the state of each thread (i.e., running, blocking, or ready), among other features. Most recently, Fleming [16] proposes a variation on UML sequence diagrams in which hatching of the activation bar denotes thread state and object states denote the effects of operations on mutexes and condition variables.

3. Experiment

The overall goal of our experiment was to compare the use of different types of UML diagrams (UML 2.0 sequence diagrams and UML 2.0 state diagrams) for different tasks related to the comprehension, implementation, and debugging of concurrent software. The participants were fifteen Computer Science students drawn from upper-level undergraduate classes and from graduate classes during the spring semester of 2010. Students were volunteers and were paid $50 for their time. The study materials included a demographic survey, six computer-based training modules, five pre-tests (one quiz for each of the first five training modules), and a post-test. Part I of the post-test comprised 24 comprehension questions that involved reasoning about what could happen next in a particular execution scenario. Part II questions involved identifying errors, evaluating and creating models and diagrams, and writing code.

In this paper we provide a detailed analysis of participant explanations of their answers to Part I questions, in which they were asked to supply both a yes/no answer to whether a particular set of program events could occur next and in the stated order, and also to explain their reasoning. These explanations of student reasoning provided the basis for our identification of misconceptions. Questions in this part of the post-test were based on the "Single-lane Bridge" problem. The problem states that a bridge over a river is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any one time. To simplify this problem, we define the cars that move from left to right as red cars and those that move from right to left as blue cars. To avoid a safety violation, only one kind of car is allowed to be on the bridge at a time. Cars exit the bridge in the order in which they entered and the leading car may exit the bridge at any time. We structure this system so that each colour of car is implemented as a thread, and the shared bridge object is implemented as a monitor with two associated condition variables **okToEnter** and **okToExit.** The basic functions for entering and exiting the bridge are *redEnter(), redExit(), blueEnter()* and *blueExit()*. We assume a C++ implementation using the *pthreads* library, in which explicit calls to lock() and unlock() are invoked on mutex locks. Then for each of the given scenarios, we asked whether a particular event sequence could happen next.

4. Analysis

Although Part I of the post-test consisted of objective questions, we initially found it difficult to evaluate the responses in a way that accurately reflected the students' understanding of the system. Consider question 1.b, shown in Figure 1 and describing a scenario in which two threads, **redCar1** and **redCar2,** exist in the system. Thread **redCar1** invokes the *redEnter()* method and has already returned when a context switch occurs and the **redCar2** thread begins to run. One of the sub-questions asks whether it is now possible for the **redCar2** thread to invoke the *redEnter()* method and block on the monitor lock. The answer to this question should be NO. Only two threads exist in the system and **redCar1** should have released the monitor lock before it returned from the *redEnter()* method. Thus, it is not possible for **redCar2** to block on the monitor lock.

1. Suppose that only two threads exist in the system: **redCar1** and **redCar2.** Suppose further that **redCar1** has invoked the *redEnter()* method, and has returned. A context switch occurs and the **redCar2** thread starts to run.

Could the following event sequence happen next? Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(b) **redCar2** invokes *redEnter()*, then blocks on the monitor lock.
YES NO

Figure 1 -- Question 1.b

In answering this question, 9 out of 15 subjects chose the correct answer (NO). However, in looking closely at their explanations, we found that 7 of them thought that the monitor lock would only block blue car threads and regarded the monitor lock in the question as an **okToEnter** condition variable. One of them misunderstood the meaning of the term "block" as "own" or "has" and thought that **redCar1** already owned the monitor lock since it was on the bridge and that **redCar2** could thus not own the same lock. Another student, however, did not understand the question and thought that **redCar2** should not "block" on the monitor lock but lock the monitor lock. Thus, by reading the explanations given by the students we found that actually none of the 9 students who gave the correct answer really understood the monitor lock and its mechanism.

We also found that although each question was designed to test some specific misconceptions, a failure in one particular question might not actually stem from the misconception the question intended to examine. Instead, the failure might be rooted in some other misconceptions. We found further that some misconceptions could cause general failures in reasoning about many different scenarios. Consider questions 4.d and 4.e as an example (Figure 2).

4. Suppose that only three threads exist in the system: **redCar1**, **redCar2** and **blueCar1.** Suppose further that **redCar1** is running and has just invoked the *redEnter()* method and the *redEnter()* method has returned. A context switch occurs and the **redCar2** thread begins running and invokes the *redEnter()* method. **redCar2**'s invocation of the *redEnter()* method has not returned.

