Concurrency Manager

Sérgio Soares[*] and Paulo Borba[†]
Centro de Informática
Universidade Federal de Pernambuco

Intent

Provide an alternative to method synchronization with the aim of increasing system performance. *Concurrency Manager* uses knowledge about the semantics of the methods in order to block only conflicting execution flows, allowing the non–conflicting ones to execute concurrently.

Motivation

The advent of web–based information systems significantly increased the number of concurrent programs. Concurrent programs must control concurrency to guarantee safe implementations, which avoid interference that lead systems to inconsistent states and behaviors. To implement some of these controls we need to use programming language features, such as blocking methods to avoid their concurrent execution in the same object. In the Java [6] programming language we can do this synchronizing methods with the `synchronized` method modifier, which forbid concurrent execution of methods within an object.

However, implementation of such features brings performance overhead, serializing the execution of some operations. There are several approaches concerned about guaranteeing performance increasing, removing unnecessary synchronization of Java concurrent programs [3, 1, 5]. They show the negative impact in efficiency of the Java concurrency control mechanisms. This negative impact demands alternatives to increase programs' performance.

Method synchronization guarantees that all concurrent execution of a method within an object will be serialized. With this approach we can allow or block all concurrent execution of a method. However, if some execution flows cannot be concurrently executed, but others can, we need an intermediary approach.

Example

Consider an address book application with a class `AddressBook` that has a method to register addresses. The method verifies, before registering an address in the system, if

[*]Supported by CAPES. Email: scbs@cin.ufpe.br
[†]Partially supported by CNPq, grant 521994/96–9. Email: phmb@cin.ufpe.br

there is an address with the same email of the address being registered. Two concurrent executions that try to register addresses with the same email may get the same answer: there is not an address with the email of the objects being registered. In this case both execution flows will try to register the objects, which may turn the system to an inconsistent state, or raise an unexpected error. We can conclude that this method cannot be concurrently executed if the objects (addresses) being registered have the same email, otherwise concurrent execution is allowed. Therefore the address' email can be used to decide if an execution flow can or cannot be concurrently executed. Figure 1 shows an UML [4] class diagram of this application.

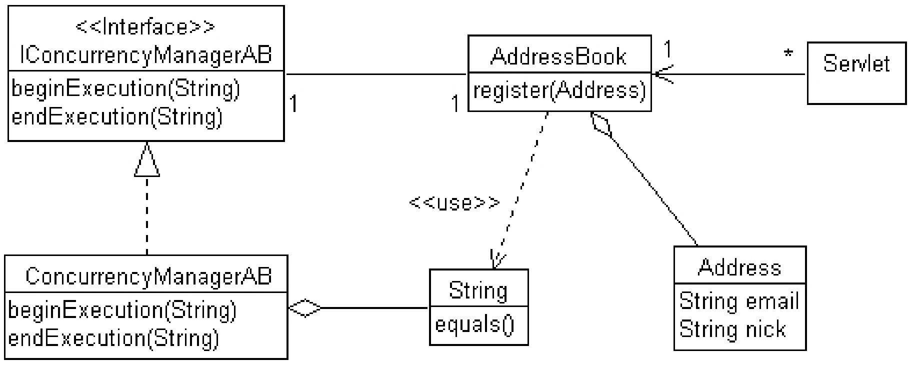


Figure 1: Address Book application's class diagram.

The class `ConcurrencyManager` is user to control the `AddressBook` `register` method execution. The method should ask permission to the `ConcurrencyManager` before executing, calling the `beginExecution` method with the address's email as the argument. If another address is being registered with the same email by other execution flow, this execution is blocked until the executing flow terminates.

The email information is part of the method semantics, as the method's parameters and the object's state. The *Concurrency Manager* pattern uses such information to block only the conflicting execution flows.

Either persistent application that uses databases management systems (DBMS) must make some concurrency controls. So this pattern can be also used in such systems, besides the idea that persistent systems already make all concurrency controls using the DBMS features, which is not true [9].

