Applying Idioms for Synchronization Mechanisms Synchronizing communication components for the One-dimensional Heat Equation

Jorge L. Ortega Arjona

[1]Departamento de Matemáticas Facultad de Ciencias, UNAM
jloa@fciencias.unam.mx

Abstract. *The Idioms for Synchronization Mechanisms is a collection of patterns related with the implementation of synchronization mechanisms for the communication components of parallel software systems. The selection of these idioms take as input information (a) the design pattern of the communication components to synchronize, (b) the memory organization of the parallel hardware platform, and (c) the type of communication required.*

In this paper, it is presented the application of the idioms for synchronization mechanisms to implement communication components for the One-dimensional Heat Equation. The method used here takes the information from the Problem Analysis, Coordination Design, and Communication Design, selecting an idiom for synchronization mechanisms, and providing elements about its implementation.

1. Introduction

For the last forty years, a lot of work and experience has been gathered in concurrent, parallel, and distributed programming around the synchronization mechanisms originally proposed during the late 1960s and 1970s by E.W. Dijkstra [4], C.A.R. Hoare [6, 7, 8], and P. Brinch-Hansen [1, 2, 3]. Further work and experience has been gathered today, such as the formalization of concepts and their representation in different programming languages.

Synchronization can be expressed in programming terms as language primitives, known as synchronization mechanisms. Nevertheless, merely including such synchronization mechanisms into a language seems not sufficient for creating a complete parallel program. They neither describe a complete coordination system nor represent complete communication subsystems. To be applied effectively, the synchronization mechanisms have to be organized and included within communication structures, which themselves have to be composed and included in an overall coordination structure [12].

Common synchronization mechanisms for concurrent, parallel and distributed programming can be expressed as idioms, that is, as software patterns for programming code in a particular programming language. Several of such synchronization mechanisms have been already expressed as idioms: the Semaphore idiom, the Critical Region idiom, the Monitor idiom, the Message Passing idiom and the Remote Procedure Call idiom [12]. All these idioms are presented by describing the use of the synchronization mechanism with a particular parallel programming language, rather than a formal description of their theory of operation.

The objective of this paper is to show how the idioms that provide a pattern description of well-known synchronization mechanisms can be applied for a particular programming problem under development. The description of synchronization mechanisms as idioms should aid software designers and engineers with a description of common programming structures used for synchronizing communication activities within a specific programming language, as well as providing guidelines on their use and selection during the design and implementation stages of a parallel software system. This development of implementation structures constitutes the main objective of the Detailed Design step within the Pattern-based Parallel Software Design method [12].

When implementing the components that act as synchronization mechanisms within the communication components of a parallel program, it is important to carefully consider how both communication and synchronization are carried out by such synchronization mechanisms. Idioms for Synchronization Mechanisms [12] stands out from many of the sources, references, and descriptions available about how to implement the synchronization between communicating components (or processes) of a parallel program, with the following advantages:

- The Idioms for Synchronization Mechanisms represent programming constructs that express synchronization beyond what is properly included within the parallel programming language, but giving the impression that their use is actually part of the parallel language.
- The Idioms for Synchronization Mechanisms attempt to reproduce good programming practices, describing some common programmed structures used to detail and implement the synchronization required by a Design Pattern for Communication Components. Thus, their objective is to help the software designer or programmer understand and master features and details of the parallel programming language at hand, by providing low-level, language specific descriptions of code that are used to synchronize between parallel processing components. These Idioms, then, help to solve recurring programming problems in such a parallel programming language. There has been extensive experience and research about such codification in several different parallel programming languages, but unfortunately, they have not been related or linked with general communication structures or overall structures of parallel programs.
- The Idioms for Synchronization Mechanisms are descriptions that relate a synchronization function (in run-time terms) with a coded form (in compile-time terms). In many parallel languages, synchronization mechanisms are implemented so their run-time function has little or no resemblance to the code that performs it. Both, function and code, are difficult to relate, so the software designer or programmer cannot notice how communication and synchronization are carried out by coded components. The Idioms here try to relate function and code, providing dynamic and static information about the synchronization mechanisms.
- Idioms for Synchronization Mechanisms describe common coded programming structures based on data exchange and function call. As such, they are guidance about how to achieve synchronization between processing components. This is a key for the success or failure of communication. Hence, the Idioms proposed here are classified based on (a) the memory organization and (b) the type of communication between parallel components. These issues deeply affect the selection of

synchronization mechanisms and the implementation of communication components.

