Proceedings of the 2nd International Conference on Computer and Information Science and Technology (CIST'16)
Ottawa, Canada – May 11 – 12, 2016
Paper No. 132

Verifying Consistency of Process Communications between Design and Implementation of Concurrent Systems

Ming Zhu[1], Peter Grogono[1], Olga Ormandjieva[1], Kunsheng Zhao[2]
[1]Department of Computer Science and Software Engineering, Concordia University
1455 De Maisonneuve Blvd. W., Montreal, Canada
zhu_ming@encs.concordia.ca; grogono@cse.concordia.ca; ormandj@cse.concordia.ca
[2]School of Information Study, McGill University
3661 Rue Peel, Montreal, Canada
kunsheng.zhao@mail.mcgill.ca

Abstract - Category theory is considered to be a suitable means for verifying consistency of process communications between design and implementation of concurrent systems. In this paper, certain features of a proposed categorical framework for the verification are studied by using a Client/Server example. In particular, Communicating Sequential Processes (CSP), Erasmus, abstraction, and category theory are used to verify the consistency of process communications between design and different implementations of the example.

Keywords**:** concurrent system; verification; category theory; CSP; Erasmus

1. Introduction

Verifying consistency of process communications between design and implementation of concurrent systems is considered as a difficulty [1], due to the different abstraction levels of design and implementation. Particularly, for those concurrent systems designed in a process algebra and implemented in a process-oriented language, it is challenging to manage the consistency between design phase and implementation phase. Research [2], [3] and [4] propose a categorical framework to bridge the gap of inconsistency between design and implementation of concurrent systems. As a continuation, the aim of this paper is to demonstrate the verification of consistency of process communications by using the framework.

The rest of this paper is organized as follows: Section 2 provides background knowledge and related work on Communicating Sequential Processes (CSP), the process-oriented programming language Erasmus, and the categorical framework. Section 3 demonstrates the application of the categorical framework to different scenarios of a Client/Server example. Section 4 concludes this paper and proposes directions for future research.

2. Background and Related Work

In this section, the background and related work on our research are introduced.

2.1. Communicating Sequential Processes (CSP)

CSP was proposed by Hoare as a process algebra towards specification-oriented [5], and then has been refined by Roscoe [6]. Most of process-oriented programming languages are considered to be derived from CSP [7]. CSP specifies and models processes in a concurrent system that communicate with their external environment. The construction of a process depends on a set of all events that occur on the process. This set of all events is called an *alphabet*. A process in CSP can be represented by *failures* [5]. Given a process P, a failure of P is of the form (s, X). It means that P can engage in the *trace* of events s, and then refuse to do anything more, although its environment is prepared to engage in any of the events of X [5]. Also, there are several operations defined on process, which includes *prefixing*, *recursion*, *deterministic choice*, and *nondeterministic choice* [6]. Processes can be assembled together as a system, where they can *communicate* with each other. If one process needs to communicate to another process, a channel is required between them to receive inputs and send outputs. The notion of *parallel* along with the symbol || is introduced to describe communications between processes.

CSP is widely used and studied. For example, traces and failures are used to analyze the liveness and correctness of processes [8]; semantics for revivals, stuckness and the hierarchy of CSP model are discussed [9]; and it is used for understanding particular issues in concurrent and real-time systems [10].

2.2. Erasmus

Several signs suggest that the next paradigm may be process-oriented programming [11]. Erasmus is a process-oriented programming language based on the idea of CSP but with some differences [11]. An Erasmus program consists of *cells*, *processes*, *ports*, *protocols* and *channels*. A cell, containing a collection of one or more processes or cells, provides the structuring mechanism for an Erasmus program. A process is a self-contained entity which performs computations, and communicates with other processes through its ports. A port, which is of a type of protocol, serves as an interface of a process for sending and receiving messages. A protocol specifies the type and the orderings of messages that can be sent and received by the ports of the type of this protocol. A channel, which is of a type of protocol, links two ports and so enables two processes to communicate.

Some research is proposed to study communications in Erasmus. It includes constructing a fair protocol that allows arbitrary and nondeterministic communication between processes [12], and describing an alternative construct that allows a process to nondeterministically choose between possible communications on several channels [13].

