

A Discrete Event Simulation Model for the Analysis of Software Quality Attributes

Verónica Bogado, Silvio Gonnet, Horacio Leone

INGAR, Instituto de Desarrollo y Diseño (UTN-CONICET),
Santa Fe, Argentina, 3000
{vbogado, sgonnet, hleone} @santafe-conicet.gov.ar

Abstract

A discrete event simulation model for evaluating quality attributes, employing the software architecture, is proposed in this work. A metamodel of the software architecture domain that includes the concepts required for measuring quality attributes at runtime is specified. So, a simulation model is built from it, following the principles of hierarchy and modularity, assembling simple blocks to obtain complex blocks. DEVS framework is applied to obtain a decoupled model from the simulator, and the DEVS formalism is used to specify the elements of the simulation model.

The objective of this approach is to provide information about the quality attributes that can be measured at runtime, introducing the discrete event simulation in the context of the software architecture design. This quantitative information will assist the architect to make decisions about the design of the system.

Keywords: Quality Attribute, Software Architecture, DEVS, Simulation Model.

1 INTRODUCTION

Today, in the software industry is known that architectural design is very important to build a final product that satisfies user requirements. Software architecture (SA) provides the basis for analyzing information related to the quality attributes of the product. The evaluation of software architectures can be done at any stage of the software development, but it has major impact at early stages of the process. It provides information to understand the system and its characteristics, detecting early errors to make decisions to improve the system quality (an architecture that better responds to the user requirements). Nevertheless, the architect has to acquire knowledge of a variety of techniques that are different in use and application, so this makes difficult to the companies to find human resources with the required “know-how”. In this context, it is necessary to develop methods and tools that give support to the software architects at the design stage.

Several works present different approaches to evaluate software architectures. Some of them are focused on the architecture itself depending on the subjectivity of the stakeholders and experts; an example of this is the ad-hoc assessment proposed in [1]. Other approaches are focused on quality attributes, using quantitative or qualitative analysis based on Markov Decision Process (MDP, [2], [3]), Petri Net (PN, [4]), or Queueing networks (QN, [5]). Furthermore, there are techniques that are based on scenarios (for example, Use Case Maps- UCM, [6]) in combination with formalisms such as Markov Process or Queueing Theory to evaluate quality attributes [7]. On the other hand, experimental approaches have an increasing relevance in the software architecture area because they give some “view” of the execution during the analysis to the stakeholders. Examples of this type of techniques are prototyping and simulation [8].

All techniques are important to the area, but today systems are more complex and changeable. So, there is a need for tools that include not only the software architecture but several quality attributes and functional requirements in the analysis, being adaptable enough to incorporate future changes for validating new scenarios. Simulation has shown to be a powerful instrument for studying the states the system could pass, obtaining values for evaluating different scenarios. This allows studying the impact, on the system behavior, when a particular variable is changed at the design stage. In the last years, discrete event simulation has an increasing relevance in this area and many of

its techniques have been successfully applied in other design domains due to its capability to model dynamic complex systems.

In this way, we present a novel approach based on discrete event theory in order to simulate software products using the software architecture. So, the behavior of the system can be a subject of early study, measuring quality attributes and validating scenarios specified in the software requirements. In this article, we focus on the simulation model for analyzing quality attributes that can be measured at runtime, including functional aspects in the software architecture. A framework for modeling and simulation is applied with the purpose of keeping the model separated from the complexity of the simulator. Discrete Event System Specification (DEVS, [9]) formalism is used to specify the different elements in the simulation model. DEVS represents elements in a modular and hierarchical way and provides a powerful level of abstraction, expression and organization. These characteristics are suitable to represent concepts of software architecture and their relationships, providing scalability to the simulation model and a formal support without extra costs of implementing components at early stage of the development.

The rest of the paper is organized as follows: Section 2 describes a conceptual model that captures information for evaluating software architectures, considering functional and quantitative aspects. Section 3 presents a framework for modeling and simulation, and DEVS fundamentals. Section 4 explains the simulation model, DEVS models for concepts of the software architecture. Section 5 describes the implementation of the hierarchy of DEVS models (proposed simulation model) for the software architecture domain using DEVJSJAVA. Section 6 explains the process to apply the proposal. Section 7 shows an example to describe how the proposed model can be applied to a specific case. Section 8 discusses the use of DEVS formalism in the software architecture domain. Finally, Section 9 presents conclusions and future work.

2 A CONCEPTUAL MODEL FOR THE ANALYSIS OF QUALITY ATTRIBUTES

Software architecture design is a complex activity, which requires the knowledge of different aspects such as views, elements, relationships between elements, quality attributes, metrics, scenarios, and techniques like UCM ([10], [11], [12], [3]). The last one complements an architectural model with functional features [6], under the concept of responsibility and the causal relationship [13].

Therefore, the required concepts are captured from the general and common language used to design architectures and to validate them according to the quality attributes values. These concepts and their relationships are represented in a conceptual model to evaluate the software architecture at runtime (*SAEM*). This model (Fig. 1) captures information about the architectural design, metrics that can be used to analyze indicators of the quality of the system, and functional aspects that include the behavior of the system.

Concepts taken from the dynamic view (*SAView* concept in Fig. 1) of the software architecture and their relationships are the fundamental structures. The *ArchitecturalElement* concept is an abstract entity, which represents structures that have runtime presence, and they are included in the dynamic view of the system. Two important concepts are specialized from it, *Component* and *ConnectionMechanism*, which together with the *Responsibility* concept conform the main structure. They represent essential elements to build software architectures, describing the system behavior too.

The *Component* concept is an abstract entity, which is a generalization of two types of components, where one represents simple structures (*SimpleComponent* concept in Fig. 1) and the other complex structures (*CompositeComponent* concept in Fig. 1) respectively. The *SimpleComponent* corresponds to a software entity that could have some runtime presence such as a process, an object, etc., and it is in charge of a set of responsibilities. A more complex structure is the *CompositeComponent*, which depends on a set of components for carrying out the assigned responsibilities. It can be composed by both simple components and composite components, delegating the responsibilities to them, because responsibilities are only assigned to *SimpleComponent*.

A *Responsibility* is a statement about software objects. It could be an action that an object performs, knowledge that an object maintains, or a major decision that an object makes that affect others [10]. The relationship between responsibilities is the kind of cause-effect (*Causes* relationship in Fig. 1), where the fulfillment of the one or more responsibilities implicates the execution of others, which are activated to be performed.

The forms of interaction between software elements, such as simple components, are the connection mechanisms, which are captured in the *ConnectionMechanism* concept. If connection mechanisms are complex connectors, they can have assigned responsibilities too.

Finally, for a quantitative evaluation of quality attributes, other concepts such as quality attribute values and measures (metrics) are associated with the responsibilities, since they are the smallest units at runtime. The *QualityAttributeValue* concept depends on the measures that are applied to evaluate quality scenarios. The *Measure* concept maintains information about the needed values for the calculation of a quality indicator.