Which of the following event sequences could happen next? Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(d) A context switch occurs, and the **redCar1** thread begins to run. **redCar1** then invokes *redExit() and this invocation returns*.
YES NO

(e) A context switch occurs, the **redCar1** thread begins to run. **redCar1** then invokes the *redExit()* method and blocks on the monitor lock.
YES NO

Figure 2 -- Question 4.d and 4.e

These two questions are aimed at testing the subjects' ability to consider multiple possible interleavings in an execution. The answer to both of the questions should be YES since the question only describes that **redCar2**'s invocation of the *redEnter()* method is interrupted by a context switch but does not mention whether **redCar2** holds the monitor lock or not when interrupted. Three possible interleavings exist here. One is that **redCar2** has invoked the method but has not yet obtained the monitor lock. The second is that **redCar2** invoked the method, holds the monitor lock and has not yet released it. Another possibility is that **redCar2** has already released the monitor but not yet returned from the *redEnter()* method. The first and the third situations could lead to event sequences described in 4.d and the second situation could lead to event sequences described in 4.e.

Organizing students' answers to these two questions, we have the following table (Table 1).

	4.d	4.e	Subjects
1	YES	YES	102, 139, 132
2	YES	NO	108, 109, 113, 122, 126,138, 141, 142, 145
3	NO	NO	110, 119
4	YES	No answer	128

Table 1 – Subjects' Answers to Questions 4.d and 4.e

Apparently, most of the students were not able to answer both of these questions correctly and the majority failed on question 4.e. However, by looking closely at their explanations, we found the reason for the failure does not truly stem from students' inability to consider the possible interleaving, as expected. Actually, all 9 subjects failed in 4.e because of misconceptions about the monitor lock. Some of them confused it with the **okToExit** or **okToEnter** condition variables. Others were ignorant of the mechanism of the monitor lock so they succeeded in question 4.d, which does not deal with the monitor lock concept but failed in 4.e. Also worth noting is that most students reasoning about these two questions was based on "story-level" understandings, as seen in explanations such as "**redCar1** is free to exit" or "nothing blocks **redCar1** to exit", etc. Actually, none of them considered the event sequence at the implementation level, which again highlights their misconceptions of the context switch and its properties.

Thus, we found students' misconceptions about concurrency and synchronization cannot be captured in a simple list of confusions or misunderstandings of concepts, terminologies and mechanisms. Rather, they are correlated with one another, interacting in a seemingly hierarchical architecture so that it is not possible to examine higher level misconceptions without first teasing out the impact of lower-level misconceptions, or ensuring that participants first have a firm grasp of lower level concepts. In other words, to understand higher level concepts, students must first rid themselves of lower level misunderstandings.

4.1 Misconception Pyramid

We introduce a misconception pyramid (Figure 3), which captures common misunderstandings that students exhibited when reasoning about a concurrent system, and the hierarchical structure of the misconceptions according to the difficulty and dependency relations of understanding the concepts in that level. Understanding concepts at higher levels of the pyramid requires an understanding of the concepts at lower levels first. Descriptions of the types of misconceptions one might find at each level are presented in Table 2, which was constructed based on misconceptions identified in the literature and also those that we encountered in our analysis of subjects' explanations of their reasoning in this study.

The bottom level of the pyramid is the description level and includes misconceptions such as misunderstanding of the requirements, constraints and other details of a concurrent system at the level of the "story" about the red cars and blue cars. For example, some subjects wrote explanations such as "**redCar2** should wait for **redCar1** to invoke *redEnter()* method first" or "**redCar1** should block the bridge first" demonstrate one common misconception at this level: that the thread labels **redCar1** and **redCar2** were the actual running order of the threads.

