

# *Building Simulation Models to Evaluate Web Application Architectures*

*María Julia Blas, Silvio Gonnet, Horacio Leone*

Consejo Nacional de Investigaciones Científicas y Técnicas  
Instituto de Desarrollo y Diseño INGAR, CONICET-UTN  
Santa Fe, Argentina  
{ mariajuliablas, sgonnet, hleone }@santafe-conicet.gov.ar

**Abstract**— Software quality has become a critical issue in software engineering because affects systems development costs, delivery schedules, and user satisfaction. In specific software products, an early quality evaluation can be done by using simulation techniques over the architectural design. However, not all architectural designs can be transformed into simulation models. This paper presents a set of simulation models based on standard architectural components that helps to compose web architectures simulation models in order to analyze its behavior. It also includes the definition and modeling of different types of mechanisms that help to simulate consumer's behavior. These mechanisms allow generating the inputs required to the quality evaluation (that is, software requirements) and could be used in other situations. All simulation models proposed were formalized using Discrete Event System Specification (DEVS). In this context, this work gives the foundations needed to make a future evaluation of specific quality attributes for web applications.

**Keywords**— *Cloud computing; software-as-a-service; discrete event simulation; user behavior; workload modeling; DEVS.*

## I. INTRODUCTION

Cloud computing (CC) is a distributed computing paradigm that focuses on providing a wide range of users with distributed access to scalable, virtualized hardware and/or software infrastructure over the internet [1]. Most authors propose CC architectures based on the layer architectural pattern [2–4]. An overview of these designs shows that all the architectures can be simplified into two layers: *service layer* and *infrastructure layer*. According to this abstraction, the traditional role of service provider is divided into two: the *infrastructure providers* who manage cloud platforms and lease resources according to a usage-based pricing model, and *service providers*, who rent resources from one or many infrastructure providers to serve the end users [5]. The two layers abstraction also allows denoting two types of capabilities over CC environments (one per layer): *Software-as-a-Service* (SaaS) and *Infrastructure-as-a-Service* (IaaS). SaaS provides a service delivery model which allows customers to use the provider's applications running on a cloud infrastructure [6]. IaaS refers to on-demand provisioning of infrastructural resources over the internet [5].

In this context, CC has emerged as an effective reuse paradigm, where software functionality, hardware computing power, and other computing resources are delivered in the form of

services so that they become available widely to consumers [7]. Given the complexity of the paradigm, the quality requirements evaluation is not easy. A lot of research work has been done in this area. Some of the most important topics involve quality models development, study and evaluation of specific quality attributes and deployment of quality maintenance strategies [2, 6, 8–13]. Authors recently have beginning to use simulation techniques at infrastructure level in order to analyze quality properties [14, 15]. The main objective in this type of research is to estimate quality before make the deployment of the cloud infrastructure. However, the same approach can be applied in similar contexts. In fact, researchers have used simulation strategies over traditional software products in many cases [16–18].

Software architectures can be considered the earliest design of any software product. These designs can be used as a vehicle to predict and estimate the final behavior of the product [16–22]. Then, an integrated approach to analyze software quality properties can be built by combining simulation techniques with architectural designs. If this approach is applied to CC environments, designers could evaluate different service architectures over the same infrastructure before to move to the next step of the development process. In fact, they also could analyze the impact of architectural changes over the final product quality and make recommendations based on this evaluation.

However, the simulation model construction should not be a problem to be solved by the architect. To this purpose, this work presents a mapping that allows build simulation models from web architectural designs in order to make a quality evaluation in CC environments. With this aim, a metamodel to instantiate valid web architectures has already been developed in a previous work [23]. Using the existing metamodel, this paper proposes a systematic way to transform each architectural component in a simulation model. However, the simulation models definition is not the only activity required for the evaluation of architectural designs. The architects also need mechanisms that let them make an adequate representation of the consumer's behavior in order to generate the inputs to the architecture simulation model. Therefore, along with the architectural simulation models, a set of user simulation models based on different types of workloads is specified. All these simulation models have been specified using the discrete event simulation formalism DEVS [24].

The remainder of this paper is organized as follows. Section II presents the problem statement introducing the CC architectures and how it can be modeled. Section III presents the DEVS fundamentals. Section IV details the set of DEVS models developed to represent the different types of architectural components identified in web architectures. Section V describes the set of DEVS models designed to represent the user behaviors. Section VI shows a case study that transforms basic web architecture into a DEVS simulation model. Finally, Section VII is devoted to the conclusions of the work.

## II. PROBLEM STATEMENT AND MOTIVATION

### A. Cloud Computing Layered Architecture

Layered architectures are a well-known architectural pattern [25]. In this architectural style, the software system is modeled by a functional hierarchical decomposition. Dependencies between layers are controlled so each layer only can access to its immediate lower-level layer.

Usually, CC architectures are defined using the layered architectural pattern because this pattern allows defining distributed application with loose coupling (Fig. 1). By nature, cloud environments are large distributed environments that consist of many information technology (IT) resources and, therefore, cloud applications have to be decomposed into separate application components that can be distributed among the resources in this environment [4]. However, since the number of IT resources on which a cloud application relies constantly changes, the dependencies between application components should be minimized (that is, loose coupling) in order to simplify the provisioning and decommissioning tasks and to reduce the impact of failing application components.

The abstraction modeled in Fig. 1 shows five layers: *hardware layer*, *software layer*, *software infrastructure layer*, *software environment layer* and *application layer*. The *hardware layer* is responsible to make an efficient assignment of the IT resources. The *software layer* contains the specific software features that manage the underlying IT resources. The *software infrastructure layer* manages the network resources to the upper layers, providing a standardized support to deliver IT as a service. The *software environment layer* provides a platform to develop web applications. Finally, the *application layer* allows consumers accessing to the applications installed on the data-centers of a cloud provider.

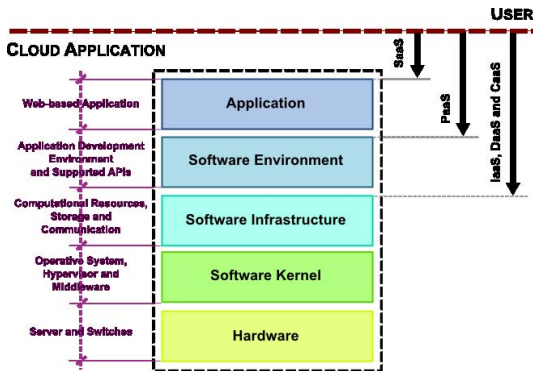


Fig. 1. Layered architecture for CC environments (adapted from [2–5]).

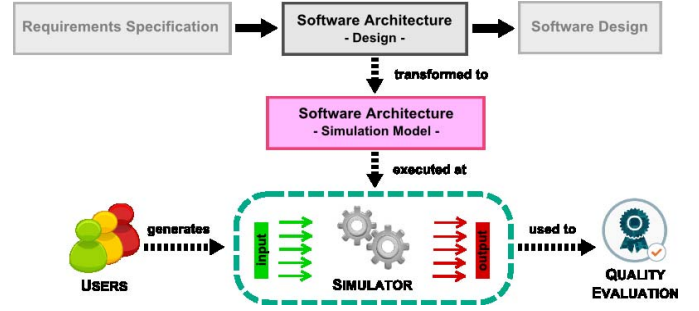


Fig. 2. Simulation approach over software architecture models.

CC employs a service-driven business model [5]. Conceptually, every layer of the architecture described can be implemented as a service to the layer above. In CC, the available services are: *infrastructure as a service* (IaaS), *data storage as a service*, (DaaS), *communication as a service* (CaaS), *platform as a service* (PaaS) and *software as a service* (SaaS). When a consumer use a web application is accessing to a SaaS. In this context, SaaS refers to providing on demand applications over the Internet.

### B. Quality Evaluation using Software Architecture Models

The delimitation of what defines an adequate level of quality in a software system is a highly context dependent question [26]. Everyone involved in the software engineering process is responsible for quality [27]. Achieving quality attributes must be considered throughout design, implementation, and deployment [25]. However, maintain the traceability of this attributes in the development process is very difficult. In this context, Bogado [16] and Rech [17] have used simulation techniques over software architecture models in order to estimate quality properties. In this work, an abstraction of these approaches is proposed (Fig. 2).

