Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
2014, Vol. 90(3) 290–319

DOI: 10.1177/0037549713518586
sim.sagepub.com
SAGE SCS

Modeling and simulation of software architecture in discrete event system specification for quality evaluation

Verónica Bogado[1,2], Silvio Gonnet[1] and Horacio Leone[1]

Abstract

Software quality is an important issue in software engineering. The evaluation of software architecture is crucial to achieve quality scenarios, which reduces development and maintenance costs. This work presents a discrete event simulation environment for the software architecture assessment considering both functional and quality aspects. Discrete event system specification (DEVS) formalism and the underlying framework are used to specify the simulation elements. DEVS is based on the system theory and follows the engineering and object-oriented fundamentals to construct complex dynamic systems. The proposed environment is built in a modular and hierarchical way that provides scalability and reusability advantages. Although the proposal is focused on three quality attributes and a few metrics, this approach enables the definition of new elements and metrics related to other quality attributes that can be visible at runtime. A traditional architecture illustrates the proposal, where preliminary computational experiments for this real software system and concrete quality scenarios demonstrate the feasibility of the integrated simulation environment for the software architecture evaluation. In addition, a discussion shows how the results could help architects make design decisions to improve software quality during development.

Keywords

Discrete event system specification, software architecture, quality attribute, software quality evaluation, simulation environment

1. Introduction

Software quality has become a critical issue because it affects systems development costs, delivery schedules, and user satisfaction.[1,2] Functional and structural quality must be considered to improve the overall quality of software products.[3,4] Structural quality focuses on the principles of software architecture (SA), where quality attributes are important. Quality attributes are non-functional requirements (NFRs, also known as quality requirements) that the system has to include and that are beyond the functionality of the software (functional requirements, FRs)[3] (e.g., availability, performance). SA plays a key role in the quality evaluation because it allows an early study of the system from operational and structural perspectives. The SA is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.[3] SA is the fundamental organization of a software embodied in its components, their relationships, the interaction with the environment, and the main decisions guiding its design and evolution.[5] In other words, SA constitutes a graspable model for how a software system is structured and how its components work together to provide support for some software qualities. The SA is developed during the earliest phases of the system production and accompanies the system throughout its lifetime; SA reduces costs, improves quality, supports delivery against requirements, and reduces risk.[3,6] Furthermore, SA is the first stage in which quality requirements could be addressed, and the early design decisions on which the architecture is based will greatly affect the resulting system.[7] Therefore, SA evaluation is crucial to obtain results that can be applied to

[1]INGAR, Instituto de Desarrollo y Diseño (UTN-CONICET), Argentina
[2]Departamento Ingeniería en Sistemas de Información, Facultad Regional Villa María, UTN, Argentina

Corresponding author:
Verónica Bogado, INGAR, Instituto de Desarrollo y Diseño (UTN-CONICET), Avellaneda 3657, S3002 GJC Santa Fe, Argentina.
Email: vbogado@santafe-conicet.gov.ar

achieve quality requirements. The main challenge is to assure stakeholders that the candidate architecture is capable of supporting the current and future business goals related to the FRs and NFRs.[8]

SA evaluation consists of quantifying some qualities, and it can be performed by using a qualitative or quantitative approach or a mix of both.[9] Most approaches employ the Markov Decision Process,[10,11] Queueing Theory,[12] or Petri Nets.[13] However, several issues must be improved: (i) formalisms introduce the complexity of the technique in the model and, consequently, stakeholders must operate on a low level of abstraction; (ii) these approaches have some limitations in representing software elements; and (iii) these approaches do not consider FRs in the models. Including FRs in the architecture specification implies an early emphasis on behavioral aspects. The system functionality is supported by its architecture through the interactions among its elements, acting as a vehicle through which system qualities are achieved. Quality attributes cannot be achieved without a unified vision of the architecture, all its elements, and their relationships. For example, to address performance requirements, it may be necessary to consider the execution time of each component and the time spent in intercomponent communication.[6] In this way, Use Case Maps (UCMs[14]) notation has gained popularity within the software domain. UCMs are models that relate FRs to structural elements, that is, causal scenarios related to architectural entities.[15] UCMs can help architects to understand emergent behavior of complex and dynamic systems, but it is an informal notation requiring other formal techniques for a dynamic evaluation.[16]

In this context, it is essential to supplement traditional techniques and tools with novel approaches oriented to the structural and functional quality that can be easily adapted to new demands of software designs. This work proposes a generic simulation environment for SA evaluation based on a Discrete Event System Specification (DEVS) framework to aid architects in the analysis of software quality attributes. A SA specified using UCM notation defines the input elements to obtain the simulation model and its background with the purpose of simulating the system operation at any stage of its development. The evaluation can be conducted under several conditions and collects measures related to quality attributes that are visible at runtime. Thus, concrete simulation environments are built from SAs following a set of rules defined in this work, hiding the formalism from the architect. The simulation has shown to be a powerful instrument for studying the system behavior and providing statistics related to features of the system under study. Furthermore, the DEVS framework provides instruments to build an adaptable simulation environment,[17] where the elements are built following the principles of modularity and hierarchy. This high level of abstraction enables us to suitably represent the concepts of the SA domain. This approach is achieved by developing two main parts of the DEVS frame: the model and the experimental frame (EF). The model represents the SA elements and some required parameters related to the quality attributes that will be measured. The model has been proposed and described in a first version in previous works,[18,19] and the features and capabilities are extended in this work. The EF is the operational formulation of the objectives. The construction of the EF is as important as the model for this environment because it will introduce quality goals associated with software quality attributes and will calculate quality indicators for each attribute to be analyzed. Therefore, the concepts of the architectural domain are translated into elements of the simulation model, and the background that interacts with the system is modeled as elements of the EF.

The quality attributes that have been previously studied include performance, reliability, and availability. Several works have considered these attributes to be critical to software quality. Performance is a dominating user requirement in the software industry and is perhaps the most studied aspect in this area. Reliability is another important aspect of the software. Several studies have addressed reliability as the major factor in software quality because this attribute quantifies failures that can make a powerful system inoperative.[20] Reliability is related directly to availability, which is typically measured as a factor of its reliability. In this sense, one attribute is a consequence of the other, with higher reliability indicating a greater availability. In software development, quantitative information to make decisions is critical. Studying SA to improve the design can impact development and maintenance costs and avoid errors that could affect the systems' users. Although quality indicators for three attributes are included into this proposal, this approach could be scalable to other quality attributes visible at runtime, such as security.

The remainder of this paper is organized as follows: Section 2 analyzes related works; Section 3 briefly describes SA and the related concepts required to analyze quality attributes; Section 4 describes the discrete event modeling and simulation (M&S) framework applied to software quality evaluation; Section 5 presents the simulation model and specifies each simulation element using DEVS; Section 6 explains the importance of the EF in the SA evaluation context and how to build it according to the quality objectives and describes the entire simulation environment; Section 7 briefly describes the implementation of the simulation environment; Section 8 illustrates the contribution with a case study that describes a traditional SA for a particular system, the simulation model, its configuration, and results, explaining how to use this information to make design decisions; and Section 9 presents conclusions and future work.

2. Related work

In recent years, the use of SA to predict quality attributes and the validation of requirements represented as quality

scenarios have become more critical in software development. In the software engineering literature, there are several approaches that support architects; some approaches are theoretical rather than empirical. We can also find methods based on scenarios for a qualitative analysis.[7] To place our proposal into context, we prefer to organize related contributions into three areas: (i) works related to SA evaluation based on quality attributes measurement; (ii) works related to SA evaluation that also consider functional aspects (UCM proposal); and (iii) works based on simulation techniques.