2.3. The Categorical Framework

It is suggested that category theory can be helpful towards discovering and verifying connections in different areas, while preserving structures in those areas [14]. In software engineering, category theory is proposed as an approach to formalizing refinement from design to implementation [15]. Specifically, category theory is used to construct the categorical framework for verifying consistency of process communications between design and implementation in research [3] and [4]. The categorical framework consists of 6 steps: (1). designing: design concurrent systems in CSP, and analyze failures of processes together with communications, (2). implementing: implement concurrent systems in Erasmus with the design refinement, (3). analyzing abstraction: abstract processes and communications out of the implementation, and analyze failures of abstracted processes as well as communications, (4). categorizing design: construct categorical models for the design with preserving structures of communications, (5). categorizing abstraction of implementation: construct categorical models for the abstraction of implementation with preserving structures of communications, and (6). verifying: construct functors to verify the categorical models of the design and the abstraction. With this categorical framework, it is able to check whether the designed communication is captured by the implementation or not [4]. In this paper, we use the Client/Server example to demonstrate the leverage of the framework.

3. Demonstrating the Categorical Framework

To demonstrate the categorical framework, a Client/Server example is developed. In the example, there are a server and a client. The server can provide two types of service, service *A* and service *B*. The client can request service *A* and service *B*. In the beginning, the client lets the server know the type of service it requests. Then, the client sends the information related to the requested service to the server. At last, the client receives the corresponding results from the server. The client can repeatedly request service from server.

According to the software development process, we develop the design in CSP based on the requirements specification of the example, then we refine the design into the implementation in Erasmus. In this paper, we develop three different scenarios in the implementation stage. In the first scenario, the server offers three types of service that are service *A*, service *B* and service *C*. In the second scenario, the server offers only one type of service that is service *A*. In the third scenario, the server offers exactly same services as the design. With the application of the categorical framework to the example, the consistency of Client/Server communications between the design and the implementation can be verified automatically. Fig.1 illustrates the process of applying the framework to the design and a scenario of the implementation.

3.1. Designing the Example

In the design stage, the server and the client are modeled as processes *Server* and *Client* respectively. As described in the specification of the example, processes *Server* and *Client* communicate the following messages sequentially: (1). the client sends a type of request, *requestA* or *requestB*, to the server, (2). the client sends a message with information of the

service, *infoA* or *infoB*, to the server, and (3). the server processes the service request, and sends the corresponding result, *resultA* or *resultB*, to the client. According to CSP, communications of *Client*/*Server* are modeled and analyzed as follows:

$$Alphabet(Server||Client) = \{requestA, infoA, resultA, requestB, infoB, resultB\}$$
$$Failures(Server||Client) = \{\{(\langle\ \rangle, X)|X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB\}\},$$
$$\{(\langle requestA\rangle, X)|X \subseteq \{requestA, requestB, resultB, resultA, infoB\}\},$$
$$\{(\langle requestB\rangle, X)|X \subseteq \{requestA, requestB, resultB, resultA, infoA\}\},$$
$$\{(\langle requestA, infoA\rangle, X)|X \subseteq \{requestA, requestB, resultB, infoA, infoB\}\},$$
$$\{(\langle requestB, infoB\rangle, X)|X \subseteq \{requestA, requestB, resultA, infoA, infoB\}\},$$
$$\{(\langle requestA, infoA, resultA\rangle, X)|X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB\}\},$$
$$\{(\langle requestB, infoB, resultB\rangle, X)|X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB\}\},$$
$$\dots\ \dots\}$$

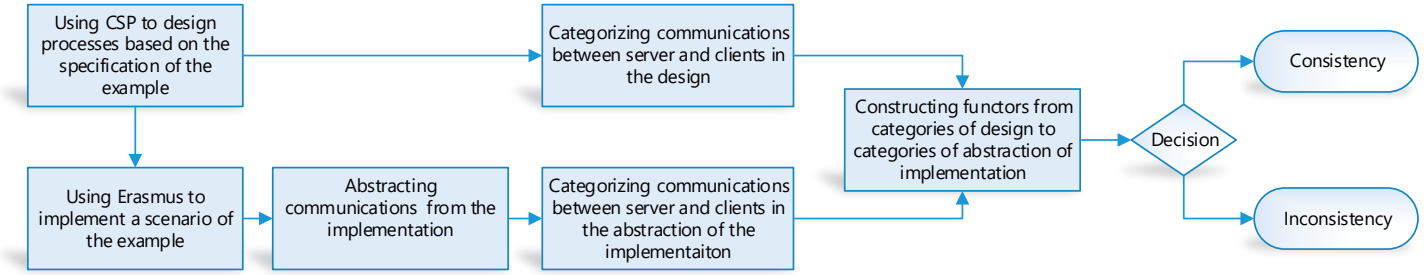

Fig. 1: Process of applying the categorical framework to the example.

