Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages

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

[1]*ETH Zurich* *firstname.lastname@inf.ethz.ch*

[2]*University of Toronto* *faraz@cs.toronto.edu*

***Abstract*—The recent turn towards multicore processing architectures has made concurrency an important part of mainstream software development. As a result, an increasing number of developers have to learn to write concurrent programs, a task that is known to be hard even for the expert. Language designers are therefore working on languages that promise to make concurrent programming "easier". However, the claim that a new language is more usable than another cannot be supported by purely theoretical considerations, but calls for empirical studies. In this paper, we present the design of a study to compare concurrent programming languages with respect to comprehending and debugging existing programs and writing correct new programs. A critical challenge for such a study is avoiding the bias that might be introduced during the training phase and when interpreting participants' solutions. We address these issues by the use of self-study material and an evaluation scheme that exposes any subjective decisions of the corrector, or eliminates them altogether. We apply our design to a comparison of two object-oriented languages for concurrency, multithreaded Java and SCOOP (Simple Concurrent Object-Oriented Programming), in an academic setting. We obtain results in favor of SCOOP even though the study participants had previous training in writing multithreaded Java programs.**

***Keywords*-empirical study; concurrency; programming languages; usability**

I. Introduction

The advent of multicore processing architectures has rapidly increased the importance of concurrency in computing. The new situation entails that many programmers without extensive concurrency training have to write concurrent programs, a task widely acknowledged as error-prone due to concurrency-specific errors, e.g. data races or deadlocks. Such errors typically arise from the incorrect usage of synchronization primitives.

To avoid these pitfalls, the programming languages community works towards integrating concurrency mechanisms into new languages. The goal is to raise the level of abstraction for expressing concurrency and synchronization, and hence to make programmers produce better code. Resulting programming models can exclude certain classes of errors by construction, usually accepting a penalty in performance or programming flexibility for the sake of program correctness.

The question remains whether these new languages can deliver and indeed make concurrent programming "easier" for the developer: both understanding and modification of existing code and the production of new correct code should be improved. It is difficult to argue for such properties in an abstract manner as they are connected to human subjects: empirical analyses of the usability of concurrent languages are needed to distinguish promising from less promising approaches, driving language research in the right direction.

Empirical studies for this purpose have to deal with two main challenges. First, to compare the usability of two languages side-by-side, additional programmer training is needed: typically, only few programmers will be skilled both programming paradigms. However, bias introduced during the training process has to be avoided at any cost. Second, a test to judge the proficiency of participants using the languages has to be developed, along with objective means to interpret participants' answers.

In this paper we propose the design of an empirical study that addresses the mentioned challenges and provides a template for comparing concurrent programming languages. In particular, we make the following contributions:

- a design for comparative studies of concurrent programming languages, based on self-study followed by individual tests;
- a template for a self-study document to learn the basics of concurrency and a new concurrent language;
- a set of test questions that allows for a direct comparison of approaches;
- an evaluation scheme for interpreting answers to the test questions, objective and reproducible;
- application of the study design to a comparison of two concrete languages, multithreaded Java and SCOOP, in an academic setting with 67 B.Sc. students.

A companion technical report available online [9] includes the complete self-study material as well as the test questions of the multithreaded Java vs. SCOOP study, for reproduction of this study or for adapting the template to other languages. A short paper [10] outlines a methodology for comparative studies of concurrent languages from a teaching perspective.

The remainder of this paper is structured as follows. In Section II we review multithreaded Java and SCOOP. Section III outlines our hypotheses. In Section IV we present an overview of the design of the study. We present the design of the training phase including the structure for a self-study

document on concurrency in Section V. The design of the test and the results of the multithreaded Java vs. SCOOP study are presented in Section VI. We discuss threats to validity in Section VII and give an overview of related work in Section VIII. We conclude and present avenues for future work in Section IX.

II. Review of SCOOP and Java Threads

As background for the main part of the paper, this section briefly reviews SCOOP (Simple Concurrent Object-Oriented Programming) [8], [11] and multithreaded Java [15], two object-oriented concurrent programming models.

A. SCOOP

The central idea of SCOOP is that every object is associated for its lifetime with a *processor*, an abstract notion denoting a site for computation: just as threads may be assigned to cores on a multicore system, processors may be assigned to cores, or even to remote processing units. References can point to local objects (on the same processor) or to objects on other processors; the latter ones are called *separate* references. Calls within a single processor remain synchronous, while calls to objects on other processors are dispatched asynchronously to those processors for execution, thus giving rise to concurrent execution.

The SCOOP version of the producer/consumer problem serves as a simple illustration of these main ideas. In a root class, the main entities *producer* and *consumer* are defined. The keyword **separate** denotes that these entities may be associated with a processor different from the current one.

```
producer: separate PRODUCER
consumer: separate CONSUMER
```

Creation of an separate object such as *producer* results in the creation of a new processor and of a new object of type *PRODUCER* that is associated with this processor. Hence in this example, calls to *producer* and *consumer* will be executed concurrently, as they will be associated with two different new processors.