In the first related area, there are contributions that study a particular quality attribute and others that consider a set of attributes. For example, some proposals employ the Markov Decision Process to evaluate the system reliability;[10] more recent works include performance and security in the analysis.[11] Other authors have proposed the application of Queueing Theory to measure performance[12] as the most common application because performance is the most visible attribute and the easiest to be modeled at runtime. Petri Nets evaluate different quality attributes, such as security, performance, and reliability, and have been important contributions in this area.[13] The aforementioned techniques make a quantitative study of specific quality attributes with mathematical fundamentals, which is very important for precision in the final results. However, critical studies have shown some limitations of these types of techniques because they appear to be too restrictive from the modeling viewpoint.[21] These approaches have some restrictions related to representing a number of situations about ''real-world'' systems; furthermore, they incorporate the complexities of the technique in the model, losing abstraction capabilities. Markov Process approaches have two main aspects that must 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 that requires specific information (such as transition probabilities, quantitative data for each component, etc.).[21] 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 related to complex system modeling as it is very difficult to solve the resulting net. Petri Nets require simplification of the modeled problem, but this simplification may cause a loss of important information related to the real problem (in this case, features of the software system that is being analyzed), affecting the final results.[22] On the other hand, empirical techniques have increasing importance, where architectural prototyping has been proposed.[23] This type of technique requires extra effort and costs to build a prototype of the software system at the beginning of the software development process. Finally, in this first group of related work, we can find contributions that propose an ad hoc assessment of the SA qualities, because it is difficult to exclude subjectivity and informality from the analysis.[24]

Currently, enterprises emphasize the importance of including functional aspects in the evaluation of SA, which in most cases have been focused only on non-functional aspects. Therefore, in this second group of related works, we can find some contributions that include functionality in the architectural models. The Software Engineering Institute (SEI) has proposed advances in this area by developing *Arche*.[25] This tool permits the analysis of performance (using Fixed Priority Scheduling) and modifiability (using Graph Theory) from an architecture that considers functional aspects.[26] Moreover, other approaches based on UCMs have been proposed to add behavior to the architectural structures in the analysis. Because UCMs is an informal notation, related works have proposed the use of Queueing Theory to analyze system performance, but they are only focused on this attribute.[27]

The last group of related contributions employs simulation techniques. RAPIDE language is an antecessor of the DEVS approach and is focused on interface connection architectures, where public interfaces are translated into actions.[28] RAPIDE provides a formal specification to find violations in the interfaces, testing the complete specification, the correctness, and the performance analysis of distributed time-sensitive systems. However, this method does not consider scenarios or functionalities that a software element provides and does not capture the internal complexity of software elements. The limitations of the aforementioned techniques lead to describe the problem of SA evaluation by mixing several techniques, causing a sophisticated simulation development. However, this complexity of the techniques obscures the problem that is being resolved. DEVS appears to be a simple and more scalable simulation approach to tackle the SA evaluation problem.[17] DEVS has been shown to be a very flexible formalism in many other domains, for example traditional applications[29] or the study of physical system quality.[30] Furthermore, some authors have developed domain-specific M&S using OO-DEVS.[31] That work proposes the application of specific design patterns (Composite, Façade, Observer) to an astronomical observatory system. Although the approach appears to be at a high level of software design, it is mainly focused on a lower level design. Thus, the evaluation is not performed immediately after the requirements specification. On the other hand, the approach is not mainly focused on the SA evaluation in terms of software quality in a general form; rather, it develops specific-domain metrics for astronomy observatory systems.

SA design is a challenging problem that requires elements to represent different types of software systems and many domain aspects that can change over time. This dynamic feature requires flexible conceptual tools that can be adapted to new requirements. Thus, this work proposes

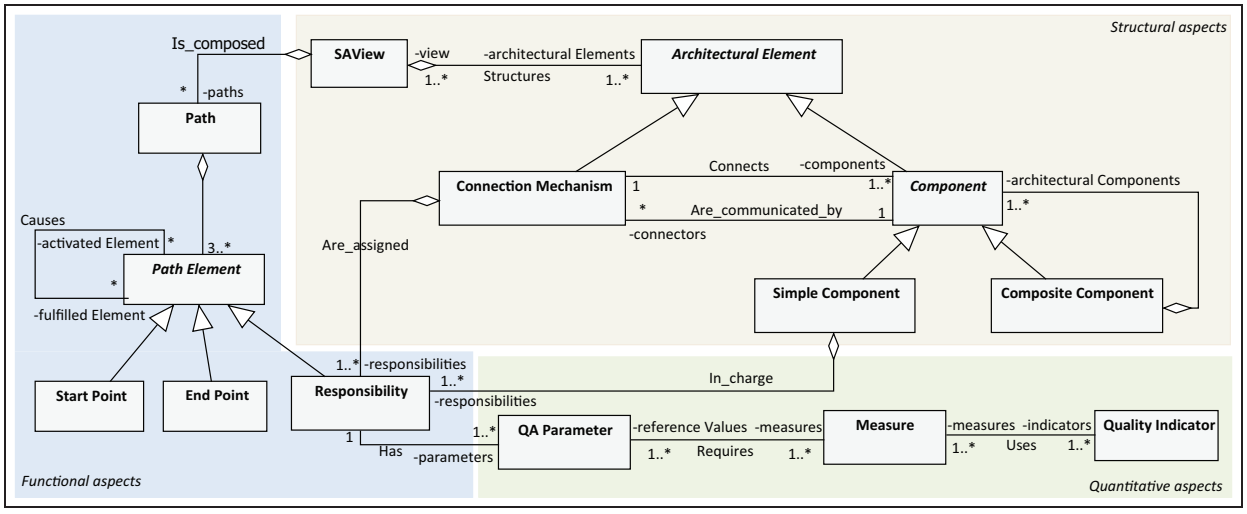

Figure 1. *SAEM*: software architecture evaluation concepts and their relationships.

a DEVS-based simulation environment to analyze SAs by measuring quality attributes from a UCM model. This proposal is very close to requirement specifications and can be applied at the earliest stages of software development, taking a high-level design employing UCM notation as an input. The concept of UCMs is based on capture FRs, taking into account the architectural elements responsible for derived functionality. This work provides continuity to very preliminary proposals in which only the performance attribute was considered,[18,19] adds new features to the initial model, and presents a set of reports that can be used by architects to make decisions. This proposal includes the discrete event simulation advantages in the architectural design context, resulting in a novel and adaptable approach to evaluate software quality. Although early SA evaluation is more common than late assessment, the proposed simulation environment can be applied at any stage with different purposes, that is, for quality prediction (early) or analysis (late).

3. Software architecture and quality attributes abstractions

SA evaluation requires structural, functional, and quantitative information that can be used to obtain quality indicators. Several contributions have proposed specific language and good practices to design SAs.[3,4,5,8] The SEI suggests the use of quality scenarios to specify quality requirements from which to build an appropriate design.[3] This design can be specified using several notations, including Unified Modeling Language (UML) and UCMs, among others. The UCM is a model that provides a dynamic view of the architectural model by adding functional aspects under the concept of responsibility.[14,32] Thus, the architecture not only includes NFRs but also FRs. These concepts are captured in the *Software Architecture Evaluation Model* (*SAEM*, Figure 1).

The *SAEM* captures concepts that specify a dynamic view of the SA, elements, and their relationships by adding one more granularity level to the responsibilities. Furthermore, UCM concepts are integrated with quantitative aspects related to quality. This model represents the following entities: responsibility (*Functional aspects*, Figure 1), simple and composite components, connection mechanisms, and the relationships among these artifacts. These elements model the SA view (*SAView*, Figure 1) and are the *Structural aspects* (Figure 1).

An architectural element (*ArchitecturalElement*, Figure 1) is an abstract concept that represents structures with a presence at runtime, providing a dynamic view of the system. Two important specialized concepts are included, namely component (*Component*, Figure 1) and connection mechanism (*ConnectionMechanism*, Figure 1). Moreover, the components can be classified into two types: simple (*SimpleComponent*, Figure 1) and complex (*CompositeComponent*, Figure 1). A simple component is a software entity that could have some runtime presence (e.g., a process) and is in charge of a set of responsibilities. The composite component can be composed of both simple and composite components, and responsibilities are delegated to each component. The forms of interaction of the software elements, such as the components, are the connection mechanisms. If the connection mechanisms are complex connectors, they could also have assigned responsibilities.