Applicability

Use *Concurrency Manager* when

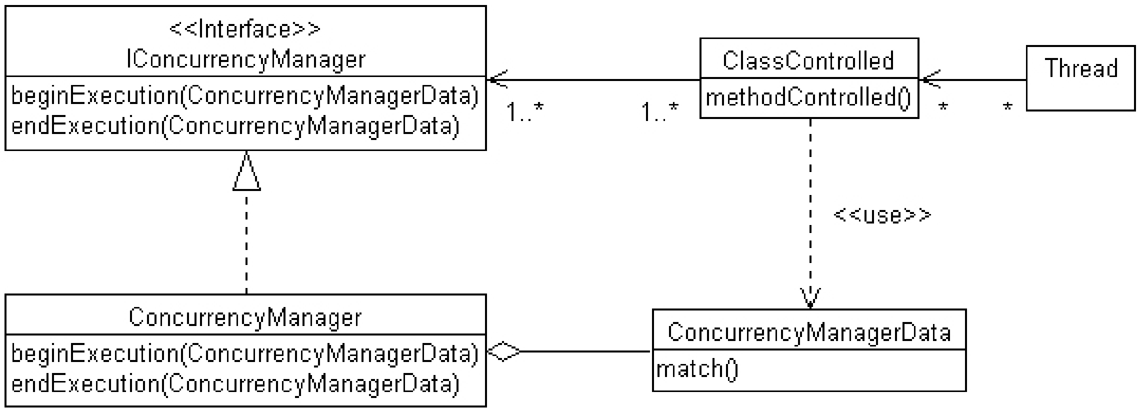
- You need to control concurrent access to an object, blocking just some concurrent execution of a method within the object, allowing others. In the previous example concurrent addresses registration can be executed since the addresses have not the same email. Only the registrations of addresses with the same emails must be serialized (synchronized).

- You need to control concurrency in more than one method; some flows can execute concurrently in the methods and others cannot. For example, in a banking application you can concurrently execute the methods deposit and withdraw, but for different accounts. The execution of deposit and withdraw for a same account must be serialized to avoid inconsistencies.

- You need to control methods in different objects. The objects must have the same instance of the pattern to manage the concurrent execution in the objects. In this case, one instance of the pattern manages methods execution in several objects.

Structure

The structure of *Concurrency Manager* is presented in the Figure 2 using an UML [4] class diagram. This diagram is a generic diagram, if compared with the diagram presented in Figure 1, which defines a class to encapsulate the information used to decide if an execution flow should be blocked.



<<Interface>>
IConcurrencyManager
beginExecution(ConcurrencyManagerData)
endExecution(ConcurrencyManagerData)
1..*
1..*
ClassControlled
methodControlled()
*
*
Thread
<<use>>
ConcurrencyManager
beginExecution(ConcurrencyManagerData)
endExecution(ConcurrencyManagerData)
ConcurrencyManagerData
match()

Figure 2: *Concurrency Manager* pattern's class diagram.

Participants

The participants of the pattern are

- `ClassControlled`. A concrete class with methods that can be concurrently executed in some cases and cannot in others. The classes must have one or more instances of the `ConcurrencyManager` to synchronize its methods.

- `Thread`. The thread that executes the controlled method(s) of the *ClassControlled* objects.

- `IConcurrencyManager`. An interface responsible to abstract the pattern implementations and its extensions.

- `ConcurrencyManager`. A concrete class, which implements the `IConcurrency-Manager` interface, and is responsible to control the concurrent execution of the `ClassControlled` objects. This control is made based in the operation's semantics, which is encapsulated in the `ConcurrencyManagerData` object.

- `ConcurrencyManagerData`. A concrete class with the information used to forbid a method execution. The class also has a method `match` to compare two instances of the class. This method implementation is responsible to define, based in the information of their attributes, if a `ConcurrencyManagerData` object matches any `ConcurrencyManagerData` object already inserted in the manager, which means that the current execution flow may conflict with another one, and hence must be blocked.

Collaborations

Figure 3 shows a collaboration diagram modeling how a method can use the concurrency manager to control concurrent execution. After being called by an execution flow (message 1), the controlled method creates a `ConcurrencyManagerData` object with the relevant information to decide if an execution flow can execute concurrently (message 1.1). After that, the manager's `beginExecution` method is called with the created data as argument (1.2). Based in the stored data objects, the manager verifies if there is a data object that matches the object passed by the controlled method (1.2.1). If the objects match, the controlled method is blocked (1.2.2); otherwise, the data object is stored in the manager (1.2.3) and the controlled method executes. Just before terminating, the controlled method calls the manager's `endExecution` method with the same data object created in the beginning (1.3). This method call removes the data from the manager (1.3.1) and notifies the blocked execution flows (1.3.2), which become ready to execute again.