- The Idioms for Synchronization Mechanisms represent programmed forms as regular organizations of code, aiming to allow software designers to understand the synchronization between component, and therefore, reducing their cognitive burden. Moreover, if these idioms are used and learnt, they ease understanding legacy code, since programs tend to be easier to understand.
- The Idioms for Synchronization Mechanisms are based on the common concepts and terms originally used for inter-process communication [4, 6, 1, 7, 2, 8, 3], and as such, they are a vehicle to develop terminology for implementing synchronization components for parallel programs.

Nevertheless, as it is obvious, the Idioms for Synchronization Mechanisms present the disadvantage of being non-portable, since they depend on features of the parallel programming language. This does not exclude that several idioms for expressing synchronization mechanisms can be developed for the different parallel programming languages available.

2. Specification of the System

In the paper, *Applying Architectural Patterns for Parallel Programming. Solving the One-dimensional Heat Equation* [11], the Communicating Sequential Elements (CSE) Architectural Pattern was selected as a viable solution for the coordination within the parallel program that solves the One-dimensional Heat Equation. In order to apply the ISM, some information is required related to the CSE Pattern, such as the parallel platform and programming language.

We will use a SUN SPARC Enterprise T5120 Server, a multi-core, shared memory parallel hardware platform, with 1-8 Core UltraSPARC T2 1.2 GHz processors (capable of running 64 threads), 32 Gbytes RAM, and Solaris 10. The programming language will be Java.

3. Specification of the Communication Components

In the paper *Applying Design Patterns for Communication Components. Communicating CSE components for the One-dimensional Heat Equation* [13], the Shared Variable Channel (SVC) Design Pattern was selected as a viable solution for the communication components of the CSE pattern for solving the One-dimensional Heat Equation. In order to apply the ISM, some information related with the Shared Variable Channel Pattern is required. This information is summarized as follows.

3.1. The Shared Variable Channel pattern

The communication components are defined so they enable the exchange of temperature values between neighboring wire segments as ordered data [13]. Therefore, the Shared Variable Channel pattern is an adequate solution for such communications. Hence, the design of the communication components can proceed as follows [9, 11, 13].

- **Description of the communication**. The parallel program that solves the One-dimensional Heat Equation problem is being developed on a multi-core, shared

memory parallel hardware platform, programmable using the Java programming language. The CSE pattern describes a coordination in which multiple **Segment** objects act as concurrent processing software components, each one applying the same temperature operation, whereas **Channel** objects act as communication software component which allow exchanging temperature values between sequential components. Every **Segment** object communicates by sending its temperature value from its local space to its neighboring **Segment** objects, and receiving in exchange their temperature values. This communication is normally asynchronous, considering the exchange of a single temperature value, in a one to one fashion [9, 11, 13].

The **Channel** communication component acts as a single entity, allowing the exchange of information between processing software components. Given that the available parallel platform is a multi-core, shared memory system, the behavior of a channel component is modelled using shared variables. Thus, a couple of shared variables are used to implement the channel component as a bidirectional, shared memory communication means between elements. It is clear that such shared variables require to be safely modified by synchronizing read and write operations from the elements. Hence, the Java programming language provides the basic elements for developing synchronization mechanisms (such as semaphores or monitors). This is required to preserve the order and integrity of the transferred temperature values.