The main objective of our approach is to transform an architectural design into an executable simulation model. Since the architecture simulation model is used as a representation of the real software, its inputs must be obtained from the study of different users behaviors. Given a set of inputs, the model execution produces a set of outputs that could be used to obtain quality measures. These measures could be useful to predict the final quality of the software product modeled by the architecture.

However, the application of this approach on CC environments is not easy because the architectural designs are not standardized. The CC architectural patterns are not completely established in the architects' community yet. Besides, a unique representation technique has not been developed and the understanding of CC architectures is still very confused. Frequently the architectural representations mix infrastructure components with application components. These representations are very difficult to understand and lead to the readers to misunderstanding the architectural design. In order to solve the lack of representation structure at software level on CC environments, in [23] authors specify a metamodel to define web application architectures. This model take a set of design patterns proposed by Fehling et al. [4] and proposes a set of conceptual elements that resume the components of any SaaS architecture (that is , it

only models the components related to the *application layer*). The identified components are divided in three categories: *management components*, *application components* and *functional components*. The management components are used to manage automatically the application behavior. The *application components* are divided in two types: *defined application components* and *undefined application components*. The *defined application components* involve a set of standard elements which can be used by the architect to build its design. In contrast, by the *undefined application components* only a conceptual definition was made. This component type refers to domain specific elements of the web application and, therefore, should be modeled by the architect using a composition of the *functional components*. The way in which the application must be implemented is shown in this composition. Finally, the functional components are a set of elements in which each element refers to a specific functional responsibility. These standard components are the ones that should be used to model the behavior of specific undefined application components. Table I summarizes the set of elements included in each category identified as part of the metamodel.

The metamodel proposed can be used as a reference to build web application architectures in a standardized way. So that, each architectural component can be transformed in a specific simulation model. Given that the relationships and compositions are clearly identified in the metamodel, the links between simulation models can be established in the same way that in the original design. This paper proposes a set of discrete simulation models formalized using DEVS that refer to each type of architectural component identified as part of the metamodel. Each simulation model is developed using the behavioral description of each architectural component including the actions related with fault states. These simulation models are detailed in Section IV.

However, in order to complete the architectural simulation approach is necessary to have some mechanism that provides the model input. The metamodel concepts are restricted to architectural components and, therefore, nothing says about the consumers' behavior (that is, user request patterns). According to Fehling [4], the *workload* is the utilization of IT resources on which an application is hosted. That is, workload is a consequence of users accessing the application or jobs that need to be handled automatically and, therefore, takes different forms according to the type of IT resource on which it is measured. Recently, some research works have proposed solutions to this type of problems over the infrastructure resources [14, 28]. In this context, the workload can be seen simply as the resource petitions or execution requests of specific tasks repeated according a temporal pattern. Then, the workload result can be seen as the different types of temporal patterns of user requests to the web application. This approach can be used into the architectural simulation given that different types of users can be identified by different types of temporal patterns, e.g. static and periodic patterns. Discrete simulation models were designed and implemented in order to generate user requests according to different types of temporal patterns. This approach gives a way to specify user workloads in terms of temporal user requests. All the simulation models developed for user behaviors are detailed in Section V.

TABLE I. ARCHITECTURAL COMPONENTS IDENTIFIED IN THE METAMODEL PROPOSED IN [23] CLASSIFIED BY CATEGORIES.

Type		Name	Description [4]
Management Component		Elastic Load Balancer	The number of synchronous accesses to the application is used to determine the number of required application component instances.
		Elastic Queue	The number of asynchronous accesses to the application is used to adjust the number of required application component instances.
		Watchdog	Applications deal with failures automatically by monitoring and replacing application component instances.
		Elastic Manager	The application resources utilization is used to determine the number of required application component instances.
		Provider Adapter	The provider interfaces are encapsulated and mapped to unified interfaces used in applications.
		Configuration Manager	Use a centralized configuration to provide a unified behavior that can be adjusted simultaneously to all application components.
Application Component	Defined	Load Balancer	Determines the number of synchronous accesses to the application.
		Message Queue	Determines the number of asynchronous accesses to the application (that is, access via messaging).
	Undefined	Stateful Component	Multiple instances of an application component synchronize their internal state to provide a unified behavior.
		Stateless Component	State is handled external of application components to ease their scaling-out and to make the application more tolerant to component failures.
Functional components		Idempotent Processor	Component designed to detect duplicate messages and inconsistent data and to be immune to these conditions.
		Data Access Component	Accesses data elements, hiding the complexity of the access, enabling additional consistency and adjustability of handled data elements to different customer requirements.
		Data Abtractor	The data provided is abstracted to support eventually consistent data storage through the use of abstractions and approximations.
		Processing Component	Possibly long running processing functionality is handled by separate components to enable elastic scaling.
		Batch Processing Component	Requests are delayed until environmental conditions make their processing feasible.
		User Interface Component	Guarantees interactive synchronous access to applications and interaction asynchronously to application-internal elements.
		Multi-component Image	Virtual servers host multiple application components that may not be active at all times to reduce provisioning and decommissioning operations.
		Timeout based Message Processor	Clients acknowledge message receptions and processing to ensure that all messages are handled by an application. If a message is not acknowledged after a certain timeout, it is processed by a different client.
		Transaction based Message Processor	Components receive messages or read data and process the obtained information under a transactional context to ensure that all received messages are processes and all altered data is consistent after processing, respectively.



### III. DISCRETE EVENT SYSTEM SPECIFICATION (DEVS)

DEVS is a simulation modeling formalism that provides the basis for simulating systems of systems in a virtual environment [29]. This formalism allows describing any system behavior at two levels. At the lowest 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. At the higher level, a *coupled DEVS* describes a system as a network of coupled components which can be atomic DEVS models or coupled DEVS models [30].

#### A. Classic DEVS with Ports

Zeigler et al. [24] specified the *Classic DEVS with Ports* as an structure for describing single, sequentially executed, timed automata for discrete event systems. This type of model allows receiving input values by multiple ports at the same time. The formal specification of this DEVS model is detailed in (1).

$$DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta) \quad (1)$$

where:

$X = \{ (p, v) \mid p \in InPorts, v \in X_p \}$  is the set of inputs in which *InPorts* is the set input ports and  $X_p$  is the set of possible values for the  $p$  input port.

$Y = \{ (p, v) \mid p \in OutPorts, v \in Y_p \}$  is the set of outputs in which *OutPorts* is the set output ports and  $Y_p$  is the set of possible values for the  $p$  output port.

$S$  is the set of sequential states.

$\delta_{ext}: Q \times X \rightarrow S$  is the external state transition function where  $Q = \{ (s, e) \mid s \in S, 0 \leq e \leq ta(s) \}$  is the set of total states and  $e$  is the elapsed time in the  $s$  state.

$\delta_{int}: S \rightarrow S$  is the internal state transition function.

$\lambda: S \rightarrow Y$  is the output function.

$ta: S \rightarrow R_0^+$  is the time advance function.

#### B. Coupled DEVS

In the DEVS formalism, *atomic DEVS* captures the system behavior, while *coupled DEVS* describes the structure of system [29]. In fact, the design of coupled models implies the building of models from others models, which become components of it. Equation (2) details the formal specification of this DEVS model type.

$$N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select) \quad (2)$$

where:

$X = \{ (p, v) \mid p \in InPorts, v \in X_p \}$  is the set of inputs in which *InPorts* is the set input ports and  $X_p$  is the set of possible values for the  $p$  input port.

$Y = \{ (p, v) \mid p \in OutPorts, v \in Y_p \}$  is the set of outputs in which *OutPorts* is the set output ports and  $Y_p$  is the set of possible values for the  $p$  output port.

$D$  is the set of component reference names.

$M_d$  is the set of DEVS models that compose the coupled model.

$EIC \subseteq \{ ( (N, ip_N), (d, ip_d) ) \mid ip_N \in InPorts, d \in D, ip_d \in InPorts_d \}$  is the external input coupling.

$EOC \subseteq \{ ( (d, op_d), (N, op_N) ) \mid op_N \in OutPorts, d \in D, op_d \in OutPorts_d \}$  is the external output coupling.

