# Formal Verification for the Absence of Deadlock in the Manager Workers Pattern

Jorge Luis Ortega-Arjona and Francisco Hernández-Quiroz

Facultad de Ciencias
Universidad Nacional Autónoma de México
Ciudad Universitaria, D.F. 04510, MEXICO
`jloa,fhq@ciencias.unam.mx`

**Abstract.** The Architectural Patterns for Parallel Programming are descriptions of the fundamental organizational features of common top-level coordinations observed in parallel software systems. They represent a means to capture and express experience in the design and development process of parallel software. Nevertheless, by now, these software patterns have been described in informal terms, in which very little can be stated about the properties present in the final parallel software system.
The present paper presents an initial approach for studying and documenting logical properties of an architectural pattern for parallel programming. In particular, the objective here is to formally verify the property known as "absence of deadlock" for the Manager-Workers pattern, a widely used architectural pattern for parallel programming, by means of formal verification using CCS and $\mu$-calculus. The aim is to establish under what conditions this architectural pattern is deadlock-free, and whether this formal verification can be ported later to other Architectural Patterns for Parallel Programming.[1]

**Key words**: Formal Verification, Absence of Deadlock, Manager-Workers pattern.

## 1 Introduction

Software patterns describe in a very general and abstract way a general problem in software design, and link it with a particular structure of software components that solve the general problem [3, 4]. Among all the software patterns, and in the area of parallel programming, the Architectural Patterns for Parallel Programming have been proposed as the *fundamental organizational descriptions of the common top-level structure observed in a group of parallel software systems* [8, 9]. They can be viewed as templates, expressing and specifying some structural properties of their communication and synchronization subsystems, and the responsibilities and relationships between them. The selection of an architectural

[1] This work was made possible thanks to the support of a project grant from our university (PAPIIT IN109010).

Received 30/04/10
Final version 19/05/10
Accepted 14/05/10

pattern for parallel programming is considered to be a fundamental decision during the design of the overall coordination of a parallel software system [9].

Architectural Patterns for Parallel Programming are defined and classified according to the requirements of order of data and operations, and the nature of their processing components. Requirements of order dictate the way in which parallel computation has to be performed, and therefore, impact on the software design [8, 9].

Nevertheless, even though these architectural patterns have served as guidance to the software designer or engineer, they still remain as a documented informal description about how to partition and communicate a problem among several parallel software components. In these terms, it would be advantageous to have further information about the performance and logic properties of the resulting parallel software system.

The objective of the present paper is to provide a formal verification that an important logical property of concurrency, namely the "absence of deadlock", is present in the Manager-Workers pattern (MW pattern hereafter), as an instance of an architectural pattern for parallel programming. For this, the MW pattern will be expressed as a CCS process [7] and absence of deadlock will be represented by a modal-mu calculus formula [5] satisfied by the process. The aim is to establish the conditions under which the MW pattern is deadlock-free, and if such a formal verification technique can be ported later to other Architectural Patterns for Parallel Programming. For our purposes here, deadlock is defined as the situation in which no process can take any further action but, at the same time, at least a process has a pending task [13].

[12] applied a similar approach to verification of mutual exclusion in parallel algorithms.

## 2 The Manager-workers pattern

The MW pattern is a variant of the Master-Slave pattern [3] for parallel systems, considering an activity parallelism approach where the same operations are performed on ordered data. The variation is based on the fact that components of this pattern are proactive rather than reactive. Each processing component simultaneously performs the same operations, independent of the processing activity of other components. An important feature is to preserve the order of data [8, 9].

The MW pattern has multiple data sets processed at the same time. So, a MW structure is composed of a manager component and a group of identical worker components. The manager is responsible of preserving the order of data. On the other hand, each worker is capable of performing the same independent computation on different pieces of data. It repeatedly seeks a task to perform, performs it and repeats; when no tasks remain, the program is finished. The execution model is the same, independent of the number of workers (at least one). If tasks are distributed at run time, the structure is naturally load balanced: while a worker is busy with a heavy task, another may perform several shorter tasks. This distribution of tasks at runtime copes with the fact that data pieces may exhibit different size. To preserve data integrity, the manager program takes care of what part of the data has been operated on, and what remains to be computed by the workers [8, 9].