A responsibility (*Responsibility*, Figure 1) represents the smallest executable unit in this software model. Moreover, a responsibility is the unit of measurement because it can be associated with a number of parameters (*QAParameter*, Figure 1). These parameters are used to

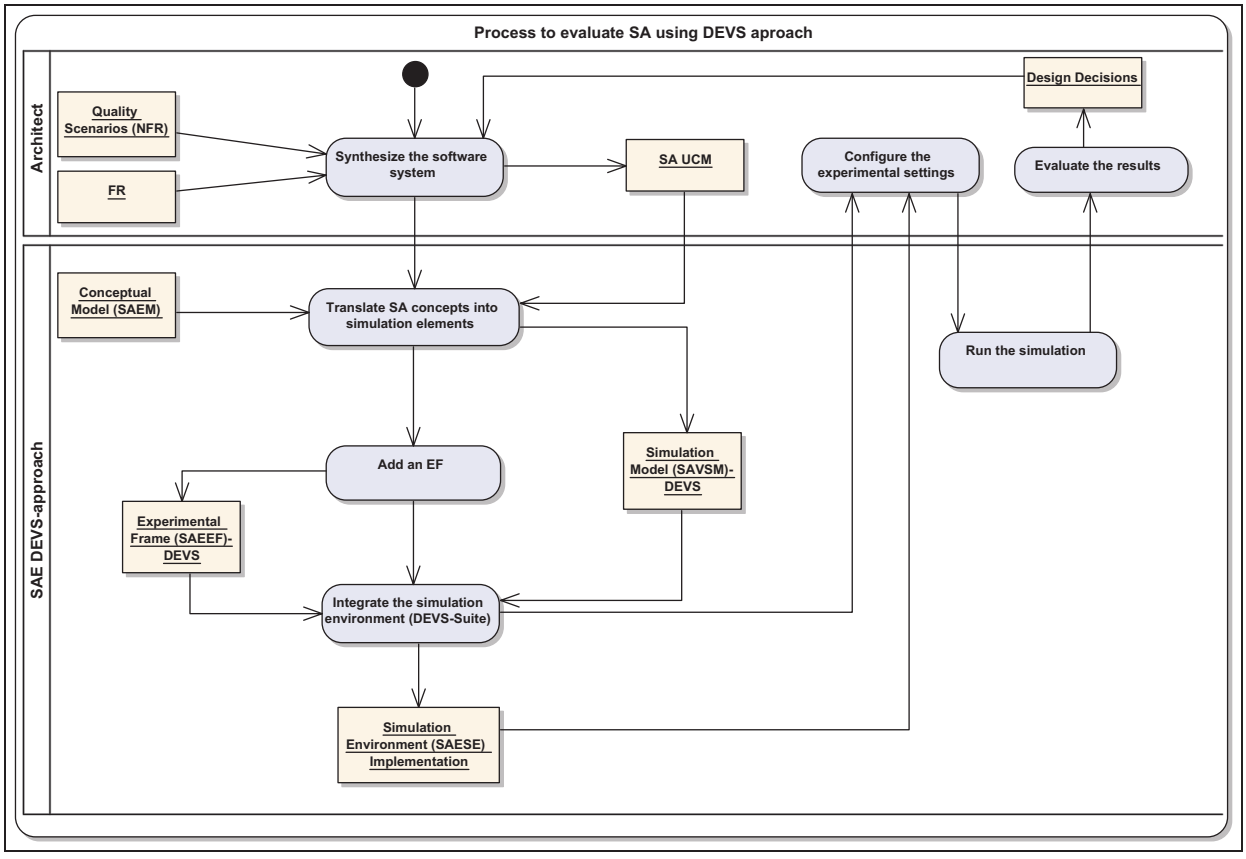

Figure 2. Discrete event system specification (DEVS) process to evaluate software quality.

model the behavior of different dynamic aspects that can be found in a software at the execution time. These parameters are related to different quality metrics that provide measures (*Measure*, Figure 1) related to quality indicators (*QualityIndicator*, Figure 1). The parameters can be used to create statistics according to the architect's interests, which allow for quality objectives to be accomplished. In this way, the *SAEM* includes numerical quality attribute values that provide quantitative analysis capabilities (*Quantitative aspects*, Figure 1).

The UCM is composed of paths that may traverse components (*Path*, Figure 1). A path is interpreted as a visual representation of a scenario (*Functional aspects*, Figure 1). Paths are routes along which chains of causes and effects propagate through the system. The start point of the path is a waiting place for a stimulus (*StartPoint*, Figure 1). Its immediate effect is the execution of the first responsibility along the path. This in turn is a cause relative to the next point along the path (relationship *Causes*, Figure 1); this pattern continues as the causes accumulate to result in each next effect. The path ends where the end bar is reached (*EndPoint*, Figure 1). The path is considered progressive in the terms of each point (*responsibility*) along it advances the path toward the end.[15]

A discrete event approach applied to evaluate software quality from the SA requires a set of steps. Figure 2 shows the process and elements associated with this approach, from the user's requirements (*FRs* and *NFR*) to the software quality evaluation. This process includes several activities, some of which must be completed by the architect.

The first activity involves the synthesis of SA and the information needed to evaluate its quality guided by the user requirements (*Synthesize the software system*, Figure 2). The SA is represented by a UCM model (product *SA UCM*, Figure 2). This model is a required input to apply the proposed evaluation approach and provides instances of the elements that have been captured in the *SAEM*, as we have described. These elements are translated into simulation elements using the *SAEM* and a set of transformation rules that define the correspondence between SA elements and simulation elements in DEVS (activity *Translate SA concepts into simulation elements*, Figure 2). Consequently, the simulation model for the paths of the given SA view is obtained (*Simulation Model (SAVSM)-DEVS*, Figure 2). This view is a particular case (instance) of the generic simulation model (*SAVSM*) specified using DEVS formalism in this contribution.

Table 1. Measures taken from each responsibility.

ID	Metric	Description
rta	Turnaround time per request	Time that a responsibility requires to answer a request (responsibility execution).
fdt	Downtime per failure	Time that responsibility stays "failed" each time a failure occurs.
frt	Recovery time per failure	Time that a responsibility needs to return to a normal operation after a failure has occurred.
fn	Number of Failures	Number of failures that have happened to the moment in a responsibility.

On the other hand, the environment that interacts with the software system is modeled in the EF. The EF includes information taken from the requirements specification, particularly from quality scenarios, which enable the *start* and *end points* of *paths* in the UCM (Figure 1). This model provides the elements to build an EF adapted to the quality goals (activity *Add an EF* and product *Experimental Frame (SAEEF)-DEVS*, Figure 2), which define quality measures that are relevant to the system. This EF is specified using DEVS formalism and it is added to the simulation environment to ''give life'' to the software system that is being simulated.

Both the simulation model and EF are integrated into the simulation environment (product *Simulation Environment (SAESE) Implementation*, Figure 2). The architect must configure the experimental settings (parameters) and *run the simulation* (Figure 2). Finally, the architect validates the quality scenarios and makes decisions that can improve the system design using the information provided by the execution of the simulation model (activity *Evaluate the results*, Figure 2). These *design decisions* can modify the SA and its elements or responsibilities to achieve the quality goals.

Although this approach is mainly useful during the early stages of the software development, it can be applied at any stage. If a project is new, the view of the SA can be obtained from user requirements and quality scenarios (*NFRs*) to identify elements of the architecture and FRs to extract responsibilities and the causalities among them. If the project is intended to modify an implemented system in use, the architecture can be obtained by employing tools such as the SWAG Kit (SA Group, http://www.swag.uwaterloo.ca/swagkit/) or ''manually'' (ad hoc) by reconstructing the architecture elements and identifying the responsibilities.

3.1 Software quality metrics

In this work, metrics related to three quality attributes that can be measured at runtime are discussed: performance, availability, and reliability. In our model, the responsibilities are the main providers of quantitative information, as shown in Table 1. Indirect metrics can be calculated from the information that responsibilities provide. These indirect metrics are quality indicators of the system, or per intermediate components that contain the involved responsibilities (Table 2).