$IC \subseteq \{ ( (a, op_a), (b, ip_b) ) \mid a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b \}$  is the internal coupling.

*Select* is the tie-breaking function (used in classic DEVS but eliminated in parallel DEVS).

### IV. SIMULATION MODELS FOR WEB ARCHITECTURES

#### A. Functional Component (FC)

A FC is an architectural element that belongs to the lower level of web architectures. The metamodel proposed in [23] identifies a set of FCs that share the following characteristics:

- *It can fail*: A FC can be induced into a fault or failure state while is active. Therefore, fault events must be created inside the component.
- *Faults and failures are spread to upper levels*: The occurrence of an error in a FC may involve errors in the following level of components.
- *If a failure occurs, then probably an incorrect behavior of the FC will have place*: The incorrect processing is given as a consequence of a failure state.

In order to model all the FCs, a generic simulation model was developed. Equation (3) formalized its definition.

$$FC = (X_{FC}, Y_{FC}, D_{FC}, \{M_{FC,d} \mid d \in D_{FC}\}, EIC_{FC}, EOC_{FC}, IC_{FC}) \quad (3)$$

where:

$X_{FC} = \{ (p, v) \mid p \in FCIP, v \in X_{FC,p} \}$  is the set of inputs where:

- $FCIP = \{ fromInfrastructure, requestToProcess \}$
- $X_{FC, fromInfrastructure} = X_{FC, requestToProcess} = \{ request_1, \dots, request_n \}$

$Y_{FC} = \{ (p, v) \mid p \in FCOP, v \in Y_{FC,p} \}$  is the set of outputs where:

- $FCOP = \{ requestReady, toInfrastructure, fault, F_R, F_U, state \}$
- $Y_{FC, requestReady} = Y_{FC, toInfrastructure} = \{ request_1, \dots, request_n \}$
- $Y_{FC, fault} = Y_{FC, FR} = Y_{FC, FU} = \{ R_0^+ \}$
- $Y_{FC, state} = \{ fallen, working \}$

$D_{FC} = \{ activator, error, processor \}$

$$M_{FC,d} = \begin{cases} M_{FC, activator} = Act \\ M_{FC, error} = EG \\ M_{FC, processor} = P \text{ where } P \text{ is a processor model.} \end{cases}$$

$EIC_{FC} = \{ ((FC, fromInfrastructure), (processor, fromInfrastructure)), ((FC, requestToProcess), (processor, requestToProcess)), (FC, requestToProcess), (activator, in) \}$

$EOC_{FC} = \{((processor, requestReady), (FC, requestReady)), ((processor, toInfrastructure), (FC, toInfrastructure)), ((processor, fault), (FC, fault)), ((processor, F_R), (FC, F_R)), ((processor, F_U), (FC, F_U)), ((processor, state), (FC, state))\}$

$IC_{FC} = \{((activator, state), (error, state)), ((error, f_{ONP}), (processor, f_{ONP})), ((error, f_{OIP}), (processor, f_{OIP})), ((error, f_{CNP}), (processor, f_{CNP})), ((error, f_{CIP}), (processor, f_{CIP})), ((error, F_R), (processor, F_R)), ((error, F_U), (processor, F_U)), ((processor, requestReady), (activator, out)), ((processor, state), (activator, componentState))\}$

Fig. 3 depicts the model structure. The FC model has two input ports and six output ports. Over the input ports receives the requests that should be processed and the responses of petitions that this component has made to the infrastructure. In contrast, by the output ports sends the processed requests, petitions to infrastructure, error events and its current state. The model's composition consists of three DEVS models: *Activator*, *Error Generator* and *Processor*.

When an input event takes place over the *requestToProcess* port, the event is spread to the *Activator* in order to start the model operation (if the component was in standby state). This event is also spread to the *Processor* in which will be waiting its turn to be processed. If the component was in standby state, the *Activator* sends the start signal to the *Error Generator* since it must start the generation of faults and failures. When the *Error Generator* determines that an error has occurred, a numerical signal is sent in the port that refers to the specific type of error generated. This signal is reported to the *Processor* with aim to define the following state of the FC given the type of error occurred. When an error event is received, the *Processor* stops its normal operation and reports an error state. Along with the state report, the *Processor* spread the error signal accepted to the architectural components included in the upper level. If the *Activator* receives an error state, stops the operation of the *Error Generator* by sending an event with *off* value by its output port. Since some error situations can be overcome, when the *Processor* detected that the error state has finished, sends a new event that represents the normal operation state. Once the pending request has been processed by the *Processor*, the request is spread to the next architectural component using the *requestReady* port. However, this request is also spread to the *Activator* in order to inform that the FC has one less request to process. If in any moment the *Activator* detected that there are not requests to process in the FC, the model knows that the FC should not be active. Then, it sends an event with *off* value by its output port.

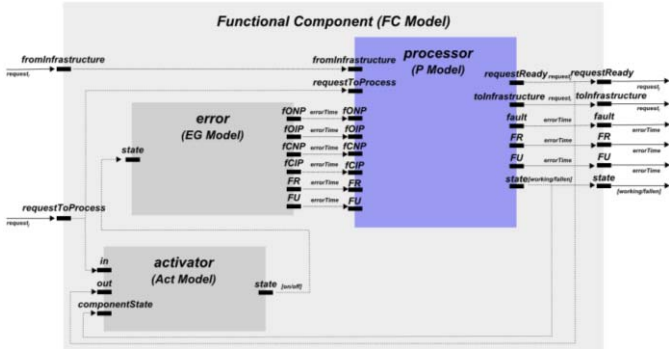


Fig. 3. Input and output interfaces and model coupling on the FC model.

### 1) Activator (Act)

The Act model acts as an intermediate component that turns on and off all the other elements that compose the FC. Its operation is based on the number of requests waiting to be processed by the FC. Equation (4) details the model definition.

$$Act = (X_{Act}, Y_{Act}, S_{Act}, \delta_{ext,Act}, \delta_{int,Act}, \lambda_{Act}, ta_{Act}) \quad (4)$$

where:

$X_{Act} = \{(p, v) \mid p \in ActIP, v \in X_{Act,p}\}$  is the set of inputs where:

- $ActIP = \{in, out, componentState\}$
- $X_{Act, componentState} = \{working, fallen\}$
- $X_{Act, in} = X_{Act, out} = \{request_1, \dots, request_n\}$

$Y_{Act} = \{(p, v) \mid p \in ActOP, v \in Y_{Act,p}\}$  is the set of outputs where:

- $ActOP = \{state\}$
- $Y_{Act, state} = \{on, off\}$

$S_{Act} = (phase \times \sigma \times requestsToProcess)$  where:

- $phase \in \{waiting, turning\ on, turning\ off, stopping, rejecting\}$
- $\sigma \in R_0^+$
- $requestsToProcess \in N_0$

$$\delta_{ext,Act} = \begin{cases} \delta_{ext,Act}[(waiting, \sigma, 0), e, (request, in)] = (turning\ on, 0, 1) \\ \delta_{ext,Act}[(waiting, \sigma, requestsToProcess), e, (request, in)] = (waiting, \infty, requestsToProcess + 1) \\ \delta_{ext,Act}[(waiting, \sigma, 1), e, (request, out)] = (turning\ off, 0, 0) \\ \delta_{ext,Act}[(waiting, \sigma, requestsToProcess), e, (request, out)] = (waiting, \infty, requestsToProcess - 1) \\ \delta_{ext,Act}[(waiting, \sigma, requestsToProcess), e, (fallen, componentState)] = (stopping, 0, requestsToProcess) \\ \delta_{ext,Act}[(rejecting, \sigma, requestsToProcess), e, (working, componentState)] = (waiting, \infty, 0) \\ \delta_{ext,Act}[(p, \sigma, req), e, (x, port)] = (p, \sigma - e, req) \text{ en otro caso} \end{cases}$$

$$\delta_{int,Act} = \begin{cases} \delta_{int,Act}(turning\ on, \sigma, requestsToProcess) = (waiting, \infty, requestsToProcess) \\ \delta_{int,Act}(turning\ off, \sigma, requestsToProcess) = (waiting, \infty, requestsToProcess) \\ \delta_{int,Act}(stopping, \sigma, requestsToProcess) = (rejecting, \infty, requestsToProcess) \end{cases}$$

$$\lambda_{Act} = \begin{cases} \lambda_{Act}(turning\ on, \sigma, requestsToProcess) = (state, on) \\ \lambda_{Act}(turning\ off, \sigma, requestsToProcess) = (state, off) \\ \lambda_{Act}(stopping, \sigma, requestsToProcess) = (state, off) \end{cases}$$

$$ta_{Act}(s) = \sigma$$

The model's initial state is  $s = (waiting, \infty, 0)$ . When receives the first event over the *in* port, the model increments a unit on *requestsToProcess* and changes to *turning on* phase. During this phase, the model sends an *on* signal to its output port in order to initiate the FC operation. Then, returns its phase value to *waiting*. Every time that an event have place in the *in* (*out*) port, the actual value of *requestToProcess* is incremented (decremented). If in any moment the *requestToProcess* value is 0, the model sends an *off* signal by its output port with aim to stop the FC operation. When a fallen event is received in the *componentState* port, the Act stops the FC operation and starts to reject all the entries until receive a working signal.