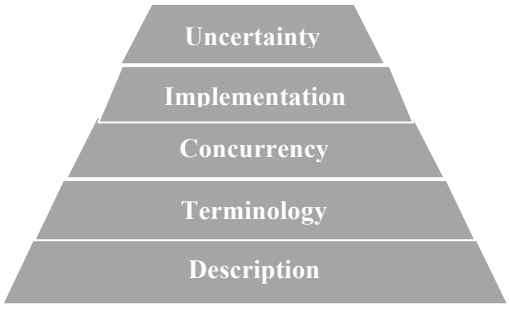

Figure 3 -- Pyramid of Misconceptions

The next level of the pyramid includes misconceptions related to terminology we used in describing concurrent scenarios. A typical example is the misunderstanding of the meaning of "block on" a conditional variable/monitor lock as "hold/own" a conditional variable/monitor lock. This kind of misconception can be seen throughout the explanations given by subjects in our study. Most students who held this kind of misconception did so consistently, causing them to fail on a particular group of questions. Typical students' explanations that illustrate this level of misconception include but are not limited to "**okToEnter** is already blocked" or "monitor is already blocked by **redCar2**".

The third level of the pyramid is the concurrency level, which includes misconceptions about basic thread behaviors such as context switching and the thread life cycle. For example, some students seemed to think that a context switch could not happen while a thread was executing in a critical section and many students thought that a context switch is not allowed during the execution of a method and regarded the whole method body as uninterruptible. Some typical students' explanations are "**redCar2** should receive return call then switch out" or "because **redCar2** has not done its activity (so it cannot be context switched out)".

Description Level	
D1	Misconceptions of system and/or problem descriptions
Terminology Level	
T1	Misconceptions of the meaning of "invoke/call" a method
T2	Misconceptions of the meaning of "return" from a method/invocation
T3	Misconceptions of "block" on a monitor lock as "hold/has" a monitor lock
T4	Misconceptions of "block" on a conditional variable as "hold/has" a conditional variable
Concurrency Level (thread behavior)	
C1	Misconceptions about context switching
C2	Misconceptions about the thread life cycle
Implementation Level	
I1	Misconceptions about conditional variables and the wait/signal mechanism
I2	Misconceptions about monitor lock
I3	Misconceptions about block and unblock mechanism
Uncertainty Level	
U1	Confused about space of executions and thread interleavings

Table 2: Misconception Pyramid Table

```
Invoke
……
Lock the monitor lock
    Check conditional variables
        Access and modify shared variable
        ……
Release monitor lock
    Signal on conditional variables
……
Return
```

Figure 4 – Basic Monitor Programming Function Structure

The fourth level of the pyramid is the implementation level, which is related to detailed implementation mechanisms such as the monitor lock and condition variables and their functionalities. By investigating the subjects' answers and explanations in our study, we found that few subjects were clear on the basic monitor programming structure shown in figure 4. We believe that this is greatly related to students' misunderstandings in the three previous levels. If students do not understand the context switch, they are not able to appreciate the actual purpose and corresponding mechanism of the monitor lock. Misunderstandings of different terminologies also lead to confusion about the workings of monitor programming structures and functions.

The top level of the pyramid is concerned with failures in dealing with uncertainty; that is, the inability to envision or manage all the possible threads interleavings and execution scenarios. While

this problem is often cited as the main source of difficulty in the comprehension of concurrent program executions, we found that this level of difficulty was not seen in our study, as students tended to fail much earlier in the pyramid, and thus were not even exposed to these higher-level issues. Whether a detailed investigation of participant reasoning processes would find the same to be true in other studies of comprehension of concurrent program executions is an open question.

An alternative representation of the pyramid might combine the two lower levels them into a single level, in which Description and Terminology sit side-by-side, supporting the Concurrency level. Further, another approach to layering might think of the top layer of the pyramid, which we term "Uncertainty" as dealing with dynamic analysis issues, and the second layer of the pyramid as dealing with static analysis issues, with both layers together dealing with implementation-related issues.

4.2 Subject Profile

Next, we introduce the subject profiles shown in Table 3. These subject profiles reflect the types and frequency of occurrence of each subject's misconceptions. The first column of the table indicates the subjects' ID number. The other columns correspond to items in the misconception pyramid table. Each cell of (subject, item) is the number of (answer, explanation) pairs of that subject that demonstrate the corresponding type of misconception.