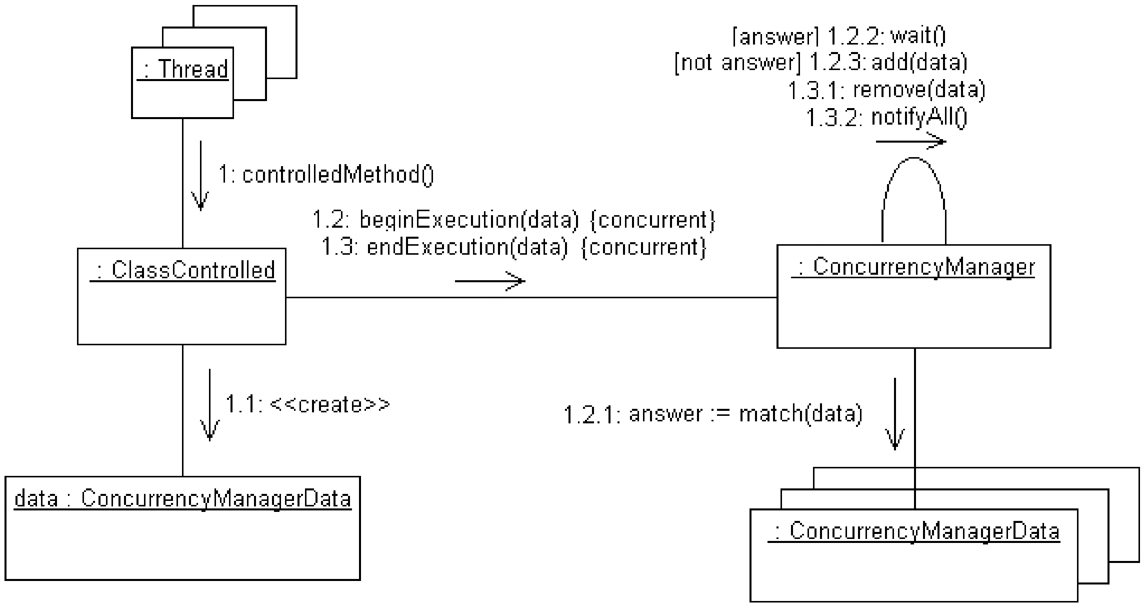


Figure 3: *Concurrency Manager*'s collaboration diagram.

In the collaboration diagram (Figure 3) the constructs { *concurrent* } (messages 1.2 and 1.3) means that in the presence of multiple flows of control, the operation will be treated as atomic. Java supports this construct with the `synchronized` method modifier.

Figure 4 shows a sequence diagrams that specifies an execution of a controlled method without conflicting flows.

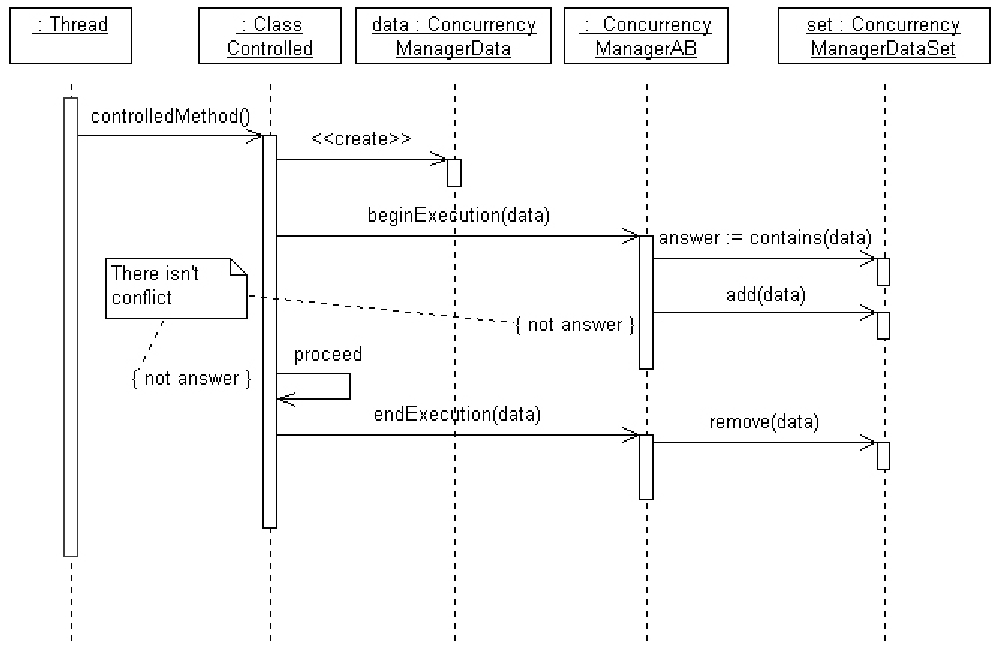


Figure 4: *Concurrency Manager*'s sequence diagram of an execution flow without conflict.