Both *producer* and *consumer* access an unbounded buffer

```
buffer: separate BUFFER [INTEGER]
```

and thus their access attempts need to be synchronized to avoid data races (by *mutual exclusion*) and to avoid that an empty buffer is accessed (by *condition synchronization*). To ensure mutual exclusion, processors that are needed for the execution of a routine are automatically locked by the runtime system before entering the body of the routine. The model prescribes that separate objects needed by are routine are *controlled*, i.e. passed as arguments to the routine.

For example, in a call *consume*(*buffer*), the separate object *buffer* is controlled and thus the processor associated with *buffer* gets locked. This prevents data races on this object for the duration of the routine. For condition synchronization, the condition to be waited upon can be explicitly stated as a precondition, indicated by the keyword **require**. The evaluation of the condition uses wait semantics: the runtime system automatically delays the routine execution until the condition is true. For example, the implementation of the routine *consume*, defined in the consumer, ensures that an item from *a_buffer* is only removed if *a_buffer* is not empty:

```
consume (a_buffer: separate BUFFER[INTEGER])
    require
        not (a_buffer.count = 0)
    local
        value: INTEGER
    do
        value := a_buffer.get
    end
```

Note that the runtime system further ensures that the result of the call *a_buffer.get* is properly assigned to *value* using a mechanism called *wait by necessity*: while the client usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result of this call.

The corresponding producer routine does not need a condition to be waited upon (unboundedness of the buffer):

```
produce (a_buffer: separate BUFFER[INTEGER])
    local
        value: INTEGER
    do
        value := new_value
        a_buffer.put (value)
    end
```

In summary, the core of SCOOP offers the programmer: a way to spawn off routines asynchronously (all routines invoked on separate objects have this semantics); protection against object-level data races, which by construction cannot occur; a way to explicitly express conditions for condition synchronization by preconditions with wait semantics. These are the main reasons for SCOOP's claim to make concurrent programming "easier", as some concurrency mechanisms are invoked implicitly without the need for programmer statements. This comes at the cost of a runtime system taking care of implicit locking, waiting, etc.

B. Java Threads

In multithreaded Java[1] (Java Threads for short), no further abstraction level is introduced above threads. Hence in the producer/consumer problem, both the producer and the consumer are threads on their own (inheriting from class *Thread*) and share a buffer as in the following code example:

```
Buffer buffer = new Buffer();
Producer producer = new Producer(buffer);
Consumer consumer = new Consumer(buffer);
```

[1]We consider "traditional" multithreaded Java, without the advanced features implemented in later versions of its concurrency library.

Once the threads are started

```
producer.start();
consumer.start();
```

the behavior defined in the *run*() methods of *producer* and *consumer* will be executed concurrently.

Mutual exclusion can be ensured by wrapping accesses to the buffer within **synchronized** blocks that mention the object that is used as a lock (in this case *buffer*):

```
public void consume() throws InterruptedException {
    int value;
    synchronized (buffer) {
        while (buffer.size() == 0) {
            buffer.wait();
        }
        value = buffer.get();
    }
}
```

Condition synchronization can be provided by injecting suitable calls to *wait*() and *notify*() methods, which can be invoked on any synchronized object. For example in the *consume*() method, *wait*() is called on *buffer* under the condition that the buffer is empty and puts the calling process to sleep. For proper synchronization, the *notify*() method has in turn to be called whenever it is safe to access the buffer, to wake up any threads waiting on the condition:

```
public void produce() {
    int value = newValue();
    synchronized (buffer) {
        buffer.put(value);
        buffer.notify();
    }
}
```

In summary, the core of Java Threads offers: a way to define concurrent executions within an object-oriented model; no automatic protection against object-level data races, but a monitor-like mechanism based on **synchronized** blocks; monitor-style *wait*() and *notify*() calls to implement condition synchronization. In comparison with SCOOP, the runtime system is less costly as the programmer is given more responsibility to correctly apply the offered concurrency mechanisms.

III. Hypotheses

Stating the research questions to be answered is an essential part of the design of any empirical analysis. In the case of our comparative study, a suitable abstract hypothesis is given by the frequently used claim of language designers that programming is simplified by the use of a new language:

> It is easier to program using SCOOP than using Java Threads.

Note that, to support intuition, we explain our study template here and in the following with the concrete languages SCOOP and Java Threads.

A broad formulation such as the above leaves open many possibilities for refinement towards concrete hypotheses:

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

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

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

For the comprehension and correctness tasks we focus on programs having the same functionality, while for the debugging task we require them to have only the same size (close correspondence in number of classes, attributes, functions, and overall lines of code). This is because we want to separate the debugging task from the program's semantics in as far as possible, focusing on syntactic or "shallow" semantic errors. Asking for the detection of deeper semantic errors would be conceivable as well, but would introduce a possibility for misinterpretation: within the same task, one would have to specify what the program is supposed to do, opening the possibility for misunderstanding of either the program or its specification.

