Building DEVS Models from the Functional Design of Software Architecture Components to Estimate Quality

María J. Blas[1], Horacio P. Leone[1], Silvio M. Gonnet[1]

[1]Instituto de Desarrollo y Diseño INGAR – Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) – Universidad Tecnológica Nacional (UTN)
Avellaneda 3657 – Santa Fe – CP 3000 – Argentina

{mariajuliablas, hleone, sgonnet}@santafe-conicet.gov.ar

Abstract. *Software architectures can be used as a vehicle to improve the study of quality properties in the early stages of development. This paper proposes an automatic mapping between the design of architectural components and the specification of DEVS atomic models with aims to evaluate all-purpose quality metrics. Then, we use the functional description of architectural components (that address functional requirements) to estimate the architecture adjustment to non-functional requirements. The guidelines for structuring the simulation models are defined starting from the design of high-level components. To illustrate the proposal, web-based architecture is used as proof of concepts.*

1. Introduction

Software engineering encompasses processes, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality [Pressman and Maxim 2019]. From a broad point of view, quality refers to the degree to which software products meet their stated requirements. Specifically, *software quality* can be defined as *"the capability of a software product to satisfy stated and implied needs under specified conditions"* [ISO/IEC 2011]. Then, quality is the basic parameter of software engineering efforts whose primary goal is the delivery of maximum stakeholder value while balancing cost and schedule.

Quality is everyone's business [Li, Chen and Cheung 2000]. In software development, the quality encompasses requirements, specifications, design, and implementation of the system. Each artifact produced during development has its individual quality properties. These properties provide a feasible context that will allow achieving quality in the final product. Take *software architecture* as an example. The *architecture* of a computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass, Clements and Kazman 2012]. Regarding the design itself, system engineers usually inspect the architecture to determine whether it is acceptable. Typically, this determination is made by a human engineer inspecting a set of architectural representations and using heuristics to judge whether they will result in a viable system that, when built, will meet the system requirements [Rodano and Giammarco 2013]. There is no such thing as an inherently good or bad architecture: architectures are either more or less fit for some stated purpose [Bass, Clements and Kazman 2012]. Hence, one of the vexing challenges of software architecture is the problem of satisfying the functional specifications of the system to be created while at the same time meeting its non-functional needs [Harrison and Avgeriou 2007].

The *software architecture* is a pivotal vehicle to address and guarantee non-functional software qualities such as security, maintainability, extensibility, and portability [Heijstek, Kühne and Chaudron 2011]. From this perspective, the *software architecture* can be used as a vehicle to improve the study of quality properties in the early stages of development. Instead of applying heuristics for architectural evaluation, innovative approaches combine new types of techniques (as additional evaluation methods) with the traditional ones (such as SAAM and ATAM). This is the case of Modeling and Simulation (M&S). A deeper discussion regarding the use of M&S for software architectures in comparison with other traditional approaches is presented in [Blas, Leone and Gonnet 2020].

Software architectures are design models that refer to discrete-event systems. Under this conceptualization, the user requests are seen as the events to which architectural components react. Hence, discrete-event simulation models can be used to perform their evaluation quantitatively [Bogado, Gonnet and Leone 2014; Blas, Gonnet and Leone 2016; Reussner et al. 2016]. Most approaches use the Discrete Event System Specification (DEVS) formalism [Zeigler, Muzy and Kofman 2018] to define the simulation model.

Building a simulation model that provides a quality estimation using the software architecture as a sketch is not easy. Prior to defining the model design, quality measures should be defined (i.e. the *simulation goal*) with aims to ensure the measurement of non-functional requirements. Also, the architectural representation should be taken into account with aims to ensure the correctness of the *simulation model structure*. Combining both fields (software quality and architecture representation) in a single simulation model is a complex task [Blas 2019].