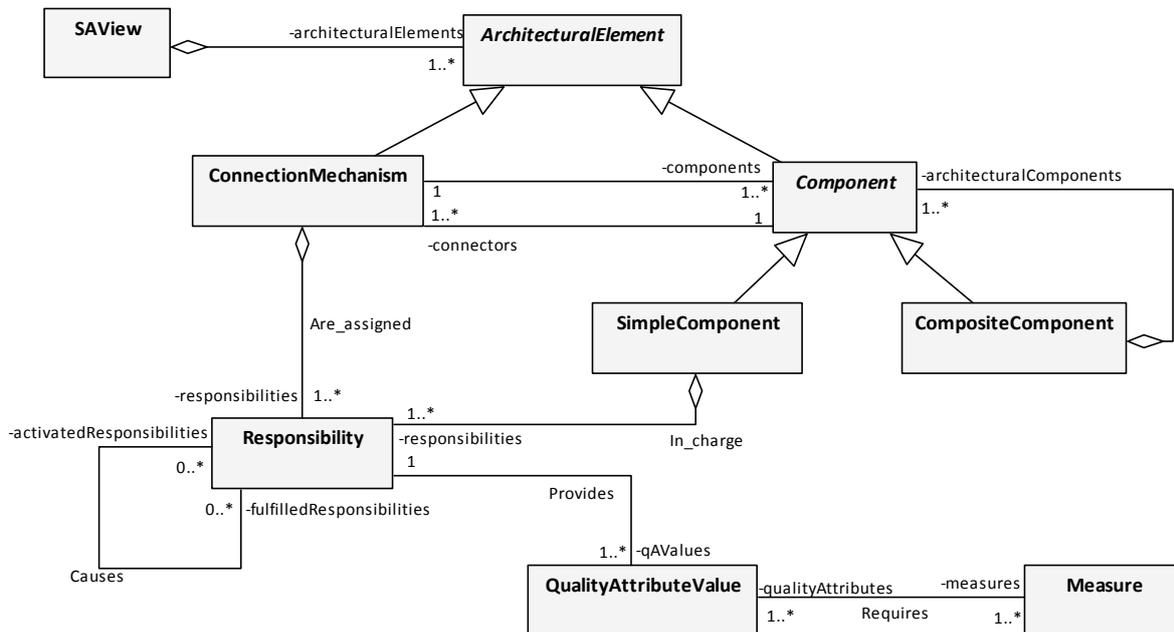


Figure 1: Conceptual Model for the Evaluation of Software Architectures (SAEM)

3 AN OVERVIEW OF DISCRETE EVENT MODELING AND SIMULATION

DEVS is a formalism for simulating discrete event system. It defines system behavior (input/output events and states, with the respective functions) and the system structure.

Zeigler in [9] has proposed a conceptual framework for modeling and simulation with the purpose of providing a support to the DEVS formalism. The framework underlying this formalism has three basic entities (Fig. 2):

- *Model*: is the system specification, defining the structure and behavior needed to generate data comparable to data from the real world. Generally, a simulation model is a set of instructions, rules, equations, constraints, for generating input/output behavior.
- *Simulator*: is a kind of agent (any computation system) that executes the instruction of the model, generating the behavior.
- *Experimental Frame*: is the specification of the conditions under which the system is observed, allowing the experimentation and validation of the model.

These entities are linked by two relationships. The first, *Modeling relationship*, links real system and model, it is used to represent the system (problem) and to validate the model with the real world. The other relationship, *Simulation relationship*, links model and simulator, it is employed to assure the simulator correctness; it guarantees that the simulator generates the output trajectory given an initial state and an input trajectory.

Keeping separate these three entities gives some benefits such as the fact of that the same model can be executed by different simulators, or that several experiments can be interchanged to study different situations. This flexibility has the cost of complexity, but in many cases is a future investment.

As shown in Fig. 2, this contribution is focused on the model entity of the simulation approach. The simulation model for the evaluation of quality attributes is based on the specification of simple primitive DEVS models (atomic DEVS) and with them complex DEVS models (coupled DEVS). This means that complex DEVS models are constructed of simpler ones. In this way, how models are connected is defined, building a hierarchy of DEVS for the simulation model. DEVS models employed in this work are specified in the following subsections.

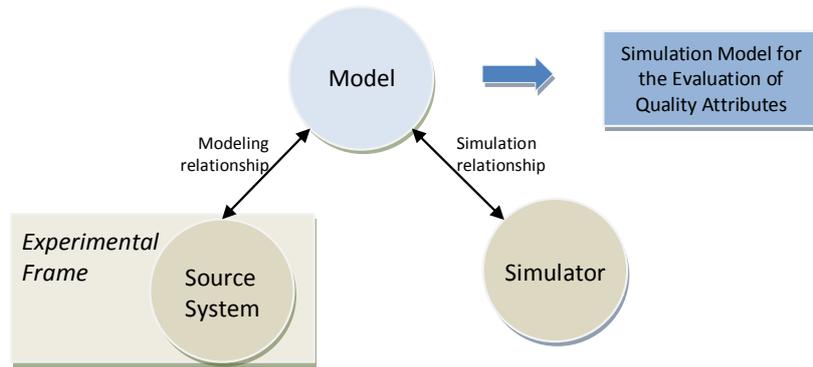


Figure 2: M&S Framework. Simulation model for the analysis of quality attributes, based on [9]

3.1 Atomic DEVS Model with Ports

This model allows multiple ports to receive values at the same time [9]. The specification is as follows:

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

$X = \{(p, v) | p \in InPorts, v \in X_p\}$ Set of inputs. It is a set of pairs consisting of an input port and a value. *InPorts* is the set of possible input ports and X_p is the set of possible values for p input port.

$Y = \{(p, v) | p \in OutPorts, v \in Y_p\}$ Set of outputs. It is a set of pairs consisting of an output port and a value. *OutPorts* is the set of possible output ports and Y_p is the set of possible values for p output port

S Set of sequential states

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

$\delta_{int}: S \rightarrow S$ Internal state transition function

$\lambda: S \rightarrow Y$ Output function

$ta: S \rightarrow R_{0,\infty}^+$ Time advance function

3.2 Coupled DEVS Model

Coupled DEVS model is built with other DEVS models, which become components of it. To specify coupled models requires: external interfaces (input/output ports and values), components (names/references to the component models and their specification with DEVS), and coupling relationships (external and internal couplings). So, it is specified in the following form [9].

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

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ Set of inputs. It is a set of pairs consisting of an input port and a value. *IPorts* is the set of possible input ports and X_p is the set of possible values for p input port.

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ Set of outputs. It is a set of pairs consisting of an output port and a value. *OPorts* is the set of possible output ports and Y_p is the set of possible values for p output port

D Set of component names

M_d Component models, for each $d: M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \lambda, ta)$

$EIC \subseteq \{(N, ip_N), (d, ip_d) | ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$ External Input Coupling

$EOC \subseteq \{(d, op_d), (N, op_N) | op_N \in OPorts, d \in D, op_d \in OPorts_d\}$ External Output Coupling

$IC \subseteq \{(a, op_a), (b, ip_b) | a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$ Internal Coupling

Select: tie-breaking function (used in classic DEVS but eliminated in parallel DEVS)

4 DISCRETE EVENT SIMULATION MODEL FOR THE ANALYSIS OF QUALITY ATTRIBUTES

This work is focused on the specification of elements that represent basic structures of the software architecture domain in the discrete event simulation model. The relationships between the software architecture concepts and the simulation elements are defined to translate the conceptual model to evaluate software architecture (Fig. 1) into a hierarchical simulation model (Table 1).

The simulation model is built considering the guidelines suggested by Zeigler [9] and other authors [15]. The specification is composed of DEVS models, which are structured in a modular and hierarchical way. The elements or building blocks are defined following some characteristics that each block (element of the simulation model) has to fulfill [15]:

- *Self-contained*: refers to the use of local information and local processes.
- *Interoperable*: determines that a building block may cooperate with other building blocks. All the system elements form the system that will be simulated.
- *Reusable*: means that the simulation building blocks could be instantiated more than once in the same model or in different simulation models for several studies.
- *Replaceable*: indicates that a building block in a simulation model may be removed from it and another building block may take its place.
- *Encapsulated*: keeps secret mechanism inside, encapsulating the internal structure of a building block in the simulation model. The purpose is to hide the complexity of the mechanism from the user.

Table 1: Correspondence between conceptual elements (software architecture) and simulation elements (DEVS)

Conceptual Model Elements	Simulation Model Elements
Responsibility	Atomic DEVS with ports
SimpleComponent	Coupled DEVS
ConnectionMechanism	Coupled DEVS
Relation: Causes (between responsibilities)	I/O Ports, Couplings between ports
Relation: In_charge (SimpleComponent-Responsibility)	Couplings between ports
Relation: Are_assigned (ConnectionMechanism-Responsibility)	Couplings between ports
SAView	Coupled DEVS

Therefore, the main concepts of the architectural model, represented by the conceptual model in Fig. 1, are transformed into elements of the simulation model. Consequently, a hierarchy of DEVS models is obtained. An atomic DEVS model is defined for the *Responsibility* concept (Table 1), and three coupled models are specified, one for the *SimpleComponent* concept, other for the *ConnectionMechanism* concept and another one for the *SAView* concept (Table 1).

