Empirical Assessment of Languages for Teaching Concurrency: Methodology and Application

Sebastian Nanz[1] Faraz Torshizi[2] Michela Pedroni[1] Bertrand Meyer[1]

[1]*ETH Zurich* [2]*University of Toronto*

firstname.lastname@inf.ethz.ch *faraz@cs.toronto.edu*

Abstract

Concurrency has been rapidly gaining importance in computing, and correspondingly in computing curricula. Concurrent programming is, however, notoriously hard even for expert programmers. New language designs promise to make it easier, but such claims call for empirical validation. We present a methodology for comparing concurrent languages for teaching purposes. A critical challenge is to avoid bias, especially when (as in our example application) the experimenters are also the designers of one of the approaches under comparison. For a study performed as part of a course, it is also essential to make sure that no student is penalized. The methodology addresses these concerns by using self-study material and applying an evaluation scheme that minimizes opportunities for subjective decisions. The example application compares two object-oriented concurrent languages: multithreaded Java and SCOOP. The results show an advantage for SCOOP even though the study participants had previous training in writing multithreaded Java programs. The lessons should be of use to educators interested in teaching concurrency, to researchers looking for objective ways of assessing teaching techniques, and to researchers who want to avoid bias in assessing an approach or tool that they have themselves designed.

1. The need to assess concurrency proposals

Concurrent programming has been practiced for over 40 years, but was until recently perceived as a task for specialists in high-performance computing, operating systems and networking. The move to parallel architectures, in particular multicore processors, has changed this situation, making concurrency part of mainstream software development.

Computer science education should reflect this evolution by assigning a substantial role to teaching concurrency. Concurrent programming is, however, a difficult endeavor, subject to such errors as data races or deadlocks. To help avoid these pitfalls, the programming language community works on new languages that express concurrency and synchronization at higher levels of abstraction and exclude entire classes of errors by construction.

The question remains whether these new languages can deliver and indeed make concurrent programming easier to teach and use: they should help understand existing code, modify it, and produce new correct code. The original description of a new approach usually argues for it on the basis of conceptual arguments, but these arguments must be complemented by empirical analyses of usability and teachability.

Two challenges confront such empirical studies. First, they must avoid bias, especially when one of the approaches under study was designed by the experimenters themselves. Second, they must rely on an objective test of the students' eventual success at mastering the subject matter. In addition, studies performed as part of a course must be careful to avoid any damaging influence on the teaching process; for example it is usually not possible to separate students into two groups, one using a classical approach and the other a new technique: finding that one group performs better would be good for the study but bad for the students of the other group. Such a naïve study setup would violate educational ethics.

Objectivity is especially challenging for experiments that assess an approach designed by the experimenters themselves. Ideally, other researchers should perform such assessments. They are not always available, however, and the original designers have a legitimate interest in evaluating their own designs. In so doing, they must protect themselves against the risk of experimenter bias. We have gone to great lengths, as described below, to avoid this risk.

We have developed a methodology for empirical studies that addresses these challenges through a general design for comparing concurrent programming languages, and applied the methodology and template to a pilot study. A companion report available online [2] includes the material of that study (which others are welcome to reuse) and its results on our sample populations.

The example study assessed two object-oriented concurrent approaches:

- Java Threads offers a way to define concurrent executions within an object-oriented model, using a monitor-like mechanism based on synchronized blocks to express mutual exclusion. Monitor-style *wait()* and *notify()* calls implement condition synchronization.
- SCOOP [1], originally defined for Eiffel, is explicitly designed to make concurrent programming easier by providing concurrency as a simple extension to standard object-oriented mechanisms. SCOOP handles many details of synchronization and message passing without explicit programmer intervention. An object will access any other object handled by a different thread of control through variables declared *separate*, ensuring proper concurrent semantics for access (routines invoked on separate objects are spawned off asynchronously) and synchronization (calls synchronize on their arguments and wait on preconditions).

Section 2 describes the experimental design. Section 3 presents the self-study approach and Section 4 the evaluation methodology. Section 5 summarizes results, and Section 6 concludes.