## 2) Error Generator (EG)

The objective of the EG model is to generate different types of error events according to the following set of configuration parameters:

- *Time between errors (tbe)*: The error generation process follows an exponential distribution.
- *Fault duration (minimum and maximum time)*: The faults duration time follows a uniform distribution.
- *Failure reparation time (minimum and maximum time)*: The failure reparation follows a uniform distribution.
- *Failure occurrence probability (fop)*: Given  $fop=F$  and  $x$  as random number where  $\{F,x\} \in [0;1]$ . If  $x < F$  then the error event is a failure. In other case, is a fault.
- *Failure reparation probability (frp)*: Given  $frp=R$  and  $x$  as random number where  $\{R,x\} \in [0;1]$ . If  $x < R$  then the error event is a failure which can be fixed. In other case, is an irreparable failure.
- *Fault persistence probability (fpp)*: Given  $fpp=C$  and  $x$  as random number where  $\{C,x\} \in [0;1]$ . If  $x < C$  then the error event is a persistent fault. In other case, is an occasional fault that will last a fixed time.
- *Inadequate processing probability (ipp)*: Given  $ipp=I$  and  $x$  as random number where  $\{I,x\} \in [0;1]$ . If  $x < I$  then an incorrect processing will be consequence of the fault. This incorrect processing will be active while the fault exists. In other case, the fault occurrence will not guarantee the inadequate processing.

The combination of these parameters gives six types of possible error events: occasional fault with normal processing, occasional fault with inadequate processing, chronic fault with normal processing, chronic fault with inadequate processing, repairable failure and unrepairable failure. Each error type generated by the model is sent by a specific output port. Equation (5) formalized the EG model specification.

$$EG = (X_{EG}, Y_{EG}, S_{EG}, \delta_{ext,EG}, \delta_{int,EG}, \lambda_{EG}, ta_{EG}) \quad (5)$$

where:

$X_{EG} = \{(p,v) \mid p \in EGIP, v \in X_{EG,p}\}$  is the set of inputs where:

- $EGIP = \{state\}$
- $X_{EG,state} = \{on, off\}$

$Y_{EG} = \{(p,v) \mid p \in EGOP, v \in Y_{EG,p}\}$  is the set of outputs where:

- $EGOP = \{f_{ONP}, f_{OIP}, f_{CNP}, f_{CIP}, F_R, F_U\}$
- $Y_{EG,p} = \{R_0^+\} \forall p \in EGOP$

$S_{EG} = (phase \ x \ \sigma)$  where:

- $phase \in \{active, inactive\}$
- $\sigma \in R_0^+$

$$\delta_{ext,EG} = \begin{cases} \delta_{ext,EG}[(inactive,\sigma),e,(on,state)] = (active,tbe) \\ \delta_{ext,EG}[(active,\sigma),e,(off,state)] = (inactive,\infty) \\ \delta_{ext,EG}[(active,\sigma),e,(on,state)] = (active,\sigma-e) \\ \delta_{ext,EG}[(inactive,\sigma),e,(off,state)] = (inactive,\sigma-e) \end{cases}$$

$$\delta_{int,EG}(active,\sigma) = (active,tbe)$$

$$\lambda_{EG}(active,\sigma) = (errorType,errorTime)$$

$$ta_{EG}(s) = \sigma$$

The initial state is  $(inactive,\infty)$ . When an on events is received by the *state* port, the model changes to  $(active,tbe)$  where *tbe* is a random value obtained from *getTimeBetweenErrors()*. This function returns a value according to an exponential distribution configured with the parameters set. The model will be in *active* phase until receives an *off* signal. In this case, the EG will return to the initial state. During the *active* phase, the EG will generate different error events which will be sent by the corresponding output port. The error type is generated by *getErrorType()*. This function uses the parameters *fop*, *frp*, *fpp* and *ipp* along with random numbers to determine the type of event to be sent. Inside the error event, the model sends a real value that represents the error duration time (*errorTime*). This value is obtained from a function called *getErrorTime()* that is configured according to the minimum and maximum times provided in the model.

## 3) Processor (P)

Given that each FC has its own behavior, one P model was described for each specific component. In all the cases the models contemplate two types of operation: normal processing and inadequate processing. For space reasons the models developed are not detailed in this work.

## B. Defined Application Component (DAC)

A DAC is an architectural element that belongs to the upper level of web architectures. It refers to application components that architects can simply use in their designs. Equation (6) details the DAC definition and Fig. 4 shows its structure.

$$DAC = (X_{DAC}, Y_{DAC}, D_{DAC}, \{M_{DAC,d} \mid d \in D_{DAC}\}, EIC_{DAC}, EOC_{DAC}, IC_{DAC}) \quad (6)$$

where:

$X_{DAC} = \{(p,v) \mid p \in DACIP, v \in X_{DAC,p}\}$  is the set of inputs where:

- $DAIP = \{requestToProcess, fromManagemComponent\}$
- $X_{DAC,requestToProcess} = X_{DAC,fromManagementComponent} = \{request_1, \dots, request_n\}$

$Y_{DAC} = \{(p,v) \mid p \in DACOP, v \in Y_{DAC,p}\}$  is the set of outputs where:

- $DACOP = \{requestReady, toManagementComponent, state\}$
- $Y_{DAC,requestReady} = \{request_1, \dots, request_n\}$
- $Y_{DAC,toManagementComponent} = \{information_1, \dots, information_n\}$
- $Y_{DAC,state} = \{working, fallen\}$

$D_{DAC} = \{receptor, component, calculator\}$

$$M_{DAC,d} = \begin{cases} M_{DAC,receptor} = R \\ M_{DAC,component} = DACS \text{ (component specification)} \\ M_{DAC,calculator} = FFC \end{cases}$$

$$EIC_{DAC} = \{((DAC,requestToProcess),(receptor, requestToProcess)), ((DAC, fromManagementComponent), (component, fromManagementComponent))\}$$



$EOC_{DAC} = \{((component, requestReady), (DAC, requestReady)), ((component, toManagementComponent), (DAC, toManagementComponent)), ((component, state), (DAC, state))\}$

$IC_{DAC} = \{((receptor, requestToProcess), (component, requestToProcess)), ((receptor, fromManagementComponent), (component, fromManagementComponent)), ((component, fault), (calculator, fault)), ((component, F_R), (calculator, F_R)), ((component, F_U), (calculator, F_U))\}$

The DAC model includes three DEVS models: *Receptor*, *Defined Application Component Specification* and *Fault and Failure Calculator*. When an event occurs in the *requestToProcess* port, the *Receptor* determines if the component should accept it. If the request is accepted, the *Defined Application Component Specification* has to process it. After the processing has finished, the request is sent by the *requestReady* port. The events related with the *fromManagementComponent* port follow the same process. The *Fault and Failure Calculator* is a model used to obtain error metrics and, therefore, receives the different error signals.