Other scenario is presented in Figure 5, which shows a sequence diagram that specifies an execution of a conflicting execution flow.

Consequences

The benefits of the pattern are:

- *Performance increase.* The system performance is increased by the elimination of unnecessary synchronization. The pattern uses the system operations' semantics to block only conflicting concurrent execution. Performance tests made to analyze the efficiency impact of using this pattern showed that the performance increasing was about 20%, depending of some aspects, such as the operations workload and the number of concurrent threads. With high workload, we can see a low synchronization overhead, on the other hand, with high number of concurrent threads the synchronization overhead will be greater [9]. These aspects variation take the performance increasing in a range from 5% to 60%.

- *Reuse.* The manager class and the class responsible for the data used by the manager can be reused in several systems.

- *Extensibility and maintainability.* The pattern uses an interface to abstract different implementations of the manager. So, we can have different implementations of the pattern, for example, an implementation to be used in sequential environments, which does not make any concurrent control, simplifying its implementation before migrating the system to the concurrent environment. Note that the pattern's structure allows changing the concurrency control without making modifications in business classes. This is possible because the use of the `ConcurrencyManagerData`

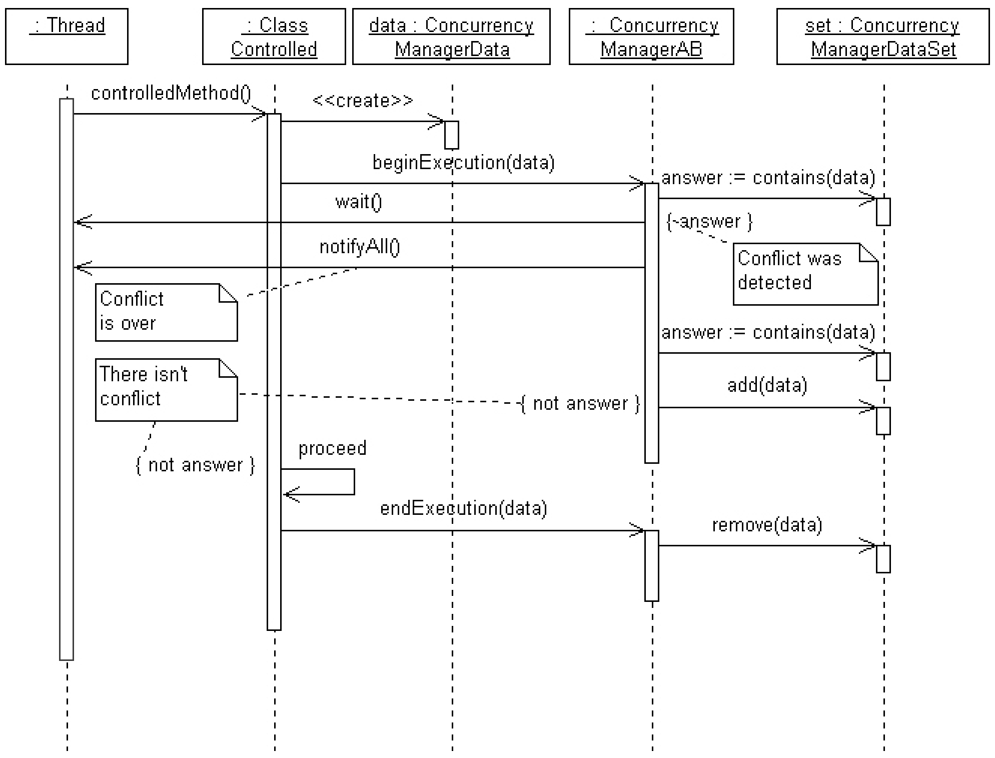


Figure 5: *Concurrency Manager*'s sequence diagram of a conflicting execution flow.

and `ConcurrencyManager` classes remove the code responsible for the concurrent control from the business classes, such as the `Address` class. This separation of concerns [7] (business and concurrency control) helps the system extensibility and maintainability.