4. Discrete event modeling and simulation and discrete event system specification fundamentals in the software architecture evaluation context

DEVS is a formalism for simulating discrete event systems. Furthermore, DEVS defines the system behavior and structure and has a conceptual framework (Figure 3), which is composed of three entities:[17] a *Model* (system specifications, structures, and behaviors required to generate data comparable to data from the real world), a *Simulator* (a computation system that executes the instruction of the model giving life to it), and an *EF* (conditions under which the system is observed, allowing for the experimentation and validation of the model). These entities are linked by two types of relationships (Figure 3): *Modeling* to represent the system and validate the model with the real world and *Simulation* to assure the simulator correctness. This contribution is focused on the simulation modeling in the SA evaluation domain. Thus, we have developed a generic simulation model and EF (Figure 3). The simulator is provided by DEVSJAVA, which is embedded in DEVS-Suite 2.1 (http://acims.asu.edu/software/devs-suite), encapsulating this part of the frame from the other two.[33] Keeping three independent entities provides some benefits; for example, the same model can be executed by different simulators, or several experiments can be changed to study different situations.

The simulation environment follows the guidelines suggested by Zeigler et al.[17] and other authors.[34] Atomic and coupled models are employed to represent the software entities with different levels of complexity (Figure 4). The concept *Responsibility* (Figure 1) is represented by an atomic parallel DEVS in the simulation model that models the actual functionality (*RM*, Figure 4). An additional atomic model is introduced to represent embedded problems in the responsibility that can cause failures (*FG*, Figure 4). The parameters (*QAParamenter*, Figure 1) are allocated in these elements as fixed parameters, which are used during the simulation to calculate quality indicators (*QualityIndicator*, Figure 1). The calculation of these metrics is explained in Section 6 (*Stat* elements responsible for the statistics). *SimpleComponent* and *ConnectionMechanism* (Figure 1) are specified as coupled

Table 2. Indicators for the software architecture evaluation.

Metric	Description	Used direct metric
Average turnaround time of the system	Average time that the system (software) requires to answer to a request.	*rta* *Number of requests*
Average throughput of the system	Average number of request served per time unit in the system (software).	*Number of requests* *Total time*
Total downtime of the system	Total time that the system (software) has been offline, considering only the downtimes.	*fdt*
Total unavailable time of the system	Total time that the system (software) has been offline, considering the downtimes and recovery times.	*fdt* *frt*
Total uptime (available) of the system	Total time that the system (software) has been online.	*fdt* *Total time*
Total number of failures of the system	Total amount of failures occurred in the system (software).	*fn*
Average turnaround time per responsibility	Average time that a responsibility requires to answer to a request.	*rta* *Number of requests per responsibility*
Average throughput per responsibility	Average number of requests served per time unit in the responsibility.	*Number of requests per responsibility* *Total time*
Average downtime per responsibility	Average time that a responsibility has been "failed".	*fdt* *fn*
Number of failures per responsibility	Number of failures occurred in a responsibility.	*fn*

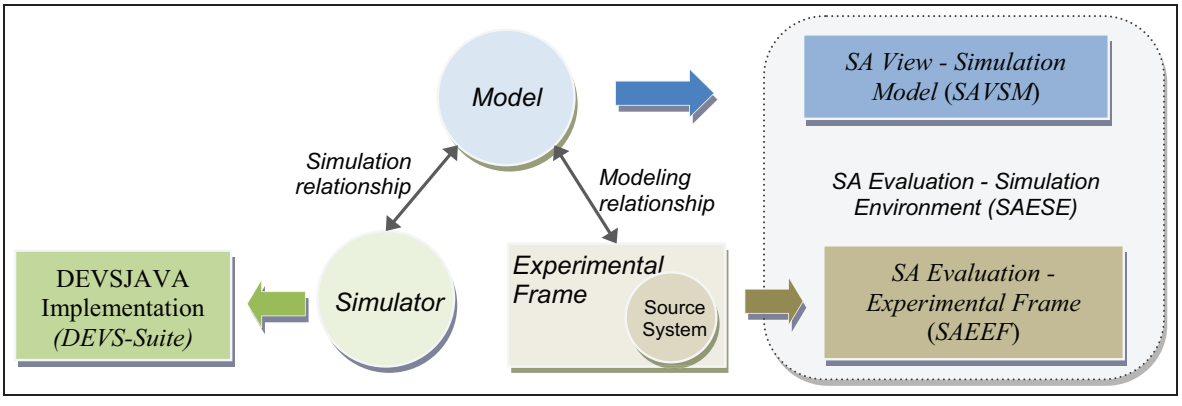

Figure 3. Discrete event system specification (DEVS) modeling and simulation framework in software architecture evaluation context, based on Zeigler et al.[17]

models called *SC* and *CM* respectively, whose elements are one or more responsibilities. These elements can be parts of *CompositeComponent* (Figure 1), which is specified as a hierarchical DEVS called *CC*, which can be composed by instances of *CC, CM*, and *SC* (Figure 4). These architectural elements are structured by a view (*SAView*, Figure 1), which is translated into a hierarchical DEVS (*SAVSM*) and represents the simulation model. Finally, the whole environment is on the top of the hierarchy (*SAESE*, Figure 4), where the simulation model (*SAVSM*) and experimental frame (*SAEEF*) are components of it. The two DEVS models employed in this work are the atomic parallel model and coupled model.

4.1 Atomic Parallel DEVS

The Atomic Parallel DEVS model has multiple ports to receive values at the same time. It differs from a classic DEVS in allowing all imminent components to be activated and to send their outputs to other components. Instead of having a single input per port it has a bag of inputs, with the possible multiple occurrences of its elements and it has a confluent transition function to solve collisions between internal and external events. The specification of this DEVS is as follows:[17]