With aims to provide a partial solution to this problem, in this paper we propose an automatic mapping between the design of architectural components and the specification of DEVS atomic models in order to evaluate all-purpose quality metrics. The architectural components studied in this paper are located at low-level design (i.e. functional components). To illustrate the approach, we use a web-based architecture as proof of concepts. The main contribution of this paper is the strategy proposed to obtaining a runnable DEVS atomic model from the functional definition of an architectural component.

The remainder of this paper is organized as follows. Section 2 introduces the M&S approach used as a guideline to study quality using the software architecture design. Section 3 presents the DEVS-centered modeling strategy that allows building the simulation model specification for measuring a set of generic quality properties. Finally, Section 4 is devoted to conclusions and future work.

2. Software Architecture Evaluation using M&S

Quality estimation of software architecture using M&S involves the understanding of *i)* the *software product domain* that describes the *architecture* to be evaluated, and *ii)* the *M&S formalism* that will be used for describing the *discrete-event specification* of the *architecture*. To accomplish *i)*, the evaluation requires defining: *i)* the set of *quality attributes* to be measured during the simulation, and *ii)* the way in which *architecture components* should be structured to build the simulation model. On the other hand, to achieve *ii)*, the definitions required in *i)* must be set.

The goal of the final simulation model is to perform the behavior of the software product following its architectural components as a prototype with aims to measure a set of pre-defined quality properties. At the core of the simulation model, the components and relationships detailed in the architecture provide the functional behavior. Depending on the type of component, different strategies can be used to get this behavior. Then, the following subsections detail how the statements described in *i)* were defined with aims to structure the functional behavior of the simulation model in a generic DEVS specification. Such specification is detailed in Section 3.

2.1. Definition of Quality Properties

The quality model proposed in ISO/IEC 25010 [ISO/IEC 2011] classifies the software product quality using three hierarchical levels: *i)* characteristic level to represent external quality views, *ii)* sub-characteristic level to define properties that can be evaluated when the software is used in a system, and *iii)* attribute level to depict entities that can be verified or measured over the software product.

Due to attributes change among distinct types of products, the standard does not define entities to be measured. However, a set of all-purpose quality measures can be set up using the most common software attributes. Table 1 summarizes these measures including the quality properties attached to each case.

Table 1. Quality properties to be measured during the architecture simulation.

Quality*		***Software***		
Characteristic	*Sub-characteristic*	*Attribute*	*Quality Measure*	
			Description (Abrev)	*Unit*
Performance efficiency	Time behavior	Invocation time	Processing time for user requests (ET).	time
			Total time for processing a request (TSIT).	time
Reliability	Maturity	Replies accuracy	Number of processed requests (TR).	request
			Number of requests with incorrect responses (IR).	request
	Availability	Software robustness	Inactive time (FT).	time
			Operative time (TT).	time
	Fault tolerance	Software stability	Number of faults that are not failures (FNF).	fault
			Number of faults (TF).	fault

* Quality properties defined as *characteristic* and *sub-characteristic* in [ISO/IEC 2010].

The *attributes* detailed in Table 1 refer only to *internal quality properties* (i.e. the factors that affect the software itself and its developers). For example, the quality characteristic named *reliability* is defined in [ISO/IEC 2011] as *"the degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time"*. This characteristic includes the sub-characteristic named *maturity*. This quality sub-characteristic is defined as "*the degree to which a system, product or component meets needs for reliability under normal operation*" [ISO/IEC 2011]. Regarding these quality properties, a generic attribute to be measure in any software product is *replies accuracy*. This attribute is defined as "*the accuracy of the software product in responding to a specific user request*". For this attribute, Table 1 defines two metrics: the *number of requests processed* (*TR*) and the *number of requests with incorrect responses* (*IR*).

As a result of the quality properties definition, each quality measure becomes a simulation goal. That is, the architectural simulation model should lead to the calculation of the set of metrics detailed in Table 1.

2.2. Representation of Software Architectures

A software architecture design representation is a description of the highest-level concept of a system in its environment [Kruchten 2003]. The architectural principles related to the topology of the architecture should be obvious in any design. A style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [Garlan and Shaw 1993]. The use of patterns offers a reusable and proven way to partition a system with known consequences to quality attributes [Harrison and Avgeriou 2007].