- **Structure and dynamics.** This section takes information of the Shared Variable Channel design pattern, expressing the interaction between the software components that carry out the communication between parallel software components for the actual example.
    1. *Structure.* The structure of the Shared Variable Channel pattern applied for designing and implementing channel communication components of the CSE pattern is shown in Figure 1 using a UML Collaboration Diagram [5]. Notice that the channel component structure allows an asynchronous, bidirectional communication between two sequential elements. The asynchronous feature is achieved by allowing an array of temperatures to be stored, so the sender does not wait for the receiver [10, 13].

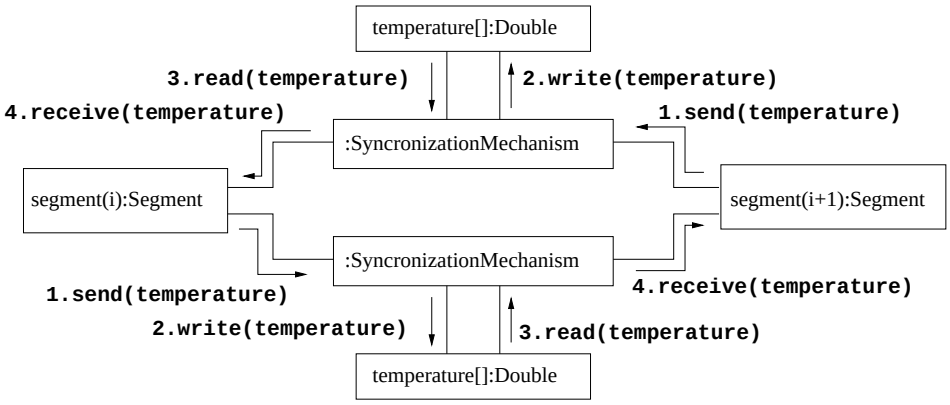

Figure 1. UML Collaboration Diagram of the Shared Variable Channel pattern used for asynchronously exchange temperature values between sequential components of the CSE solution to the One-dimensional Heat Equation.

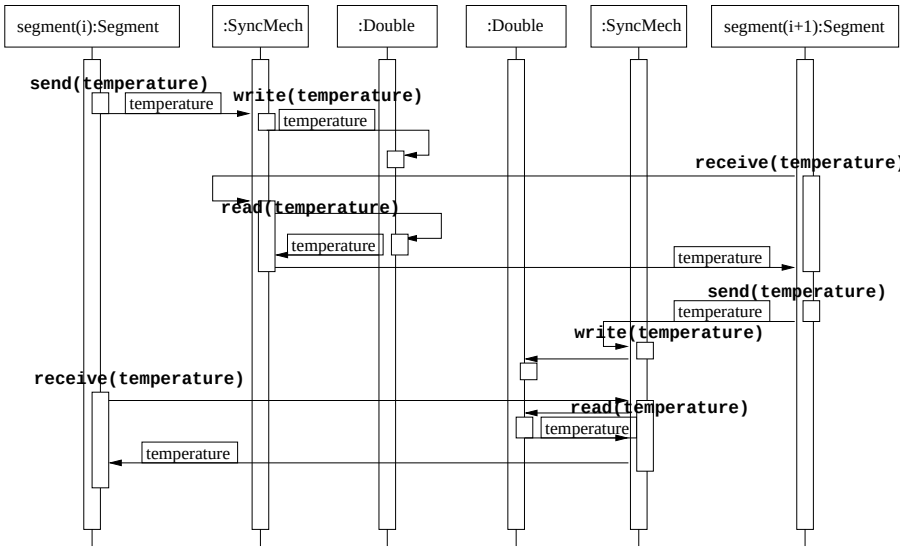

Figure 2. UML Sequence Diagram for the Shared Variable Channel pattern applied for exchanging temperature values between two neighboring sequential elements of the CSE solution for the One-dimensional Heat Equation.

2. *Dynamics.* This pattern actually emulates the operation of a channel component within the available shared memory, multi-core parallel platform. Figure 2 shows the behavior of the participants of this pattern for the actual example.

