A Set of Patterns for Concurrent and Parallel Programming Teaching

Manuel I. Capel[1], Antonio J. Tomeu[2], and Alberto G. Salguero[2(✉)]

[1] College of Informatics and Telecommunications, University of Granada, 18017 Granada, Spain
manuelcapel@ugr.es
[2] College of Engineering, University of Cádiz, 11519 Cádiz, Spain
{antonio.tomeu,alberto.salguero}@uca.es

Abstract. The use of key parallel-programming patterns has proved to be extremely helpful for mastering difficult concurrent and parallel programming concepts and the associated syntactical constructs. The method suggested here consists of a substantial change of more traditional teaching and learning approaches to teach programming. According to our approach, students are first introduced to concurrency problems through a selected set of preliminar program code-patterns. Each pattern also has a series of tests with selected samples to enable students to discover the most common cases that cause problems and then the solutions to be applied. In addition, this paper presents the results obtained from an informal assessment realized by the students of a course on concurrent and real-time programming that belongs to the computer engineering (CE) degree. The obtained results show that students feel now to be more actively involved in lectures, practical lessons, and thus students make better use of their time and gain a better understanding of concurrency topics that would not have been considered possible before the proposed method was implemented at our University.

Keywords: Parallel design patterns · Teaching innovation
Blended learning · ICT integration lecturing model
Concurrent programming · Parallel programming · Virtual Campus

1 Introduction

An effective teaching and learning in Concurrent and Parallel Programming (CPP) cannot be only based on theoretical lectures on process management and their concurrency, but on how to program with specific syntactical constructs included in concurrent programming languages and libraries. Currently, it is of paramount importance to include practical education on programming techniques that can provide scalability, speedup and performance to programs for today's multi and many-core processors.

To learn many different parallel patterns and syntactical constructs in CPP is by no means an easy task for students, and thus they tend to avoid taking

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 203–215, 2018.
https://doi.org/10.1007/978-3-319-75178-8_17

the courses on the subject or postpone for as long as possible. Current CSE University Curricula [16], however, recognize the importance of teaching such subjects early in CS or SE curricula, which would enable future IT professionals to exploit the parallel potential that multiprocessors now offer.

The use of patterns to teach parallelism is in line with new didactics for teaching CPP [2,6,16]. The GoF catalog [4] proposes a comprehensive set of design patterns in the domain of simple object-oriented software design. Our intention is for the parallel *programming* pattern (henceforth referred to simply as *pattern*) to resemble the *parallel design* pattern [12] by describing solutions to recurrent problems in the domain of parallel and distributed software systems.

However, there are several drawbacks to conduct teaching and learning based on patterns: lack of interoperability, since some patterns are highly dependent on the platform or memory models (STM, volatile, immortal, etc.); scalability issues, especially if big data structures need to be mapped onto multicore and many-core processor architectures; impossibility of quality and performance testing, as long as for checking patterns it is necessary to simulate the execution context of each used pattern within a program code that needs to be verified.

We dealt with these issues by defining a selected set of patterns for obtaining optimal scalable parallel software code. Our approach is based on a new method that involves *blended learning* [7], i.e., students can check/compile/run codes generated from this set of key patterns. The student's work is supervised and evaluated by teachers aided by the Virtual Campus (Moodle supported) platform at our University. Students can therefore import program code into the programming language environment that they know and start working with the proposed pattern in order to produce correct program code. Each learning session is completed with a series of exercises to reinforce the students understanding of each pattern introduced.

The paper concludes with an evaluation of the satisfaction degree of students on the pilot course on concurrency, parallelism and real-time programming that we taught over the last three years. As result of the teaching experience, our model has been suggested for application to other courses on programming by the officers in charge of educational issues at the University of Cádiz, and is in process of implementation as a Massive Open Online Course (MOOC).