4.1 DEVS Model for Responsibility Concept (RM)

The *Responsibility* concept is the smallest (primitive) unit of the dynamic elements in the proposed architectural model. Responsibilities are linked among them by a cause-effect relationship (*Causes*). The *fulfilled responsibilities* active other responsibilities (*activated responsibilities*).

This element (Fig. 3 (a)), from an architectural model, is translated into an atomic DEVS with ports, in the simulation model, called *Responsibility Model (RM)* (Fig. 3(b)). Specifying a DEVS with ports provides adaptability to the model, allowing an easier model evolution. Thus, when some other aspects need to be considered, inputs ports could be easily introduced. In the same way, when extra information from *RM* is required to be analyzed, output ports could be added easily. These changes do not impact in the whole DEVS for the *Responsibility* concept.

Each *RM* calculates its output values, related to its own states or to the measurement of quality attributes. Currently, the parameters included in the simulation elements would allow measuring performance aspects. Each element calculates the execution time to compute the turnaround time of the system when a stimulus arrives.

The relationships between responsibilities (*RM*) in the simulation model are given by the input/output ports, with their respective constraints to keep the cause-effect relationships.

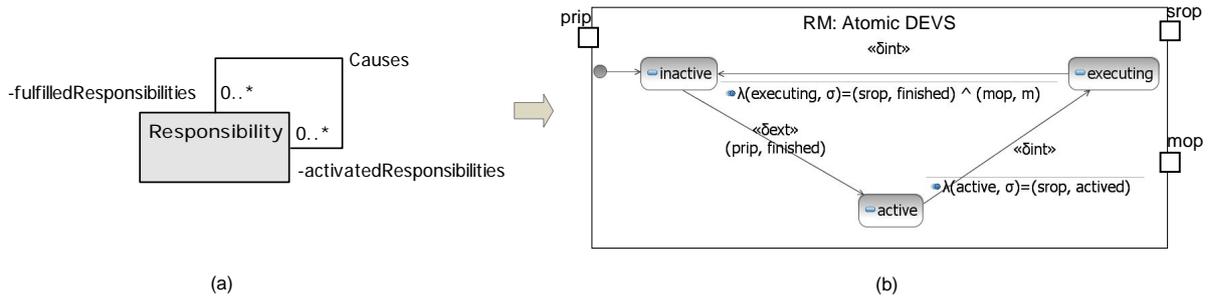


Figure 3: (a) *Responsibility* concept and the causal relationship, from architectural model. (b) *RM* (atomic DEVS with ports) with the transition diagram inside for representing the dynamic

To sum up, formally, the *Responsibility* concept is specified (DEVS) defining the set of inputs, outputs and states, the external and internal transition functions, the output function and the time advance function.

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

4.1.1 Set of Inputs (X_{RM})

The set of input values represents information about the fulfilled responsibilities and indicates the level in which each predecessor responsibility is completed. The only needed information for activating a responsibility initially is to know if previous responsibilities have finished their executions.

This set of inputs could be increased by incorporating more details to the model, with the purpose of evaluating other quality aspects of the system. In consequence, maintaining a model with ports allows that these changes can be easily added.

$$X_{RM} = \{(prip, finished)\}$$

where $prip \in RIP$, $finished \in X_{RM,prip}$

Set of input ports:

$RIP = \{prip\}$ where $prip$ is the input port (*activatedResponsibilities*, Fig. 3(a)) that is connected to the output ports of other responsibilities (*fulfilledResponsibilities*, Fig. 3(a)).

Set of input values for $prip$:

$$X_{RM,prip} = \{finished\}$$

This set might incorporate other values when other aspects need to be considered.

4.1.2 The Set of States (S_{RM})

A responsibility reflects a point where the system makes a change in its state, because it is interrogated or affected by other architectural element. So the states allow the designer to analyze if the system is active (*active*), in execution (*executing*) or passive (*inactive*) mode in this point.

For this model, a state is given by a phase and a sigma value. So, the set of states is defined as follows.

$$S_{RM} = \{inactive, active, executing\} \times R_0^+$$

where the possible phases are (Fig. 3 (b)):

- *inactive*: passive state, waiting for an external event. The system stays in this phase until an event occurs and interrupts the system condition.
- *active*: transitory state. This phase starts an internal transition that generates a needed output for the system evaluation. This stage indicates the execution of a responsibility has been started. The duration of this phase is null and it cannot be interrupted by external events.

- *executing*: this stage indicates that the responsibility is being performed, where execution means the processing of code in software domain.

σ : resting time in the state.

The quantitative aspects, represented in the conceptual model and related to the quality attributes, are modeled as fixed parameters of the atomic model or as calculated metrics in the simulation. They are measured during the execution. Thus, adding parameters to *RM* will allow the evaluation of quality attributes in different scenarios based on the execution of the system.

At present, the model only includes one parameter that provides information for analyzing dynamics aspects:

- *execution_time*, fix parameter of the model, indicates the time that an architectural element needs to carry out a responsibility; this means time that a responsibility uses to be performed. It sums up the metrics associated to the *Measure* concept in the conceptual model (Fig. 1).

4.1.3 Set of Outputs (Y_{RM})

A responsibility is in charge of emitting two types of output values. One related to its state, data of interest for the successor responsibilities (activated responsibilities as consequence of the execution of others), and other related to the values used for measuring aspects about quality attributes.

$$Y_{RM} = \{(srop, activated), (srop, finished), (mop, m)\}$$

where $m \in Y_{RM, mop}$, $srop \in ROP$, $mop \in ROP$

Set of output ports:

$ROP = \{srop, mop\}$ where *srop* is the output port for the events generated with the purpose of notifying other responsibilities and *mop* is the measure output port for evaluating quality attributes (in this work only was considered a performance indicator).

Set of values for *srop*:

$$Y_{RM, srop} = \{activated, finished\}$$

The possible values are:

- *activated*: this value is emitted when the execution of the responsibility has been started. The responsibility is ready for the execution.
- *finished*: this value is generated when the responsibility has been carried out, after its execution has been finished.

Set of values for *mop*:

The set of values for this port indicates the needed execution time to perform the responsibility.

$$Y_{RM, mop} = R_0^+$$

4.1.4 Internal Transition Function ($\delta_{RM, int}$)

The internal transition function (Fig. 3 (b)) defines the next state for the responsibility, as result of the elapsed time without an external event has taken place.

The *active* transitory state indicates that the responsibility can be carried out, because their predecessors (fulfilled responsibilities) have been finished in a suitable form. It communicates its activated state to the corresponding simulation elements; it has been started. So that, an internal transition is required that allows emitting an event using the correct port and changes automatically its state to *executing*.

$$\delta_{RM, int}(active, \sigma) = (executing, execution_time)$$

The other internal transition happens when the execution time has been elapsed, returning to the passive state (*inactive*), in standby, waiting for other external event.

$$\delta_{RM, int}(executing, \sigma) = (inactive, \infty)$$

4.1.5 External Transition Function ($\delta_{RM, ext}$)

This function makes a state transition when an external event has happened (Fig. 3 (b)). In other words, this change occurs when *finished* value is received from the fulfilled responsibilities. Therefore, *x* is equals to *finished*.

$$\delta_{RM, ext}(phase, \sigma, e, x) \begin{cases} (active, 0) & \text{if phase=inactive} \\ (phase, \sigma - e) & \text{if phase= active or executing} \end{cases}$$

4.1.6 Output Function (λ_{RM})

The output function generates output values and then makes the internal transition of states (Fig. 3 (b)).

$$\lambda_{RM}(active, \sigma) = (srop, actived)$$

$$\lambda_{RM}(executing, \sigma) = (srop, finished) \wedge (mop, m) \text{ where } m \text{ corresponds to the execution time.}$$

4.1.7 Time Advance Function (ta_{RM})

This function defines the time that the responsibility has been in a state. For the proposed model: $ta(s) = \sigma$

$$ta_{RM}(active, \sigma) = 0 \text{ where } s=(active, \sigma) \text{ is a transitory state.}$$

$$ta_{RM}(executing, \sigma) = execution_time \text{ where the } execution_time \text{ is the specified parameter for this model initially.}$$

$$ta_{RM}(inactive, \sigma) = \infty \text{ where } s=(inactive, \sigma) \text{ is a passive state.}$$

4.2 DEVS Model for the Simple Component (SC)

The simple component in the architectural model is in charge of a set of responsibilities (Fig. 4 (a)). So, this structure can be mapped to a coupled DEVS model (Fig. 4 (b)). The relationship between the simple component and their responsibilities can be represented as a hierarchy of DEVS models, structured in a modular way.