Given that software architectures should satisfy the functional specifications of the system to be created, most architectural styles define different types of elements to be used for functional design. Commonly, two architecture levels are defined. Architects employ *low-level components* to define *high-level components*. Then, *high-level components* refer to domain components specifically designed to fulfill some software functionality. Instead, *low-level components* refer to basic actions, functions, or procedures that combined allow getting complex behaviors. Links are frequently allowed on both levels. In the case of *low-level components*, links describe an execution flow. For *high-level components*, links describe interactions.

Take *web-based architectures* as an example. Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content [Pressman and Maxim 2019]. The architectural patterns for *web-based software architectures* proposed in [Fehling et al. 2014] employ three types of architectural components: *i) application components* as elements used to define functional requirements, *ii) management components* as elements used to watch the performance of *application components*, and *iii) functional components* as basic functionalities used to build the complex behaviors of *application components*. A full analysis of these types of architectural components is presented in [Blas, Leone and Gonnet 2019].

Figure 1 presents an architecture composed of three high-level components (*Load Balancer*, *Elastic Load Balancer*, and *Presentation and Business Logic*) and three low-level components (*User Interface*, *Processing,* and *Data Access*). In this example, two types of *application components* are used: *i) generic components* that refer to components frequently used as standard templates in *web-based software* (such as *Load Balancer*), and *ii) domain-specific components* that refer to components specifically used to define software functionalities (i.e. *Presentation and Business Logic*). Then, the behavior of *generic application components* is well-know. Meanwhile, the behavior of *domain-specific components* varies from one architecture to another according to the functional components employed in the design.

Since functional requirements describe the required behavior of the system in terms of required activities [Pfleeger and Altee 2006], *high-level components* are the ones that should be studied during the simulation. That is, quality measures should be obtained over *high-level components*. However, at the basic level, *low-level components* are the ones that perform functionalities. For example, the behavior of *domain-specific components* (Figure 1) is detailed as a sequence of *functional components*. Then, the simulation models for *high-level components* (i.e. *domain-specific components* in Figure 1) should be designed following the structure of *low-level components* (i.e. *functional components* in Figure 1) with aims to get the quality measures.

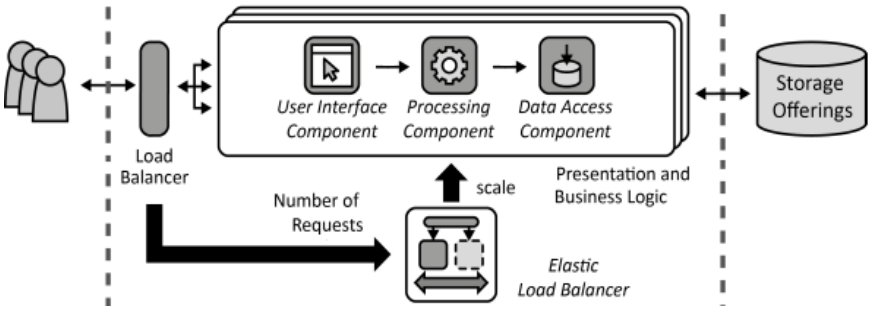

Figure 1. Example two-tier web architecture (adapted from [Fehling et al. 2014]).

Section 3.4 presents a DEVS-based solution for building the simulation model of the *high-level component* named *Presentation and Business Logic* using *low-level components* as phases. These phases are designed considering the quality measures.

3. DEVS Atomic Model for High-Level Components

3.1. DEVS Formalism

DEVS is a modular and hierarchical formalism based on systems theory that provides a general methodology for the construction of reusable models at two distinct levels [Wainer and Mosterman 2010]. At the lower level, an *atomic DEVS* describes the autonomous behavior of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input events and how it generates output events. On the other hand, at a higher level, a *coupled DEVS* describes a system as a network of DEVS components.