The paper is organized as follows: Sect. 2 examines the didactical objectives of the course; Sect. 3 details the suggested teaching model and its development in practical tasks and student assignments; Sect. 4 describes the most important patterns in the set selected for the study; Sect. 5 details how the experiment was evaluated and results analyzed; and finally, Sect. 6 outlines the conclusions reached and future work to be developed.

2 Course on Concurrent, Parallel and Real-Time Programming

The teaching and learning objectives of the experiment outlined in this article aim not only to generally improve the quality of the theory content of lessons

but also to increase student involvement in classes through a more practical ICT integration in classical theory content teaching, which includes the core concepts:

1. Fundamental concurrent programming concepts: mutual exclusion, race conditions, synchronization, concurrent systems properties (15%).
2. Mutual exclusion: algorithms for shared memory multiprocessors (20%).
3. Monitors: Hoare's model, signal semantics, concurrent property verification (safety, liveness and fairness) (20%).
4. Message passing and distributed parallel programs: RPC and RMI models, MPI, rendez-vous (15%).
5. Real-time systems: periodic task scheduling based on static priority assignment, scheduling tests, priority inversion anomaly, aperiodic and sporadic task scheduling (30%).

Table 1. Lecture hours and a selected set of patterns from the last course

Course topics	%	Hours (lectures + lab)	No. of patterns used	Pattern names
Fundamentals	15	4.5 + 6	2	Thread creation(*), race-condition
Mutual exclusion	20	6 + 8	2	Lamport's protocol(*), Peterson's algorithm
Monitors	20	6 + 8	2	Readers/writers, passing the baton
Message passing	15	4.5 + 6	4	Rendez-vous, broadcast, geometric parallelism, tumor growth
Real-time systems	30	9 + 2	2	Observer, priority ceiling
Total	100	30 + 30	12	-

Table 1 shows the number of lecture hours allocated to each course topic, the number of patterns typically used to teach each one and a possible selection of patterns that covers all the important concepts of the course. Topics taught on previous courses and reexamined on this one are labelled with an (*).

As with any lecture on general computer programming techniques, we are particularly concerned that the content taught on CPP courses is both clear and *conceptually significant*. We agree with other authors [5] that the use of programming patterns, together with a documentary base of code samples, improves comprehension of the material taught. These patterns must be easily available to students in lectures [14,15].

By compiling and executing the program code arising from the application of one of these patterns once it has been presented by the teacher, students become more actively involved and participate more in lessons [9], and therefore they are following a *blended learning* method that is identified as the most successful for teaching programming contents effectively [7]. There has been, consequently, a significant increase of the time that students spent in the practical work done in our course [11].

3 Teaching Model for Concurrent Programming

In a previous paper [1], we proposed a new concurrent program development process, which students undertake to complete the assignments during the lectures. Students are involved in the initial program design although they are not required to design it from scratch. An initial design was validated by teachers and the students are provided with pre-selected input data to check their implementations (following the above development process). With this work students are ready to apply programming patterns to specific applications.

In our approach, students' assignments comprise the following parts:

1. a set of active components or *processes*.
2. a concurrent ADT or *shared resource*.
3. a localized communication structure.

Students have to develop a solution that uses these elements particularized for each exercise. Communication and synchronization between processes is only carried out through this *shared resource*.

3.1 Predicative Specification Model

The students must develop a formal specification (*pre-*, *post-*, *invariant*) of the initial shared resource design and its operations. We use a specification language that admits a first-order logic semantics as in Logic of Programs [10] but we decided to keep a similar style of specifications to a single-assignment procedural language. The language also includes Z-like mathematical annotations for easy specification of data structures and this facilitates translation into an OO programming language.

Formal resource specification consists of three sections:

1. the declaration of the resource's operations,
2. the definition of the correct states of the resource as a type invariant (*Semantics Domain* section) and
3. the specification of the behavior of operations as pre- and post-conditions (`CPre` and `CPost annotations`).

Pre-, post-conditions and type or class invariants are part of the *design by contract* software construction method [13].