## 2.1 Structure

The Manager-Workers pattern is represented as a manager, preserving the order of data and controlling a group of processing elements or workers. Usually, only one manager and several identical worker components simultaneously exist and process during the execution time. In this architectural pattern, the same operation is simultaneously applied in effect to different pieces of data by worker components. Conceptually, workers have access to different pieces of data. Operations in each worker component are independent of operations in other components.

The structure of this pattern involves a central manager that distributes data among workers by request. Therefore, the solution is presented as a centralized network, the manager being the central common component. An Object Diagram, representing the network of elements that follows the Manager-Workers structure is shown in Figure 1.

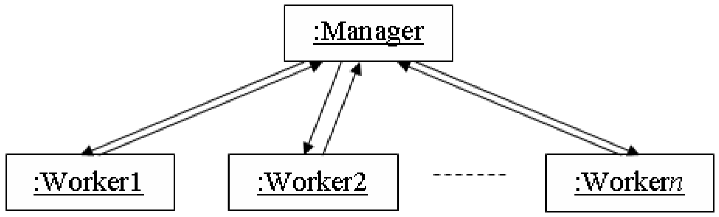

**Fig. 1.** Object Diagram of the Manager-Workers pattern.

## 2.2 Participants

– Manager. The responsibilities of a manager are to create a number of workers, to partition work among them, to start up their execution, and to compute the overall result from the sub-results from the workers.

– Worker. The responsibility of a worker is to seek for a task, to implement the computation in the form of a set of operations required, and to perform the computation.

## 2.3 Dynamics

A typical scenario to describe the run-time behavior of the Manager-Worker pattern is presented, where all participants are simultaneously active. Every worker performs the same operation on its available piece of data. As soon as it finishes processing, it returns a result to the manager, requiring more data. Communications are restricted between the manager and each worker. No communication between workers is allowed (Figure 2).

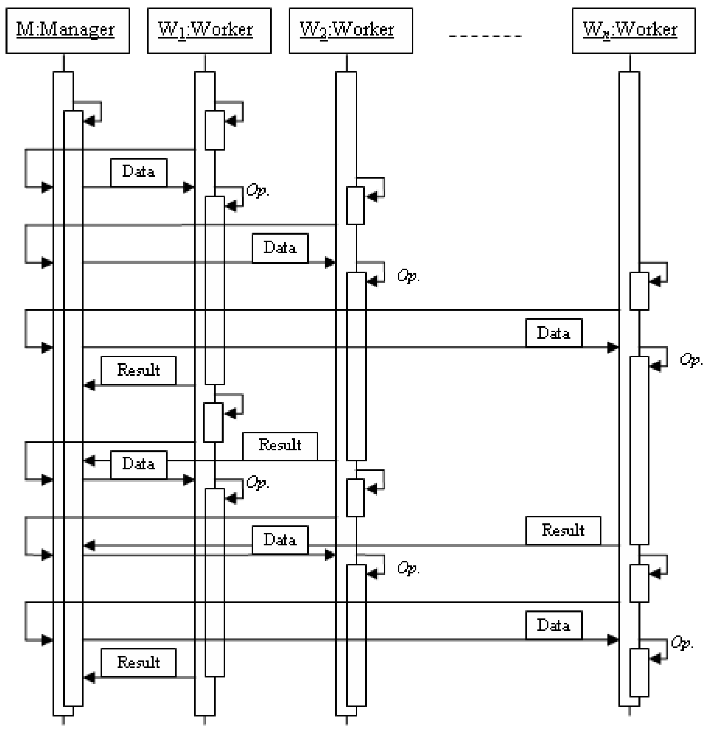

**Fig. 2.** Interaction Diagram of the Manager-Workers pattern.

In this scenario, the steps to perform a set of computations is as follows:

1. All participants are created, and wait until a computation is required to the manager. When data is available to the manager, this divides it, sending data pieces by request to each waiting worker.
2. Each worker receives the data and starts processing an operation Op. on it. This operation is independent of the operations on other workers. When the worker finishes processing, it returns a result to the manager, and then, requests for more data. If there is still data to be operated, the process repeats.
3. The manager is usually replying to requests of data from the workers or receiving their partial results. Once all data pieces have been processed, the manager assembles a total result from the partial results and the program finishes. The non-serviced requests of data from the workers are ignored.

## 3 CCS

Milner designed the Calculus of Communicating Systems [7] for modelling concurrency in systems that communicate in a message-passing synchronous style. Message are sent via two-end channels. Access to channels can be *open* or *restricted*. A message is consumed once communication has taken place and therefore is no longer available to anybody else.