   In this scenario, a point to point, bi-directional, asynchronous communication exchange of temperature values of type `Double` is carried out, as follows[13]:
   - The `segment(i)` sequential element sends its local `temperature` value by issuing a `send(temperature)` operation to the sending Synchronization Mechanism.
   - This Synchronization Mechanism verifies if the `segment(i+1)` sequential element is not reading the `temperature` shared variable. If this is the case, then it translates the sending operation, allowing a `write(temperature)` operation of the data item on `temperature`. Otherwise, it blocks the operation until the `temperature` can be safely written.
   - When the `segment(i+1)` attempts to receive the temperature value, it does so by issuing a `receive(temperature)` request to the Synchronization Mechanism. This function returns a `double` type representing the temperature value stored in the shared variable `temperature`. Again, only if its counterpart sequential element (here, `segment(i)`) is not writing on `temperature`, the Synchronization Mechanism grants a `read(temperature)` operation from it, returning the requested `temperature` value. This achieves the send and receive operations between neighboring segment elements.
   - On the other hand, when data flows in the opposite direction, a similar procedure is carried out: the local `temperature` value

of `segment(i+1)` is sent by issuing a `send(temperature)` operation to the Synchronization Mechanism.

3. *Functional description of software components.* This section describes each software component of the Shared Variable Channel pattern as the participant of the communication sub-system, establishing its responsibilities, input, and output [13].
   (a) **Synchronization Mechanisms.** This kind of components is used to synchronize the access to the shared variables. Notice that they should allow the translation of `send()` and `receive()` operations into adequate operations for writing to and reading from the shared variables. Normally, synchronization mechanisms are used to keep the order and integrity of the shared data. In this paper, the objective is to obtain the development and implementation of these synchronization mechanisms for the available shared memory parallel platform and in the Java programming language.
   (b) **Shared Variables.** The responsibility of the shared variables is to store the temperature values exchanged by sequential elements. These shared variables are designed here as simple variables that buffer during communication, for actually achieving an asynchronous communication.

4. Detailed Design

In the Detailed Design step [12], the software designer selects one or more idioms as the basis for synchronization mechanisms. From the decisions taken in the previous steps (Specification of the Problem [11], Specification of the System [11], and Specification of Communication Components [13]), the main objective now is to decide which synchronization mechanisms are to be used as part of the communication substructures.

4.1. Specification of the Synchronization Mechanism

– **The scope.** This section takes into consideration the basic previous information for solving the One-dimensional Heat Equation. The objective is to look for the relevant information for choosing a particular idiom as a synchronization mechanism.

  For the One-dimensional Heat Equation, the factors that now affect selection of synchronization mechanisms are as follows:

  * The available hardware platform is a shared memory multicore computer, this is, a shared memory parallel platform, programmed using Java as the programming language.
  * The CSE pattern is used as an architectural pattern, requiring two types of software components: elements and channels [11].
  * The Shared Variable Channel design pattern is selected for the design and implementation of communication components to support asynchronous communication between elements [11].

Based on this information, the procedure for selecting an Idiom for Synchronization Mechanisms for the One-dimensional Heat Equation is as follows [12]:

(a) *Select the type of synchronization mechanism.* The Shared Variable Channel pattern requires a synchronization mechanism that controls the access and exchange of temperature values between elements as software components that cooperate. These temperature values are communicated using a shared variable. Hence, the idioms that describe this type of synchronization mechanism are the Semaphore idiom, the Critical Region idiom, and the Monitor idiom [12].

(b) *Confirm the type of synchronization mechanism.* The use of a shared memory platform confirms that the synchronization mechanisms for communication components in this example are semaphores, critical regions, or monitors.

(c) *Select idioms for synchronization mechanisms.* Communication between elements needs to be performed asynchronously that is, no element should wait for any other element. This is normally achieved using the Shared Variable Channel. Nevertheless, this design pattern requires synchronization mechanisms directly supported by the Java programming language. In Java, the Monitor idiom allows to develop a mechanism used here to show how implementation of the Shared Variable Channel pattern can be achieved using this idiom.