3.2 Validation and Code Generation

First, a model checker can be used to check that the invariant is not violated as Fig. 1 shows. TLC [8] model checker is given to validate the entire `System`. The logic of the processes is encoded into TLA+ and combined with the resource specification so as to explore the interleavings that the real system can afford. By considering this validation scenario, stronger invariants can therefore be proved. Figure 1 points out that a *test generation* tool can be used for testing a large set

of traces that explore all the system states up to a given depth. A typical tester executes between 500 and 1,000 different traces of the system to be checked. By exploring traces it is also possible to locally detect any malfunction of flawed parts of the system. Students can use testers to discover what is wrong with their implementations.

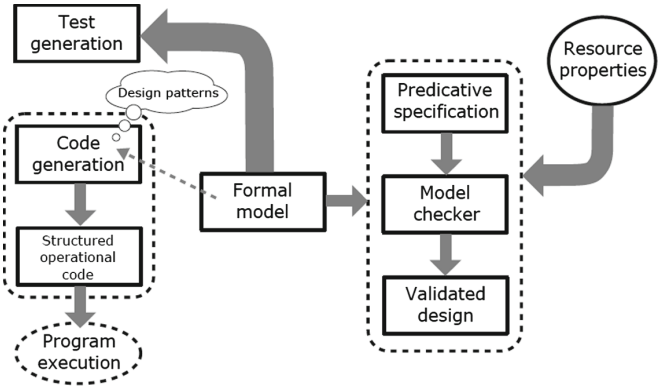


Fig. 1. Suggested development of the teaching model

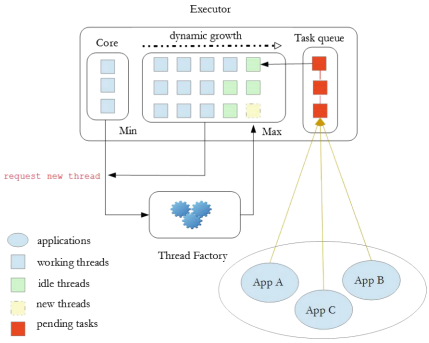
Students are then instructed to deliver a code snippet that implements the concurrent shared resource behavior. The programming work done by the students has to prove the correct use of the parallel and concurrent constructs taught during the course and it is graded as the 50% of the assignment. By doing so, a set of design patterns are used to transform the formal model of a resource specification into C++11 or Java code. This transformation is susceptible of being automated for specific cases. Concurrent properties (safety, fairness, etc.) must have been assured through correct synchronization programming. Different synchronization idioms are suggested for programming thread interaction (*notify–notifyAll*, *locks* and *conditions*, *MPI operations*, etc.) to students to implement the required code.

4 Set of Selected Patterns

Since our *programming patterns* are aimed at the coding level and, unlike *algorithmic skeletons* or *structured parallel patterns* [3], they do not hide concurrent instructions or synchronization operations. The connection topology or low-level dependencies are not hidden either in the parallel algorithms used.

The main role of a parallel design pattern is to find solutions for different aspects that must be addressed when designing a parallel application or algorithm, i.e., to be capable of finding concurrency, and then to determine a suitable algorithm structure and to define its supporting structure (data, communications, user-interface). Finally, the implementation mechanism must be described.

In order to give a general overview of our teaching method, we present here only three of the patterns included in the set of Table 1 and the rest of those can be found in the prior publication [1].



Initial: #idleThreads ← N; finished(k)← false;
Invariant: #idleThreads <= size(TaskQueue)
Reachability: $\diamond\Box task(\bullet)\ at\ finished(\bullet)$
Liveness:$\Box \diamond thread(i)\ at\ freeThread(i)$

Fig. 2. Shared resource model for the executor pattern

4.1 Thread Creation: Executor Pattern

In this pattern tasks can be considered as logical units of work and threads are a mechanism by which tasks can run asynchronously. A graphical high-level model of the pattern is shown in Fig. 2.