Formally speaking, we have a countable set of channels $\alpha$, $\beta$, $\gamma$, $\alpha_0$, ... The basic actions are

- sending a message: $\alpha!m$, where $\alpha$ is a channel;
- receiving a message: $\alpha?m$;
- synchronous communication between process without any disclosure to outsiders: $\tau$.

Let us denote by the variables $\lambda$, $\lambda'$ any of the first two types of actions. If $\lambda$ is the action $\alpha?m$, then $\bar{\lambda}$ is the complementary action $\alpha!m$ and viceversa.

*Processes* are made out of basic actions and the simple process *nil* compounded by *prefixing* by basic actions, *non-deterministic choice*, *parallel composition*, *restriction* of channels and *relabelling* of channels. In BNF:

$$p ::= nil \mid \lambda\, .\, p \mid (p + p) \mid (p \parallel p) \mid p \backslash L \mid p[f],$$

where $L$ is a set of channels and $f$ is a one-to-one mapping of channels.

Finally, a process can be defined recursively by an equation

$$N \equiv_{\mathrm{def}} p$$

where $N$ is a name and $P$ is a process in the above grammar (possibly containing instances of the name $N$).

The semantics of CCS is usually expressed in terms of structural operational semantic rules [11] with labelled transitions:

Prefixed process

$$\lambda \,.\, p \xrightarrow{\lambda} p$$

Choice

$$\frac{p \xrightarrow{\lambda} q}{(p+r) \xrightarrow{\lambda} q} \qquad \frac{r \xrightarrow{\lambda} q}{(p+r) \xrightarrow{\lambda} q}$$

Parallel composition

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'} \qquad \frac{p_0 \xrightarrow{\lambda} p_0' \quad p_1 \xrightarrow{\bar{\lambda}} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Channel restriction

$$\frac{p \xrightarrow{\lambda} q}{p \backslash L \xrightarrow{\lambda} q \backslash L} \quad \lambda \notin L \cup \bar{L}$$

Channel relabelling

$$\frac{p \xrightarrow{\lambda} q}{p[f] \xrightarrow{f(\lambda)} q[f]}$$

Recursively defined processes

$$\frac{p \xrightarrow{\lambda} q}{P \xrightarrow{\lambda} q} \quad \text{where } P \equiv_{\text{def}} p.$$

### 3.1 Manager-Workers in CCS

Our translation of the Manager-Workers pattern to CCS focuses on the messages sent from the manager to assign a task, and the results sent back by the worker. The specific task performed for the worker can be left unspecified.

The components of the main process (represented by $M$) will be as follows: (a) a task being assigned (represented by $T$); (b) the worker which receives the assignment (represented by $W$); (c) the assignment itself (represented by $A$). Subscripts will be used to distinguish between different workers and tasks.

$$\begin{aligned}
W_i &\equiv_{\text{def}} (\alpha_i?m \,.\, P \,.\, \alpha!m \,.\, \mathit{nil}) \\
T_i &\equiv_{\text{def}} (\alpha_i!m \,.\, \alpha?m \,.\, A_i) \\
A_i &\equiv_{\text{def}} (T_i \parallel W_i)\backslash\{\alpha_i\} + \mathit{nil} \\
M &\equiv_{\text{def}} C \parallel (A_1 \parallel \cdots \parallel A_n)
\end{aligned}$$

A further explanation is needed:

- Worker $W_i$ is expecting through channel $\alpha_i$ an assignment. When this happens, it proceeds to perform the task, which is represented by the unspecified process $P$. The only condition we will assume later is that $P$ is deadlock free itself. For this assumption to be realistic, $P$ should not depend on any external synchronizing event. Then the result is sent back through the same channel.
- Task $T_i$ contains the complementary communication actions of $W_i$. Once a task is finished control is handed over to the assignment process.
- Assignment $A_i$ can choose either calling in a task *and* a worker to perform it, or consider the work done and become *nil*. Observe how channel $\alpha_i$ is restricted in order to guarantee integrity of the communication with $W_i$.
- $M$ is the manager creating tasks and combining results in process $C$, while in parallel assigns tasks to different workers. Please do note that $C$ is also left unspecified and again the only condition required is that $C$ is deadlock free.

A final but important remark: this translation has made explicit important facts about synchronization between actions from manager and workers which by no means were stated in the English description of this pattern and that will be critical to have a deadlock free system (as it will be proved later).