2. Overview of the experimental design

2.1. Hypotheses

An empirical study should address clearly defined research questions. The hypothesis we tested was:

SCOOP concepts are better picked up by students than Java Threads concepts.

This abstract and broad question can be refined into more concrete hypotheses:

Hypothesis I (Program comprehension) Students can comprehend an existing program written in SCOOP more accurately compared to an existing program having the same functionality written in Java Threads.

Hypothesis II (Program debugging) Students can find more errors in an existing program written in SCOOP than in an existing program of the same size written in Java Threads.

Hypothesis III (Program correctness) Students make fewer programming errors when writing programs in SCOOP than when writing programs having the same functionality in Java Threads.

The combination of these hypotheses reflects the observations that it is critical for students to be able both to write correct programs (III) and to understand existing programs, correct or incorrect (II and III).

2.2. Experimental procedure

The student participants, ideally with no previous exposure to either language, are split randomly into two groups: the SCOOP group works with SCOOP and the Java group works with Java Threads. The study has two phases, run in close succession: a training phase and an evaluation phase. The challenges noted earlier affect both phases: the study should remove any bias both during training and during evaluation.

A self-study approach avoids bias during the training phase. We have prepared material for both the Java and SCOOP approaches and ask students to review it on their own. They can do so individually but are encouraged to work in groups of two or three. The self-study time is limited to 90 minutes; tutors are available to discuss any questions that the students feel are not adequately answered in the material. Section 3 gives more details on the measures used to avoid bias in the training phase.

The evaluation phase uses a simple pen & paper test setup. Students work individually, with a maximum of 120 minutes, and are supervised by tutors. Section 4 gives more details on bias avoidance in the design of the evaluation.

3. Training phase: the self-study

The training process can introduce bias into a teaching study, arising for example from two instructors' different teaching styles. The use of self-study material is intended to circumvent this issue. The material has the following structure:

Java Threads
§1 Concurrent execution
– Multiprocessing and multitasking
– Operating system processes
§2 Threads
– The notion of a thread
– Creating threads
– Joining threads
§3 Mutual exclusion
– Race conditions
– Synchronized methods
§4 Condition synchronization
– The producer/consumer problem
– The methods *wait()* and *notify()*
§5 Deadlock
Answers to the exercises

SCOOP
§1 Concurrent execution
– Multiprocessing and multitasking
– Operating system processes
§2 Processors
– The notion of a processor
– Synch. & asynch. feature calls
– Separate entities
– Wait by necessity
§3 Mutual exclusion
– Race conditions
– The separate argument rule
§4 Condition synchronization
– The producer/consumer problem
– Wait conditions
§5 Deadlock
Answers to the exercises

The only prerequisite for working with these documents is a solid knowledge of the sequential base language of the chosen approach, here Java or Eiffel. Although the approaches differ considerably, the documents closely mirror each other:

§1 This section is identical in both documents, introducing basic notions of concurrent execution in the context of operating systems.

§2 This section addresses the writing of concurrent programs. The central notion is *thread* for Java Threads and *processor* for SCOOP. After completing this section, students should be able to introduce concurrency into a program, but not yet synchronization.

§3 This section introduces mutual exclusion. It explains race conditions and their avoidance using synchronized blocks in Java and separate arguments in SCOOP.

§4 This section introduces the concept of condition synchronization. The need is explained with the producers/consumers example, and the solutions in Java, i.e. *wait()* and *notify()*, and SCOOP, i.e. execution of preconditions with wait semantics, is explained.

§5 This section introduces the concept of a deadlock.

Every section includes, in both variants, an equal number of exercises to check understanding of the material. Solutions are given at the end of the document.

As noted in Section 1, it would be improper to expose students to one approach only. In our setup, students perform both self-studies; the only difference is the order, assigned randomly. Students are assessed (next section) after the first self-study. After they have taken both self-studies, we provide a short traditional-style lecture which summarizes both approaches and answers questions. We feel that the difference of order in which the two approaches are studied has a negligible pedagogical effect in the end and so does not harm any student.

4. Evaluation phase

The evaluation phase should avoid bias and includes three tasks, each directly designed to help assess one of the three hypotheses presented earlier.