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

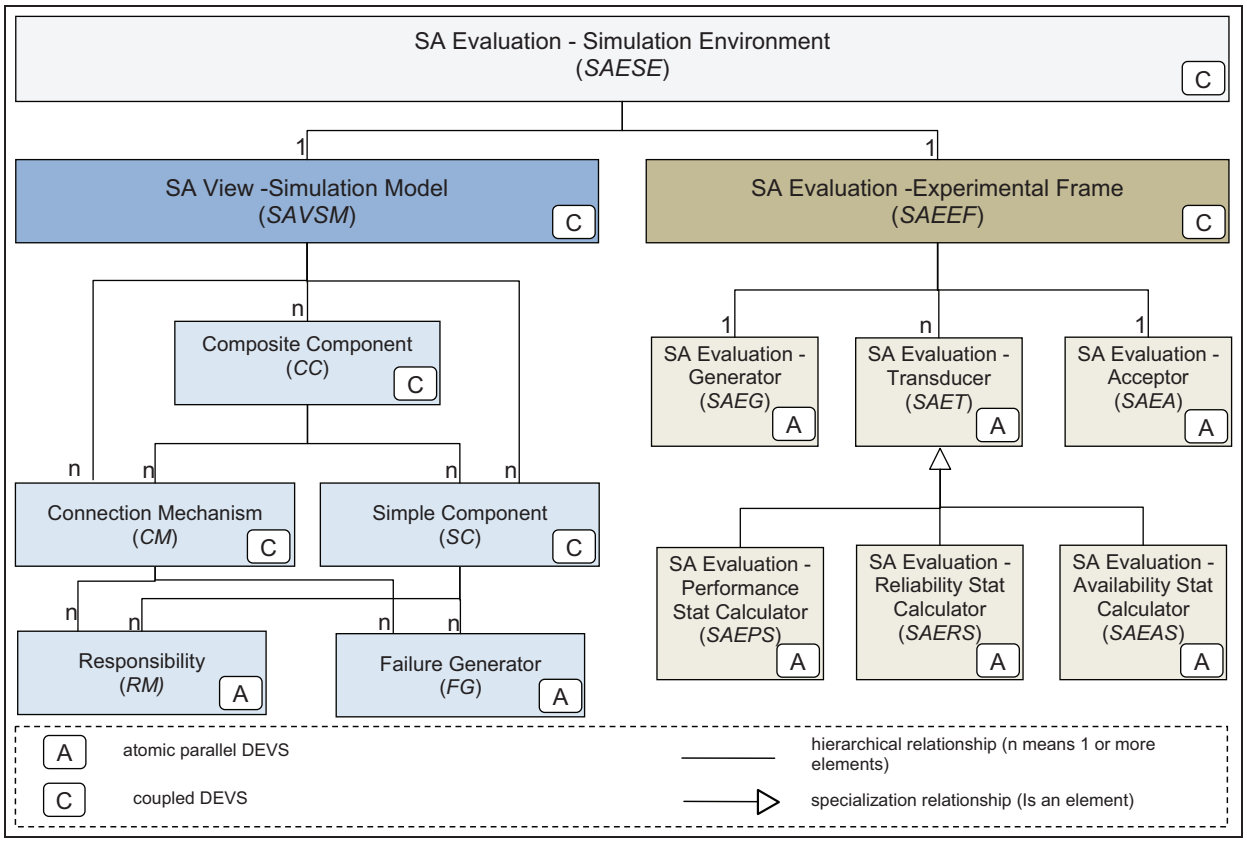

Figure 4. Hierarchy of simulation elements (*SA Evaluation – Simulation Environment*).

where

$X = \{(p, v)|p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output ports and values

S is the set of states;

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

δ_{int}: $S \rightarrow S$ is the internal state transition function;

δ_{con}: $Q \times X^b \rightarrow S$ is the confluent transition function;

λ: $S \rightarrow Y$ is the output function;

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

4.2 Coupled DEVS

A coupled DEVS model specifies which models become components of it and how they are connected. It is defined as follows:[17]

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

where

$X = \{(p, v)|p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of component names;

M_d represents the component models, $d \in D$;

$EIC \subseteq \{((N, ip_N),(d, ip_d))|ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$ is the external input coupling;

$EOC \subseteq \{((d, op_d),(N, op_N))|op_N \in OPorts, d \in D, op_d \in OPorts_d\}$ is the external output coupling;

$IC \subseteq \{((a, op_a),(b, ip_b))|a, b \in D; op_a \in OPorts_a; ip_b \in IPorts_b\}$ is the internal coupling.

5. Simulation model for the software architecture evaluation: dynamic software view

The simulation model represents the required elements to evaluate the quality of software through its architecture by employing a UCM model. This model is described using a bottom-up approach, from the leaf-elements of Figure 4 to the simulation model.

5.1 Responsibility: RM atomic model

RM is an atomic model that represents one FR that must be satisfied by a component of the SA (Figure 1). The

responsibility is the smallest unit of the dynamic elements in the *SAEM*, so it is the basis on which to build more complex dynamic elements. *RM* takes base measures used to calculate quality indicators (Table 1).

We propose to model responsibility behavior with five phases (Figure 5): *inactive, active, executing, failed*, and *recovering. RM* is in phase *inactive* when the system is not providing the functionality. *RM* is in phase *active* when the system is ready to provide the functionality. The responsibility assumes *executing* when the system is actually providing the functionality. If for some reason the system cannot provide the functionality, the responsibility goes to phase *failed*. From here, the responsibility moves automatically to *recovering*, which represents the recovery process that might be required when a failure interrupts the execution. *RM* has two input ports (Figure 1): *prip* and *intfailip*. Port *prip* is connected to the output ports of other responsibilities for representing the causal relationship between them. Thus, if two responsibilities are connected by a link *Causes*, then the output port (*srop*) of the fulfilled responsibility (*fulfilledElements*, Figure 1) is linked to port *prip* of the activated responsibility (*activatedElements*, Figure 1). Port *intfailip* is linked to the failures generator representing the problems that might occur in the responsibility.

In this model, a state is indicated by one of the described phases, a sigma value, and a finite queue of requests (sequence *requests*), defined in (5). *RM* is initialized in *inactive* (Figure 5), the passive state of waiting for a request. When a request ($request_i$) arrives at *prip* (external transition with $x = (prip, request_i)$, Figure 5), *RM* changes to *active*, a transitory state ($\sigma = 0$). This transitory state generates an output indicating that the execution of a responsibility has been started (internal transition with $y = (stateop, activated)$, Figure 5). *RM* then evolves to the *executing* phase and remains there during the time required to process the request if non-failure occurs. The time that the responsibility requires to execute a response to a request is specified by the internal transition between *active* and *executing*. The time is computed by function *executionTime*(), which follows a probability distribution according to the features of the responsibility in a particular system; this parameter is set by the architect, otherwise it is assumed uniform by default. After processing the request, *RM* returns to the initial state (*inactive*) if the queue of pending requests is empty; otherwise, the RM stays in the *executing* stage processing the next request.

When *RM* is in *executing*, it can stop its execution due to a failure. To model the behavior of the internal failures, this DEVS can be coupled to a failure generator (*FG*). The failure event ($fail_k$) arrives at *intfailip* (external transition with $x = (intfailip, fail_k)$, Figure 5). Therefore, *RM* changes to state *failed*, where the time in this state is defined by the function $downtime(fail_k)$. This function defines the downtime according to the particular failure and responsibility, following a defined behavior set by the architect (or uniform as the default). Once this time is over, *RM* changes to *recovering.* The responsibility can immediately recover from a failure ($\sigma = 0$) or can require additional time to return to the normal execution. This time is defined by the function *recoveryTime*(), which returns time values according to the failure and responsibility, following a probability distribution defined by the architect (by default, this behavior is uniform). We have assumed that the responsibility involves the task of defining its execution time and its recovery time when a failure occurs, depending on its features. However, the downtime due to failure is defined by the type of the failure, which is specified at the moment in which the failure is generated by *FG*.

RM is in charge of emitting three types of output values related to the requests (data of interest for other entities of the model), to its state, and to the values used for

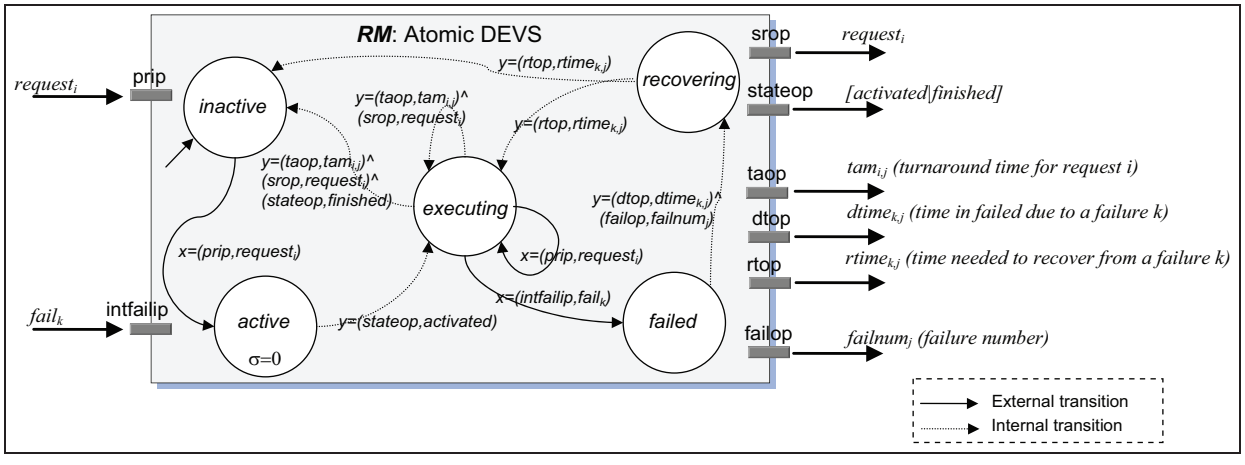

Figure 5. *RM* (responsibility), atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

measuring quality attributes. *srop* (Figure 5) emits the requests to the next responsibility determined by the relationship *Causes* (Figure 1). Each event is sent when the responsibility finishes processing the request (internal transitions with y = (*srop, request*$_i$), Figure 5). *stateop* sends the value *activated* to indicate that the responsibility has started the activity (internal transition with y = (*stateop, activated*), Figure 5), and the value *finished* to indicate that it returns to state *inactive* when no requests are pending (internal transition with y = (*stateop, finished*), Figure 5). The other output ports are used to send measures (Table 1) to the entities of the EF that are in charge of calculating quality indicators. In this way, a value that indicates the turnaround time for a request ($tam_{i,j}$) is sent to *SAEPS* from *taop* (internal transitions with y = (*taop,* $tam_{i,j}$), Figure 5), the time ($dtime_{k,j}$) that the responsibility was ''failed'' due to a failure is sent to *SAEAS* from *dtop* (internal transition with y = (*dtop,* $dtime_{k,j}$), Figure 5), the time needed to recover ($rtime_{k,j}$) from a ''failed'' situation is sent to *SAEAS* using *rtop* (internal transitions with y = (*rtop,* $rtime_{k,j}$), Figure 5), and the number of the failure k ($failnum_j$) is sent to *SAERS* using *failop* (internal transition with y = (*failop,* $failnum_j$), Figure 5).