When students attend first courses on concurrent and parallel programming, they program by assigning a thread per task, or sequentially executing all the applications's tasks on a single thread. Assigning one thread per task is a bad solution that might lead to poorly performant implementations, and a sequential approach yields extremely bad application responsiveness. We propose to our students to learn and use a high-level pattern for obtaining performance for thread creation and launching in concurrent applications.

The *executor* pattern can be seen as a variant of the *producer-consumer* concurrency paradigm. Application activities that submit tasks to the *executor* monitor have a *producer*-behavior and the executor's threads that pick up from the queue and execute tasks have a *consumer*-behavior. The fundamental idea that supports the executor pattern is to set up an adjustable number of threads that sit idle, waiting for any pending work on the task queue that they can perform.

Predicative Specification of the Executor Pattern. The resource's operations are `executeTask()`, `freeThread()` and `nextTask()` and must be defined in the context of a monitor to synchronize the concurrent access to resource `TaskQueue` list. The correct states of the resource are defined as the invariant in the Semantics Domain section of Fig. 3, i.e., the number of pre-created threads cannot exceed the maximum number of tasks waiting on the queue.

The behavior of the operations above is expressed in the form of pre- and post-conditions (`CPre` and `CPost annotations`). When an application has a task to execute, it calls the method `executeTask(i)`, which inserts the task into the `TaskQueue` list and informs the executor's thread-pool that there is a new executable task. One of the idle threads calls `nextTask()` and starts executing the returned task; when the execution of the *task-i* finishes, the program calls `freeThread(i)` method and goes back to waiting for the next task to perform.

After doing the shared resource specification, the student must choose the correct concurrent language *idioms*, i.e., *notify()*, *locks*, *conditions*, etc. in order to correctly synchronize operations on the *TaskQueue* shared resource and this part of the exercise will then be completed.

```
TaskQueueExecutor
Operations
  executeTask(processId == i)
  freeThread(processId == i)
  nextTask():processId == i
Semantics  Domain:
   Type: TaskQueue(0..N-1) == seq N, processId: 0..N-1
   Invariant: #idleThreads <= size(TaskQueue);
   CPre: size(TaskQueue) > 0 and #idleThreads > 0;
   int nextTask(){} //operation
   CPost: size(TaskQueue) == size(TaskQueue)@pre - 1;

   CPre: #idleThreads >0 and size(TaskQueue) >= 0;
   void executeTask(i){}  //operation
   CPost: size(TaskQueue) == size(TaskQueue)@pre + 1;

   CPre: size(TaskQueue) >= 0 and #idleThreads > 0;
   void freeThread(i){} //operation
   CPost: #idleThreads >= 0 and
        #idleThreads == #idleThreads@pre - 1
```

Fig. 3. `TaskQueue`-monitor specification for the executor

4.2 Monitors: Readers and Writers Protocol

The problem of readers and writers is one of the classic problems in concurrent programming (Fig. 4). There is a shared resource that two types of processes try to access: the *readers* access the resource to obtain information, but do not modify it; the *writers* modify the shared resource when they get access to it. Because the readers do not modify the shared resource, multiple readers may be accessing it at the same time. However, no other process can access the shared resource while a writer is already in.