IV. Overview of the experimental design

In this section we give an overview of the design of our study; subsequent sections will detail the training phase and the test phase that are part of this design. We start by explaining the basic study setup, using the example of SCOOP vs. Java Threads, then discuss participants' backgrounds in our concrete study.

A. Setup of the study

As we want to analyze how programming abstractions for concurrency affect comprehension, debugging, and correctness of programs, the study requires human subjects. We have run the study in an academic setting, with 67 students of the Software Architecture course at ETH Zurich in Spring semester 2010. All participants were B.Sc. students, 86.2% in their 4th semester, the others in higher semesters.

This population was split randomly into two groups: the *SCOOP group* (30 students) worked during the study with SCOOP and the *Java group* (37 students) worked with Java Threads. To confirm that the split created groups with similar backgrounds we used both self-assessment and a small

number of general proficiency test questions, as detailed below in Section IV-B.

The study had two phases, which we run in close succession of each other: a *training phase*, run during a two-hour lecture session, and a *test phase*, run during an exercise session later on the same day. Two challenges for a study design present themselves:

- *Avoiding bias during the training phase.* We kept the influence by teachers to a minimum through the use of self-study material, discussed further in Section V.
- *Avoiding bias during the evaluation of the test.* For this we developed a number of objective evaluation schemes, discussed further in Section VI.

In the following we give a brief account of the practical procedure of running the study.

1) Training phase: During the training phase, the participants were given self-study material, depending on their membership in the SCOOP or Java group. The participants were encouraged to work through the self-study material in groups of 2-3 people, but were also allowed to do this individually. The time for working on the study material was limited to 90 minutes. Tutors were available to discuss any questions that the participants felt were not adequately answered in the self-study material.

2) Test phase: During the test phase, participants filled in a pen & paper test, depending on their membership in the SCOOP or Java group. They worked individually, with the time for working on the test limited to 120 min (calculated generously). The tutors of the Software Architecture course invigilated the test and collected the participants' answers at the end of the session.

B. Student backgrounds

To learn about the students' backgrounds and to confirm that the random split created groups with similar backgrounds we used both self-assessment and a small number of general proficiency test questions; this information was collected during the test phase.

1) Self-assessed programming proficiency: We collected information regarding the current study level of the students and any previous training in concurrency. This confirmed that all students were studying for a B.Sc. degree, and had furthermore taken the 2nd semester Parallel Programming course at ETH, thus starting with similar basic knowledge of concurrency. All students were familiar with Java Threads, as this was the language taught in the Parallel Programming course (we discuss this further in Section VII).

Concerning programming experience we asked the participants to rate themselves on a scale of 5 points where 1 represents "novice" and 5 "expert" regarding their experience in: programming in general; concurrent programming; Java; Eiffel; Java Threads; SCOOP. Figure 1 shows the results with means and standard deviations. Both groups rate their general programming knowledge, as well as their experience with concurrency, Java, and Eiffel at around 3 points, with insignificant differences between the groups. This confirms a successful split of the students into the groups from this self-assessed perspective.

Furthermore, the Java group achieved a higher self-assessed mean for knowledge of Java Threads, and analogous for the SCOOP group. The knowledge of SCOOP, which none of the students was familiar with initially, ranked significantly lower than the knowledge of Java Threads.

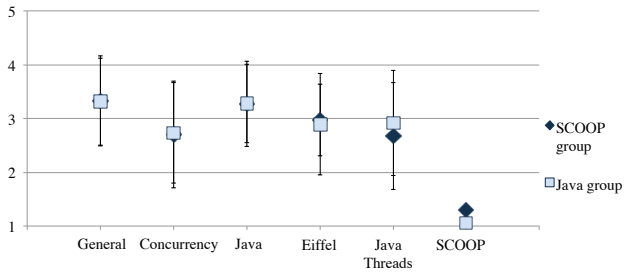

Figure 1. Self-assessed programming proficiency

2) General proficiency test: To confirm that the participants have enough knowledge in the base language – Java in the case of Java Threads, and Eiffel in the case of SCOOP – the test included an understanding task: participants were asked for the output of a given program (4 classes, plus an additional wrapper class in Java; approximately 80 lines of code). To assess participants' concurrency knowledge, we also asked multiple choice and text questions on multiprocessing, process states, data races, mutual exclusion, and deadlock. On both accounts, the students of the two groups achieved very similar results (which we have to omit for brevity), confirming again the successful split into groups.

V. Training phase

When running a comparative study involving novel programming paradigms, study subjects who are proficient in all of these will typically be the exception, making a training phase mandatory. The training process can however also introduce bias, for example if the teaching style of two teachers differs. Requiring the presence of teachers for the study makes it also harder to re-run it elsewhere, as a teacher trained in the subject has to be found.