Therefore, an *atomic DEVS* defines the system *behavior* while a *coupled DEVS* defines the system *structure*. In this context, our approach uses an *atomic DEVS* to define the behavior of *high-level architecture components*. Then, the final model can be obtained structuring architectural interactions among the simulation models built for *high-level components*.

3.2. Simulation Model Definition

3.2.1. Input and Output Events

Following the architecture design, a *high-level component* receives user requests to be processed. Considering that any software component is executed over infrastructure, the behavior of *high-level components* is influenced by their state during hardware execution. To include this influence as part of the model, we consider *running* and *not-running* as possible execution states. Hence, the simulation model inputs are defined as two distinct events named *user_request* and *execution_state*.

As Table 1 shows, two types of quality measures should be calculated during the simulation: *measures related to user requests* (*ET*, *TSIT*, *TR*, and *IR*) and *measures related to the software itself* (*FT*, *TT*, *FNF*, and *TF*). For the first set, measures are directly calculated in the *user_request*. That is, a *user_request* is modeled to include the quality measures related to its processing. Table 2 presents the properties included in the *user_request* event in order to capture these metrics. After a *user_request* is processed in a *high-level component*, the *user_request* should be transferred to the next component defined in the architecture. Then, *user_request* is also an output event.

Table 2. User request information.

Name	*Description*
execution_time[1]	Processing time.
total_time[2]	Total time used to solve the request (processing and waiting).
incorrect[3]	Boolean value that is true if the request is processed in a fault function (false in other case).

[1] Quality metric *ET* of the actual request.
[2] Quality metric *TSIT* of the actual request.
[3] To measure the quality metric *IR*.

On the other hand, the simulation model includes an explicit output event named *component_state* to measure software processing states. This output captures the state information from three perspectives: hardware, activity and processing. According to the quality measures to be calculated during the simulation, a *high-level component* can be: *i) running* or *not-running* from the hardware perspective, *ii) active* or *in failure* from the activity perspective, and *iii) ok* or *in fault* from the processing perspective. Not all combinations are possible. For example, if a component is *not-running*, it cannot be in activity or processing. Table 3 resumes available combinations for *component_state*.

Table 3. Component state information.

Hardware State	*Activity State*	*Processing State*
not-running	N/A	N/A
running	failure	N/A
running	active	ok
running	active	fault

N/A = not applicable.

3.2.2. State Definition

An atomic DEVS is based on a sequence of deterministic states. The state definition should include all the information required to describe the behavior of the model.

For high-level architectural components, this information is related to its processing state (*phase*), the processing time attached to the actual processing state (*sigma*), the user request been processed by the component (*request*), and the possibility of having a failure (or fault) in a low-level component (*function_state*). Hence, the state of the model is structured as *{ phase, sigma, request, function_state }*.

3.2.3. Low-Level Components as DEVS Phases

Initially, any *high-level component* is *waiting* for some *user_request*. When a *user_request* arrives, the component should perform its behavioral description (saving the *user_request* as the value of *request* in the next state). If new requests arrive during the processing, the component just ignores them.

The behavioral description of a *high-level component* is given by the execution flow depicted using the set of *low-level components* that compose them. Hence, a *low-level component* can be seen as a basic function that can act correctly or not during a certain period (i.e. *processing_time*). The correctness in the function behavior is given by the parameter *fault_probability*. Then, for each *low-level component* included in the *high-level component*, two possible phases are considered using the *fault_probability*: *processing* and *processing with faults*. Before executing some function, the variable *function_state* is used to define the next *phase*. The value of this variable is calculated using the *fault_probability* of the next function to be executed. Once the execution of a

function ends (correctly or not), the control is given to the next function in the sequence according to the *function_state*. The evolution among phases is detailed following the sequence of *low-level components*. When the sequence final function is executed, the *request* is sent as output. After that, the component returns to *waiting*.