Subject	D1	T1	T2	T3	T4	**C1**	C2	**I1**	**I2**	I3	U1	Total
102	2	1	2	1	2	1	1	5	9	0	0	24
108	3	2	1	0	0	3	1	3	10	0	0	23
109	3	0	0	0	0	8	1	0	11	0	0	23
110	7	3	4	4	2	2	1	8	9	0	0	40
113	2	2	1	1	1	6	1	12	11	0	0	37
119	1	0	4	0	2	4	1	8	11	0	0	31
122	0	0	4	0	0	1	0	0	9	0	0	14
126	0	7	0	0	0	2	1	4	14	0	0	28
128	NA											
132												
138	1	4	0	0	1	8	1	2	9	0	0	26
139	1	4	7	1	2	9	1	7	9	0	0	41
141	2	4	5	2	6	4	1	7	9	0	0	40
142	0	0	1	2	0	1	0	3	10	0	0	17
145	0	0	0	0	0	1	1	1	14	0	0	17
Avg	3	3	4	2	2	**7**	2	**9**	**19**	0	0	

Table 3: Subject Profile Table

While 13 of the 15 subjects provided sufficient explanations for us to build profiles, 2 out of 15 (subjects 128 and 132) provided almost no explanations for their answers, which made it impossible to evaluate their misconceptions. Perhaps the most noticeable characteristic of the subject profile is that no misconceptions of items **I3** or **U1** are found, but that subjects show a very high frequency in demonstrating misconceptions in **I1** and **I2**. This reinforces the idea that students' misconceptions form a hierarchical structure in which lower level failures not only cause higher level misconceptions but also isolate students from higher level concepts.

Another interesting characteristic of the subject profile is that the most common misconceptions are **I2, I1** and **C1, which** are misconceptions about monitor locks, condition variables and context switching. Causality relations exist among these misconceptions; for example, a subject's incomplete understanding of when and how a context switch could occur causes their misunderstanding of the functionality and mechanism of monitor lock, which thereafter causes them to confuse monitor lock with condition variable. We plan to conduct additional studies to further explore the validity of this idea.

Based on the collected data, we can make some statements about particular subject's comprehension of concurrency. For example, we could generalize that subject 139 is almost ignorant of concurrency

concepts and synchronization mechanisms since he demonstrated all kinds of misconceptions at different levels, while subject 122, who just showed consistent misconceptions in a limited range of items, apparently has a much better comprehension of concurrency. This is also validated by the Part I scores of these two subjects, as seen in Table 5 and illustrated in figure 8.

4.3 Subject Evaluation

Although a subject profile allows us to characterize both a single subject's understanding of concurrent systems and the whole subject sample, we introduce two metrics to better quantify the evaluation. One is the breadth of range of misconceptions (denoted as Metric B) and the other is the weighted severity of misconceptions (denoted as Metric S).

$$B_{subject} = \frac{count\,((subject,item) > 0)}{count\,((subject,item))} \qquad S_{subject} = \sum_{items} (subject, item) * W_{item}$$

Figure 5 – Evaluation Metrics

Metric B for a single subject is the percentage of misconceptions the subject has regarding the whole pyramid of misconceptions, as illustrated in figure 5. For example, subject 122 exhibited misconceptions in 3 of 11 categories, so B_{122} = 3/11 or 0.27, while subject 139 exhibited misconceptions in 9 of 11 categories for B_{139} = 9/11 or 0.82. With metric B we are able to evaluate how many different misconceptions a particular subject has. A larger B illustrates more widely spread misconceptions of a particular subject.

Level 0: Description Level		
D1	0.3	
Level 1: Terminology Level		
T1	0.268	0.067
T2		0.067
T3		0.067
T4		0.067
Level 2: Concurrency Level		
C1	0.2	0.1
C2		0.1
Level 3: Implementation Level		
I1	0.135	0.045
I2		0.045
I3		0.045
Level 4: Uncertainty Level		
U1	0.097	

Table 4: Misconception Item Weight Table

Subject	Part1	Metric B	Metric S
102	18	0.82	1.832
108	24	0.64	2.086
109	25	0.36	2.295
110	15	0.82	4.036
113	24	0.82	2.67
119	19	0.64	2.057
122	29	0.27	0.773
126	21	0.45	1.579
128	19	N/A	N/A
132	23	N/A	N/A
138	29	0.64	2.03
139	11	0.82	2.958
141	24	0.82	2.959
142	27	0.45	0.886
145	24	0.36	0.875

Table 5: Subject Performance Table

The S metric, however, evaluates misconceptions on another dimension. It is designed to characterize the severity of single subject's misconceptions. Thus, to compute the S metric, we must first assign a weight to each misconception item. As we pointed out before, lower level misconceptions are likely to cause higher level misconceptions. Also, lower level misconceptions impede a subject's understanding of a system more than higher level misconceptions do. Therefore, we simply use an inverse ratio of the level to assign weights. Table 4 illustrates how the weights are assigned.