To avoid these problems, we focused on the use of self-study material. Bias could also be introduced when writing this material, but the quality of the material can be judged externally, adding to the transparency of the study. In addition, re-running the study is much simplified.

A. Self-study material

A course on concurrency can easily take a whole semester. The self-study material we were using and are proposing as a template can be worked through in 90 minutes and thus appears unduly short. However, the material has to be judged

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

Figure 2. Structure of the self-study material

in conjunction with the questions of the test; our results in Section VI show that participants can actually acquire solid basic skills in the limited time frame. A pre-study with six participants, which allowed us to gain various helpful feedback on the study material, confirmed also that the study material can be worked through in 90 minutes.

For teaching the basics of a concurrent language, we suggest the basic structure shown in Figure 2, side-by-side for Java Threads and SCOOP. The only prerequisite for working with these documents is a solid knowledge of the (sequential) base language of the chosen approach, i.e. Java and Eiffel. It is apparent that the documents closely mirror each other, although they describe two different approaches:

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

§2 This section concerns the creation of concurrent programs. Here the central notion for Java Threads is that of a thread, for SCOOP it is that of a processor (compare Section II). At the end of the second section, participants should be able to introduce concurrency into a program, but not yet synchronization.

§3 This section introduces mutual exclusion. Race conditions and their avoidance using **synchronized** blocks in Java and **separate** arguments in routines in SCOOP are presented.

§4 This section introduces 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.

Furthermore, in every section of the self-study material, there is an equal number of exercises to check understanding of the material; solutions are given at the end of the document. The Java Threads document had 18 pages including exercises and their solutions, the SCOOP document 20 pages. The self-study material is available online [9].

B. Students' feedback

To learn about the quality of the training material, we also asked for feedback on the self-study material participants had worked through; this information was collected during the test phase.

Figure 3 gives an overview of the answers to our questions on this topic, rated on a Likert scale of 5 points (where 1 corresponds to "strongly disagree" and 5 to "strongly agree"). Most of the students felt that the material was easy to follow and provided both enough examples and exercises, with insignificant differences between the groups. Both groups also felt that 90 minutes were enough time to work through the material, where the Java group felt significantly better about this point; this might be explained by the fact that the Java group knew some of the material from before. Overall most students agreed, but not strongly, that self-study sessions are a good alternative to traditional lectures.

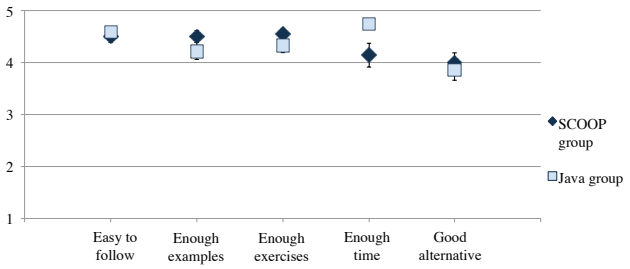

Figure 3. Feedback on the self-study material

The overall very positive feedback to the self-study material was confirmed by a number of text comments, and by the tutors invigilating the sessions, who reported that students explicitly expressed that they liked the format of the session.

VI. Test phase and study results

In this section we present the design of the test and our test evaluation scheme, and report on the results of the concrete study concerning Java Threads vs. SCOOP. After some general remarks, we describe Tasks I to III with their individual evaluation schemes and results, and conclude with a brief interpretation of results. The test material is available online [9].

A. General remarks

The participation of the students in the test was high at 84.8% out of 79 students registered in the course. No special incentives such as a prize were given, and the students were told beforehand that their performance in the test cannot affect their grades. The students were told a week in advance that the lecture and the exercise session on the day of the study would be devoted to the study of two concurrent programming techniques.

Our goal was to focus on the correctness of answers, rather than the speed of producing them. For this reason, we allowed for ample time to complete the test (120 minutes); consequently, all students were able to hand in before the time was up. However, time to completion is an important complementary measure in our setup and therefore we asked students to self-assess the time needed. Completion times turned out to be comparable in both groups: students in the Java Threads group took 54.4 minutes on average, SCOOP students 61.2 minutes on average – the difference between the means was not significant at a 95% confidence level (exact significance level: 8.6%).

B. Task I: Program comprehension

Task I was developed to measure to what degree participants understand the semantics of a program written in a specific paradigm, and thus to test Hypothesis I. Rather than having the semantics described in words, which would make answers ambiguous and their evaluation subjective, we let participants predict samples of a program's output. This task is interesting for concurrent programs, as the scheduling provides nondeterministic variance in the output.