The liabilities of the pattern are:

- *Increased number of classes.* The class hierarchy becomes more complex because new classes and interfaces are added, decreasing legibility and maintainability.
- *Increased indirection.* In order to introduce our control technique we must delegate some calls to methods, which seems to decrease system performance. In fact, this lost of efficiency is recompensed because only conflicting execution flows are blocked.
- *Complexity.* The controlled method's code is more complex than using the `synchro–nized` modifier, because to implement the pattern we must add about four new lines of code. This contributes to decrease the system legibility and maintainability.
- *Risk of deadlock.* When applying the concurrency control technique to a method, the programmer may forget to properly call the manager's `endExecution` method allowing execution flows to block indefinitely.

Implementation

To implement the *Concurrency Manager* we can use several approaches.

- `ConcurrencyManagerData` set. The simplest approach is the one where the manager keeps a set of `ConcurrencyManagerData` objects. When a thread asks permission to execute a method passing a `ConcurrencyManagerData` object, the manager uses the `ConcurrencyManagerData match` method to verify if there is another object that matches this `ConcurrencyManagerData`. If there is not, the `ConcurrencyManagerData` object is inserted in the `ConcurrencyManagerData` set and the execution may proceed. If there is any, the execution flow is blocked until the execution that inserted the object matched by the `ConcurrencyManagerData` finishes. The `ConcurrencyManagerData` class may store a simple key, as a string, or more complex information that is necessary to decide if an execution flow can execute concurrently with others.

- *State machine.* We can also implement a state machine in the `ConcurrencyManager` class. The manager should have a table to store all the system execution. Probably the `ConcurrencyManagerData` class has to store the name of the method to execute and the object id of the object being executed. With this information, the manager can decide if this execution can be done at this time, verifying if this execution sequence is according with the state machine definition.

Sample Code

Consider the application that registers addresses and verifies, before register an address, if there is an object in the system with the same email of the object to be registered, see Figure 1. A possible concurrent execution that tries to register two objects with the same email may turn the system to an inconsistent state. Consider that in a concurrent execution both threads make the email verification and receive a reply that there is not an object with the email of the ones being registered, so, the threads will try to insert the objects. Note that these execution flows cannot execute concurrently, but the flows that try to insert addresses with different emails can be concurrently executed. Therefore, we can implement the *Concurrency Manager* pattern to control these executions.

The following implementation of the *Concurrency Manager* makes a simplification of the pattern. The `ConcurrencyManager` class keeps a keyword set (`String` set) that is used to control concurrent execution over then, instead to keep a `ConcurrencyManagerData` set. To implement this set we use the `java.util.HashSet`, a class that implements an object set without any concurrency control.

```
public class ConcurrencyManager {
    private HashSet keys;
    public ConcurrencyManager() {
        keys = new HashSet();
    }
```

We need to define a method to receive a `String` parameter to inform that a thread will start to execute some operation over this `String`. If this keyword is already in the set the execution is blocked, meaning that another thread is executing over this keyword. The `HashSet` class has a method to verify if there is an object in the set. We use this method to find if the keyword is already in the keyword set. We also use the method wait inherited from Object class, superclass of all Java classes. This method blocks an execution until being notified to resume it.

```
    public synchronized void beginExecution(String keyword) {
        try {
            while (!keys.add(keyword)) {
                wait();
            }
        }
        catch(InterruptedException ex) {
            throw new RuntimeException("Unexpected error");
        }
    }
```

We also need a method to inform that the execution over a String (keyword) is finished. This method removes the keyword of the set, using the HashSet remove method, and releases a thread that is blocked waiting to execute by calling the method notifyAll, other method inherited from Object.

```
    public synchronized void endExecution(String keyword) {
        try {
            if (!keys.remove(keyword)) {
                throw new RuntimeException("Keyword not found");
            }
        }
        finally {
            notifyAll();
        }
    }
}
```

Now we need programming in the business class the code that negotiates with the manager. The conflicting executions are the ones that try to register objects with a same email. So we use the ConcurrencyManager class to synchronize only the conflicting execution, which are the ones registering addresses with the same email. The keyword to be used is the object's email, so if two threads try to register two objects with the same email, the second one's execution is blocked until be released by the first one. The following example shows how the ConcurrentManager class is used in this example.

```
public class AddressBook {
    private AddressCollection addresses;
    private ConcurrencyManager manager;
    ...
```

The AddressBook class defines an AddressCollection, which is responsible to store the Address objects, and a ConcurrencyManager that is responsible to control the concurrency over the method register.

```
      public void register(Address address) throws EmailException {
 1:         String email = address.getEmail();
 2:         try {
 3:             manager.beginExecution(email);
 4:             if (!addresses.hasEmail(email)) {
 5:                 addresses.insert(address);
 6:             }
 7:             else {
 8:                 throw new EmailException();
 9:             }
10:         }
11:         finally {
12:             manager.endExecution(email);
13:         }
      }
  }
```