4.1. Task I: Program comprehension

The goal of Task I is to measure to what degree students understand the semantics of a program written in the approach they self-studied, and thus to test Hypothesis I. Asking the students to describe the semantics in words would lead to ambiguous answers and subjective evaluation. Instead, we give them programs and ask them to predict the output. This task is interesting for concurrent programs, as scheduling introduces nondeterminism in the output.

The concrete programs in Java Threads and SCOOP, each about 80 lines of code, print character strings of length 10, with 7 different characters available. The test asks the students to write down three of the strings that might be printed by the program.

The evaluation needs an objective and automatic measure of the correctness of a proposed answer. A simple boolean measure stating whether a sequence is correct would be too coarse, as any careless mistake leads to marking the entire solution incorrect. Instead, the assessment uses the Levenshtein distance, a common metric for measuring the difference between two sequences. For every answer s proposed by the student, the algorithm computes the minimum distance of s to elements of the correct answer; the measure of performance for Task I is the mean of these minima for the three answers provided by the student.

4.2. Task II: Program debugging

To analyze debugging proficiency and assess Hypothesis II, we provide programs, each about 70 lines of code and seeded with six bugs. All bugs are of a syntactic nature, so that a student can solve the exercise without understanding the effect of the program. For example, for Java Threads the bugs include a call of *notify()* on a non-synchronized object; for SCOOP they include assigning a separate object to a non-separate variable. Students were asked for the line of an error and a short explanation of why it is an error.

The evaluation assigns points according to the following scheme: one point for identifying the line where an error was hidden; one additional point for a correct explanation. The reason for this approach is that a students may recognize that there is something wrong in a particular line, but might not know the exact reason that would allow correcting it.

4.3. Task III: Program correctness

To analyze program correctness (Hypothesis III), the third task requires students to implement a program that shares an object with two integer fields *x* and *y* between two threads. One thread continuously tries to set both fields to 0 if they are both 1, the other tries the converse. Like the others, this is a pen and paper exercise.

To avoid subjective influences, every answer to be graded starts out with ten points, and points are deducted according to the number and severity of errors. The grading process is correspondingly split into three steps:

1. Step 1 examines all answers to determine the types of errors students made.
2. Step 2 assigns to each type a severity level, expressed as a number of points to be deducted (1 to 3).
3. Step 3 performs a new pass on all answers, deducting points as determined by step 2.

The severity levels are defined as follows: 1-point errors are those that can also occur in a sequential context; 2-point errors can only arise in a concurrent setting, but still allow concurrent execution; 3-point errors prevent concurrent execution.

5. Results: Java Threads vs SCOOP

This paper concentrates on the methodology of designing empirical studies for evaluating the usability and teachability of concurrent languages. A separate report [2] describes in detail the results of the pilot study assessing Java Threads vs SCOOP. The results favor SCOOP even though the study participants had previous training in Java Threads. Given the extra care that we took to avoid experimenter bias, the study reinforces our trust in the usability and teachability of SCOOP; independent assessment by others would be most welcome.

As a side indication, the students reported in course evaluations (on the spot and at semester end) that they greatly enjoyed the self-study format.

6. Conclusion

Given the multitude of proposals for new concurrent languages, empirical studies are urgently needed to judge which are suitable for teaching. We have presented a methodology and study template to compare concurrent languages, relying on self-study material and student evaluation. In future work, the template could be applied to more languages and also developed further, for example by focusing more strongly on one of the hypotheses.

We hope that the methodology and the general study design can be useful not only to educators interested in the specific issue at hand — teaching concurrency — but also to others confronted with the common problem of assessing an approach that one has designed while avoiding experimenter bias and achieving a strong guarantee of objectivity.

Acknowledgments This work is part of the SCOOP project at ETH, which has benefited from grants from the Hasler Foundation, the Swiss National Fonds, Microsoft (Multicore award), ETH (ETHIIRA). F. Torshizi has been supported by a PGS grant from NSERC.

References

[1] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, 2nd edition, 1997.

[2] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer. A Comparative Study of the Usability of Two Object-oriented Concurrent Programming Languages. http://arxiv.org/abs/1011.6047, 2010.