Therefore, the metric S can be calculated as the expected value of severity of different misconception items according to formula illustrated in figure 5, in which W_{item} is the weight of the corresponding misconception item. Applying these two metrics to subjects in our study, we get the subject performance table (Table 5). For example, subject 122 exhibited 4 misconceptions of type T2 (w=0.067), 1 misconception of type C1 (w = 0.1), and 9 misconceptions of type I2 (w = 0.045). S_{122} is thus 4 * 0.067 + 1 * 0.1 + 9 * 0.045 = 0.773.

To illustrate the validity of these two metrics, Metric B and Metric S, we explore the correlation between these values and students' total score of Part I in the post-test.

Figure 6 illustrates the correlation between metric B and the score of part I. As we see, although the high scores are not strictly determined by metric B, the metric characterizes how poorly a student may perform in reasoning about concurrency and synchronization scenarios, and overall shows the expected negative correlation (the greater the breadth of misconceptions, the lower the score).

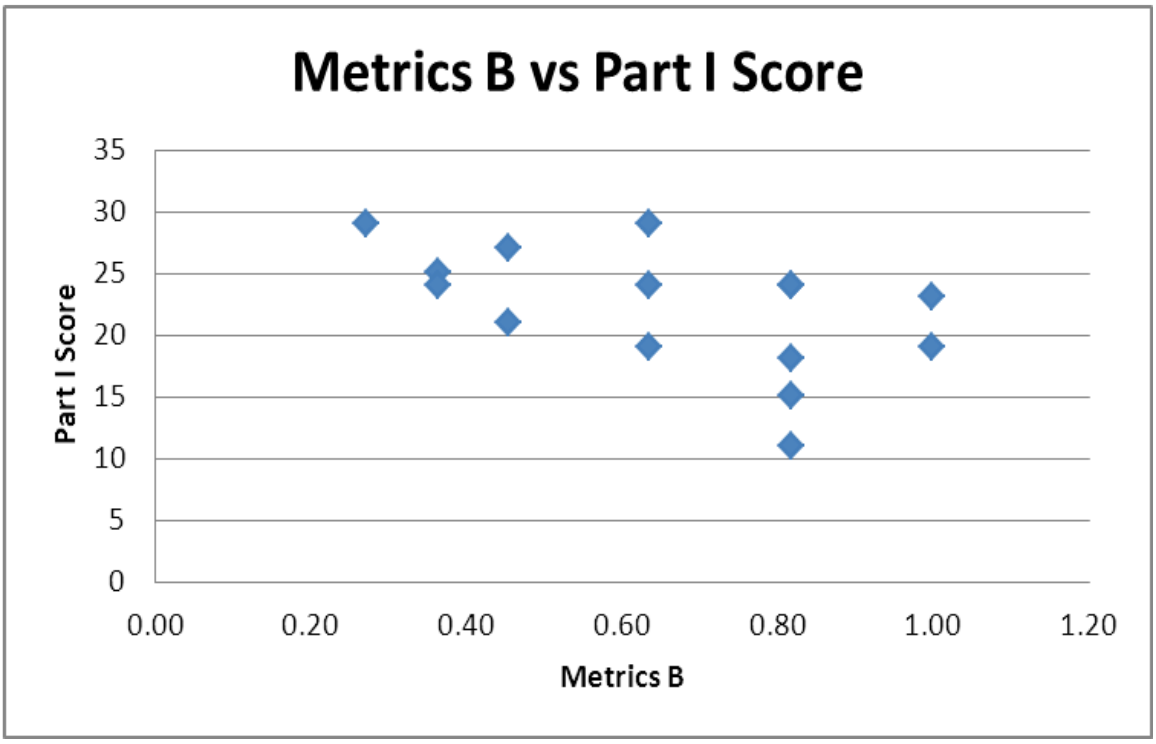

Figure 6 – Correlation between Metric B and Part I Score, Pearson correlation = -0.527

Figure 7 illustrates the correlation between metric S and the score of part I. Unlike metric B, the metrics S seems to have a better (negative) correlation with score when metric S is small. As metric S becomes large, the correlation becomes random. This is reasonable, since when a subject has no idea of a concept in concurrency, they tend to reason about the corresponding scenario based on understanding of one possible sequential execution, which randomly coincides with the actual execution sequence under concurrency.