### 1) Receptor (R)

This model detects if a specific request should be accepted in the DAC. Equation (7) details its specification.

$$R = (X_R, Y_R, S_R, \delta_{ext,R}, \delta_{int,R}, \lambda_R, ta_R) \quad (7)$$

where:

$X_R = \{(p, v) \mid p \in RIP, v \in X_{R,p}\}$  is the set of inputs where:

- $RIP = \{requestToProcess, fromInfrastructure\}$
- $X_{R,fromInfrastructure} = X_{R,requestToProcess} = \{req_1, \dots, req_n\}$

$Y_R = \{(p, v) \mid p \in ROP, v \in Y_{R,p}\}$  is the set of outputs where:

- $ROP = \{requestToProcess, fromInfrastructure\}$
- $Y_{R,fromInfrastructure} = Y_{R,requestToProcess} = \{req_1, \dots, req_n\}$

$S_R = (phase \times \sigma \times actualRequest)$  where:

- $phase \in \{waiting, resending infrastructure message, resending request\}$
- $\sigma \in R_0^+$
- $actualRequest \in \{request_1, \dots, request_n\}$

$$\delta_{ext,R} = \begin{cases} \delta_{ext,R}[(waiting, \sigma, actualRequest), e, (request, requestToProcess)] = (resending request, 0, request) \\ \delta_{ext,R}[(waiting, \sigma, actualRequest), e, (request, fromInfrastructure)] = (resending infrastructure message, 0, request) \\ \delta_{ext,R}[(waiting, \sigma, actualRequest), e, (request, fromInfrastructure)] = (waiting, \sigma - e, actualRequest) \text{ en otro caso} \end{cases}$$

$$\delta_{int,R} = \begin{cases} \delta_{int,R}(resending request, \sigma, actualReq) = (waiting, \infty, \phi) \\ \delta_{int,R}(resending infrastructure message, \sigma, actualReq) = (waiting, \infty, \phi) \end{cases}$$

$$\lambda_R = \begin{cases} \lambda_R(resending request, \sigma, actualReq) = (requestToProcess, removeFromPath(actualRequest)) \\ \lambda_R(resending infrastructure message, \sigma, actualReq) = (fromInfrastructure, removeFromPath(actualReq)) \end{cases}$$

$$ta_R(s) = \sigma$$

The R model knows the DAC identifier. Given that the requests have the path of components that must follow; the R compares the value of the next component that must treat the request with the DAC identifier. To obtain the identifier of the next component, the R uses the function *nextComponentId()*. If these values are equal, then the DAC must accept the request. This behavior happens when the model spreads the input request by the *requestToProcess* output port. In other case, the request is rejected and the R returns to the *waiting* phase.

### 2) Fault and Failure Calculator (FFC)

The FFC model counts the different types of error events that take place inside the DAC in order to allow a posterior quality analysis. Equation (8) formalizes the FFC model specification.

$$FFC = (X_{FFC}, Y_{FFC}, S_{FFC}, \delta_{ext,FFC}, ta_{FFC}) \quad (8)$$

where:

$X_{FFC} = \{(p, v) \mid p \in FFCIP, v \in X_{FFC,p}\}$  is the set of inputs where:

- $FFCIP = \{fault, F_R, F_U\}$
- $X_{FFC,fault} = X_{FFC,FR} = X_{FFC,FU} \in \{R_0^+\}$

$Y_{FFC} = \{(p, v) \mid p \in FFCOP, v \in Y_{FFC,p}\}$  is the set of outputs where  $FFCOP = \{\phi\}$

$S_{FFC} = (phase \times \sigma)$  where:

- $phase \in \{waiting\}$
- $\sigma \in R_0^+$

$$\delta_{ext,FFC}[(waiting, \sigma), e, (errorTime, fault)] = (waiting, \infty)$$

$$ta_{FFC}(s) = \sigma$$

The FFC model has three input ports. On the first port receives the fault events without details of its permanence or associated processing strategy, while in the second and third port receives the repairable and unrepairable failures respectively. This model has no output ports since only counts the error events. When an input event occurs, the model increments the counter associated in one unit and returns to the *waiting* phase.

### 3) Defined Application Component Specification (DACS)

Each DAC has its own behavioral specification. Therefore, for each DAC a DACS model was created. In this context, each DACS model represents the specific behavior of the defined components proposed in the metamodel. For space reasons the designed models are not detailed in this work.

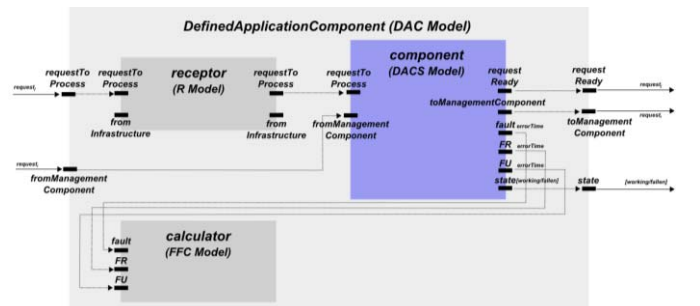


Fig. 4. Input and output interfaces and model coupling on the DAC model.

### C. Undefined Application Component (UAC)

A UAC is an architectural element that belongs to same architectural level that the DAC components. However, the difference between DAC and UAC is that the behavior of UAC is not defined. Then, the software architect must use the FC elements to specify a sequence of operations that define its behavior. Equation (9) describes the UAC definition and Fig. 5 shows the model structure.

$$UAC = (X_{UAC}, Y_{UAC}, D_{UAC}, \{M_{UAC,d} | d \in CD_{UAC}\}, EIC_{UAC}, EOC_{UAC}, IC_{UAC}) \quad (9)$$

where:

$X_{UAC} = \{(p,v) | p \in UACIP, v \in X_{UAC,p}\}$  is the set of inputs where:

- $UAIP = \{requestToProcess, fromInfrastructure\}$
- $X_{UAC,requestToProcess} = X_{UAC,fromInfrastructure} = \{request_1, \dots, request_n\}$

$Y_{UAC} = \{(p,v) | p \in UACOP, v \in Y_{UAC,p}\}$  is the set of outputs where:

- $UACOP = \{requestReady, toInfrastructure, state\}$
- $Y_{UAC,requestReady} = Y_{UAC,toInfrastructure} = \{request_1, \dots, request_n\}$
- $Y_{UAC,state} = \{working, fallen\}$

$D_{UAC} = \{receptor, sequence, sender, calculator, state\}$

$$M_{UAC,d} = \begin{cases} M_{UAC,receptor} = R \\ M_{UAC,sequence} = FCS \\ M_{UAC,sender} = SIS \\ M_{UAC,calculator} = FFC \\ M_{UAC,state} = SP \end{cases}$$

$EIC_{UAC} = \{((UAC, requestToProcess), (receptor, requestToProcess)), ((UAC, fromInfrastructure), (receptor, fromInfrastructure))\}$

$EOC_{UAC} = \{((sequence, requestReady), (UAC, requestReady)), ((sender, toInfrastructure), (UAC, toInfrastructure)), ((state, componentState), (UAC, state))\}$

$IC_{UAC} = \{((receptor, requestToProcess), (sequence, requestToProcess)), ((receptor, fromInfrastructure), (sequence, fromInfrastructure)), ((sequence, toInfrastructure), (sender, toInfrastructure)), ((sequence, fault), (calculator, fault)), ((sequence, F_R), (calculator, F_R)), ((sequence, F_U), (calculator, F_U)), ((sequence, state), (state, componentState))\}$

The UAC model includes five DEVS models: *Receptor*, *Functional Component Sequence*, *Sender Information Setter*, *State Processor* and *Fault and Failure Calculator*. Some of these models were detailed in the previous section. The rest is described in the next section.

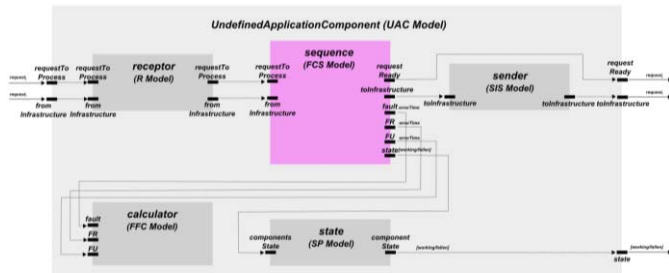


Fig. 5. Input and output interfaces and model coupling on the UAC model.