The concrete programs in Java Threads and SCOOP (5 classes, plus an additional wrapper class in Java; ca. 80 lines of code) were printing strings of characters of length 10, with 7 different characters available. In total, the programs' possible outputs contained 28 such sequences, but the participants were neither aware of this number nor the length of the strings. The test asked the participants to write down three of the strings that might be printed by the program.

1) Evaluation: To evaluate the results of Task I, we aimed to find an objective and automatic measure for the correctness of an answer sequence. The obvious measure – stating whether a sequence is correct or not – appeared too coarse-grained. For example, some students forgot to insert a trailing character that was printed after the concurrent computation had finished. Such solutions, although they might show an understanding of concurrent execution as expressed by the language, could have only be marked "incorrect".

We therefore considered the Levenshtein distance [6] as a finer-grained measure, a common metric for measuring the difference between two sequences. In our case, we had to compare not two specific sequences, but a single sequence s with a set C of correct sequences. Our algorithm computes the Levenshtein distance $dist$ between s and every element $c \in C$, and then takes the minimum of the distances:

$$L_{min}(s) = min\ \{dist(s, c) : c \in C\}$$

This corresponds to selecting for s the Levenshtein distance to one of the *closest* correct sequences. As the participants were asked for three such sequences, we took the mean of all three minimal Levenshtein distances to assign a measure to a participant's performance on Task I:

$$\frac{1}{3} \cdot \sum_{i=1,2,3} L_{min}(s_i)$$

Example To illustrate our evaluation algorithm, consider the following example:

Given sequence	A closest correct sequence	*dist*
ATSFTSFPML	ATSFTSFPML	0
ATSFMTSFPL	ATSFPTSFML	2
APTSFTSFM	APTSFTSFML	1

In this case we obtain $\frac{1}{3} \cdot (0 + 2 + 1) = 1$.

By using a general metric such as the Levenshtein difference, equal weight is given to all errors in the sequence. In future work, defining a customized distance measure could allow to distinguish between errors and to analyze in detail the language aspects that are confusing for students.

2) Results: The results for Task I are displayed in Figure 4 with means and standard deviations.

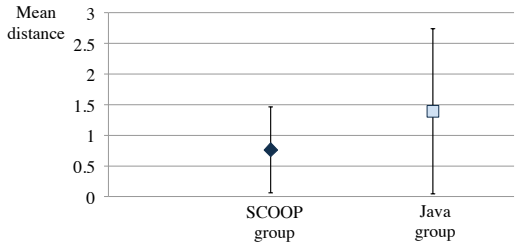

Figure 4. Results Task I

A two-tailed independent samples t-test gives that the means can be assumed to be different at a confidence level of 95% (exact significance level 3.3%). This implies that the SCOOP group with the lower mean performed *better* at Task I than the Java group.

C. Task II: Program debugging

To analyze program debugging proficiency, we provided programs (3 classes, ca. 70 lines of code) that were seeded with 6 bugs. For Java Threads the bugs included the following types

- Calling *notify*() on a non-synchronized object
- Creating a **synchronized** block without a synchronization object
- Failing to catch an *InterruptedException* for *wait*()

and for SCOOP they included:

- Assigning a **separate** object to a non-separate variable
- Passing a **separate** object as non-separate argument
- Failing to control a **separate** object

Participants were asked for the line of an error, and a short explanation why it is an error.

1) Evaluation: The evaluation assigned every participant points, according to the following scheme:

- 1 point was assigned for pointing out correctly the line where an error was hidden;
- 1 additional point was assigned for describing correctly the reason why it is an error.

The rationale for splitting up the points in this way was that participants may recognize that there is something wrong in a particular line (in this case they would get 1 point), but might or might not know the exact reason that would allow them to fix the error; depending on whether they could actually debug the error, they would get another point.

2) Results: The results for Task II are displayed in Figure 5. A two-tailed independent samples t-test showed a significant difference between the results of the Java and the SCOOP group at a confidence level of 95% (exact significance level 4.2%). This implies that the SCOOP group with the higher mean performed *better* at Task II than the Java group.

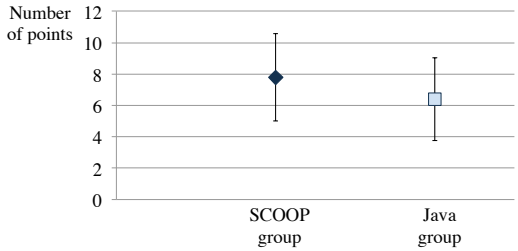

Figure 5. Results Task II

D. Task III: Program correctness

To analyze program correctness, the third task asked participants to implement a program where an object with two integer fields x and y is shared between two threads. One thread continuously tries to set both fields to 0 if they are both 1, the other thread tries the converse. As a pen & paper exercise, the usual compile-time checks that are able to find many of the errors made were not available.

1) Evaluation: Even in everyday teaching routine, the grading of a programming exercise can be challenging, and is often not free of subjective influences by the corrector. To avoid such influences in the evaluation of Task III, we used a deductive scheme in which every answer to be graded starts out with 10 points, and points are deducted according to the number and severity of the errors it contains.