By regarding metric B and metric S as two orthogonal vectors that characterize an individual subject's misconceptions in concurrency and viewing the origin point in a coordinate system as an ideal expert who does not demonstrate any misconceptions in understanding a concurrent system, we are able to calculate the Euclidian distance of a particular subject from the ideal expert. This Euclidian distance may be regarded as a combination of metric B and metric S. In figure 8, we plot this new evaluation with the total score of part I for every subject. Regardless of the two subjects, number 132 and number 128, who did not given enough clues for us to conclude their misconceptions, other subjects tend to form a reverse correlation of their Part I score and the Euclidian distance from an ideal expert.

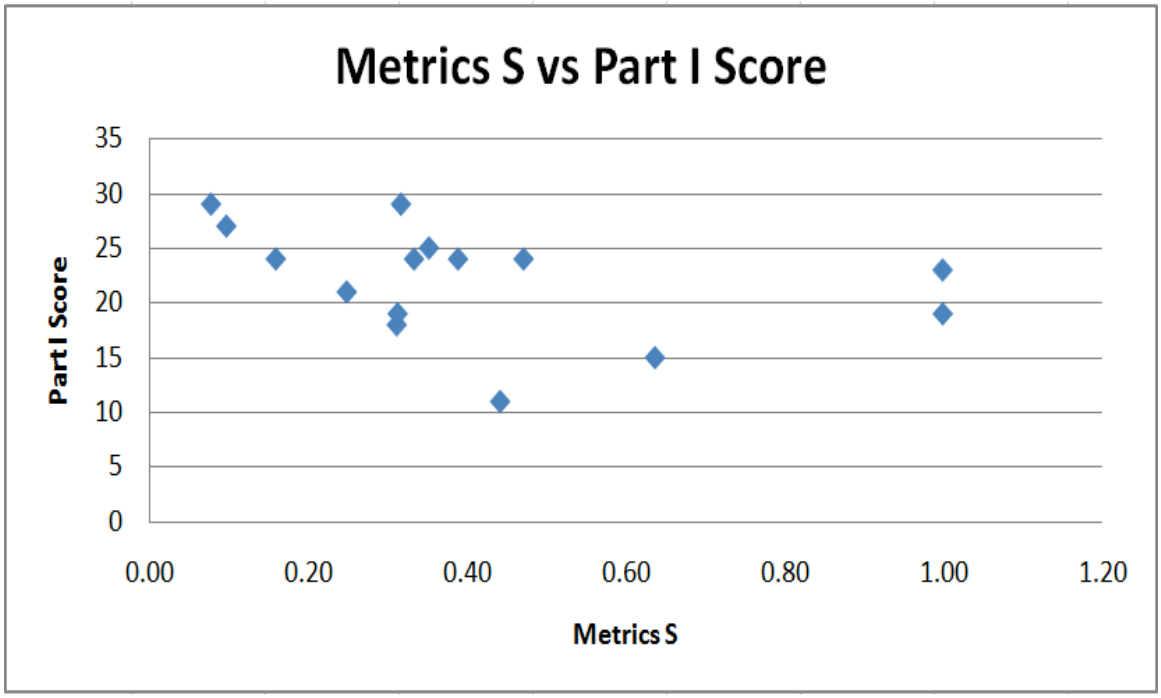

Figure 7 – Correlation between Metric S and Part I Score, Pearson correlation = -0.386