#### 1) Sender Information Setter (SIS)

Given that messages to infrastructure represent petitions; its solutions should be received in the same component that makes the requests. To do this, the SIS adds the UAC identifier into the components list that must process the actual request using *addComponentToPathFirst()*. After that, the model sends the request by its output port. Equation (10) shows its definition.

$$SIS = (X_{SIS}, Y_{SIS}, S_{SIS}, \delta_{ext,SIS}, \delta_{int,SIS}, \lambda_{SIS}, ta_{SIS}) \quad (10)$$

where:

$X_{SIS} = \{(p,v) | p \in SISIP, v \in X_{SIS,p}\}$  is the set of inputs where:

- $SISIP = \{toInfrastructure\}$
- $X_{SIS,toInfrastructure} = \{req_1, \dots, req_n\}$

$Y_{SIS} = \{(p,v) | p \in SISOP, v \in Y_{SIS,p}\}$  is the set of outputs where:

- $SISOP = \{toInfrastructure\}$
- $Y_{SIS,toInfrastructure} = \{req_1, \dots, req_n\}$

$S_{SIS} = (phase \times \sigma \times actualMessage)$  where:

- $phase \in \{waiting, sending infrastructure message\}$
- $\sigma \in R_0^+$
- $actualMessage \in \{request_1, \dots, request_n\}$

$\delta_{ext,SIS}(waiting, \sigma, actualMessage), e, (request, toInfrastructure)] = (sending infrastructure message, 0, request)$

$\delta_{int,SIS}(sending infrastructure message, \sigma, actualMessage) = (waiting, \infty, \phi)$

$\lambda_{SIS}(sending infrastructure message, \sigma, actualMessage) = (toInfrastructure, addComponentToPathFirst(actualMessage))$

$ta_{SIS}(s) = \sigma$

#### 2) State Processor (SP)

The UAC state is defined as the global state of all the FCs included inside of it. Only when all FCs fail, the UAC fails. The SP model is designed to determine the state of the UAC using the states reported by the FC elements. With this aim, the SP uses the *failedComponents* counter that increases (decrements) in one unit every time that the model receives an input event with *fallen* (*working*) value. If the value of *failedComponents* achieves the number of FC included in the UAC, a *fallen* event is sent by its output port. Once these indicators take different values, a *working* event is sent by its output port. Equation (11) describes its formalization.

$$SP = (X_{SP}, Y_{SP}, S_{SP}, \delta_{ext,SP}, \delta_{int,SP}, \lambda_{SP}, ta_{SP}) \quad (11)$$

where:

$X_{SP} = \{(p,v) | p \in SPIP, v \in X_{SP,p}\}$  is the set of inputs where:

- $SPIP = \{componentsState\}$
- $X_{SP,componentsState} = \{working, fallen\}$

$Y_{SP} = \{(p,v) | p \in SPOP, v \in Y_{SP,p}\}$  is the set of outputs where:

- $SPOP = \{componentState\}$
- $Y_{SP,componentState} = \{working, fallen\}$



$S_{SP} = (\text{phase } x \sigma x \text{ failedComponents})$  where:

- $\text{phase} \in \{ \text{waiting}, \text{sending fallen state}, \text{sending working state}, \text{inactive} \}$
- $\sigma \in R_0^+$
- $\text{failedComponents} \in N_0$

$$\delta_{ext,SP} = \begin{cases} \delta_{ext,SP}[\text{waiting}, \sigma, \text{failedComponents}], e, (\text{fallen}, \text{componentsState})] = (\text{sending fallen state}, 0, \text{failedComponents} + 1) \\ \delta_{ext,SP}[\text{waiting}, \sigma, \text{failedComponents}], e, (\text{fallen}, \text{componentsState})] = (\text{waiting}, \infty, \text{failedComponents} + 1) \\ \delta_{ext,SP}[\text{waiting}, \sigma, \text{failedComponents}], e, (\text{working}, \text{componentsState})] = (\text{waiting}, \infty, \text{failedComponents} - 1) \\ \delta_{ext,SP}[\text{inactive}, \sigma, \text{failedComponents}], e, (\text{working}, \text{componentsState})] = (\text{sending working state}, 0, \text{failedComponents} - 1) \\ \delta_{ext,SP}[\text{inactive}, \sigma, \text{failedComponents}], e, (\text{fallen}, \text{componentsState})] = (\text{inactive}, \infty, \text{failedComponents}) \end{cases}$$

$$\delta_{int,SP} = \begin{cases} \delta_{int,SP}[\text{sending fallen state}, \sigma, \text{failedComponents}] = (\text{inactive}, \infty, \text{failedComponents}) \\ \delta_{int,SP}[\text{sending working state}, \sigma, \text{failedComponents}] = (\text{waiting}, \infty, \text{failedComponents}) \end{cases}$$

$$\lambda_{SP} = \begin{cases} \lambda_{SP}[\text{sending fallen state}, \sigma, \text{failedComponents}] = (\text{componentState}, \text{fallen}) \\ \lambda_{SP}[\text{sending working state}, \sigma, \text{failedComponents}] = (\text{componentState}, \text{working}) \end{cases}$$

$$ta_{SP}(s) = \sigma$$

### 3) Functional Component Sequence (FCS)

The FCS model is a coupled model that only specifies a set of input and output ports. The components included in the model must be derived from the original architectural design using the FC model proposed to each component.

## V. USERS SIMULATION MODELS

The consumers of a web application (or software service) can be considered users. In order to model the requests generation process, different types of workloads were used to simulate the temporal behavior of users [4]. Table II summarizes and schematizes the proposed workloads.

### A. Request Generation Process

Given that the requests generation process is independent of users' behavior, a unique coupled model was created to simulate this activity. This model was called RG.

Fig. 6 depicts the structure of the RG model in terms of component elements. The definitions of the models that compose the RG model are described in the next section. For space reasons, the formalization of these models is not included in this work.

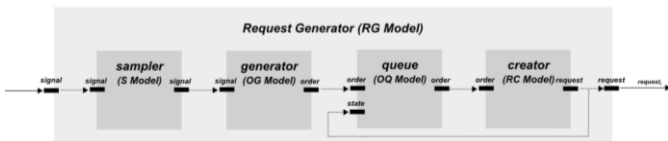

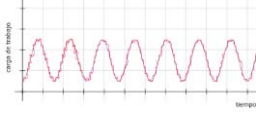
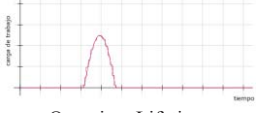
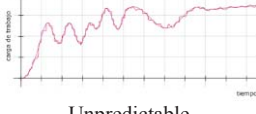



Fig. 6. Input and output interfaces and model coupling on the UAC model.

TABLE II. WORKLOADS USED TO MODEL TEMPORAL BEHAVIORS ([4]).

Type	Description
 Static	Static workloads are characterized by a more-or-less flat utilization profile over time within certain boundaries.
 Periodic	In a periodic workload the resources follow a peaking utilization at reoccurring time intervals.
 Once in a Lifetime	Is a special case of periodic workload in which the peaks of periodic utilization can occur only once in a very long time frame.
 Unpredictable	The random workloads are a generalization of the periodic workloads since are not predictable.
 Continuously Changing <sup>a</sup>	The utilization grows or shrinks constantly over time.

<sup>a</sup> The figure corresponds to a continuously increasing workload.

### 1) Sampler (S)

The S model receives a signal and sends an output event with the last value received according to a fixed time interval.

### 2) Order Generator (OG)

The OG model receives an integer value that represents the number of requests to be created. Then, it sends as many output events as the value received. Inside of each output event, the model sets a request type identifier. The available values to this identifier are configured in the model parameters.

### 3) Order Queue (OQ)

The OQ model implements a queue of orders. If the processor is available, all input events that arrive to the *order* port are immediately spread to the output port. However, if the processor is not available, the events are queued. When the processor informs its availability (using the *state* input port), one order is removed from the queue. This order is sent by the output port and the state of the processor is actualized to unavailable.

### 4) Request Creator (RC)

The RC model receives an order with a request identifier and creates the request that must be sent to the architecture model. The information related with each request type is obtained from an external file accessed by the model parameters.