(d) *Verify the selected idioms.* Checking the Context and Problem sections of the Monitor idiom [12]:

   * Context: *'A concurrent, parallel or distributed program in which two or more software components execute simultaneously on a shared memory parallel platform, communicating by shared variables. Each software component accesses at least one critical section that is, a sequence of instructions that access the shared variable. At least one software component writes to the shared variable'.*
   * Problem: *'To maintain the integrity of data, it is necessary to give a set of software components synchronous and exclusive access to shared variables for an arbitrary number of read and write operations'.*

   Comparing these sections with the synchronization requirements of the actual example, it seems clear that the Monitor idiom can be used as the synchronization mechanism for the communication. The use of a shared memory platform implies the use of semaphores, critical regions, or monitors, whereas the need for asynchronous communication between elements points to the use of shared variables. Nevertheless, given that Java directly supports monitors, it is therefore decided that the Monitor Idiom is used as the

basis for the synchronization mechanism.

The design of the parallel software system can now continue using the Solution section of the Monitor idiom, directly implementing it in Java.

– *Structure and Dynamics.*

(a) **Structure.** The Monitor Idiom is used for implementing the synchronization mechanisms of the channel communication components for the CSE pattern. The Monitor idiom in Java is presented as follows. Notice that the monitor allows a synchronization over the shared variables [12].

```
class Monitor{
    ...
    // declarations of shared variables and local data
    private type shared_variables;
    private type local_data;
    ...
    // declarations of methods
    public synchronized type method(type formal_parameters){
        ...
        // operations_on_shared_variables
        ...
    }
}
...
int main(){
    ...
    monitor m;
    ...
    m.method(actual_parameters);
}
```

(b) **Dynamics.** Monitors are used in several ways as synchronization mechanisms. Here, monitors are used for mutual exclusion. The Monitor idiom actually synchronizes the operation of the element components over shared variables within the available shared memory, multi-core parallel platform. Figure 3 shows a UML Sequence diagram of the possible execution of two participants of this idiom as the synchronization mechanism within the Shared Variable Channel pattern. Two parallel software components `segment(i)` and `segment(i+1)` share an array that represents the temperature and, since it is encapsulated within the monitor, it can only be accessed through invocations to the monitor's methods.

In this scenario, the synchronization over the shared variable is performed as follows:

∗ The mutual exclusion between parallel software components starts when `segment(i)` invokes `s`end(temperature). Assuming that the `monitor` is free at that moment, `segment(i)` obtains its lock and performs `w`rite(temperature), which allows accesses to the shared variable of type `D`ouble.

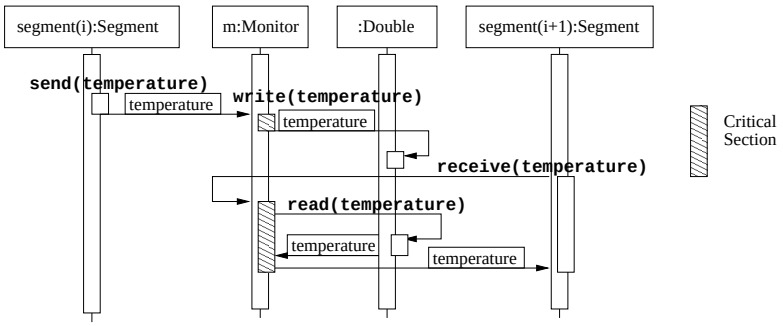

Figure 3. UML Sequence Diagram for the Monitor idiom.

  - As long as `segment(i)` remains inside the `monitor` `segment(i+1)` may attempt to invoke methods of the `monitor`. However, as `segment(i)` owns the `monitor`'s lock, `segment(i+1)` is not able to succeed. Thus, it has to wait until `segment(i)` leaves the `monitor`.
  - Only when `segment(i)` leaves the `monitor`, `segment(i+1)` is able to access it. Notice that even though the two software components proceed in parallel, only one software component is able to access the `monitor`, and so only this component is able to access the shared variables inside the critical section at once.