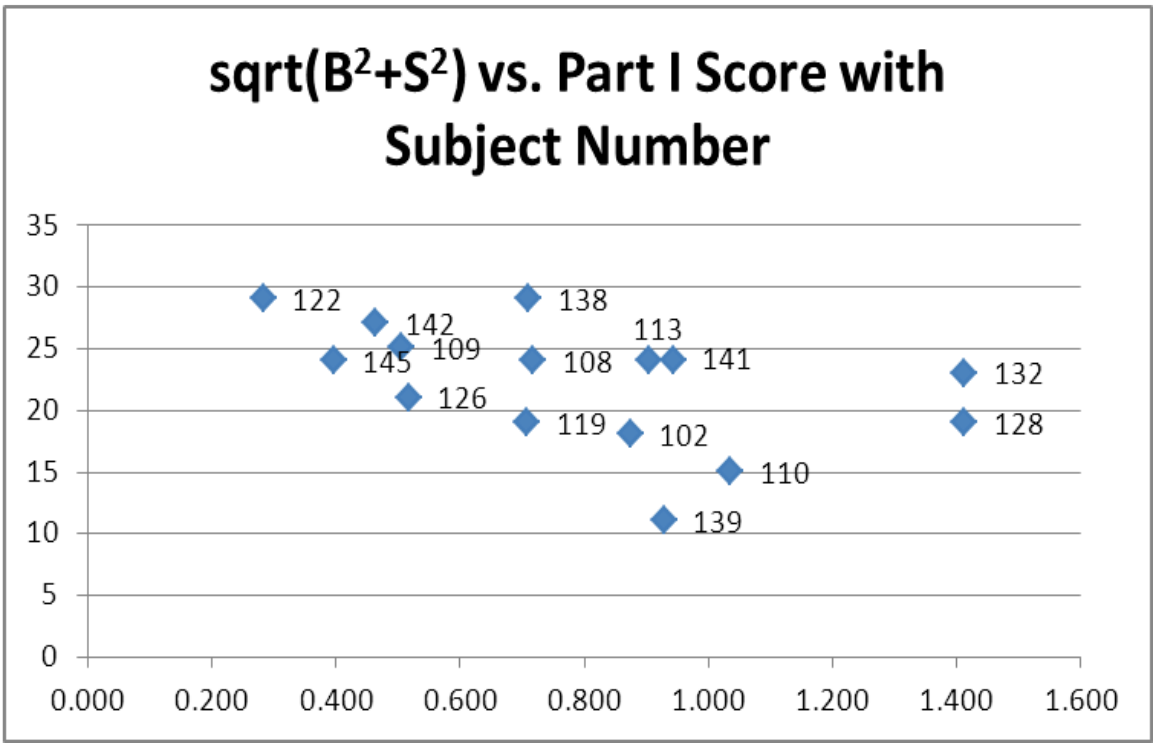

Figure 8 – Correlation between sqrt(B^2+S^2) and Part I Score, with Subject Number

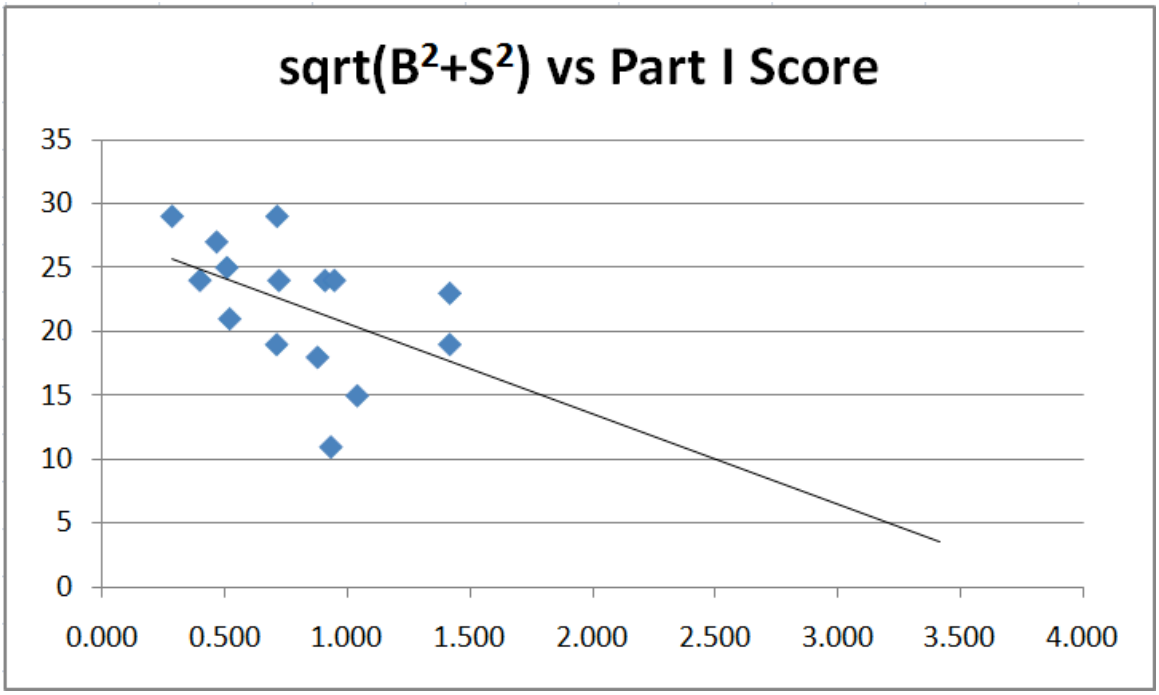

Figure 9 – Correlation between sqrt(B^2+S^2) and Part I Score with Linear Prediction, Pearson's correlation = -0.476