3.2. Implementing Scenario 1

In the implementation stage, for this scenario, process *Server* is implemented to provide 3 types of service, and process *Client* is implemented to request 3 types of service. The Erasmus code for the implementation is as follows:

```
Match = protocol {requestA|requestB|requestC|infoA|infoB|infoC|^resultA|^resultB|^resultC}
Server = process p: +Match{
  loop select{  ||p.requestA; p.infoA; p.resultA;
                ||p.requestB; p.infoB; p.resultB;
                ||p.requestC; p.infoC; p.resultC;  }}
Client = process e: -Match{
  loop case{    ||e.requestA; e.infoA; e.resultA;
                ||e.requestB; e.infoB; e.resultB;
                ||e.requestC; e.infoC; e.resultC;  }}
Main = cell{ m: Channel Match;  Server(m);  Client(m); }
```

3.3. Analyzing the Abstraction of the Implementation of Scenario 1

According to research [3] and [4], the implementation of scenario 1 is abstracted as follows:

```
Server = loop{select{|p.requestA; p.infoA; p.resultA;
                     |p.requestB; p.infoB; p.resultB;
                     |p.requestC; p.infoC; p.resultC}}
Client = loop{case{  |e.requestA; e.infoA; e.resultA;
                     |e.requestB; e.infoB; e.resultB;
                     |e.requestC; e.infoC; e.resultC}}
```

According to CSP and research [3] and [4], communications between *Server* and *Client* in the abstraction of the implementation are modeled and analyzed as follows:

$$Alphabet(Server||Client) = \{requestA, infoA, resultA, requestB, infoB, resultB, requestC, infoC, resultC\}$$
$$Failures(Server||Client) =$$
$$\{\{(\langle\ \rangle, X)|X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB, requestC, infoC, resultC\}\},$$

$\{(\langle requestA\rangle, X) | X \subseteq \{requestA, requestB, resultB, resultA, infoB, requestC, infoC, resultC\}\},$
$\{(\langle requestB\rangle, X) | X \subseteq \{requestA, requestB, resultB, resultA, infoA, requestC, infoC, resultC\}\},$
$\{(\langle requestC\rangle, X) | X \subseteq \{requestA, requestB, resultB, resultA, infoA, requestC, infoB, resultC\}\},$
$\{(\langle requestA, infoA\rangle, X) | X \subseteq \{requestA, requestB, resultB, infoA, infoB, requestC, infoC, resultC\}\},$
$\{(\langle requestB, infoB\rangle, X) | X \subseteq \{requestA, requestB, resultA, infoA, infoB, requestC, infoC, resultC\}\},$
$\{(\langle requestC, infoC\rangle, X) | X \subseteq \{requestA, requestB, resultA, infoA, infoB, requestC, infoC, resultB\}\},$
$\{(\langle requestA, infoA, resultA\rangle, X) | X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB, requestC, infoC, resultC\}\},$
$\{(\langle requestB, infoB, resultB\rangle, X) | X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB, requestC, infoC, resultC\}\},$
$\{(\langle requestC, infoC, resultC\rangle, X) | X \subseteq \{requestB, resultB, requestA, resultA, infoA, infoB, requestC, infoC, resultC\}\},$
$\ldots\ldots\}$

3.4. Categorizing Communications and Verifying the Consistency for Scenario 1

The implementation of scenario 1 not only provides *serviceA* and *serviceB*, but also offers *serviceC*. According to Definition 4.1 and Proposition 1 in research [4], we can try to construct a functor from the category of the design to the category of the abstraction of the implementation, in order to check whether the process communications in the implementation are consistent with the process communications in the design.