– *Synchronization Analysis.* This section describes the advantages and disadvantages of the Monitor idiom as a base for the synchronization code proposed [12].

  (a) **Advantages**

  - Two parallel `segment` software components are allowed to execute non-deterministically and at different relative speeds, each acting as independently of the others as possible.
  - Synchronization is carried out by atomic or indivisible operations over the `monitor`.
  - Each `segment` software component is able to execute the critical section within the `monitor`, accessing the shared variables that represent temperatures in a safe and secure manner. Any other software component attempting to enter the `monitor` is blocked, and should wait until the current `segment` software component finishes its access.
  - The shared variables maintain their integrity during the entire communication exchange.
  - The use of the `monitor` enforces the correct use of operations over the shared variables.

  (b) **Liabilities**

  - Mutual exclusion using monitors needs to be implemented at the compiler level. The compiler commonly associates a semaphore with each monitor. However, this imple-

mentation introduces potential delays when there the semaphore is committed to a `wait()` operation when a monitor procedure is called.

* Mutual exclusion is sometimes not sufficient for programming concurrent systems. Conditional synchronization is also needed (a resource may be busy when it is required, a buffer may be full when a write operation is pending and so on). Therefore, most monitor-based systems provide a new type of variable called a condition. These condition variables should be incorporated during programming: they are needed by the application and the monitor implementation, managing them as synchronization queues.
* A software component must not be allowed to block while holding a monitor lock. If a process has to wait for condition synchronization, the implementation must release the monitor for use by other software components and queue the software component on the condition variable.
* It is essential that a component's data is consistent before it leaves the monitor. It might be desirable to ensure that a component can only read (and not write) the monitor data before leaving.
* The implementation of monitors based on semaphores has a potential problem with the `Signal()` operation. Suppose a signaling component is active inside the monitor and another component is freed from a condition queue and is thus potentially active inside the monitor. By definition, only one software component can be active inside a monitor at any time. A solution is to ensure that a `Signal()` is immediately followed by exit from the monitor that is, the signaling process is forced to leave the monitor. If this method is not used, one of the software components may be delayed temporarily and resume execution in the monitor later.
* Monitors, as programming language synchronization mechanisms, must be implemented with great care and always with awareness of the constraints imposed by the mechanism itself.

5. Implementation

In this section, the communication components and their respective monitors are implemented as described in the Detailed Design step, using the Java programming language [11, 13]. So, the implementation is presented here for developing the channel as communication and synchronization components. Nevertheless, this design and implementation of the whole parallel software system goes beyond the actual purposes of the present paper.

5.1. Communication components – Channels

A class `Monitor` is used as the synchronization mechanism component of the Shared Variable Channel pattern, in order to implement the class `Channel` as follows [13]:

```
public final class Channel {
    private Monitor m0 = null;
    private Monitor m1 = null;
    public Channel(){
       m0 = new Monitor();
       m1 = new Monitor();
    }
    public void send0(Channel c, double temp){
       if(temp == null) throw new NullPointerException();
       m0.write(temp);
    }
    public void send1(Channel c, double temp){
       if(temp == null) throw new NullPointerException();
       m1.write(temp);
    }
    public double receive0(Channel c){
       return m0.read();
    }
    public double receive1(Channel c){
       return m1.read();
    }
}
```

Each channel component is composed of two monitors which allow the bi-directional flow of data through the channel. In order to keep straight the direction of each message flow, it is necessary to define two methods for sending and another two methods for receiving. Each method distinguishes on which monitor of the channel the message is written. The channel is capable of allowing a simultaneous bi-directional flow. In the present example, this is used to enforce the use of the Jacobi relaxation [11]. In fact, using *(a)* a channel communication structure with two-way flow of data, *(b)* making each one of them asynchronous, and later, *(c)* taking care on the communication exchanges between segment components, are all design previsions for avoiding any potential deadlock. In parallel programming, it is generally advised that during design, all previsions should be taken against the possibility of a deadlock [13].