## B. Users Behavior

Each user behavior was designed combining the RG model with a continuous signal that represents a specific workload. Fig. 7 shows the models implemented in PowerDEVS<sup>1</sup>.

<sup>1</sup> PowerDEVS - Available at: <http://sourceforge.net/projects/powerdevs/>

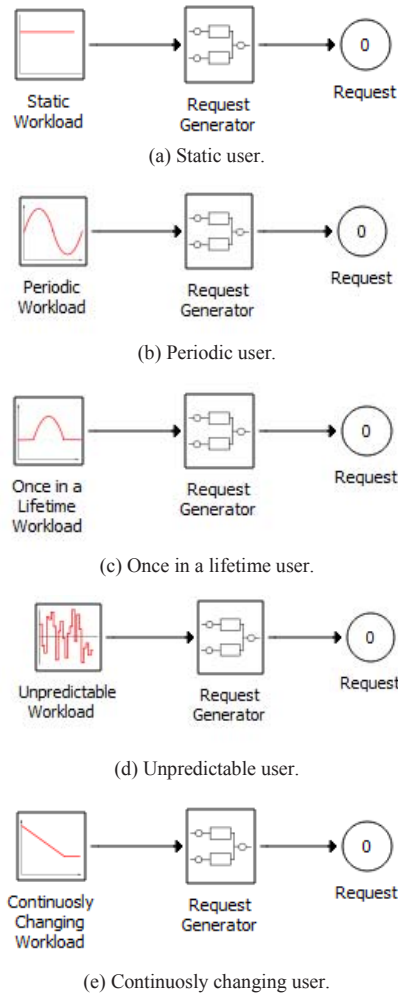


Fig. 7. Different types of user models.

## VI. CASE STUDY: THREE TIER WEB APPLICATION ARCHITECTURE TRANSFORMED TO SIMULATION MODELS

The tier software architectures provide an architectural style that allows build flexible and reusable web applications. These two properties make this style ideal to be applicable over CC environments.

The most used style is the three-tier architecture. In this type of architecture the presentation logic, business logic, and data handling are realized as separate tiers to scale stateless presentation and compute-intensive processing independently of the data tier, which is harder to scale and often handled by the cloud provider.

### A. Three-Tier Cloud Application Proposed

The three-tier architecture proposed by Fehling et. al. [4] was used as a previous case study in [23]. Fig. 8 presents the original architectural design. As the figure shows, each tier is composed by a set of defined application components, undefined application components, functional components and management components presented in Table I. However, the user behaviors are not detailed given that its specification is not really a part of the architectural design.

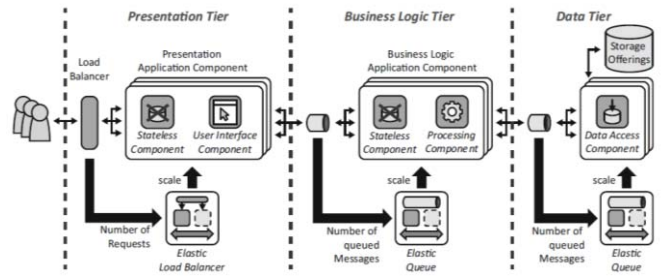


Fig. 8. Three-tier cloud application (from [4]).

### B. Modeling the Web Architecture using the Metamodel

The architecture proposed in Fig. 8 shows not only the web application architecture but also the infrastructure architecture. In order to model the web architecture with the metamodel proposed in [23], a separation of components was made. Given that along with the metamodel a design tool was developed, the final web architecture was described using this tool. Fig. 9 shows the final architecture.

### C. From Architectural Design to Simulation Models

Each type of architectural component has been associated with its own DEVS model. In this context, a direct transformation can be made:

- *User Interface Component*, *Processing Component* and *Data Access Component* correspond to the simulation model presented in Eq. (3).
- *Load Balancer* and *Message Queue* correspond to the simulation model presented in Eq. (6).
- *Presentation*, *Business Logic* and *Data* correspond to the simulation model presented in Eq. (9).

Given that *Elastic Load Balancer* and *Elastic Queue* are architectural components that still have not been modeled in terms of discrete event simulation elements; its transformation is not presented in this work.

Fig. 10 schematizes the components transformation according to the previous statements. It also shows how the simulation models should be related in order to maintain the original design. Since the composition of each model was already detailed as part of precedent work, each model is represented as a black box in which only the input and output ports of the containers are specified.

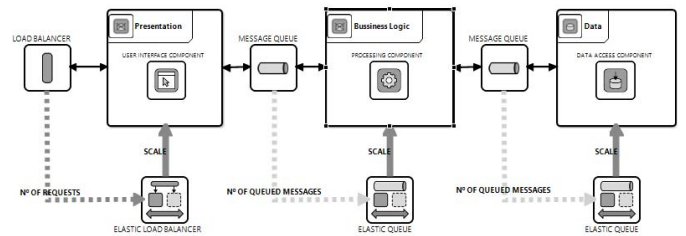


Fig. 9. Three-tier web cloud application modeled with the design tool that gives support to the metamodel components. Only web application is modeled. The details of the infrastructure architecture are omitted since its specification is not included as part of the metamodel.

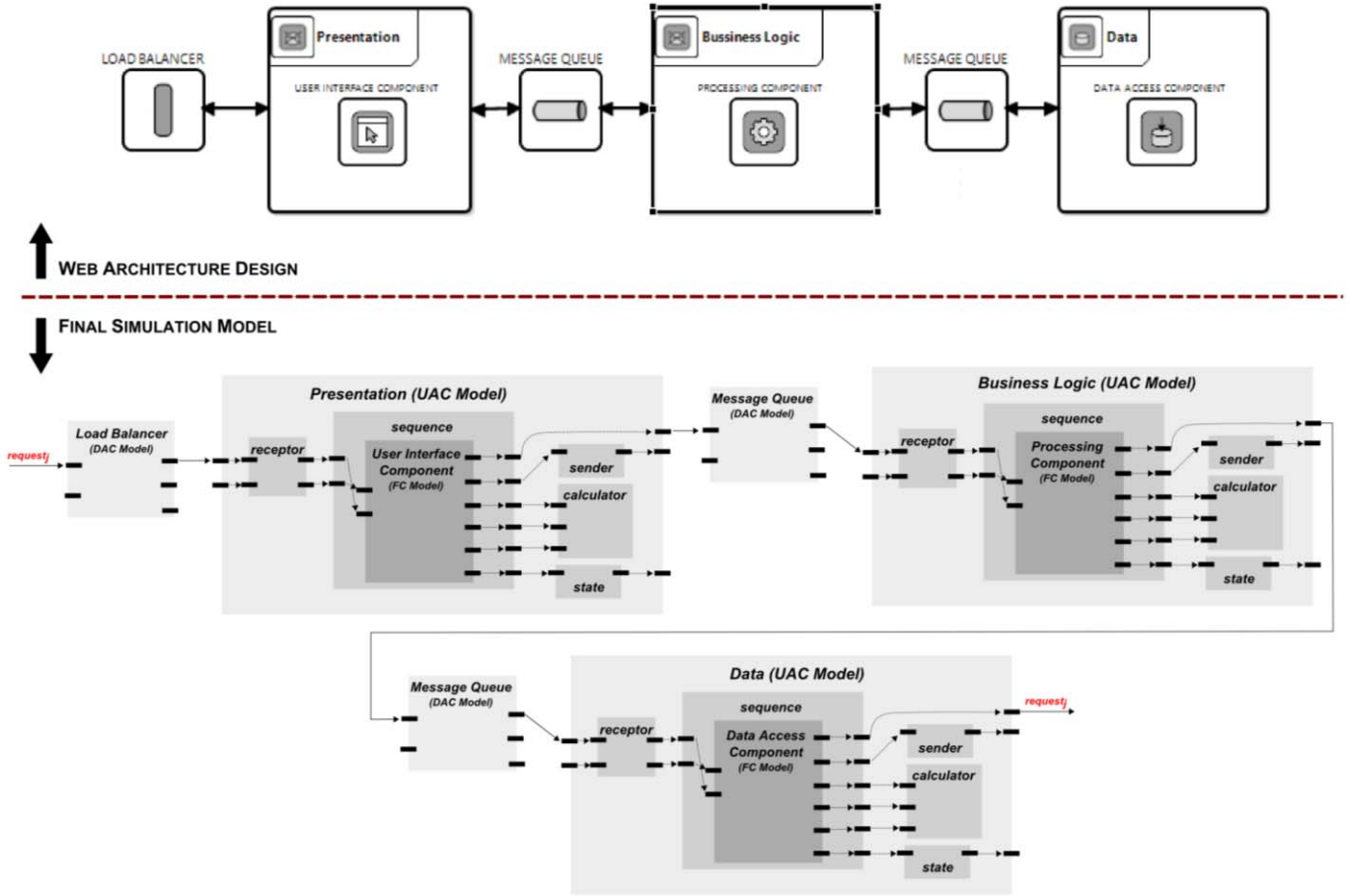


Fig. 10. Final DEVS model of the three-tier web application proposed.