The *Responsibility* is formally specified in DEVS as follows:

$$RM = \left\{X_{RM}, Y_{RM}, S_{RM}, \delta_{ext,RM}, \delta_{int,RM}, \delta_{con,RM}, \lambda_{RM}, ta_{RM}\right\} \tag{3}$$

where

$$X_{RM} = \{(p, v) | p \in RIP, v \in X_{RM,p}\} \tag{4}$$

is the set of input ports and values. RIP = {*prip, intfailip*} is the set of input ports, $X_{RM,prip} = \{request_i | i = 1..n\}$ is the set of requests (i is the request ID and n is the number of requests), $X_{RM,intfailip} = \{fail_k | k = 1..q\}$ is the set of the failures (k is the failure ID and q is the number of failures).

$$S_{RM} = \{inactive, active, executing, failed, recovering\} \times R_0^+ \times X_{RM,prip}{}^* \tag{5}$$

is the set of sequential states. $X_{RM,prip}{}^*$ is a sequence of requests.

$$Y_{RM} = \{(p, v) | p \in ROP, v \in Y_{RM,p}\} \tag{6}$$

is the set of output ports and values. ROP = {*srop, stateop, taop, dtop, rtop, failop*} is the set of output ports, $Y_{RM,srop} = \{request_i | i = 1..n\}$ is the set of requests, $Y_{RM,stateop}$ = {*activated, finished*} is the set of responsibility macro states, $Y_{RM,taop} = \{tam_{i,j} | i = 1..n, tam_{i,j} \in R_0^+\}$ is the set of turnaround times for requests i by the responsibility (j = responsibility ID), $Y_{RM,dtop} = \{dtime_{k,j} | k = 1..q, dtime_{k,j} \in R_0^+\}$ is the set of times that the responsibility was failed due to a failure k, $Y_{RM,rtop} = \{rtime_{k,j} | k = 1..q, rtime_{k,j} \in R_0^+\}^+\}$ is the set of times required by the responsibility to recover from a failure k, and $Y_{RM,failop} = \{failnum_j | failnum_j \in Z_0^+\}$ is the set of number of failures in the responsibility.

External transition function ($\delta_{ext,\ RM}$):
$\delta_{ext,RM}$((*inactive, σ, requests*), *e*,(*prip, request*$_i$)) = (*active*, 0, *add*(*requests, request*$_i$))
where σ is the time remaining in the state, e is the elapsed time in the state and *add*(*requests, request*$_i$) is a function that returns a sequence with the value *request*$_i$ appended to the queue (sequence) *requests.*
$\delta_{ext,RM}$((*executing, σ, requests*), *e*, (*intfailip, fail*$_k$)) = (*failed, dtime, requests*)
where *dtime* = *downtime*(*fail*$_k$) that returns a value that represents the time that the responsibility will be unavailable because of the failure *fail*$_k$.
$\delta_{ext,RM}$((*executing, σ, requests*), *e*, (*prip, request*$_i$)) = (*executing, σ-e, add*(*requests, request*$_i$))

Internal transition function ($\delta_{int,\ RM}$):
$\delta_{int,RM}$(*active, σ, requests*) = (*executing, etime, rem*(*requests*))
where *etime* = *executionTime*(*get*(*requests*)), a function that calculates time execution following a probability distribution. Function *get*(*requests*) returns the first element of the sequence *requests*. Function *rem(requests)* removes the first element of the sequence *requests*, and returns the updated sequence.
$\delta_{int,RM}$(*executing, σ, requests*) = (*inactive*, ∞, *requests*), if *requests* is empty.
$\delta_{int,RM}$(*executing, σ, requests*) = (*executing, etime, rem*(*requests*)), if *requests* is not empty
where *etime* = *executionTime*(*get*(*requests*)).
$\delta_{int,RM}$(*failed, σ, requests*) = (*recovering, rtime, requests*)
where *rtime* = *recoveryTime*(), which calculates the time required to return to a normal state following a probability distribution.
$\delta_{int,RM}$(*recovering, σ, requests*) = (*inactive*, ∞, *requests*), if *requests* is empty.
$\delta_{int,RM}$(*recovering, σ, requests*) = (*executing, etime, rem*(*requests*)), if *requests* is not empty
where *etime* = *executionTime*(*get*(*requests*)).

Confluent transition function ($\delta_{con,\ RM}$):
$\delta_{con,RM}(s,ta(s), x) = \delta_{ext,RM}(\delta_{int,RM}(s), 0, x)$

Output function (λ_{RM}):
λ_{RM}(*active, σ, requests*) = (*stateop, activated*)
λ_{RM}(*executing, σ, requests*) = (*stateop, finished*) ^ (*taop,* $tam_{i,j}$) ^ (*srop, request*$_i$), if *requests* is empty
λ_{RM}(*executing, σ, requests*) = (*taop,* $tam_{i,j}$) ^ (*srop, request*$_i$), if *requests* is not empty
λ_{RM}(*failed, σ, requests*) = (*failop, failnum*$_j$) ^ (*dtop,* $dtime_{k,j}$)
λ_{RM}(*recovering, σ, requests*) = (*rtop,* $rtime_{k,j}$)

Time advance function (ta_{RM}):
$ta_{RM}(s) = \sigma$.

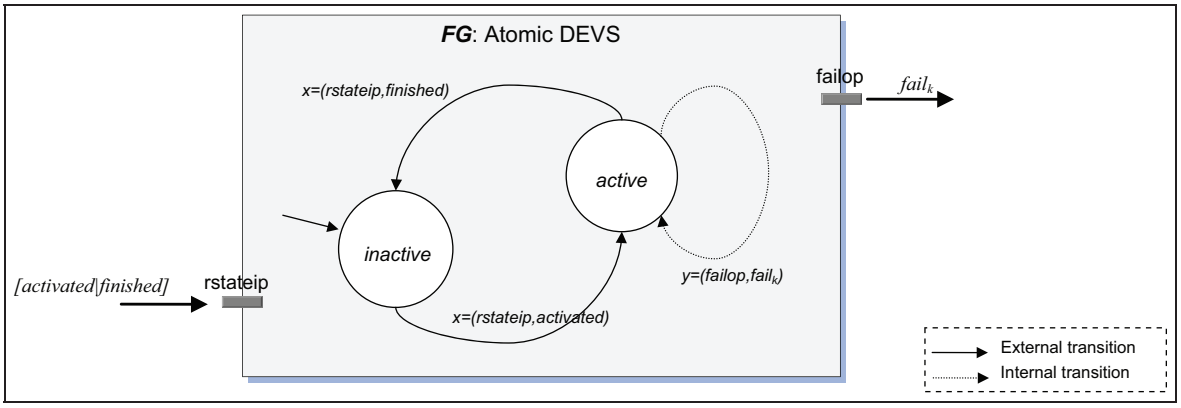

Figure 6. *FG*, atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

The quantitative aspects in the conceptual model (Figure 1) are modeled as fixed parameters of the atomic model or as measures. Thus, adding parameters to *RM* will allow for the evaluation of quality attributes under different scenarios of the system. *QAParameter* (Figure 1) is represented in this DEVS as fixed parameters, which are used to model the behavior of the execution time and the recovery time (*ref_execution_time* and *ref_recovery_time*). They reference values that are defined according to a probability distribution; they could represent a mean value (e.g., exponential) or a limit (e.g., uniform).