## 4 Model Checking

For expressing properties of processes we will be using a version of modal $\mu$-calculus extended with special constants [5]. $\mu$-calculus is a propositional multimodal logic with an additional least-fixed point operator for recursive formulas. The modal propositional part of the language is essentially Hennessy-Milner logic [6].

$\mu$-calculus has as atomic propositions the logical constants $V$ y $F$ (we will add some more atomic propositions later). The modalities in HML are labelled by CCS basic actions. In BNF:

$$D ::= V \mid F \mid \neg D \mid D \vee D \mid D \wedge D \mid \langle\lambda\rangle D \mid \langle\cdot\rangle D \mid \mu X \,.\, D$$

We can add the following abreviations:

$$[\lambda]D \equiv_{\text{def}} \neg\langle\lambda\rangle\neg D \qquad [\,\cdot\,]D \equiv_{\text{def}} \neg\langle\cdot\rangle\neg D$$

We define inductivevly the satisfaction relation $\models$ between CCS processes and HML formulas:

$$\begin{array}{lll}
p \models V & & \forall p \in \text{CCS} \\
p \not\models F & & \forall p \in \text{CCS} \\
p \models \langle\lambda\rangle D & \text{iff} & \exists p' \,.\, p \xrightarrow{\lambda} p' \wedge p' \models D \\
p \models \langle\cdot\rangle D & \text{iff} & \exists p', \lambda \,.\, p \xrightarrow{\lambda} p' \wedge p' \models D \\
p \models [\lambda] D & \text{iff} & \forall p' \,.\, p \xrightarrow{\lambda} p' \Rightarrow p' \models D \\
p \models [\,\cdot\,] D & \text{iff} & \forall p', \lambda \,.\, p \xrightarrow{\lambda} p' \Rightarrow p' \models D \\
p \models \mu X \,.\, D & \text{iff} & p \models D_{[X:=\mu X \,.\, D]}
\end{array}$$

For instance, the process $(\alpha?m \,.\, \mathit{nil}) \parallel (\beta!n \,.\, \mathit{nil})$ satisfies formula $\langle\alpha?m\rangle[\,\cdot\,]V$ because

$$(\alpha?m \,.\, \mathit{nil}) \parallel (\beta!n \,.\, \mathit{nil}) \xrightarrow{\alpha?m} \mathit{nil} \parallel (\beta!n \,.\, \mathit{nil})$$

and the latter process can perform only one action and therefore the only possible transition is

$$\mathit{nil} \parallel (\beta!n \,.\, \mathit{nil}) \xrightarrow{\beta!n} \mathit{nil} \parallel \mathit{nil}$$

and $\mathit{nil} \parallel \mathit{nil} \models V$. On the other hand, the same process does not satisfy $[\,\cdot\,]\langle\alpha?m\rangle V$, for if it performs firstly the transtion (and we are obliged to take into account every possibility by the operator $[\,\cdot\,]$)

$$(\alpha?m \,.\, \mathit{nil}) \parallel (\beta!n \,.\, \mathit{nil}) \xrightarrow{\alpha?m} \mathit{nil} \parallel (\beta!n \,.\, \mathit{nil})$$

we will have

$$\mathit{nil} \parallel (\beta!n \,.\, \mathit{nil}) \not\models \langle\alpha?m\rangle V.$$

## 4.1 Formal properties

The other side of model checking verification requires to express the desired/ undesired properties as formulas in a logical language. In this case, we need to express deadlock as a HML formula. Following [13], we introduce a new atomic formula satisfied by processes with no pending tasks.

$$\begin{array}{ll}
\mathit{nil} & \models \mathit{terminal} \\
\lambda \,.\, p & \not\models \mathit{terminal} \\
p + q & \models \mathit{terminal} \text{ if } p \models \mathit{terminal} \text{ and } q \models \mathit{terminal} \\
p + q & \not\models \mathit{terminal} \text{ otherwise} \\
p \parallel q & \models \mathit{terminal} \text{ if } p \models \mathit{terminal} \wedge q \models \mathit{terminal} \\
p \backslash L & \models \mathit{terminal} \text{ if } p \models \mathit{terminal} \\
p[f] & \models \mathit{terminal} \text{ if } p \models \mathit{terminal}
\end{array}$$