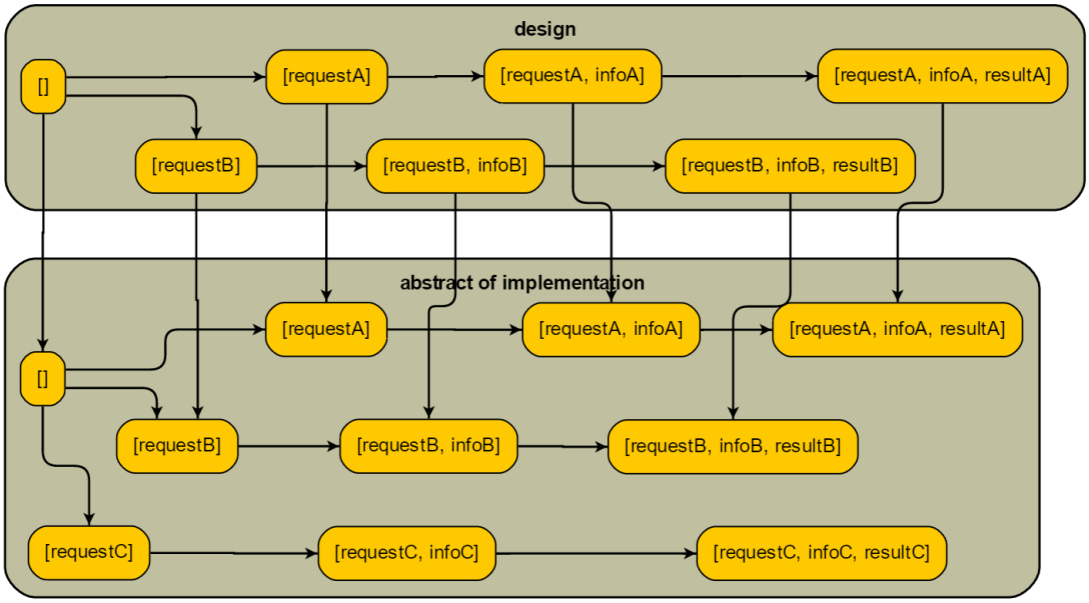

Fig. 2: The functor form the category of the design to the category of the abstraction of the implementation (scenario 1).

The successful construction of the functor indicates the consistency of process communications between the design and the implementation (See Fig.2). Furthermore, it shows that the implementation offers more than the design.

3.5. Implementing Scenario 2

In this scenario, process *Server* is implemented to provide only one type of service, and process *Client* is implemented to request the service from *Server*. The Erasmus code for the implementation is as follows:

```
Match = protocol {requestA |infoA |^resultA}
Server = process p: +Match{
     loop{ p.requestA; p.infoA; p.resultA; }}
Client = process e: -Match{
     loop{ e.requestA; e.infoA; e.resultA; }}
Main = cell{m: Channel Match;  Server(m);  Client(m);}
```

3.6. Analyzing the Abstraction of the Implementation of Scenario 2

According to research [3] and [4], the implementation of scenario 2 is abstracted as follows:

```
Server = loop{p.requestA; p.infoA; p.resultA}
Client = loop{e.requestA; e.infoA; e.resultA}
```

According to CSP and research [3] and [4], communications between *Server* and *Client* in the abstraction of the implementation are modeled and analyzed as follows:

$$Alphabet(Server||Client) = \{requestA, infoA, resultA\}$$
$$Failures(Server||Client) = \{\{(\langle\ \rangle, X)|X \subseteq \{resultA, infoA\}\},$$
$$\{(\langle requestA\rangle, X)|X \subseteq \{requestA, resultA\}\},$$
$$\{(\langle requestA, infoA\rangle, X)|X \subseteq \{requestA, infoA\}\},$$
$$\{(\langle requestA, infoA, resultA\rangle, X)|X \subseteq \{resultA, infoA\}\},$$
$$\ldots\ldots\}$$

3.7. Categorizing Communications and Verifying the Consistency for Scenario 2

The implementation of scenario 2 just provides *serviceA*. There is no *serviceB* in the implementation. According to Definition 4.1 and Proposition 1 in research [4], we can try to construct a functor from the category of the design to the category of the abstraction of the implementation, in order to check whether the process communications in the implementation are consistent with the process communications in the design.