5.2 *Failure generator:* FG *atomic model*

A system failure occurs when the system no longer delivers a service consistent with the specification, where such the failure is observable by the users, either humans or other systems.[3] In this way, the *Failure Generator* (*FG*, Figure 6) was included into the simulation model to reproduce failures that can be introduced by a fault or error of developers or by other sources. In this model, we consider

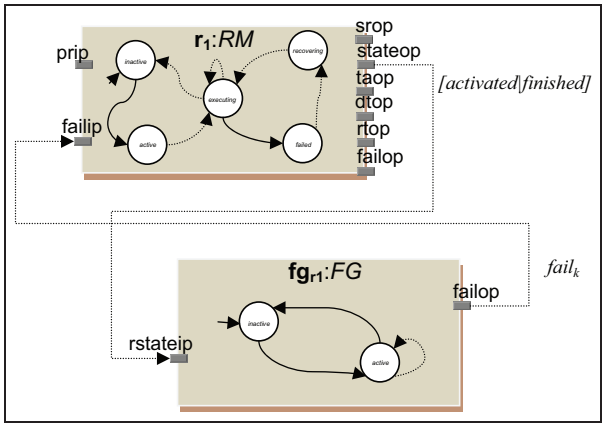

Figure 7. Failure generator (instance of *FG*) and its relationship with the responsibility (instance of *RM*).

situations that cause system problems, where there is a downtime associated with the failure.

The architect can choose whether to include this element. We prefer to keep the *FG* (Figure 6) decoupled from the responsibility (*RM*, Figure 5) to obtain a more flexible model. The situation where the responsibility does not have failures is modeled without this element. However, if the architect needs to model different scenarios where the software elements can fail, the corresponding responsibility can be related to an instance of this element. Although a failures generator can be modeled in the EF, a generator per responsibility allows the treatment of failures following the principle of modularity. This principle assumes that the failures of each software element (responsibility) are independent from other responsibilities, that developers or other sources introduce errors in an independent form in each responsibility, and that the complexity of the responsibility is different from the others, depending on the type of functionality it represents.

Figure 7 shows an instance of *RM* (r_1) coupled to an instance of *FG* (fg_{r1}) to introduce the probability of failure. Another concept associated with a failed system is the time required to repair it. In the simulation model, this time can be considered by setting the corresponding parameters (*ref_recovery_time*) in the responsibility.

FG is coupled to the responsibility associated with it by its input and output ports (Figure 7). *FG* has an input port (Figure 6), *rstateip*, which receives the state values sent by the responsibility associated to it. When the responsibility is *active*, this generator can emit failures for each given time period. The values are emitted using the output port, *failop*, and represent failures ($fail_k$). *FG* remains in phase *active* until the responsibility returns to a passive phase (*inactive*). Internally, this element changes from *active* to *active* each time, by an internal transition, to emit the failures. This interval of time is defined by the function *timeBetweenFail*(). The formal specification of this

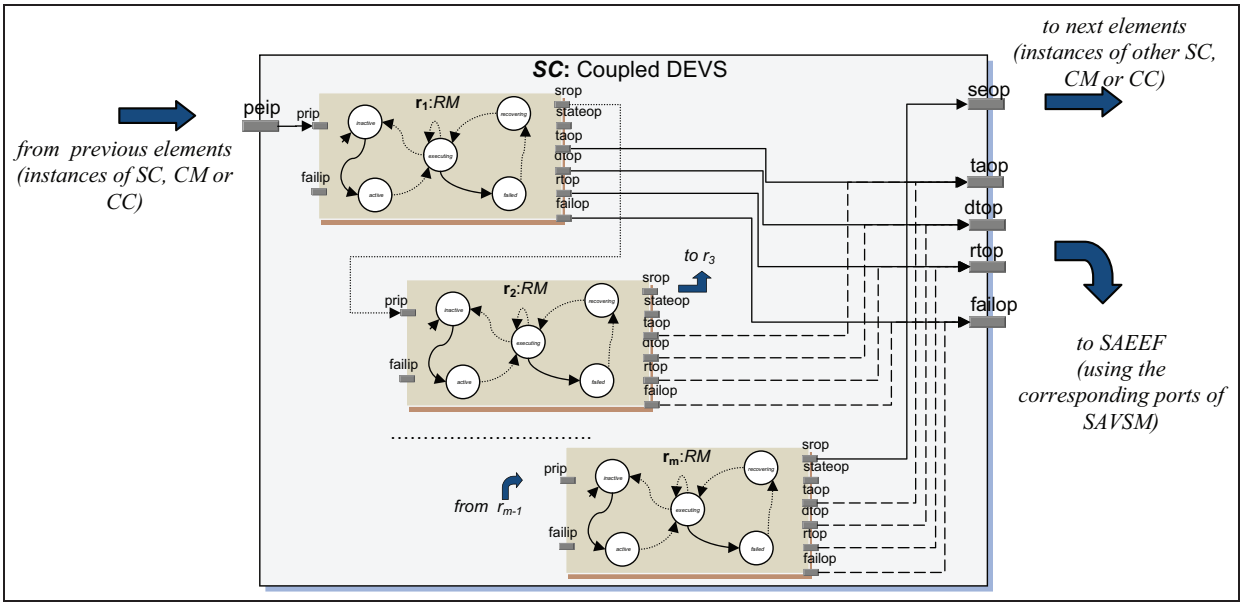

Figure 8. *SC*, simple component with a set of responsibilities considering that they are free of failures.

element as parallel atomic model is provided in Appendix A, Section A.1.

5.3 *Architectural elements:* SC, CM, *and* CC *coupled models*

In this work, we propose three architectural elements that are necessary to build a SA: a simple component, connection mechanism, and composite component. Their structure and interfaces conform to a hierarchical representation, where each one can be modeled as a coupled DEVS with its own internal components.

The simple component in the SA domain is in charge of a set of responsibilities (*In_charge*, Figure 1). The responsibilities are linked among them by a cause–effect relationship, where the fulfilled responsibilities activate other responsibilities by sending requests. Therefore, a coupled model called *SC* is defined to model this structure and behavior (Figure 8), where the relationships between the simple component and their responsibilities can be represented as a hierarchy of DEVS models. The responsibilities, or instances of *RM*, are linked through their input/output ports following cause–effect relationships specified in the SA model.

The input interface of *SC* (Figure 8) propagates the values to the internal components (responsibilities). The interface has a port called *peip* that receives requests ($request_i$) from the previous elements (instances of *SC, CM*, or *CC* according to the SA) and is connected to the input ports of the corresponding responsibilities (instances of *RM*). The requests that arrive at this port indicate that the previous elements have finished their execution to answer to the initial request. The output interface is represented by a set of output ports (Figure 8) and propagates events generated by the components of this model (instances of *RM*) to other simulation elements. *SC* has five output ports: *seop* emits events to the successor elements ($request_i$) from the corresponding responsibility, which is the last in the causal flow; *taop* returns the corresponding turnaround time ($tam_{i,j}$ – metric calculated in each responsibility); *dtop* propagates the emitted values from the *dtop* of each responsibility; *rtop* propagates the emitted values from the *dtop* of each responsibility; and *failsop* propagates the emitted values from the *failop* of each responsibility. The last four ports send measures, which are propagated following the hierarchy of elements to *SAVSM* that sends these values to the EF to calculate the system metrics.