A function can also fail. When a *low-level component* presents a fault, this fault can become a failure according to a *failure_probability*. If a function fails, the *high-level component* cannot continue working. That is, once a failure is detected in a *high-level component*, the entire component fails. Then, a new phase is added to the model: *failure*. This phase is used to evolve the simulation model when any *low-level component* fails. In our approach, failures cannot be fixed. Hence, once the component achieves the *failure* phase, the simulation model stays in this phase until the hardware information indicates that the component is *not-running*.

In any case, when hardware information notifies that the component is *not-running*, the simulation model changes to the *inactive* phase. Once the component is *inactive*, the model stays in this phase forever.

Table 4 summarizes the transitions following the prior description. The guidelines for updating the *quality_measures* of the *request* are the following: *i) execution_time* set as *execution_time* + *processing_time* and *total_time* set as *total_time* + *processing_time* if the phase is *processing* or *processing with faults*, ii) *incorrect* set as *true* if the next phase is *processing with faults*.

3.3. Building an Example: Defining a Web-based Application Component

With aims to provide proof of concepts, the *high-level component* named *Presentation and Business Logic* (Figure 1) is used as an example. In this case, the *high-level component* is composed of three *functional components*: *User Interface Component* (UIC), *Processing Component* (PC), and *Data Access Component* (DAC). Per each *functional component*, three parameters are included: *processing_time*, *fault_probability*, and *failure_probability*. Then, for example, the parameters related to UIC are named $processing\_time_{UIC}$, $fault\_probability_{UIC}$, and $failure\_probability_{UIC}$.

For space reasons, the formal specification of the DEVS atomic model cannot be included. However, Figure 2 shows a simplified statechart diagram of the model.

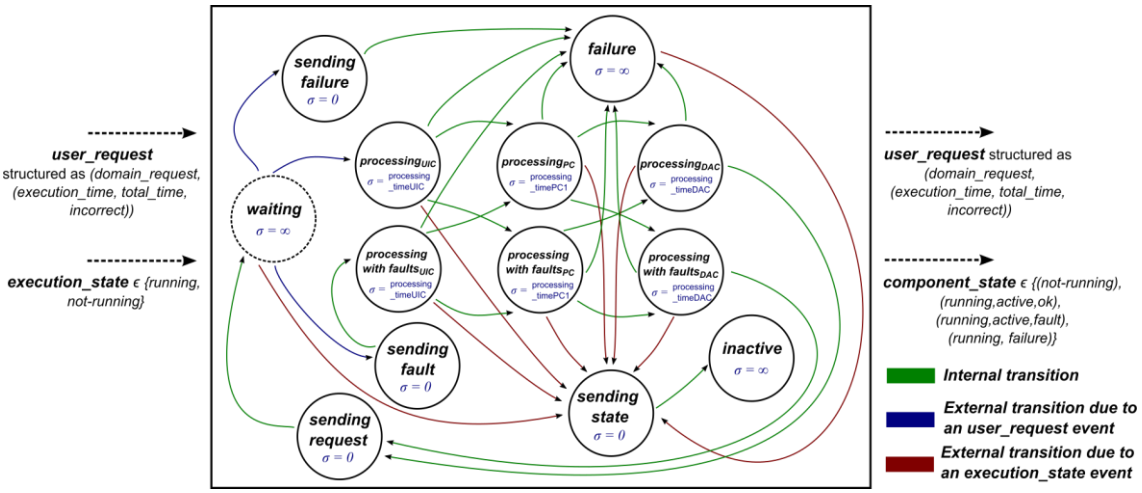

Figure 2. Representation of the "Presentation and Business Logic" model.

Table 4. Transitions for a high-level component with N low-level components.