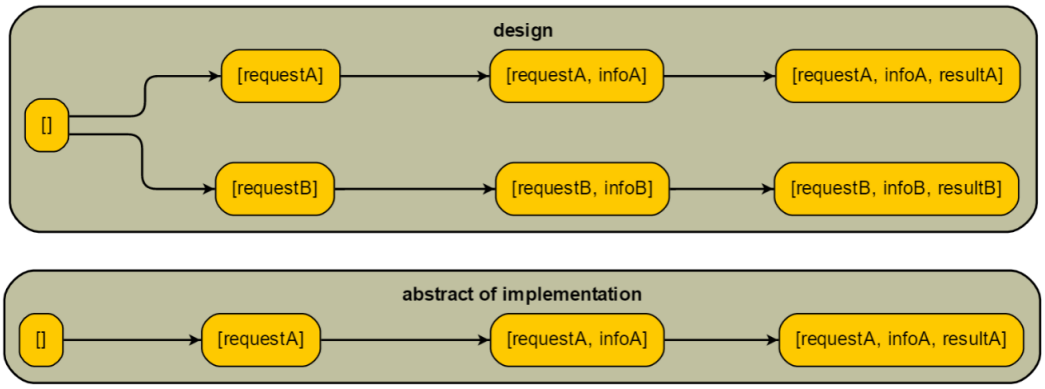

Fig. 3: the category of the abstraction of the implementation and the category of the design (scenario 2).

Clearly, the functor cannot be constructed from the category of the design to the category of the abstraction of the implementation (See Fig.3). It indicates that not all the communications of the design are captured in the implementation. For this scenario, communications related to *serviceB* is not implemented. Namely, the process communications in the design are inconsistent with the process communications in the implementation.

3.8. Implementing Scenario 3

In this scenario, process *Server* provides *serviceA* and *serviceB*, and process *Client* requests both services from *Server*. The Erasmus code for the implementation is as follows:

```
Match = protocol {requestA |requestB |infoA |infoB |^resultA |^resultB}
Server = process p: +Match{
     loop select{    ||p.requestA; p.infoA; p.resultA;
                     ||p.requestB; p.infoB; p.resultB;}}
Client = process e: -Match{
     loop case{      ||e.requestA; e.infoA; e.resultA;
                     ||e.requestB; e.infoB; e.resultB;}}
Main = cell{m: Channel Match;  Server(m);  Client(m);}
```

3.9. Analyzing the Abstraction of the Implementation of Scenario 3

According to research [3] and [4], the implementation of scenario 3 is abstracted as follows:

```
Server = loop{select{|p.requestA; p.infoA; p.resultA
                     |p.requestB; p.infoB; p.resultB }}
Client = loop{case{  |e.requestA; e.infoA; e.resultA
                     |e.requestB; e.infoB; e.resultB }}
```

According to CSP and research [3] and [4], communications between *Server* and *Client* in the abstraction of the implementation are modeled and analyzed as follows:

$$Alphabet(Server||Client) = \{requestA, infoA, resultA, requestB, infoB, resultB\}$$
$$\begin{aligned} Failures(Server||Client) = \{&\{(\langle\ \rangle, X)|X \subseteq \{requestA, resultA, infoA, requestB, infoB, resultB\}\},\\ &\{(\langle requestA\rangle, X)|X \subseteq \{requestA, resultA, requestB, infoB, resultB\}\},\\ &\{(\langle requestB\rangle, X)|X \subseteq \{requestA, infoA, resultA, requestB, resultB\}\},\\ &\{(\langle requestA, infoA\rangle, X)|X \subseteq \{requestA, infoA, requestB, infoB, resultB\}\},\\ &\{(\langle requestB, infoB\rangle, X)|X \subseteq \{requestA, infoA, resultA, requestB, infoB\}\},\\ &\{(\langle requestA, infoA, resultA\rangle, X)|X \subseteq \{requestA, resultA, infoA, requestB, infoB, resultB\}\},\\ &\{(\langle requestB, infoB, resultB\rangle, X)|X \subseteq \{requestA, resultA, infoA, requestB, infoB, resultB\}\},\\ &\dots\dots\} \end{aligned}$$

3.10. Categorizing Communications and Verifying the Consistency for Scenario 3

The implementation of scenario 3 contains both *serviceA* and *serviceB*. These two services are specified in the example, and are designed as well. According to Definition 4.1 and Proposition 1 in research [4], we can try to construct a functor from the category of the design to the category of the abstraction of the implementation, in order to check whether the process communications in the implementation are consistent with the process communications in the design.