Moreover, in case of modifying the present implementation for executing on a distributed memory parallel system, it would be necessary only to substitute the implementation of the class `Channel`, but using the Message Passing Channel pattern [10, 12] as a base for its definition.

5.2. Synchronization Mechanism – Monitors in Java

Based on the Monitor idiom and their implementation in the Java programming language, the basic synchronization mechanism that controls the access to the temperature array is presented as follows:

```
import java.util.Vector;
class Monitor {
    private int numMessages = 0;
    private final Vector temperatures = new Vector();
    public final synchronized void write(double temp){
```

```
        if(temp == null) throw new NullPointerException();
        numMessages++;
        temperatures.addElement(temp);
        if(numMessages <= 0) notify();
    }
    public final synchronized double read(){
        double temp = 0.0d;
        numMessages--;
        while(numMessages < 0){
            try{
                wait();
                break;
            }
            catch(InterruptedException e){
                if(numMessages >=0) break;
                else continue;
            }
        }
        temp = temperatures.firstElement();
        temperatures.removeElementAt(0);
        return temp;
    }
}
```

The class `Monitor` presents two synchronized methods, `write()` and `read()`, which enables the safe modification of the temperatures buffer and provides a mechanism for asynchronous communication between `segment` components. This class is used in the following implementation stage as the basic element of the channel components.

6. Summary

The Idioms for Synchronization Mechanisms are applied here along with a method for selecting them, in order to show how to select an idiom that copes with the requirements of the communication components present in the CSE solution to the One-dimensional Heat Equation problem. The main objective of this paper is to demonstrate, with a particular example, the detailed design and implementation that may be guided by a selected idiom. Moreover, the application of the Idioms for Synchronization Mechanisms and the method for selecting them is proposed to be used during the Detailed Design and Implementation for other similar problems that involve the calculation of differential equations for a one-dimensional problem, executing on a shared memory parallel platform.

7. Ackowledgements

This work is part of an ongoing research in the Departmento de Matemáticas. Facultad de Ciencias, UNAM, funded by project IN109010-2, PAPIIT-DGAPA-UNAM, 2010.

References

[1] P. Brinch-Hansen, *Structured Multiprogramming.* Communications of the ACM, Vol. 15, No. 17. July, 1972.

[2] P. Brinch-Hansen, *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering, Vol. 1, No. 2. June, 1975.

[3] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.

[4] E.W. Dijkstra *Co-operating Sequential Processes*, In Programming Languages (ed. Genuys), pp.43-112, Academic Press, 1968.

[5] M. Fowler, *UML Distilled.* Addison-Wesley Longman Inc., 1997.

[6] C.A.R. Hoare, *Towards a theory of parallel programming.* Operating System Techniques, Academic Press, 1972.

[7] C.A.R Hoare, *Monitors: An Operating System Structuring Concept.* Communications of the ACM, Vol. 17, No. 10. October, 1974.

[8] C.A.R. Hoare *Communicating Sequential Processes.* Communications of the ACM, Vol.21, No. 8, August 1978.

[9] J.L. Ortega-Arjona *The Communicating Sequential Elements Pattern. An Architectural Pattern for Domain Parallelism*, Proceedings of the 7th Conference on Pattern Languages of Programming (PLoP2000), Allerton Park, Illinois, USA, 2000.

[10] J.L. Ortega-Arjona *Design Patterns for Communication Components*, Proceedings of the 12th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2007), Kloster Irsee, Germany, 2007.

[11] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. Solving the One-dimensional Heat Equation*, Proceedings of the 14th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2009), Kloster Irsee, Germany, 2009.

[12] J.L. Ortega-Arjona *Patterns for Parallel Software Design.* John Wiley & Sons, 2010.

[13] J.L. Ortega-Arjona *Applying Design Patterns for Communication Components. Communicating CSE components for the One-dimensional Heat Equation*, Submitted to the 15th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2010), Kloster Irsee, Germany, 2010.