Phase	*Final Phase Evolution*	*Description*
waiting	waiting → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending the output *component_state = (not-running)*.
	waiting → processing$_1$**	The model goes directly from *waiting* to *processing$_1$*.
	waiting → processing with faults$_1$**	The model goes from *waiting* to *sending fault* (transient state). Then, the model changes to *processing with faults$_1$* after sending the output *component_state = (running, active, fault)*.
	waiting → failure**	The model goes from *waiting* to *sending failure* (transient state). Then, the model changes to *failure* after sending the output *component_state = (running, failure)*.
processing$_i$***	processing$_i$ → processing$_{i+1}$	The model goes from *processing$_i$* to *processing$_{i+1}$* after sending the output *component_state = (running, active, ok)*. In the new state, the *quality_measures* of the *request* are updated.
	processing$_i$ → processing with faults$_{i+1}$	The model goes from *processing$_i$* to *processing with faults$_{i+1}$* after sending the output *component_state = (running, active, fault)*. In the new state, the *quality_measures* of the *request* are updated.
	processing$_i$ → failure	The model goes from *processing$_i$* to *failure* after sending the output *component_state = (running, failure)*.
	processing$_i$ → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending the output *component_state = (not-running)*.
processing$_N$	processing$_N$ → waiting	The model goes from *processing$_N$* to *sending request* (transient state) after sending the output *component_state = (running, active, ok)*. In the new state, the *quality_measures* of the *request* are updated. Then, the model changes to *waiting* after sending the output *user_request = (request)*.
	processing$_N$ → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending the output *component_state = (not-running)*.
processing with faults$_i$***	processing with faults$_i$ → processing$_{i+1}$	The model goes from *processing with faults$_i$* to *processing$_{i+1}$* after sending the output *component_state = (running, active, ok)*. In the new state, the *quality_measures* of the *request* are updated.
	processing with faults$_i$ → processing with faults$_{i+1}$	The model goes from *processing with faults$_i$* to *processing with faults$_{i+1}$* after sending the output *component_state = (running, active, fault)*. In the new state, the *quality_measures* of the *request* are updated.
	processing with faults$_i$ → failure	The model goes from *processing$_i$* to *failure* after sending the output *component_state = (running, failure)*.
	processing with faults$_i$ → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending *component_state = (not-running)*.
processing with faults$_N$	processing with faults$_N$ → waiting	The model goes from *processing with faults$_N$* to *sending request* (transient state) after sending the output *component_state = (running, active, ok).* In the new state, the *quality_measures* of the *request* are updated. Then, the model changes to *waiting* after sending *user_request = (request)*.
	processing with faults$_N$ → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending *component_state = (not-running)*.
failure	failure → inactive*	The model goes from *waiting* to *sending state* (transient state). Then, the model changes to *inactive* after sending *component_state = (not-running)*.
inactive	-	

* External transition due an input event *execution_state = (not-running)*.

** External transition due an input event *user_request* defined as *(domain_request, quality_measures)* with *quality_measures = (execution_time, total_time, incorrect).* The *domain_request* field should be defined according to the software product in development.

*** $1 \leq i \leq N\text{-}1$ con $N \neq 1$.

4. Conclusions and Future Work

In this paper, we present a DEVS-based approach for building simulation models for functional components defined in software architectures as high-level components. A set of all-purpose quality measures is used as a simulation goal with aims to provide a feasible solution for architectural evaluation in the early stages of development. The DEVS atomic model definition detailed in this paper can be used as a foundation for other types of measures related to software products (such as quality in use). The prerequisites of software architectures studied in this paper are the chance to distinguish low-level components as actions of high-level components.

Actually, the approach presented in this paper is used as a support mechanism for building the *essential models* required for quality estimation in web-based applications using the architecture design. The Routed DEVS (RDEVS) *essential models* are basically DEVS atomic models used as embedded components in a new type of discrete-event model named RDEVS *routing model* [Blas, Gonnet and Leone 2017]. The use of Routed DEVS as a formalism for building architectural simulation models was presented in [Blas, Leone and Gonnet 2020].

Given that the approach is centered on building DEVS atomic models from the design of functional components, the final goal is performing the mapping as an internal functionality of a software tool. In such a case, the modeling effort required to use the approach in real-world projects should be minimum. However, a deeper evaluation of such a modeling effort is part of the future work. Moreover, the approach may need some adjustment if other different domains are studied. For example, for software components that implement concurrency with threads, new phases may be required.