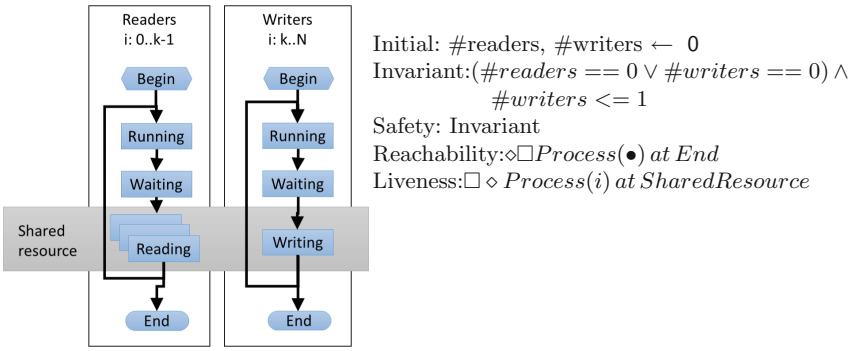


Fig. 4. Shared resource monitor pattern for reader/writer access

Predicative Specification Model. The problem begins from a situation where there are no processes accessing to the shared resource. When a process tries to access it, it must first check that there are no processes of the other type already accessing to the resource, i.e., the condition $\#readers == 0 \vee \#writers == 0$ is satisfied. In any case, there can be only one writer accessing to the resource at the same time, i.e., the condition: $\#writers <= 1$ has to be part the invariant.

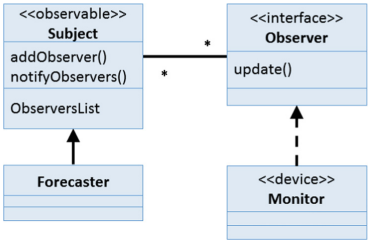
When processes of both types are trying to access the shared resource it is necessary to decide which of them can access it. The readers-writers problem can be solved by giving readers or writers higher priority to access the resource. In case of prioritizing readers, only when there are no other readers trying to access the shared resource, the lock of writers is released. On the other hand, if writers are prioritized, it will be the writers which will unlock the readers when there are no other writers trying to access the shared resource, as Fig. 5 shows.

```
SharedResource
Operations
  void* read(void* p)
  void* writer(void *p)
Semantics  Domain:
   Type: SharedResource == SQL_Type, readerId: 0..N-1,
       writerId: 0..M-1
   Invariant: (#readers==0 or #writers==0) and #writers <= 1;

   CPre: #writers == 0 and #attending_writers == 0;
   void* read(void* p){} //operation
   CPost: #readers == #readers@pre + 1;

   CPre: #writers == 0 and #readers == 0;
   void* write(void* p){}  //operation
   CPost: #writers == 1;
```

Fig. 5. SQL-`SharedResource` specification with priority to writers

Obviously, a third possibility consists of not giving priority to any of these process categories. The semaphore-based solution to the readers/writers problem with equal priorities is a little brain teaser. The students are asked to solve the equal priorities problem in order to motivate the students to follow our systematic scheme (based on *shared resource* formal specification) to find a correct solution by themselves (Fig. 5).

The correct solution to the readers/writers problem with equal priority includes many aspects of the versions with priorities, but also the need for a second level of synchronization. Before allowing a process to check the processes that are already accessing to the shared resource, a stage must be added to determine the order in which the processes have to proceed. With this problem the students learn to design multi-level synchronization protocols by using shared variables.



Invariant: $queue.q$!= $null \wedge queue.q.size() = m$
$if\ m\ items\ were\ added \wedge queue.q[i] = k_i$
$where\ k_i\ is\ the\ i\text{-}th\ item\ added$
Safety: Invariant
Liveness: $\square \diamond Subject(i).notifyObservers()$
Guarantee: $\square(Subject(i)\ at\ notifiable\_state \wedge$
$Observer(\bullet).notify() \rightarrow \diamond update())$

Fig. 6. `ObserversList` data structure for the *Observer* pattern

4.3 Real-Time Systems Design: The Observer Pattern

The standard Observer pattern was considered during the course to introduce real-time programming to students. `Observer` pattern serves to map `Subject` to `Observer` entitity-roles in algorithms and applications. The objective of the pattern is to keep consistency between the state of the object with the `Subject` role and the state(s) of the object(s) with `Observer` roles.