Our working deadlock definition implies that there is no possible action that can be performed and yet the process still has pending tasks. For the first part, a process capable of no action will satisfy trivially the formula $[\,\cdot\,]F$. But if the process has not terminated yet it will not satisfy *terminal*. Then deadlock can be defined by

$$\mathit{dead} \equiv_{\mathrm{def}} ([\cdot]F \wedge \neg \mathit{terminal}).$$

Nevertheless, this formula represents the fact of the process that already has reached deadlock. We want to say that a process can or cannot deadlock now or in the future (ie, after performing a certain number of actions). For this, we need the recursive formula

$$\mathit{e\text{-}dead} \equiv_{\mathrm{def}} \mu X\ .\ (\mathit{dead} \vee \langle\cdot\rangle X).$$

A process satisfying this formula may eventually deadlock. In the following we will check our translation of the MW pattern.

## 4.2 Deadlock absence for MW

Let us see now where our Manager-Workers representations stand regarding eventual deadlock. We need to answer

$$M \models \mathit{e\text{-}dead}?$$

According to the rules for recursive formulas, this is equivalent to

$$M \models \mathit{dead} \vee \langle\cdot\rangle\, \mathit{e\text{-}dead}?$$

Being a disjunction we need both disjuncts. Let us start with *dead*, that is

$$M \models [\cdot]F \wedge \neg \mathit{terminal}?$$

Assuming $C$ is deadlock free we can focus on the component $(A_1 \parallel \cdots \parallel A_n)$ and check whether

$$(A_1 \parallel \cdots \parallel A_n) \models [\cdot]F \wedge \neg \mathit{terminal}.$$

We have two cases: (a) *nil* is chosen in each instance of $A_i$ and we end up trivially with a collection of *nil*; (b) at least one of the assignments launches an instance of $(T_i \parallel W_i)$.

In case (a), $(\mathit{nil} \parallel \cdots \parallel \mathit{nil})$ is equivalent to *nil* and although $\mathit{nil} \models [\cdot]F$, by definition $\mathit{nil} \not\models \mathit{terminal}$ and then $\mathit{nil} \not\models \mathit{dead}$.

In case (b), by virtue of the rules of structural operational semantics

$$(T_i \parallel W_i) \xrightarrow{\tau} (\alpha?m\ .\ A_i) \parallel (P\ .\ \alpha!m\ .\ \mathit{nil})$$

which means that

$$\begin{array}{ll} (A_1 \parallel \cdots (T_i \parallel W_i) \parallel A_n) \not\models [\cdot]F & \text{and} \\ (A_1 \parallel \cdots (T_i \parallel W_i) \parallel A_n) \not\models \neg \mathit{terminal} & \end{array}$$

and therefore

$$M \not\models \mathit{dead}.$$

What about the other side of the disjunction, namely $\langle\cdot\rangle\, \textit{e-dead}$? Applying again the rule for recursive formulas, we are asking the question

$$M \models \langle\cdot\rangle\, \textit{dead} \vee \langle\cdot\rangle\langle\cdot\rangle\, \textit{e-dead}?$$

Let us consider again the first disjunct. As before, we are faced with the question

$$(A_1 \parallel \cdots \parallel A_n) \models \langle\cdot\rangle\, \textit{dead},$$

and again we can have the two cases (a) and (b). In case (a)

$$(\textit{nil} \parallel \cdots \parallel \textit{nil}) \not\models \langle\cdot\rangle\, \textit{dead},$$

this time by definition of $\models$ for the operator $\langle\cdot\rangle$ (as there is no possible action). Suppose now that a process $A_i$ chooses to launch an instance of $(T_i \parallel W_i)$. Following a similar argument as before, we have to answer now the question

$$(A_1 \parallel \cdots (T_i \parallel W_i) \parallel A_n) \models \langle\cdot\rangle\, \textit{dead}?$$

Again, we apply the rules of structural operational semantics

$$\frac{(T_i \parallel W_i) \xrightarrow{\tau} (\alpha?m \,.\, A_i) \parallel (P \,.\, \alpha!m \,.\, \textit{nil})}{(A_1 \parallel \cdots (T_i \parallel W_i) \parallel A_n) \xrightarrow{\tau} (A_1 \parallel \cdots ((\alpha?m \,.\, A_i) \parallel (P \,.\, \alpha!m \,.\, \textit{nil})) \parallel A_n)}$$

and the rules of $\models$ pose now the question

$$(A_1 \parallel \cdots ((\alpha?m \,.\, A_i) \parallel (P \,.\, \alpha!m \,.\, \textit{nil})) \parallel A_n) \models \textit{dead}?$$