Future work is devoted to expanding the number of quality metrics measured during the simulation with aims to support new quality properties at the basic architectural level. Higher levels of architectural design could be modeled hierarchically by adding new top-level models to the existing ones. In DEVS, this can be done by building *coupled models*. In RDEVS, instead, *routing* and *network models* should be defined. In this last case, the advantage is that the *essential models* are used to define the behavior of *routing models*. Hence, when the architectural design changes, only the *routing information* attached to the specification of *routing models* need to be modified. The behavior of the components (*essential model* definition) remains the same.

References

Bass, L., Clements, P. and Kazman, R. (2012). Software Architecture in Practice, Addison Wesley Publishing Company, 3rd edition.

Blas, M. (2019). Un modelo de simulación para el análisis de arquitecturas de software de aplicaciones web y en la nube. *Doctoral Thesis*. Universidad Tecnológica Nacional (Argentina).

Blas, M., Gonnet, S. and Leone, H. (2016). Building Simulation Models to Evaluate Web Application Architectures. In *Proceedings of the 2016 Latin American Symposium of Software Engineering (CLEI)*, pages 647-657.

Blas, M., Gonnet, S. and Leone, H. (2017). Routing Structure over Discrete Event System Specification: A DEVS Adaptation to Develop Smart Routing in Simulation Models, In *Proceedings of the 2017 Winter Simulation Conference*, pages 774-785.

Blas, M., Leone, H. and Gonnet, S. (2019). Modelado y Verificación de Patrones de Diseño de Arquitectura de Software para Entornos de Computación en la Nube, In *Revista Ibérica de Sistemas e Tecnologias de Informação*, vol. 35, pages 1-17.

Blas, M., Leone, H. and Gonnet, S. (2020). Modeling and Simulation Framework for Quality Estimation of Web Applications through Architecture Evaluation, In *SN Applied Sciences*, vol. 2, pages 374-395.

Bogado, V., Gonnet S. and Leone H. (2014). Modeling and Simulation of Software Architecture in Discrete Event System Specification for Quality Evaluation. In *Simulation*, vol. 90(3), pages 290-319.

Fehling, C., Leymann, F., Retter, R., Schupeck, W. and Arbitter, P. (2014). Cloud computing patterns: fundamentals to design, build, and manage cloud applications, Springer Science & Business Media.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1-39.

Harrison, N. and Avgeriou, P. (2007). Pattern-driven architectural partitioning: Balancing functional and non-functional requirements. In *2007 Second International Conference on Digital Telecommunications*, pages 21-21.

Heijstek, W., Kühne, T. and Chaudron, M. (2011). Experimental analysis of textual and graphical representations for software architecture design. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 167-176.

ISO/IEC (2011). ISO/IEC 25010:2011 Systems and Software Engineering–Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models.

Kruchten, P. (2003). The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc.

Li, E. Y., Chen, H. G. and Cheung, W. (2000). Total quality management in software development process. In *Journal of Quality Assurance Institute*, vol. 14, pages 4-6.

Pfleeger, S. L. and Altee, J. M. (2006). Software Engineering: Theory and Practice, Pearson Prentice Hall, 3rd edition.

Pressman, R. and Maxim, B. (2019). Software Engineering: A Practitioner's Approach, McGraw Hill, 9th edition.

Reussner, R., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H. and Krogmann, K. (2016). Modeling and simulating software architectures: The Palladio approach. MIT Press.

Rodano, M. and Giammarco, K. (2013). A formal method for evaluation of a modeled system architecture. In *Procedia Computer Science*, vol. 20, pages 210-215.

Wainer, G. A. and Mosterman, P. J. (2010). Discrete-Event Modeling and Simulation: Theory and Applications, CRC press.

Zeigler, B., Muzy, A. and Kofman, E. (2018). Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations, Academic Press, 3rd edition.