Predicative Specification Model. Each `Subject` entity of the application maintains a set of references to the observers attached to it. Each `Subject` has a `ObserverList` queue as its representation type, which defines `attach()` and `detach()` methods for adding and deleting observers, respectively, to that observers-list. `Subject` entities will provide a `notify()` method as well, which has to be immediately invoked whenever the subject's state experiences a change. A call to `notify()` method of the `Subject` must guarantee that the method `update()` is invoked on each of the observers in the list. A call to an observer's `update()` method changes the observers state to make it consistent with the new `Subject` state.

When the Subject's `notify()` is called, it is compulsory, according to the standard specification in Fig. 7, that the `update()` method is invoked on each attached observer, which updates the `observer` state and propagates the call to its successor on the `ObserverList` queue. Therefore, the `notify()` call needs only invoke `update()` on the first observer in the chain.

```
ObserverList
bool[N] notified=false; //N observers attached to Subject
Operations
  attach(ObserverId == i)
  detach(ObserverId == i)
  notify(): {true, false}
Semantics  Domain:
   Type: ObserverList(0..N-1) == seq N, ObserverId:0..N-1
   Invariant: #observers >= size(ObserverList);
   CPre: size(ObserverList) >= 0 and  #observers > 0;
   void attach(i){}
   CPost: size(ObserverList) == size(ObserverList)@pre + 1;

   CPre: #observers >= 0 and size(ObserverList) > 0;
   void detach(i){}
   CPost:  size(ObserverList) == size(ObserverList)@pre - 1;

   CPre: size(ObserverList) > 0 and #observers > 0;
   boolean notify(i){}
   CPost: notified(i) == true

   Guarantee: forall k:0..N-1: notified(k) == true;
```

Fig. 7. `ObserversList` and `Subject` operations specification for the Observer pattern

5 Assessment of the Teaching Experience

Learning concurrency in undergraduate courses is generally difficult for students, because of the complexity and depth of the set of concepts that they must master during the course. The main objective of the study proposed here has been to facilitate the work of the students. For this, we have chosen to conduct a teaching approach based on demonstrative teaching that uses patterns as the conceptual guide to ease the communication between teacher and class. In this way, the student got a pattern that allows her to take in new concepts in concurrent programming when these concepts are presented anew by the teacher.

The patterns we have been used to develop this study were carefully chosen to illustrate key aspects of concurrent/parallel programming, e.g., *driver-implementer* patterns allow the programmer to delegate the entire responsibility of tasks management to an *executor*, which also takes in any future asynchronous computation of those tasks; the *executor* is therefore a complex design pattern,

but at the same time it can be considered of enormous usefulness when it is well understood. Another pattern introduced here amounts to the synchronization of simultaneous accesses to a shared resource by tasks of *reader* and *writer* type. This is another situation that must be frequently tackled by programmers, and because of that we have included a specific pattern in the demostrative set that reflects such synchronization between reader and writer threads.

We assessed how the model improves the final results obtained by the students on the subject on completing our course. We also noticed more active participation of students during lecture-oriented lessons. Additionally, working time spent in the classroom became actually fun and optimized, and the breadth and depth of the contents covered increased too.

5.1 Evaluation of the Study Results

To evaluate the results obtained from the course teaching experience, we ellaborated a survey form that included four dimensions to be evaluated by the students:

(a) The concepts introduced in lectures were better apprehended through the models provided by our set of *parallel patterns*.
(b) The number of exercises was adquate.
(c) The time required to complete the assigned exercises was sufficient.
(d) The students were satisfied with this approach to the teaching of concurrency and parallelism.