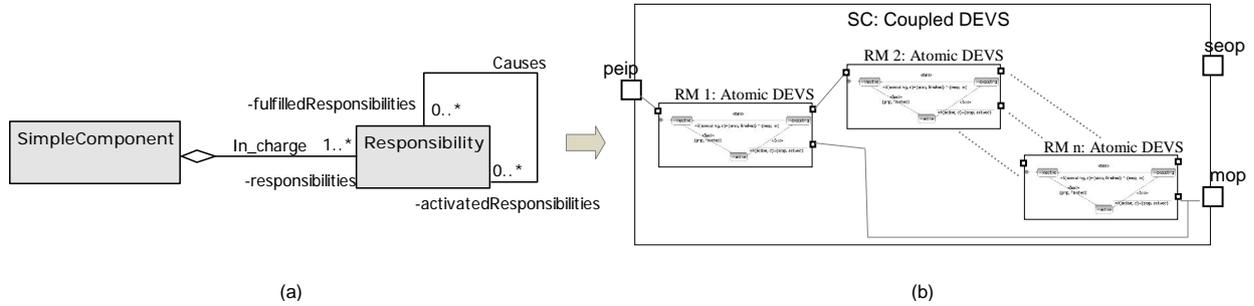


Figure 4: *SimpleComponent* concept and the relationship with the *Responsibility* concept. The simple component has a set of responsibilities in charge (*In_charge* relationship)

RM, which is specified for the *Responsibility* concept, could be instantiated as components of *SC*. The coupled model for the *SimpleComponent* concept is specified in the following form.

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

4.2.1 Set of Inputs (X_{SC})

SC has a set of input ports, where the input values are propagated to the components of this model. In this way, the input ports of this DEVS are connected to the input ports of the corresponding components, instances of *RM*. To the input ports, values that have been emitted from the previous components arrive, obtaining information to activate the suitable modules.

$$X_{SC} = \{(peip, finished)\}$$

where $peip \in SCIP$

Set of input ports:

$SCIP = \{peip\}$ where *peip* is the input port that receives information about the previous architectural elements, and it is connected to the input ports of the corresponding components.

Set of input values:

The set of input values represent information about the predecessor elements. This information is related to the execution end of the previous components, which is needed to activate the execution of the corresponding components.

$$X_{SC,peip} = \{finished\}$$

4.2.2 Set of Outputs (Y_{SC})

The output ports propagate the events generated by the components of this model to other simulation elements that represent architectural elements from the conceptual model.

$$Y_{SC} = \{(seop, activated), (seop, finished), (mop, m)\}$$

where $m \in Y_{SC,mop}$, $seop \in SCOP$, $mop \in SCOP$

Set of output ports:

$SCOP = \{seop, mop\}$ where $seop$ is the output port that emits events to the successors elements, and mop is the output port that returns a measure value, associated to indicators of quality attributes.

Set of output values for $seop$:

$$Y_{SC,seop} = \{activated, finished\}$$

where:

- *activated*: indicates that the execution of the component has been started. The corresponding responsibilities are activated inside of this component.
- *finished*: indicates that the execution of the component has been ended. No responsibility is active.

Set of output values for mop :

The output values are represented by the set of real numbers. It indicates a value of a needed measure to calculate a quality index of the software.

$$Y_{SC,mop} = R_0^+$$

4.2.3 Set of Components (D_{SC})

This set details the references to the components of the coupled model. In the present work, D_{SC} is the set of references to RM instances that are part of the SC .

4.2.4 Component Models ($M_{SC,d}$)

$$M_{SC,d} = RM \quad \forall d \in D_{SC}$$

where d is the reference name of the RM model (names of the corresponding responsibility in the architectural model).

4.2.5 External Input Coupling (EIC_{SC})

This coupling connects the inputs ports (external inputs) from the coupled model to the input ports of the corresponding components.

$$EIC_{SC} \subseteq \{(SC, peip_{SC}), (r, prip_r) \mid r \in D_{SC}, peip_{SC} \in SCIP, prip_r \in RIP_r\}$$

4.2.6 External Output Coupling (EOC_{SC})

This coupling connects the output ports from the corresponding components to the output ports of the coupled model (external outputs).

$$EOC_{SC} \subseteq \{((r, srop_r), (SC, seop_{SC})), ((r, mop_r), (SC, mop_{SC})) \mid r \in D_{SC}, seop_{SC}, mop_{SC} \in SCOP, srop_r, mop_r \in ROP_r\}$$

4.2.7 Internal Coupling (IC_{SC})

The internal coupling connects output ports from one component to input ports of another component. This must obey the causal relationship between responsibilities.

$$IC_{SC} \subseteq \{((r_1, srop_{r_1}), (r_2, prip_{r_2})) \mid r_1, r_2 \in D_{SC}, srop_{r_1} \in ROP_{r_1}, prip_{r_2} \in RIP_{r_2}\}$$

where $r_1 \Rightarrow r_2$ is the constraint for responsibility relationships.

In this model no direct feedback loops are allowed, following the DEVS clause.

$$((r_1, srop_{r_1}), (r_2, prip_{r_2})) \in IC_{SC} \Rightarrow r_1 \neq r_2 \quad \forall r_1, r_2 \in D_{SC}$$

4.3 DEVS Model for the Connection Mechanism (CM)

The connection mechanism (*ConnectionMechanism* in Fig. 1) represents a connector between two software components and, like the simple component in the architectural model, might be in charge of a set of responsibilities (Fig. 5 (a)).

The difference from the simple component is the function that it accomplishes into the software architecture. Both elements are similar in structure, but the difference lies in the functionality of each one. In the literature, there is a discussion about the importance of modeling the connectors or if using the same concept of simple component for the connections. However, in this simulation model it was introduced as other DEVS model with the purpose of decoupling elements, obtaining a more flexible model to make future changes.

The specification of the *ConnectionMechanism* concept (DEVS) follows the same principles written for the simple component, obtaining the *CM* coupled model (Fig. 5 (b)). So, the relationships between the connection mechanism and their responsibilities can be specified as a relationship of hierarchy, where the instances of *RM* that represent the responsibilities are components of *CM*.