To make this type of grading possible, the grading process was split into several phases:

1) In a first pass of all answers to Task III, attention was paid to the *error types* participants made.
2) The error types were assigned a severity, which would lead to the deduction of 1 to 3 points.
3) In a second pass of all answers, points were assigned to each answer, depending on the types of errors present in the answer and their severity.

The severity of an error was decided as follows:

Ordinary error An error that can also occur in a sequential context (1 point deduction).

Concurrency error An error that can only arise in a concurrent setting, but which is lightweight as it still allows for concurrent execution (2 points deduction).

Severe concurrency error An error that can only arise in a concurrent setting, but is severe as it prevents the program from being concurrent (3 points deduction).

Typos and abbreviations of keywords or other very minor mistakes did not lead to a deduction of points.

2) Error types: The limited size of the programming task led to few error types overall: seven for Java Threads and six for SCOOP. Figures 6 and 7 show the error types with their frequency for Java Threads and SCOOP. Error types with dark/medium/light shaded frequency bars were marked severe concurrency/concurrency/ordinary errors, respectively.

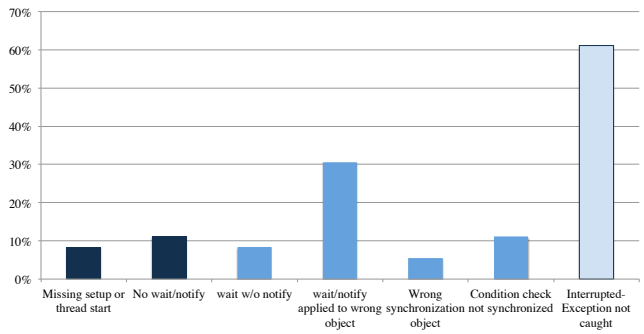

Figure 6. Error types for Java Threads

In Java Threads, we considered it a severe error if a proper setup of threads or the starting of threads was missing,

hence obtaining a functionless or non-concurrent program. In SCOOP, a direct counterpart to this error was the omission to declare the worker objects *separate*, also leading to a non-concurrent program. 8.3% of Java participants made this error, and 10.7% of SCOOP participants.

Another severe error was marked for Java Threads if the program did not contain any *wait*() or *notify*() calls, hence providing no condition synchronization. The corresponding error in SCOOP was the absence of wait conditions. Only 3.5% of the SCOOP group made this error, while 11.1% of Java participants did so, an indication that a tighter integration of synchronizing conditions into the programming language might have advantages.

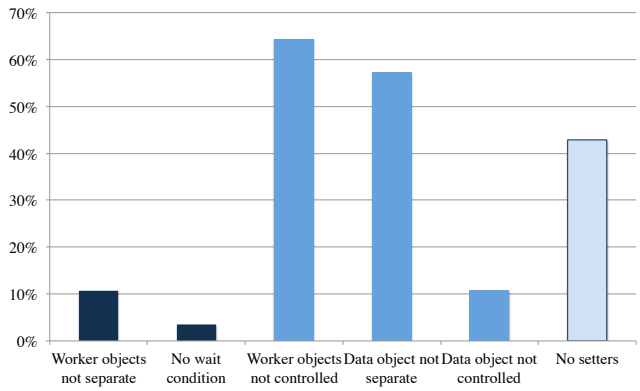

Figure 7. Error types for SCOOP

For non-severe concurrency errors and ordinary errors the comparison is no longer that straightforward. A majority of SCOOP participants did not control worker objects and did not declare the data object as *separate*. These are typical novice errors, and would be caught by compile-time checks. Also a large number of SCOOP participants did not use setter routines as needed in Eiffel, a typical ordinary error.

For Java Threads, we see an extreme peak only for not throwing an *InterruptedException* on calling *wait*(), which was classified as an ordinary error and would be caught by compile-time checks. Other concurrency errors involved the use of *wait*() or *notify*(), for example forgetting a corresponding *notify*() or applying it to a wrong object. Note that these errors cannot be caught during compile-time.

3) Results: The results for Task III are displayed in Figure 8. A two-tailed independent samples t-test does not show a significant difference between the two means (exact significance level 32.6%).

E. Interpretation of the results

The data confirms Hypotheses I and II in favor of SCOOP, leading to the conclusion that SCOOP indeed helps to comprehend and debug concurrent programs. Hypothesis III concerning program correctness could neither be confirmed nor refuted: the SCOOP group did approximately as well as

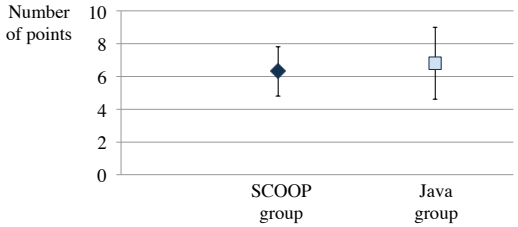

Figure 8. Results Task III

the Java group. Given the small amount of training in the new paradigm, these results are surprising, and promising for the SCOOP model.