All the four study dimensions were evaluated with a score between 1 (completely disagree) and 5 (completely agree) with the inclusion of an additional value (0) for when the student does not want to answer. The survey was made available to a sample of n = 67 students at the end of the semester. The results obtained are illustrated in Fig. 8. It is observed that the students significantly improved their understanding of the concepts explained in the theoretical classes, the number of programming exercises developed in the laboratories and their

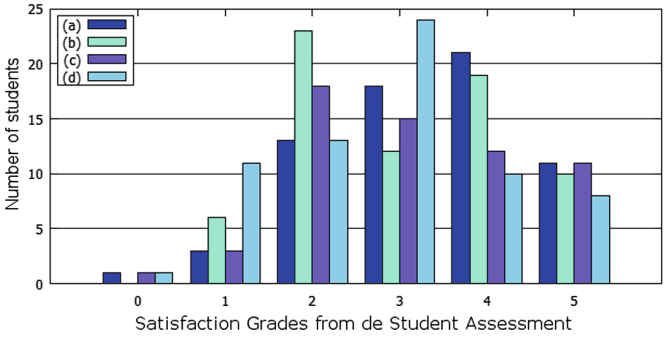


Fig. 8. Students' assesment

weekly work assignments were considered as *adequate*, they had enough time to finish the assignments and exercices, and thus the satisfaction level with the pattern-based concurrency/parallelism model was generally high or very high among the students.

6 Conclusions and Future Work

The outcome of the ICT-based experiment has been a noticeable improvement in the grades obtained by the students in the final examination of the subject. This ICT-based interactive teaching approach can easily be applied to other courses in many different areas beyond the sphere of normal university courses. Teaching projects such as MOOC could immediately benefit from our approach on many of their engineering courses since in these cases our method would only require a simple adaptation of specific course contents.

In the long term, we intend to develop a pattern-based CPP teaching tool in the Cloud which would facilitate systematic learning for any student or person interested and which would enable the method and techniques discussed in this paper to be implemented. Our future work is focused on extending the set of patterns proposed here, developing a course (MOOC) with exercises in several programming languages, such as Java, C++ and MPI. We will include new programming languages that are of interest for industry in the future.

References

1. Capel, M.I., Tomeu, A.J., Salguero, A.G.: Teaching concurrent and parallel programming by patterns: an interactive ICT approach. J. Parallel Distrib. Comput. **105**, 42–52 (2017)
2. Carro, M., Herranz, A., Mario, J.: A model-driven approach to teaching concurrency. ACM Trans. Comput. Educ. **13**(1), 1–19 (2013)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. ACM/MIT Press, New York/Cambridge (1991)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software. Addison-Wesley, Reading (1994)
5. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Reading (2006)
6. Grossman, D., Anderson, R.E.: Introducing parallelism and concurrency in the data structures course. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science and Education (SIGCSE 2012), pp. 505–510. ACM, New York (2012)
7. Hadjerrouit, S.: Towards a blended learning model for teaching and learning computer programming: a case of study. Inf. Educ. **7**(2), 181–210 (2008)
8. Joshi, R., Lamport, L., et al.: Checking Cache-Coherence Protocols with TLA+. https://www.microsoft.com/pubs/65162/fmsd.pdf. Accessed Sept 2016
9. Law, K., Lee, V., Yu, Y.: Learning motivation in e-learning facilitated computer programming courses. Comput. Educ. **55**(1), 218–228 (2010)
10. Manzano, M.: Extensions of First Order Logic. Cambridge University Press, Cambridge (1996)

11. Marowka, A.: Think parallel: teaching parallel programming today. IEEE Distrib. Syst. Online **9**(8), 1–8 (2008). Article no. 0808-o8002
12. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley, Reading (2004)
13. Mitchell, R., McKim, J.: Design by Contract, by Example. Addison-Wesley, Reading (2002)
14. Mohorovicic, S., Tijan, E.: New technologies in teaching university level programming. In MIPRO, 2010 Proceedings of the 33rd International Convention, Opatija, Croacia, pp. 1024–1028 (2010)
15. Papp-Varga, Z., Szlavi, P., Zsako, L.: ICT teaching methods-programming languages. In: Annales Mathematicae et Informaticae, vol. 35, pp. 163–172 (2008)
16. Saraswat, V.A., Bruce, K.: Curricula in concurrency and parallelism. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH 2010), pp. 281–282. ACM, New York (2010)