Given that $P$ is deadlock-free and does not require any external communication event we now that

$$\begin{array}{ll}(\alpha?m \,.\, A_i) \parallel (P \,.\, \alpha!m) & \xrightarrow{\tau}{}^{*} (\alpha?m \,.\, A_i) \parallel (\alpha!m \,.\, \textit{nil}) \text{ and} \\ (\alpha?m \,.\, A_i) \parallel (\alpha!m \,.\, \textit{nil}) & \xrightarrow{\tau} A_i \parallel \textit{nil}\end{array}$$

ie, the process can perform at least one action and therefore, as before,

$$(A_1 \parallel \cdots ((\alpha?m \,.\, A_i) \parallel (P \,.\, \alpha!m)) \parallel A_n) \not\models \textit{dead}.$$

This again resolves the left-hand side of the second disjunction, and we have to look at $\langle\cdot\rangle\langle\cdot\rangle\, \textit{dead}$ which is tantamount to

$$M \models \langle\cdot\rangle\langle\cdot\rangle\, \textit{dead} \vee \langle\cdot\rangle\langle\cdot\rangle\langle\cdot\rangle\, \textit{e-dead}.$$

Following the previous line of reasoning we will be considering either the question

$$(A_1 \parallel \cdots (A_i \parallel \textit{nil}) \parallel A_n) \models \textit{dead}?$$

which we have already answer on the negative.

But again we will have to consider a new right-hand side disjunct. This can lead us to think this process will never end. However, it is not difficult to see by now that the new questions posed can be reduced to one of the previously considered, no matter how many times a task and its corresponding worker is launched. Therefore it is safe to state that

$$M \not\models \textit{e-dead}.$$

## 5 Conclusions and Future Work

We have shown a way of translating a natural language description of the MW pattern into a syntactic construction of a formal language. This translation helped to expose and solve some vagueness implicit in the original description and it also made explicit decisions of synchronization previously left to application phase of the pattern.

The new formulation of the MW pattern was proved to comply with the property of deadlock absence. For this we used (an instance of) the modal $\mu$-calculus and standard model checking techniques. So we can ascertain that a system based on this formulation of the MW pattern will be deadlock-free (provided the additional assumptions stated here are also met).

The MW pattern is a very general one and it can be specialized into more particular patterns according to specific decisions related to coordination order, discipline of communication or any other consideration deemed relevant. Future work should consider these variants in order to prove that they are also deadlock free.

Absence of deadlock can be defined differently, as in [1, 2]. Different definitions may require different formulas in $\mu$-calculus and either a proof of equivalence or inclusion between them or separate proofs of deadlock absence.

Finally, we hope we have shown the utility of this method and how it can be applied to other Architectural Patterns for Parallel Programming.

## References

1. Andrews, G.R., *Concurrent Programming*, The Benjamin/Cummings Publishing Company, 1991.
2. Andrews, G.R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
4. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Systems*, Addison-Wesley, 1994.
5. Kozen, D., "Results on the propositional mu-calculus", *Theoretical Computer Science* 27, pp. 333–354, 1983.
6. Larsen, K.G., "Proof systems for Hennessy-Milner logic with Recursion", *Lecture Notes In Computer Science*, 299, Proceedings of the 13th Colloquium on Trees in Algebra and Programming, pp. 215–230, 1988.
7. Milner, A.J.R.G., *Communication and Concurrency*, Prentice Hall, 1989.
8. Ortega-Arjona, J.L., *Architectural Patterns for Parallel Programming. Models for Performance Estimation*, VDM Verlag, 2009.
9. Ortega-Arjona, J.L., *Patterns for Parallel Software Design.* John Wiley & Sons, 2010.
10. Najm, E., Stefani, J.-B. *Formal Methods for Open Object-based Distributed Systems*, International Workshop 1996, Paris, Chapman & Hall, 1997.
11. Plotkin, G.D., *A structural approach to operational semantics*, DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

12. Walker, J., "Automated analysis of mutual exclusion algorithms using CCS", *Formal Aspects of Computing*, 1:1, pp. 273–292, 1989.
13. Winskel, G., "A note on model checking the modal mu-calculus", *Theoretical Computer Science* 83, pp. 761–772, 1991.