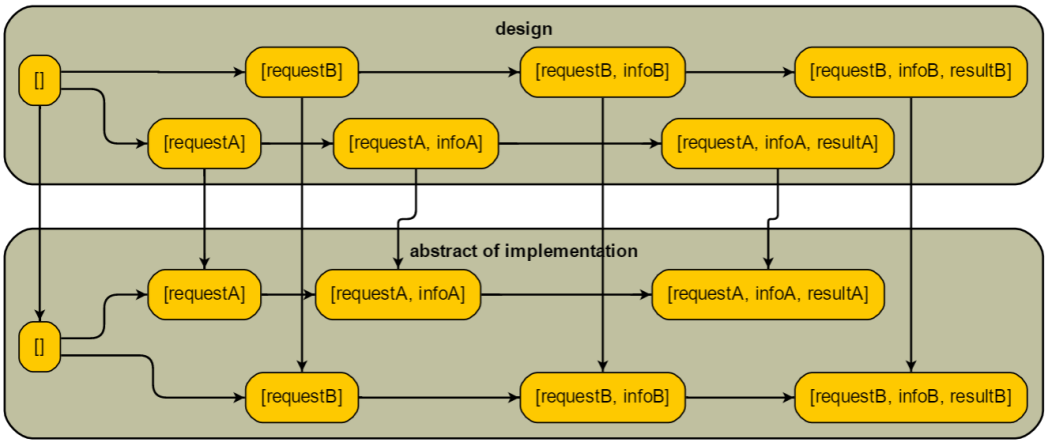

Fig. 4: The functor form the category of the design to the category of the abstraction of the implementation (scenario 3).

It is obvious that the category of design is equivalent to the category of the abstraction of the implementation (See Fig.4). The successful construction of the functor indicates the process communications in the design are consistent with the process communications in the implementation.

4. Conclusion and Future Work

As the continuation of our former research [2], [3] and [4], this paper uses the categorical framework to verify the consistency of process communications. In this framework, the design of the system is modeled and analyzed by CSP; the implementation of the system is developed in Erasmus, and then is abstracted; the categories of the design and of the abstraction of the implementation are created; and by constructing functors, the consistency of process communications between the design and the implementation is verified. By developing three different scenarios of the Client/Server example, the application of the framework to those scenarios shows correct verification results as expected.

In Future, more examples scale up to realistic systems will be analyzed by using this categorical framework. Besides, the algorithms inside the framework for automatically analyzing CSP design, abstracting implementation, and constructing categories and functors will be discussed.

References

[1] J. R. Kiniry and F. Fairmichael, "Ensuring consistency between designs, documentation, formal specifications, and implementations," in *Proceedings of 12th International Symposium on Component-Based Software Engineering*, 2009, pp. 242-261.

[2] M. Zhu, P. Grogono, O. Ormandjieva, and P. Kamthan, "Using category theory and data flow analysis for modeling and verifying properties of communications in the process-oriented language erasmus," in *Proceedings of the Seventh C* Conference on Computer Science and Software Engineering*, Montreal, 2014, pp. 24:1-24:4, 2014.

[3] M. Zhu, P. Grogono, and O. Ormandjieva, "Using category theory to verify implementation against design in concurrent systems," in *The 6th International Conference on Ambient Systems, Networks and Technologies*, London, 2015, pp. 530-537.

[4] M. Zhu, P. Grogono, O. Ormandjieva, and H. Kuang. (2016, March 23). Using Failures and Category Theory to Verify Process Communications between Design and Implementation of Concurrent Systems [Online]. Available: http://users.encs.concordia.ca/~zhu_ming/ant2016.pdf.

[5] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[6] A. W. Roscoe, *Understanding Concurrent Systems*. London, United Kingdom: Springer, 2010.

[7] A. T. Sampson, "Process-oriented patterns for concurrent software engineering," in Ph.D. dissertation, University of Kent, Kent, United Kingdom, 2008.

[8] P. Welch, "Life of occam-pi," in *Communicating Process Architectures 2013*, Edinburgh, 2013, pp. 293-318

[9] A. W. Roscoe, "Revivals, stuckness and the hierarchy of CSP models," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 3, pp. 163-190, 2009.

[10] S. Schneider, *Concurrent and Real Time Systems: The CSP Approach*. New York: John Wiley & Sons Inc., 1999.

[11] P. Grogono and B. Shearing, "Concurrent software engineering: Preparing for paradigm shift," in *Proceedings of the First C* Conference on Computer Science and Software Engineering*, 2008, pp. 99-108.

[12] P. Grogono and N. Jafroodi, "A fair protocol for non-deterministic message passing," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 2010, pp. 53-58.

[13] N. Jafroodi and P. Grogono, "Implementing generalized alternative construct for erasmus language," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, 2013, pp. 101-110.

[14] S. Awodey, *Category Theory*. New York: The Clarendon Press, 2006.

[15] C. A. R. Hoare, "Notes on an approach to category theory for computer scientists," *Constructive Methods in Computing Science*, vol. 55, pp. 245-305, 1989.