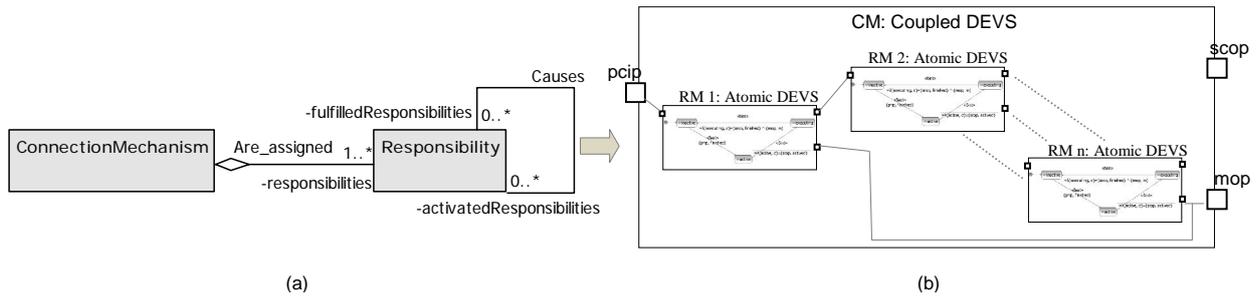


Figure 5: *ConnectionMechanism* concept and the relationship with the *Responsibility* concept. The connection mechanism has a set of responsibilities in charge (*Are_assigned* relationship)

As part of the interface, *CM* has a set of input ports, which propagate the values to the internal components. It has a port called *pcip* that receives information about the previous components (*SC* instances), and it is connected to the input ports of the corresponding components (*RM* instances). The value (*finished*) indicates that the previous components have been finished their execution, activating the corresponding connector. The output ports propagate events generated by the components of this model to other simulation elements that represent software components. *CM* has two output ports: *scop* that emits events to the successor components, and *mop* that returns a measure. Two possible values can be emitted by the first port: *activated* (the execution of the connector has been started) and *finished* (the execution of the connector has been ended, no responsibility is active).

CM has a set of elements; each element is an instance of *RM*. These elements are connected using the ports, defining the internal couplings (IC_{CM}), which obey the causal relationship between responsibilities. Furthermore, some of them are connected to the input ports of *CM* (EIC_{CM}), and others to the output ports of it (EOC_{CM}).

The coupled DEVS for the *ConnectionMechanism* concept is formally specified as follows:

$$CM = \{X_{CM}, Y_{CM}, D_{CM}, \{M_{CM,d} \mid d \in D_{CM}\}, EIC_{CM}, EOC_{CM}, IC_{CM}\}$$

$$X_{CM} = \{(pcip, finished)\}$$

$$Y_{CM} = \{(scop, activated), (scop, finished), (mop, m)\}$$

D_{CM} is the set of references to instances of *RM*

$M_{CM,d} = RM \quad \forall d \in D_{CM}$ where d is the reference name of *RM* instances; the specification of the components correspond with *RM*.

$$EIC_{CM} \subseteq \{((CM, pcip_{CM}), (r, prip_r)) \mid r \in D_{CM}, pcip_{CM} \in CMIP, prip_r \in RIP_r\}$$

$$EOC_{CM} \subseteq \{((r, srop_r), (CM, scop_{CM})), ((r, mop_r), (CM, mop_{CM})) \mid r \in D_{CM}; scop_{CM}, mop_{CM} \in CMOP; srop_r, mop_r \in ROP_r\}$$

$$IC_{CM} \subseteq \{((r_1, srop_{r_1}), (r_2, prip_{r_2})) \mid r_1, r_2 \in D_{CM}, srop_{r_1} \in ROP_{r_1}, prip_{r_2} \in RIP_{r_2}\}$$

4.4 DEVS Model for the Software Architecture View (SAVSM)

The architectural view is the highest level and it has architectural elements, components and connectors. It is translated into a coupled DEVS and it is defined in a similar form as the previous coupled DEVS. This entity represents the simulation model for the software architecture view (SAVSM). This defined DEVS can have instances of *SC* and *CM* (both coupled DEVS), which contain instances of *RM* (atomic DEVS). The components of the simulation model are related through the couplings.

5 IMPLEMENTATION OF THE DEVS HIERARCHY

The proposed specification hierarchy is implemented using DEVJSJAVA [16], which is a set of libraries that provides the needed tools for implementing DEVS models employing the JAVA programming language.

The main package is *Zdevs* that contains the *Devs* class (Fig. 6), which is the base of the hierarchy of models. This is the superclass of *Atomic* class and *Coupled* class, from the last *Digraph* class is inherited (Fig. 6).

Within the scope of this work, the simplest building blocks are the responsibilities, being atomic DEVS in the simulation model. Therefore, it is implemented as subclass of *Atomic* class (*ResponsibilityM* in Fig. 6). In this case, the internal transition function, the external transition function and the output function are rewritten according to the specification of the represented architectural concept, defining the corresponding parameters.

On the other hand, the complex structures are the components and the connection mechanisms, which are represented with coupled DEVS. Therefore, they are implemented as subclasses of *Digraph* class, called *SimpleComponentM* and *ConnectionMechanismM* respectively (Fig. 6). These complex structures are composed by responsibilities; this means that they contain atomic instances connected through the corresponding ports. The view (*SAViewM*) is implemented using *Digraph* class too, containing other coupled DEVS (*SC*, *CM* or both with the corresponding couplings).

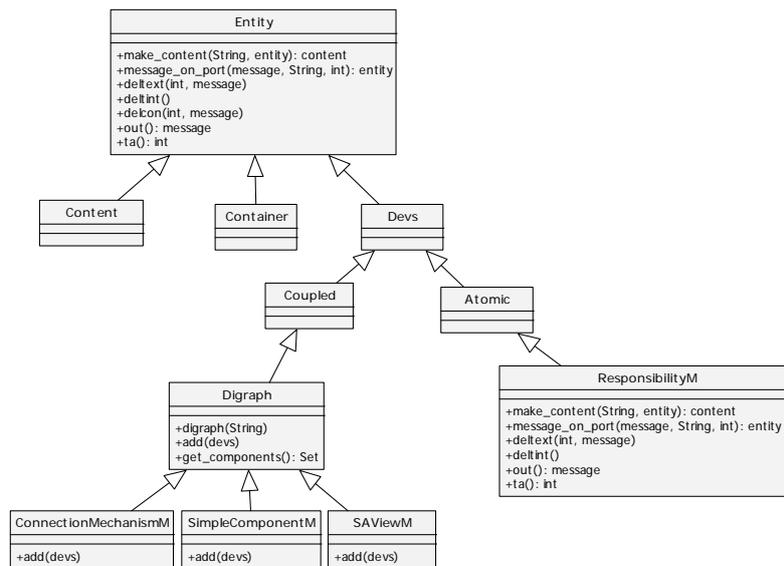


Figure 6: Implementation using DEVJSJAVA: Class hierarchy for the architectural concepts

6 DEVS- BASED MODEL AND ITS APPLICATION

The application of the proposed approach needs some initial information of the software architecture, which is specified using UCM notation by the first task of the construction process of the simulation model (Fig. 7). This model formulation requires the identification of elements of the dynamic view, their interactions with other elements and their responsibilities.

Despite the fact that this approach is mainly useful at early stage of the software development it can be applied at any stage. Initially, the view of the software architecture can be obtained from user requirements. Non-functional requirements (*NFR*) are employed to identify elements of the architecture and functional requirements (*FR*) are used to extract responsibilities, which are assigned to the elements. If the system has been implemented and there is

a need for some modifications, the architecture can be extracted employing some tool such as *SWAG Kit*¹, or “manually” (ad hoc), reconstructing the architecture elements and identifying the responsibilities too (functionalities). The non-functional requirements (*NFR*), in particular the quality requirements, are employed to define the quality scenarios (Fig. 7).

Software architecture elements and their responsibilities are represented using UCM notation [13], in order to validate scenarios. These concepts were captured in the proposed conceptual model (*SAEM*), as can be seen in Fig. 7. The process of transformation has to be done following some rules. In this way, the elements of this model are translated into elements of the proposed simulation model (*SAVSM*) using the relationships presented in Table 1.

The simulation model specified with DEVS formalism and implemented using DEVSJAVA library is tested in a development environment for simulation, last step at the top-right of Fig. 7. *DEVS-Suite* (2.0)² is employed to analyze the proposed model in structure and the dynamic of individual elements.