As we sad before, this example makes a simplification when do not use a `Concurrency–ManagerData` object to send the method information to the *Concurrency Manager*. In the `register` method of the `AddressBook` class we use the method's semantics, in this case the address' email, to avoid invalid concurrent execution, as described before, calling the `beginExecution` method of the `ConcurrencyManager` class (line 3). This method call must be done before execute the method in order to ask the manager permission to continue the execution. Therefore, other thread that tries to register another address with the same email will be blocked. Just before terminating the execution we must call the `endExecution` method (line 12) telling the *Concurrency Manager* to remove the key added in the manager's set and to release the blocked threads, if there are any. This execution is made inside a `finally` clause, which guarantees that the command will be executed, independent of what happen in the method execution, since the method execution may raise an exception before finishes its execution [6]. If this occurs and programmer forgot to call the `endExecution` method inside a `finally` block, the key will not be removed from the manager's set, which allows the executions flows, blocked because of this key, to block indefinitely.

Known Uses

The *Concurrency Manager* pattern uses the idea of semantics–based concurrency control [2]. This approach uses the operations' semantics to improve performance, decreasing operations' serialization. For example, "two operations conflict if they both operate on the same data item and one of them is a write". The concurrency pattern differs from this approach because the programmer must define what is the semantics of conflicting operations, when implementing the concurrency manager data.

A potential use of the pattern to control the concurrency is in web–based systems with a software architecture that has a business collection and a data collection for each basic class. The business collections are classes where the system policies are implemented, for example, the `AdressBook` class used in the previous sections. The data collections are classes responsible for data storage, as the `AddressCollection` class, and basic classes are classes that model the system's basic objects, for example, the `Address` class. We can mentioned many real web-based systems that use this architecture:

- A system to manage a telecommunication company's clients. The system is able to register mobile telephones and change clients and telephones services configurations. The system can be used over the Internet.
- A system for performing on–line exams. This system has been used to offer different kinds of exams, as simulations based on previous university entry exams, which help students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system responsible to control the sales in a market. This system has been used in several supermarkets and other kinds of stores.
- A system for registering health system complaints. The system allows citizens to complaint about diseases problems and to retrieve information about the public health system, such the location or the specialties of a health unit.

Our approach can be used in the business collection classes of these systems, where business polices may race conditions, as the one in the address book example.

We made performance tests [9] in a toy system that implements the pattern in order to analyze the efficiency impact. As we say in the Consequences Section, the performance increasing goes from 5% to 60%, depending on some aspects such as method workload and the number of concurrent threads.

Related Patterns

A related pattern is the *Monitor Object* [8] that synchronizes the execution of methods. This pattern also allows methods to cooperate scheduling their execution sequences by waiting and notifying each other via monitor conditions. The monitor conditions determine when a method should suspend, and when resume. In the *Monitor Object* approach, the controlled methods has to choose what is the monitor condition to wait or to notify. In the *Concurrency Manager*, the manager is the responsible to say when a method can or cannot execute concurrently. This decision is encapsulated in the manager's definition. In fact, the main *Concurrency Manager*'s goal is to allow as many as possible concurrent execution, to increase the system efficiency, on the other hand, the main *Monitor Object*'s goal is to synchronize objects methods execution. This similarity allows us to classify our pattern as a *Concurrency Pattern* [8], like the *Monitor Object* pattern.

The *Concurrency Manager* pattern may implement the Singleton design pattern to guarantee that there is a single instance of the manager. This is necessary if we try to centralize all concurrency controls in a single manager to control all the system executions.

Acknowledgments

We would like to give special thanks to Jorge L. Ortega Arjona, our shepherd, for his important comments, helping us to improve our pattern. We also thanks Gunter Mussbacher for the suggestions made at the conference.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, November 1999.

[2] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46, November 1999.

[4] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language – User's Guide.* Addison–Wesley, 1999.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM, November 1999.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison–Wesley, second edition, 2000.

[7] David L. Parnas et al. Using documentation as a software design medium. *The Bell System Technical Journal*, 60(8):1941–1977, October 1981.

[8] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, 2000.

[9] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relacional Databases (in portuguese). In *V Brazilian Symposium on Programmig Languages*, pages 252–267, Curitiba, Brazil, 23th–25th May 2001.