The question remains why SCOOP fails to help in program construction. A direct way of interpretation would be to conclude that SCOOP's strengths only affect the tasks of understanding a given program and debugging it. It does not improve constructing correct programs.

However, the first two tasks are at the Comprehension Level of Bloom's taxonomy of learning objectives [1] – level two out of a total of six levels, where a lower level means less cognitively challenging. Comprehension tasks mostly check whether students have grasped how the taught concepts work, an important prerequisite for applying them to new situations. Program construction is at a higher level; depending on the difficulty of presented tasks and previously studied examples, it could be on one of the level three to five of Bloom's taxonomy. It is possible that the training time allotted for this study was too short to enable students transfer the abstractions to the new problem presented in the test. To find out whether this was the case and SCOOP, in comparison to Java Threads, also benefits program construction, a re-run of the study with a more extensive training phase would be necessary.

VII. Threats to validity

The fact that all students of our study had previous knowledge of Java Threads, but none of SCOOP, can be expected to skew the results to benefit Java Threads. We were aware of this situation already in the planning phase of the study, and decided to run it with this group of participants nonetheless. A similar situation also frequently arises in practice: developers versed in a certain programming paradigm consider learning a new one. The study results show that even under these circumstances, the new paradigm might prove superior to the well-known one (Tasks I and II).

Another threat to internal validity is the experimenter bias, where the experimenter inadvertently affects the outcome of the experiment. A double-blind study was not an option in our case, as at least some of the results had to be analyzed by humans, at this time revealing the membership to a group in the experiment. Using automatic techniques for Task I, clearly defined errors with line numbers in Task II, and

developing the deductive scheme for Task III should however limit this bias to a minimum.

A further threat to internal validity is that results might have been influenced by the usability of the base programming languages themselves, Java and Eiffel. In self-assessment participants attributed themselves however sufficient proficiency in both languages and this was confirmed by a short test (see Section IV-B); these influences might thus be negligible. The SCOOP model can also be implemented with Java as the base language [16]; using such an implementation could eliminate this threat altogether.

As a threat to external validity, we used only students as study subjects and it is unclear how the study results generalize to other participant groups and situations. In particular, the use of development environments might greatly affect the learning experience and the potential of producing correct programs. We suggest to run further studies in the future (see Section IX-B) to explore these situations, but deem our study a "cleanroom approach" to analyzing the effects of language abstractions.

As a threat to construct validity, it is difficult to justify objectively that tasks were "fair" in the sense that they did not favor one approach over the other. However, Java Threads and SCOOP are languages that are suitable for ordinary concurrency tasks, and such tasks featured in the test. This situation would be more difficult for languages that aim for a specific application domain.

VIII. Related work

According to Wilson et al. [17], the evaluation of parallel programming systems should encompass three main categories of assessment factors: a system's run-time performance, its applicability to various problems, and its usability (ease of learning and probability of programming errors). The assessment of the factors described in the first two categories are directly related to metrics that can be collected through, for example, running benchmark test suites. But, as shown for the domain of modeling languages by Kamandi et al. [5], such metrics cannot predict the outcomes of controlled experiments with human subjects for the assessment factors of the third category "usability". Also Sadowski and Shewmaker [13] argue that usability is a key factor for the effectiveness of parallel programming and describe metrics for measuring programmer productivity.

The need for controlled empirical experiments for concurrent programming has already been recognized 15 years ago [14]. Nevertheless, only few such experiments have been carried out so far. Those that have been carried out focus on time that it takes the study participants to complete a given programming assignment.

Szafron and Schaefer [14] conducted an experiment with 15 students of a concurrent programming graduate course. They taught two parallel programming systems (one high-level system and a message-passing library system) each for 50 minutes to the entire class; students then had two weeks to solve a programming assignment in a randomly assigned system. The evaluation compared the time students worked on the assignment, number of lines, and run-time speed amongst other measures. Their results suggest that the high-level system is more usable the message passing library, although students spent more time on the task with the high-level system.

The group around Hochstein, Basili, and Carver conducted multi-institutional experiments [4], [3] in the area of high performance computing using parallel programming assignments and students as subjects. In all these experiments, time to completion is the main measure taken. The results of these studies indicate that the message passing approach to parallel programming takes more total effort than the shared memory approach. Cantonnet et al. [2] examined the influence of the language UPC on programmer productivity. They compared UPC to MPI using lines-of-code and conceptual complexity (number of function calls, parameters etc.) as metrics, obtaining results in favor of UPC.