## VII. CONCLUSIONS AND FUTURE WORK

The quality of a software system can be evaluated using simulation techniques. These techniques are also applicable to CC environments, specifically over the architectural designs of web applications. In this paper, a mapping between architectural components and simulation models is proposed in order to provide a base to a posterior quality analysis. The formalization and composition of these models are the main aspects treated in this work.

The transformation of the management components to simulation models is now been developed. When all the architectural components can be mapped to DEVS simulation models, all the case study elements will be able to be transformed into a unique simulation model. In this context, the user behavior specification should be also included as part of the architectural mapping process.

Given that the simulation approach presented is based on the metamodel developed on [23], the implementation of the simulation models can be integrated to the design tool that supports its modeling activity. In fact, the final objective of this research is to build a software tool that helps the architect to evaluate the adjustment of its design to the overall quality required by the web application. Then, the quality evaluation in CC environments will be possible.

## ACKNOWLEDGMENT

The authors wish to acknowledge the financial support received from CONICET (PIP 112-20110100906) and Universidad Tecnológica Nacional – Facultad Regional Santa Fe (UTI3803TC and UTI3804TC).

## REFERENCES

- [1] G. Lewis, "Basics about cloud computing", Software Engineering Institute Carnegie Mellon University, Pittsburgh, Sept. 2010.
- [2] A. N., Khan, M.L. Mat Kian, S. U. Khan, S. A. Mandani. "Towards secure mobile cloud computing: A survey", Future Generation Computer Systems, 2013, vol. 29, no 5, pp. 1278-1299.
- [3] F. Hu, M. Qiu, J. Li, T. Grant, D. Taylor, S. McCaleb, L. Butler, R. Hamner, "A review on cloud computing: Design challenges in architecture and security", CIT. Journal of Computing and Information Technology, 2011, vol. 19, no 1, p. 25-55.
- [4] C. Fehling, F. Leymann, R. Retter, W. Schuppeck, P. Arbitter, "Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications", Springer Science & Business Media, 2014.
- [5] Q. Zhang, L. Cheng, R. Boutaba, "Cloud computing: state-of-the-art and research challenges", Journal of internet services and applications, 2010, vol. 1, no 1, p. 7-18.
- [6] P.X. Wen, L. Dong, "Quality Model for Evaluating SaaS Service" in 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies (EIDWT), pp. 83-87, 2013.
- [7] A. Weiss, "Computing in the clouds", Computing, 2007, vol. 16.



- [8] C. Modi, D. Patel, B. Borisaniya, A. Patel, M. Rajarajan, "A survey on security issues and solutions at different layers of Cloud computing", *Journal Supercomput*, vol. 63, 2012, pp. 561–592.
- [9] J. Lee, "A quality model for evaluating software-as-a-service in cloud computing", presented at the 7<sup>th</sup> International Conference on Software Engineering Research, Management and Applications, 2009.
- [10] J. Gao, P. Pattabhiraman, X. Bai, W. T. Tsai, "SaaS performance and scalability evaluation in clouds" in 2011 IEEE 6th International Symposium on Service Oriented System Engineering (SOSE), 2011, pp. 61–71.
- [11] K. C. Huang, B. Shen, "Service deployment strategies for efficient execution of composite SaaS applications on cloud platform", *J. Syst. Softw.*, vol. 107, pp. 127–141, 2015.
- [12] H. Fan, F. K. Hussain, M. Younas, O. K. Hussain, "An integrated personalization framework for SaaS-based cloud services", *Future Gener. Comput. Syst.*, vol. 53, 2015, pp. 157–173.
- [13] R. Krebs, C. Momm, S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments", *Sci. Comput. Program.* vol. 90, part B, 2014, pp. 116–134.
- [14] D. Zehe, W. Cai, A. Knoll, H. Aydt, "Tutorial on a Modeling and Simulation Cloud Service" in *Proceedings of the 2015 Winter Simulation Conference*, pp. 103–114. IEEE Press, Piscataway, NJ, USA, 2015.
- [15] R. Buyya, R. Ranjan, R. N. Calheiros, "Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities", *High Performance Computing & Simulation, 2009 HPCS'09, International Conference on*. IEEE, 2009.
- [16] V. Bogado, S. Gonnet, H. Leone, "Modeling and simulation of software architecture in discrete event system specification for quality evaluation", *Simulation*, vol. 90, 2014, pp. 290.
- [17] J. Rech, C. Bunse, "Evaluating Performance of Software Architecture Models with the Palladio Component Model", *Model-Driven Software Development: Integrating Quality Assurance*, IDEA Group Inc, 2008, pp. 95–118.
- [18] A. Meiappane, B. Chithra, P. Venkataesan, "Evaluation of Software Architecture Quality Attribute for an Internet Banking System", *International Journal of Science Engineering Ref.*, vol. 4, 2013, pp. 1704–1708.
- [19] P. Kruchten, P. Lago, H. Van Vliet, "Building Up and Reasoning About Architectural Knowledge", in Hofmeister, C., Crnkovic, I., and Reussner, R. Eds. *Quality of Software Architectures*, Springer Berlin Heidelberg, 2006, pp. 43–58.
- [20] Z. Li, P. Liang, P. Avgeriou, "Application of knowledge-based approaches in software architecture: A systematic mapping study", *Inf. Softw. Technol.*, vol. 55, 2013, pp. 777–794.
- [21] M. L. Roldán, S. Gonnet, H. Leone, "Knowledge representation of the software architecture design process based on situation calculus", *Experts Systems*, vol. 30, 2013, pp. 34–53.
- [22] S. Giesecke, W. Hasselbring, M. Riebisch, "Classifying architectural constraints as a basis for software quality assessment", *Advanced Engineering Informatics*, vol. 21, nro. 2, pp. 169–179, 2007.
- [23] M. J. Blas, S. Gonnet, H. Leone, "Un Modelo para la Representación de Arquitecturas Cloud basadas en Capas por medio de la Utilización de Patrones de Diseño: Especificación de la Capa de Aplicación", presented at the 3<sup>o</sup> Congreso Nacional de Ingeniería Informática/Sistemas de Información (CONAIISI), Universidad Tecnológica Nacional-Facultad Regional Buenos Aires, November 19, 2015.
- [24] B. P. Zeigler, H. Praehofer, T. G. Kim, "Theory of Modeling and Simulation", Second Edition, Academic Press, San Diego, 2000.
- [25] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison-Wesley Professional, 2012.
- [26] B. Kitchenham, S. L. Pfleeger, "Software Quality: The Elusive Target", *IEEE software*, vol. 13, nro. 1, 1996, pp. 12.
- [27] R. Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill, 2010.
- [28] D. Magalhães, R. N. Calheiros, R. Buyya, D. G. Gomes, "Workload modeling for resource usage analysis and simulation in cloud computing", *Computers & Electrical Engineering*, 2015, vol. 47, pp. 69–81.
- [29] B. P. Zeigler, H. S. Sarjoughian, "Guide to Modeling and Simulation of Systems of Systems", Springer London, London, 2013.
- [30] H. Vangheluwe, "The Discrete Event System specification (DEVS) formalism", Course Notes, Course: Modeling and Simulation (COMP522A), McGill University, Montreal Canada, 2011.