The evaluation of quality attributes is automatic and it is the main part to be validated before the automation of the transformation (last part in Fig.7, which provides information of the simulation execution to make design *decisions*). The architect should provide the following input data and parameters:

- *Stimulus frequency*: events arrivals have a behavior pattern. For a better simulation the architect has to define the flow of external events, if it responds to some probabilistic distribution or it is a periodic flow. This item represents the operation mode of the system (normal or overload according to the goals).
- *Execution time* (behavior): the time that a responsibility uses to process a request is given following some probabilistic distribution, being by default uniform. This can be set by the architect to obtain better results after the simulation, according to data behavior.
- *Simulation time*: conditional value defined by the architect, which depends on the size and complexity of the system that is being simulated.

This input data may be initially estimated or obtained from previous projects or iteration, in case of iterative methodology of software development. A standard form (default values or probabilistic distributions) can be used in the first experiment to appreciate the general behavior, but it can be improved during the progress of the project, updating with new data.

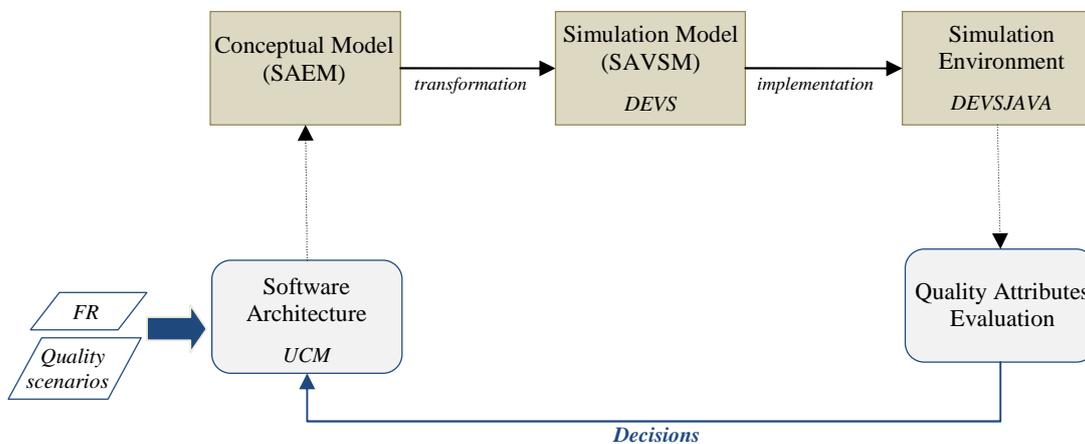


Figure 7: Quality attributes evaluation frame using DEVS approach

7 CASE STUDY: A CLASSICAL ARCHITECTURAL PATTERN

To illustrate the proposed model, a classical architectural pattern is described. The *Pipe* and *Filter* pattern is used to solve problems related to the processing of data streams, where the decomposition in subtasks are required to obtain a better result. The nature of some kinds of systems requires an implementation based on components,

¹ SWAG: Software Architecture Group. <http://www.swag.uwaterloo.ca/swagkit/>

² DEVS-Suite. <http://www.acims.arizona.edu/SOFTWARE/software.shtml>

because the conversion of a data sequence is easier if the task is divided in smaller parts rather than work with an only component.

This pattern proposes two types of elements: components called *Filters* and connectors called *Pipes*. A pipe passes the information flow from a filter to another filter, while the filter converts this information to obtain a new flow adapted to the requirements of the system. A filter can be connected to any number of input pipes and to any number of output pipes (Fig. 8).

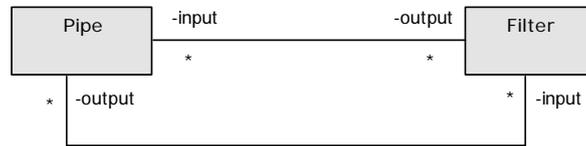


Figure 8: Conceptual model: Pipe and Filter pattern.

An example of the pipe and filter pattern in a concrete application can be seen in Fig. 9. We restrict our treatment of the case study to elements that are important to understand the concepts presented in this paper: view, component, and responsibility. The connection mechanism (*Pipe*), in this example, only has the responsibility of passing information from one filter to the next, so they are not included in the example, but if we need a more detailed evaluation they can be included in the model as instances of *CM*.

The license manager is a software management tool employed by a software company and its end-user organizations with the purpose of controlling when and how specific software products are able to be used.

The architectural view, as shown in Fig. 9, represents the client subsystem, which has three components. The first component (*Decryptor*) is in charge of decrypting the received file. Thus, an external license file arrives to be processed. The plain text file contains two main parts, license data and a digital firm. The license data include customer name (organization), license date, days counter, among others. The digital firm is encrypted, having inside a unique identifier (hash). So, this component decrypts the firm with the public key sending the file with this information to the following component. The second component (*Authenticator*) receives the file and the identifier (hash). Then it calculates, following an algorithm, the identifier (hash) using the content of the file, and finally compares these two identifiers. It sends the file and the result to the next component. The last one (*Recorder*) receives the authenticated information (file and result), and updates the license information in the DB if the result is right, otherwise it blocks the user system (error).

The architectural view, its elements and their responsibilities are represented using UCM (Fig. 9). Responsibilities are extracted from the specified functionalities representing executable units. Each component can have assigned one or more responsibilities as follows:

Decryptor:

r1: Receiving the license file

r2: Decrypting the digital firm with the public key (from client side)

Authenticator:

r3: Authenticating data comparing the corresponding information

r4: Sending the data with a report

Recorder:

r5: Receiving the authenticated data

r6: Saving the corresponding data in a DB (information about the correct or error data)

This model represents architectural elements of the system adding functional aspects to them, considering not only quality requirements but functional requirements. The responsibilities have causal relationships. Stimulus begins an execution flow obtaining a result as response.

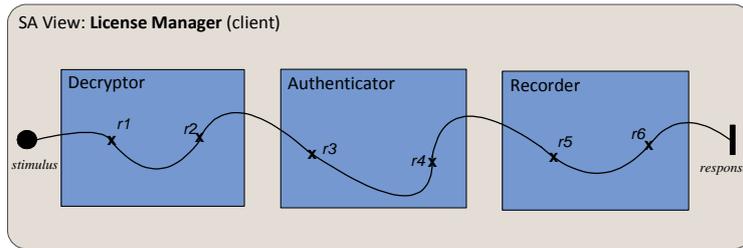


Figure 9: UCM for the Pipe and Filter pattern application

The software evaluation concepts are translated into simulation elements. In Fig. 10, software structures are translated into DEVS models. In this way, three types of elements are identified: view, simple component and responsibilities. The process of transformation starts from atomic to more complex structures (bottom-up viewpoint). First, the responsibilities are the smallest structures where *Responsibility* concept in the conceptual model (SAEM) is transformed into *RM* in the simulation model (SAVSM). So, each responsibility instance from the software architectural model is translated into an instance of *RM* in the simulation model, where the dynamic of it is defined by the functions described in the previous sections. Responsibilities are connected by ports, with the couplings for the given example, building more complex structures. In this case, one of these more complex structures is the *Filter*. So, each component that represents a *Filter* is translated into an instance of *SC* (Fig. 10), with the corresponding input and output ports, and assigned responsibilities (instances of *RM*). Finally, the view is translated into a coupled DEVS too. This simulation element has three instances of *SC*, where each one has two instances of *RM*. In this way, three levels of abstraction are represented in the simulation model obtained from the software architecture of the system that will be simulated.

The quantitative aspects are reflected in the parameters of the models. In this case, the *execution_time* variable is a metric that could provide information about the performance of the system during the simulation. This value is calculated during the simulation for each request processed by each responsibility.

DEVS-Suite (2.0) is a Parallel DEVS simulator with support for: automating design of experiments in combination with animating models and generating data trajectories at run-time. This environment allowed structural validation of the simulation elements and the dynamic of each element through manually tests. It provides tools for designing and implementing experiments. In Fig. 10, the architecture of the system is represented using the proposed approach. Thus, the specified DEVS hierarchy (*RM*, *SC*, *CM*, *SAVSM*) implemented using the classes provided by DEVJSJAVA library was integrated in this environment.