Luff [7] compares the programmer effort using traditional lock-based approaches to the Actor model, and transactional memory systems. He uses time taken to complete a task and lines of code as objective measures and a questionnaire capturing subjective preferences. The data exhibits no significant differences based on the objective measures, but the subjective measures show a significant preference of the transactional memory approach over the standard threading approach. Rossbach, Hofmann, and Witchel [12] conducted a study with undergraduate students implementing the same program with locks, monitors, and transactions. While the students felt on average that programming with locks was easier than programming with transactions, the transactional memory implementations had the fewest errors.

All of the above experiments target programmer productivity as their main focus. To measure this, the studies need to provide substantial programs and a long time range for completing them as a basis of work. By doing so, some of the control over the experimental setup is lost. Our study has a more modest goal: it tries to compare two approaches with respect to their ease of learning them and understanding and writing small programs correctly after a very short time of instruction. By narrowing the focus in such a way, we place the ability of controlling the experiment over being able to generalize the results to arbitrary situations and levels of proficiency. Given that this experiment is only a first step in a series, it seems justified to do so.

IX. Conclusion

A. Discussion

The use of programming abstractions since the 1960s has enabled the tremendous growth of computing applications

witnessed today. New challenges such as multicore programming await the developers and the languages community, but the multitude of proposals makes it hard for a new language to leave a mark. Empirical studies are urgently needed to be able to judge which approaches are promising. Since abstractions are invented for the sake of the human developer, and to finally improve the quality of written code, such studies have to involve human subjects.

Despite the need for such studies, they have been run only infrequently in recent years. One reason for this might be that there is too much focus on established languages. Hence newly proposed languages are not put to the test as they should, ultimately hampering the progress of language research. For this reason we have proposed a template for a study, which can expressly be used with novel paradigms. While established study templates are a matter of course in other sciences, they are not common (yet) in empirical software engineering. We feel that the community should draw their attention to developing templates too, as these will improve research results in the long term and provide a higher degree of comparability among studies.

The key to making our study template successful was the reliance on self-study material in conjunction with a test, and an evaluation scheme that exposes subjective decisions of the corrector. While 90 minutes for studying a new language is brief, we were actually impressed how much the participants learned, some of which handed in flawless pen & paper programs.

B. Future work

Clearly, our template should be applied to more languages in the future. Also, the set of study subjects can be varied in future studies. In an academic setting, we would ideally like to re-run the Java/SCOOP study with students who have no prior concurrency experience. Also, the study template should be used at other institutions, and in the end grow out of the academic setting and involve developers.

The template could also be developed further. For example, it would be possible to concentrate more strongly on one aspect, e.g. program correctness, and to pose more tasks to test a single hypothesis. The evaluation in Section VI-D shows that participants might have improved their results greatly if they have had access to a compiler; running the test not as a pen & paper exercise but with computer support would thus be yet another option.

Acknowledgments

We would like to thank S. Easterbrook and M. Chechik for their valuable comments and suggestions on this work. This work is part of the SCOOP project at ETH Zurich, which has benefited from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award), ETH (ETHIIRA). F. Torshizi has been supported by a PGS grant from NSERC.

References

[1] B. S. Bloom (Ed.), *Taxonomy of Educational Objectives*. London: Longmans, 1956.

[2] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the UPC language," in *Proc. IPDPS'04*, 2004.

[3] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A pilot study to compare programming effort for two parallel programming models," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1920–1930, 2008.

[4] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proc. SC'05*. IEEE, 2005, p. 35.

[5] A. Kamandi and J. Habibi, "A comparison of metric-based and empirical approaches for cognitive analysis of modeling languages," *Fundam. Inf.*, vol. 90, no. 3, pp. 337–352, 2009.

[6] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals." *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[7] M. Luff, "Empirically investigating parallel programming paradigms: A null result," in *Proc. PLATEAU'09*, 2009.

[8] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[9] 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.

[10] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer, "Empirical assessment of languages for teaching concurrency: Methodology and application," in *Proc. CSEE&T'11*. IEEE Computer Society, 2011, pp. 477–481.

[11] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.

[12] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proc. PPoPP'10*. ACM, 2010, pp. 47–56.

[13] C. Sadowski and A. Shewmaker, "The last mile: parallel programming and usability," in *Proc. FoSER'10*. ACM, 2010, pp. 309–314.

[14] D. Szafron and J. Schaeffer, "An experiment to measure the usability of parallel programming systems," *Concurrency: Practice and Experience*, vol. 8, pp. 147–166, 1996.

[15] The Java Language, http://java.sun.com/.

[16] F. Torshizi, J. S. Ostroff, R. F. Paige, and M. Chechik, "The SCOOP concurrency model in Java-like languages," in *Proc. CPA'09*. IOS, 2009, pp. 155–178.

[17] G. V. Wilson, J. Schaeffer, and D. Szafron, "Enterprise in context: assessing the usability of parallel programming environments," in *Proc. CASCON'93*. IBM Press, 1993, pp. 999–1010.