In figure 9, we plot the linear prediction of data in figure 8, which illustrates an expected inverse correlation between the evaluation of subjects' misconception and the actual performance of a subject.

Overall, we believe that metric B and metric S do a reasonable job of capturing the breadth and severity of misconceptions exhibited by individuals or by a group of individuals. The calculation of such metrics and the use of the misconception pyramid have the ability to guide instructors in assessing whether a concept or group of concepts has been sufficiently mastered by a student or class of students. The structure of the pyramid provides some insight into the order in which these concepts might be taught and suggests that intermediate evaluations be performed before moving on to higher-level concepts.

6. Conclusions and Future Work

We have presented here an initial analysis of a relatively small study of students engaged in reasoning about the execution of multi-threaded programs. We have identified a number of misconceptions exhibited by study participants, and based on these findings, have proposed a hierarchical structure of misconceptions, and metrics for evaluating the breadth and severity of these misconceptions. We present arguments to support the validity of the hierarchy and of the metrics. We propose to conduct additional studies with larger groups, to further evaluate both the pyramid and the metrics, and to further flesh out the ways that students think and learn about concurrency and synchronization.

7. References

[1] S. Carr, J. Mayo, and C.K. Shene, "ThreadMentor: A pedagogical tool for multithreaded programming", *Journal of Educational Resources in Computing*, 3(1), 2003.

[2] Cunha, J. C. and Lourenço, J., "An integrated course on parallel and distributed processing" *In Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education,* Atlanta, GA, 1998.

[3] Kolikant, Y. B.-D., "Learning concurrency: evolution of students' understanding of synchronization," *International Journal of Human-Computer Studies*, 60(2), 243–268, 2004.

[4] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In Proc. ICSE, pages 727–731, 2007.

[5] T.R.G.Green, M. Petre. "Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, 7(2), 131-174, 1996.

[6] Maria Kutar, Carol Britton and Trevor Barker. "A Comparison of Empirical Study and Cognitive Dimensions Analysis in the Evaluation of UML Diagrams." In J.Kuljis, L. Baldwin & R. Scoble (Eds). *Proc. PPIG 14*, June 2002.

[7] Mitchel Resnick, "MultiLogo: A Study of Children and Concurrent Programming," *Interactive Learning Environments*, 1(3), 153-170, 1990.

[8] Gelertner, D. "Domesticating Parallelism, " *Computer*, 19(8), 1986.

[9] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon. A study of student strategies for the corrective maintenance of concurrent software. In Proc. 30th Int. Conf.Software Eng. (ICSE 2008), pages 759–768, 2008

[10] Lu, S., Park, S., Seo, E., and Zhou, Y. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News* 36, 1 (Mar. 2008), 329-339. DOI= http://doi.acm.org/10.1145/1353534.1346323

[11] Armoni, M. and Ben-Ari, M. 2009. The concept of nondeterminism: its development and implications for teaching. *SIGCSE Bull.* 41, 2 (Jun. 2009), 141-160. DOI= http://doi.acm.org/10.1145/1595453.1595495

[12] M. Schader and A. Korthaus. Modeling Java Threads in UML. In The Unified Modeling Language: Technical Aspects and Applications, pages 122–143. Physica, 1998.

[13] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In Software Visualization, pages 163–175. 2002.

[14] K. Mehner and A. Wagner. Visualizing the synchronization of Java-Threads with UML. In Proc. VL, pages 199–206, 2000.

[15] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In Proc. ICPC, pages 123–134, 2007.

[16] Scott D. Fleming, Eileen Kraemer, R.E.K. Stirewalt, and Laura K. Dillon. Debugging Concurrent Software: The Importance of External Representations. To appear, VLHCC10.