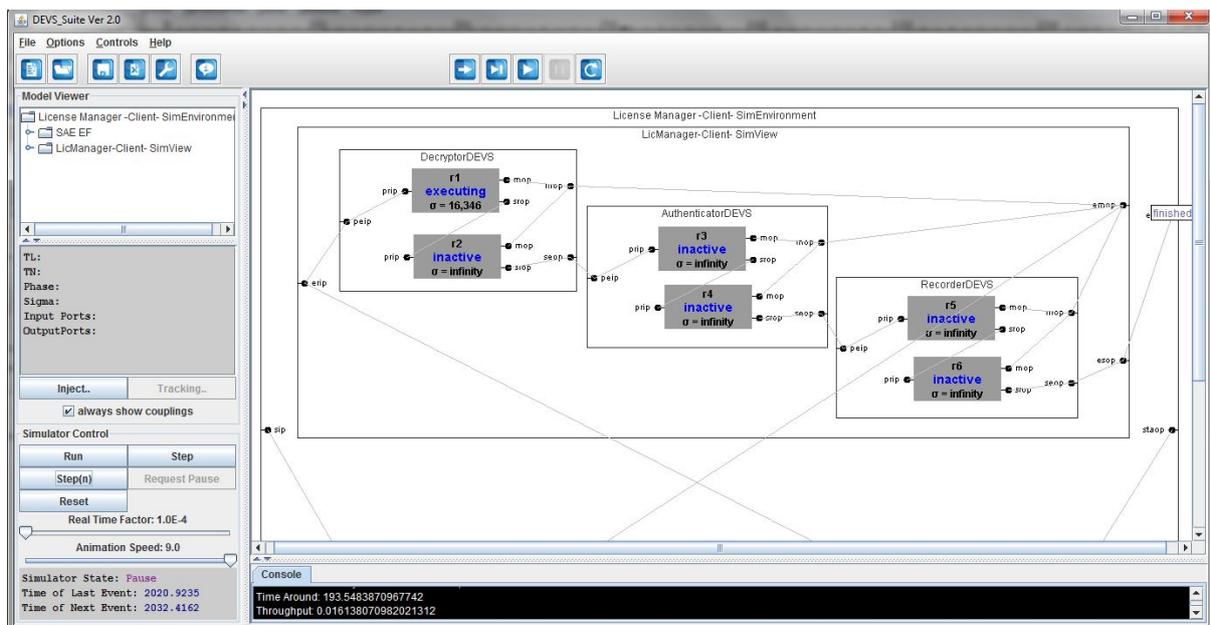


Figure 10: DEVS-Suite view: simulation elements for the architectural view of the example

The instances of the simulation elements can be seen in the *Simulation View* (at top-right of the picture, Fig. 10). The complete simulation model (defined hierarchy of DEVS models) can be seen in the *Model Viewer* (top-left, Fig. 10). Here we can see the composite elements and their components. The other parts of the window are controls and information for the simulation. For example, in the *Console* we can display the measured values for each simulation element, or system indicators of the quality attributes, such as the time around shown in Fig. 10.

Software architecture evaluation has different goals: detecting conflicts requirements, analyzing design decisions, analyzing the impact of the changes, among others.

For example, for the company may be important that the License system has a suitable response because, otherwise the clients might be blocked affecting their work. This process of validation must be efficient because some systems are critical for the company. In this way, a performance scenario is given as an example of the approach (it can be specified following the template proposed by SEI [17], as shown in Fig. 11, or in an informal form). In this scenario, time around is analyzed for a normal operation of the system.

<p>Source of stimulus: <i>Subsystem</i> (server component) Stimulus: <i>periodic requests</i> Artifact: <i>subsystem</i> (client component) Environment: <i>normal operation</i> Response: <i>authenticated license in the client subsystem</i> (processed stimuli) Response measure: <i>time around</i> within 4000 (time units=ms)</p>

Figure 11: Example of quality scenario. Concrete scenario for performance

This approach allows designers to study the system behavior, observing which components are activated and which responsibilities are being performed inside of them (*Simulation View* in DEVS-Suite), as shown in Fig. 10. The state of each element is indicated detecting which elements are executing and which are in a passive state. The simulation of the system execution allows evaluating quality scenarios related to the performance attribute (in this approximation of the model). For example, in the previous scenario we measure the time around in milliseconds, if the time used to process the requests is too high, it can indicate some problem. So the design has to be reviewed in detail, analyzing the time per component or responsibility and detecting which of them has some conflicts. This detection of elements provides information to take some design decisions, such as the application of architectural patterns, reallocation of responsibilities, or addition of new components that reallocate the work load.

The design and implementation of the experimental frame for evaluating software architectures will include an automatic experimentation in the simulation environment proposed in this work.

8 DISCUSSION

In this paper we presented a DEVS-based approach to evaluate quality attributes that can be measured at runtime based upon the software architecture of the system. Other approaches have been proposed using different techniques for analyzing different quality attributes. Some authors have classified them in theoretical and empirical techniques [8], where we can include three subclasses: one based on quality attributes, another one based on software architecture, and the last one based on scenarios.

Most approaches are in the first group being more theoretical. There are many methods based on scenarios and focused on a more qualitative analysis than in quantitative aspects [18]. Moreover, in the software architecture area, several works propose ad-hoc assessment of the software architecture qualities, but they introduce subjectivity and ambiguity in the assessment [1].

Other alternatives are mainly focused on quality attributes. For example, Markov Process has been used to evaluate the reliability, performance and security of a system ([2], [3]). Also, other contributions have employed Queueing Theory to measure performance [5]. Petri Nets are other option, which has been applied to evaluate different quality attributes such as security, performance and reliability [4]. Finally, the SEI (Software Engineering Institute) has made some contributions to deal with architecture evaluation (a software named *ArchE*), using Fixed Priority Scheduling to measure performance and Graph Theory to analyze modifiability ([19], [20], [21]).

The experience indicates that Markov Process seems to be a good approximation for evaluating software architectures; many works have shown how an architectural model can be translated into a Markov chain. Nevertheless, this approach presents two situations which have to be improved: the representation of the states that only include components from the architectural model, losing other important aspects of the architectural domain, and the analytical resolution of the models which requires specific information (such as transition probabilities, quantitative data for each component, etc) [22]. Queueing Theory provides a good performance assessment, but it

is difficult to represent other aspects of the software quality such as: security, reliability, etc. Petri Nets have limitations to model some situations such as priorities and complex system which imply problems to solve the net (known problem of Petri Nets). This problem requires simplification, but this mechanism implies to lose concepts that may be important to represent the real problem (system that is being simulated), affecting the final results.

The various approaches for the software quality evaluation presented previously provide a quantitative analysis focuses on specific quality attributes. They are based on mathematical fundamentals which are important for the consideration of final results. However, these techniques have the disadvantage of being too restrictive from the modeling point of view. They have some limitations to represent a number of situations about the real-world systems and they incorporate the complexities of the technique in the model, losing abstraction capabilities. Some critical studies have shown limitations of this kind of techniques (i.e. works focused on reliability prediction techniques such as [22]).

Many authors have exposed the enterprise needs and they have emphasized the importance of the functional aspects in the architectural evaluation, the increasing complexity of the systems and the performance as dominating quality attribute.

In this context, empirical techniques have increasing importance. Architectural prototyping has been proposed in this area to analyze: structural (components), communicational (connectors), and quality aspects (performance, security) [8]. However, the wasted effort to prototype the software and the initial costs have a major impact than other proposals. Furthermore, prototypes do not include functional aspects.

There are few contributions that include functionality in the architectural models and in the automatic evaluation. UCM provides a form to add behavior to abstract structures, but it is an informal notation [6]. However, formalisms such as Queueing Theory have been proposed to complement it in the analysis of performance [7], but lead to the same matters that have previously explained.

Simulation is another empirical technique. It allows exploring structural and behavioral aspects as well as quality indicators without implementation costs. In this way, we present a novel approach, from a conceptual model to a formalism that is more general than Petri Net and Queueing theory. DEVS is based on the system theory, so it allows clear abstractions and models and a modular construction, reducing the complexity of the systems using a hierarchical approximation. The building of simulation blocks, from basic models (atomic DEVS) to complex models (coupled DEVS), allows a suitable representation of the architectural structures, manipulating parameters for a quantitative analysis of the quality aspects. Furthermore, DEVS formalism is embedded in a simulation framework [9]. This frame defines three entities and their relationships, uncoupling the model from the simulator. Thus, the complexity of the simulator is kept outside the model, and the model is closer to the real system. It is more expressive, allowing the definition of domain-specific components and providing major semantic to the models (in this case, elements related to the software architecture and its quality).

Discrete event simulation applied to measure quality attributes using the software architecture of the system has the following items to be analyzed: i) it provides an overview of the performance of the system, ii) it allows a study of the behavior of the system and its elements (components and connectors with their responsibilities respectively), iii) its outputs include measures, such as throughput and time around, iv) it is a flexible approach, and it could be adapted to new situations, such as adding other quality aspects to be considered in the evaluation.

The last one is the most important feature because is the most difference from other techniques. DEVS is known for its capability to model complex systems without losing efficiency in the implementation of the simulation. The system theory fundamentals of DEVS are powerful tools to manage the complexities of the software architecture elements. It provides a high level of abstraction and scalability too, which are not found in other techniques. One limitation of this approach, also presented in others, might be the input data behavior employed by the model (probability distributions) for obtaining better results (metrics), because the lack of historical data is an important problem in the software industry.

9 CONCLUSION

The paper proposed an approach based on DEVS that introduces modular simulation into software architecture evaluation at early stage of the software development. Firstly, a conceptual model to evaluate software architectures has been formulated. The model represents information related to the system structure and behavior, and includes other concepts that enable quantitative analysis to validate several scenarios. Also, it incorporates the *Responsibility* concept, which provides a way to include functional aspects, representing the software execution flow, unlike other approaches that only focus on non-functional requirements (quality attributes).

We proposed the use of DEVS formalism to incorporate the advantages of discrete event simulation in the context of architectural design. This formalism builds simulation elements in a modular and hierarchical way. Furthermore,

a framework for modeling and simulation supports this formalism, keeping the model decoupled from the complexity of the simulator and from the experimental frame. These features provide flexibility, making easy to incorporate new elements in the simulation model and encapsulating the environment of the studied system under the concept of experimental frame.

We considered performance as initial quality attribute measurable at runtime, because it is a dominating quality attribute in many systems. Nevertheless, the proposal is more ambitious in order to incorporate other quality attributes indicators. For example, availability could require a new state (to simulate a software failure) in the atomic DEVS (RM) and the corresponding metrics (i.e. unavailability time). In this way, the simulation model specified using DEVS provides scalability with a formal support.

DEVS-Suite has facilitated a concrete simulator and a set of tools to implement the simulation model and its elements. This environment allowed the design of simple experiments which have been manually executed to test the proposed simulation model (structure and behavior using animation). We believe that the experimental framework of the simulation is a very important part for the results (quality metrics), so we are planning to do an exploratory study of its development, designing and implementing a specific experimental frame that responds suitably to different quality goals.

As regards the simulation model that we proposed, it could be of interest in future work to include complex architectural concepts such as architectural patterns and composite components besides required parameters for the quality metrics (atomic elements).

Finally, translating architectural concepts into simulation elements requires models transformation. MDA (Model-driven Architecture) framework will be applied, because of the importance of models in this process. The transformation rules that describe how one or more elements of the source model (architectural view) can be transformed into one or more elements in the target model (simulation model -DEVS), are currently defined in a simple and non-standardized form, and they have to be applied manually. In future works, automatic transformation will be integrated, formalizing the rules and studying standards that are widely used such as QVT (Query, Views, and Transformations) and MOF (Meta Object Facility) proposed by OMG.

Acknowledgements

The authors thank the financial support received from *Universidad Tecnológica Nacional (25/O118 – UTI1083)* and *Agencia Nacional de Promoción Científica y Tecnológica (PAE-PICT 02315)*.

References

- [1] M. Galster, A. Eberlein and M. Moussavi, “Early Assessment of Software Architecture Qualities”, in *Proc. Research Challenges in Information Science (RCIS 2008)*, 2008, pp. 81-86.
- [2] W.-L. Wang, D. Pan and M.-H. Chen, “Architecture-based software reliability modeling”, *Journal of Systems and Software*, vol. 79, Issue 1, pp. 132-146, January 2006.
- [3] V. Sharma and K. Trivedi, “Quantifying software performance, reliability and security: An architecture-based approach”, *Journal of Systems and Software*, vol. 80, Issue 4, pp. 493-509, April 2007.
- [4] K. Fukuzawa and M. Saeki, “Evaluating software architecture by coloured Petri Nets”, in *Proc. 14th International Conference on Software Engineering and Knowledge Engineering*, 2002, pp. 263- 270.
- [5] B. Spitznagel and D. Garlan, “Architecture-based performance analysis”, in *Proc. 1998 Conference on Software Engineering and Knowledge Engineering*, 1998, pp. 146-151.
- [6] D. Amyot and G. Mussbacher, “User Requirements Notation: The First Ten Years, The Next Ten Years”, *Journal of Software*, vol. 6, N° 5, pp. 747-768, May 2011.
- [7] D. Petriu and M. Woodside, “Software performance models from system scenarios in Use Case Maps”, *Lecture Notes in Computer Science, Proc. 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, vol. 2324, pp. 141-158, 2002.
- [8] H. Christensen and K. Hansen, “An empirical investigation of architectural prototyping”, *Journal of Systems and Software*, vol. 83 Issue 1, pp. 133-142, January 2010.
- [9] B. Zeigler, H. Praehofer and T. Kim, *Theory of Modeling and Simulation—Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press, 2000.

- [10] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord and B. Wood, "Attribute-Driven Design (ADD)", SEI, Version 2.0. Technical report, CMU/SEI-2006-TR-023, ESC-TR-2006-023, 2006.
- [11] ISO/IEC WD3 42010. IEEE P42010/D3. Systems and Software Engineering. Recommended practice for architectural description of software-intensive systems, 2008.
- [12] C. Hofmeister, R. Nord and D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
- [13] R. Buhr, "Making behaviour a concrete architectural concept", in *Proc. 32nd Hawaii International Conference on System Sciences*, 1999, pp. 1-5.
- [14] R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities and Collaborations*, Addison-Wesley, 2002.
- [15] A. Verbraeck and E. Valentin, "Design guidelines for simulation building blocks", in *Proc. 2008 Winter Simulation Conference (WSC)*, December 2008, pp. 923-932.
- [16] B. Zeigler and H. Sarjoughian, *Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-based Simulation Models*, 2005.
- [17] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 2003.
- [18] P. Clements, R. Kazman and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.
- [19] F. Bachmann, L. Bass and M. Klein, "Preliminary design of ArchE: a software architecture design assistant", SEI, Technical Report, CMU/SEI-2003-TR-021, ESC-TR-2003-021, 2003.
- [20] F. Bachmann, L. Bass, M. Klein, J. McGregor and P. Bianco, "Using ArchE in the classroom: one experience", Software Architecture Technology Initiative, Technical Note, CMU/SEI-2007-TN-001, 2007.
- [21] F. Bachmann, L. Bass and M. Klein, "Experience using an expert system to assist an architect in designing for modifiability", in *Proc. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, 2004, pp. 281-284.
- [22] L. K. Singh, A. K. Tripathi and G. Vinod, "Software reliability early prediction in architectural design phase: Overview and Limitations", *Journal of Software Engineering and Applications*, vol. 4 N° 3, pp.181-186, March 2011.