SC has a set of components (D_{SC}), where each component is an instance of *RM* (M_{SC}). The responsibilities are connected using the ports defining the internal couplings (IC_{SC}), which obey the causal relationship between responsibilities. If each responsibility has the possibility of failures, *SC* can also have components that are instances of *FG* (M_{SC}), which are coupled as explained in the previous section. The responsibilities that are part of a simple component compound a set of causal flows. The first responsibilities of that causal flow are connected to input port *peip* of *SC* (EIC_{SC}), the last responsibilities of the causal flow are linked to output port *seop* of *SC*, and all responsibilities are connected to the corresponding output ports that propagate the measures defining the external output couplings (EOC_{SC}).

Another software element is the connection mechanism, which could be included into a SA model depending

on the level of detail that the architect wants to represent in the model. There are many approaches, but the use of this element is not standardized. The connection mechanism (*ConnectionMechanism*, Figure 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. In this work, we have decided to include this element, providing more flexibility in the simulation model, which decouples elements to provide more tools to the architect at the moment of modeling the architecture to be evaluated. The specification follows the same principles written for the simple component, resulting in a coupled model called *CM*. The last architectural element is the composite component (*CompositeComponent*, Figure 1). This element is a component that can contain a set of other simple or composite components. The interface of this component is the same as the simple component, with equal ports. The interface only differs in the type of elements that can be coupled, including instances of *SC, CM* (if it is used to connect the components of the architecture), and other instances of *CC*. Because these two elements are similar to the simple component, we omitted a detailed description and the mathematical definition in this work.

5.4 SA view – simulation model: SAVSM hierarchical model

The SA view has a set of software elements; it is the highest level of abstraction. The view is translated into a hierarchical model and defined as the previous coupled model. This entity represents the simulation model for the SA view (Figure 9). The components of this DEVS are instances of simple components (*SC*), composite components (*CC*), or connection mechanisms (*CM*), where *CC* can have instances of other coupled models (*SC, CM*, or *CC*), *SC*, and *CM* that can contain instances of *RM* (atomic DEVS) or *FG* (atomic DEVS). The components of the simulation model are related through the corresponding couplings creating this complex model (*SAVSM*), which communicates with the environment. The environment is represented under the concept of the experimental frame (*SAEEF* – explained in the next section).

6. Experimental frame for the software architecture evaluation: system environment

The EF represents the environment that interacts with the evaluated system and is used to simulate the dynamic view of software to obtain data produced by the system under specified conditions. Zeigler et al.[17] have suggested three elements: Generator (provides input segments to the system), Transducer (observes and analyzes the system output segments), and Acceptor (monitors experiments to verify the desired conditions). Therefore, the EF for the SA evaluation acts as a stimulus provider, a measurement calculator, and an observer, where these types of elements were adapted to the context of the SA evaluation.

6.1 SAE generator: SAEG atomic model

The generator is in charge of producing events that feed the simulation model (stimulus). The output events represent requests that the users or external systems ask to the simulated software system. The generation of events (requests) is conducted in a random way following a probability distribution. The architect should choose the most appropriate distribution. Formally, the requests generator is an atomic DEVS called *SAEG* specified in Appendix A, Section A.2. To represent the interface (Figure 10), we

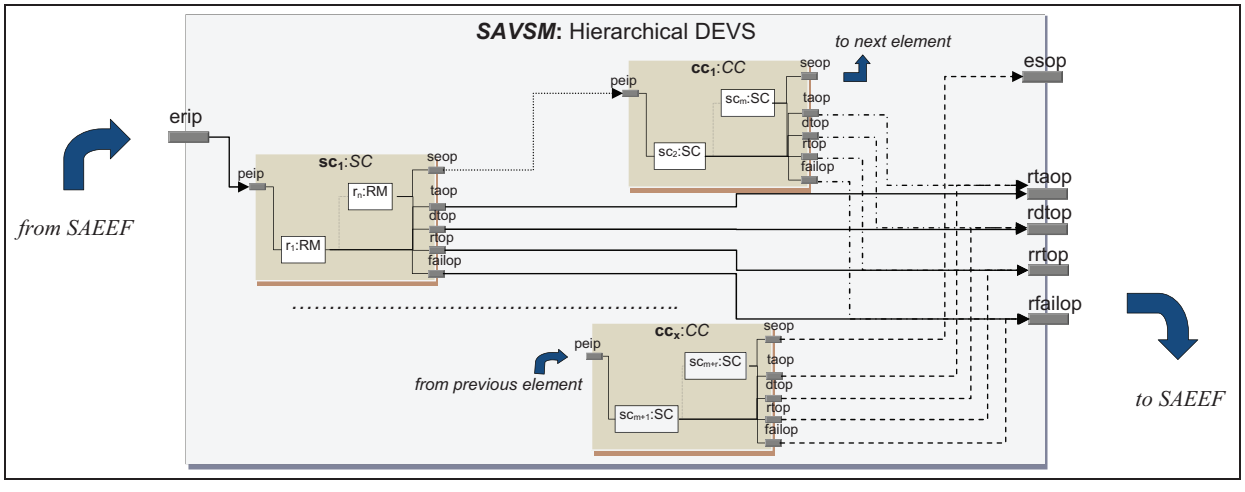

Figure 9. *SAVSM* (SA view), hierarchical discrete event system specification (DEVS) that represents the simulation model.

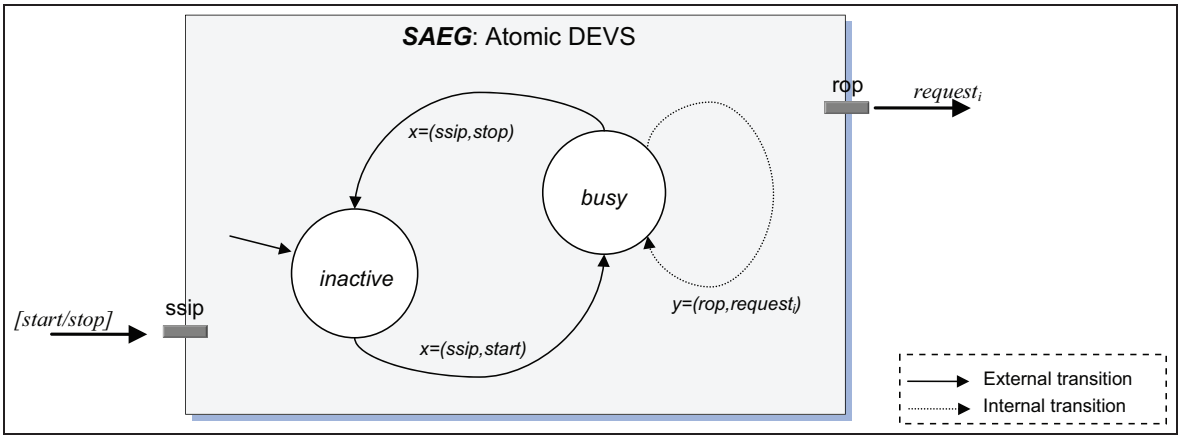

Figure 10. *SAEG*, atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

have defined an input port, *ssip*, that receives the values of a control variable that drives the simulation and an output port (*rop*) from which this element emits the requests ($request_i$) to the system that is being evaluated. Initially, the generator is in phase *inactive*, where it stays until value *start* arrives at *ssip*. Then, *SAEG* passes to *busy* (Figure 10), the phase in which the active system remains until another external event (*stop* value that arrives at *ssip*) indicates the end of the operation, thus returning to state *inactive* again (Figure 10). The internal transitions in *busy* generate the outputs. This element includes the function *timeBetweenRequest*(), which is responsible for calculating the time that elapses between output event generations. This function employs one fixed parameter, *ref_mtbr*, to specify the mean time between failures.

6.2 SAE performance stat calculator: SAEPS *atomic model*

This simulation element calculates metrics related to system performance and is an atomic parallel DEVS called *SAEPS*. The interface and dynamics are formally defined in Appendix A, Section A.3 (Figure 11). This element has a port that receives values of a control variable, *ssip*, and a port, *rtaip*, that receives the turnaround time measure of the responsibility *j* after ''serving'' the request *i* ($tam_{i,j}$). The other two ports, *sentreqip* and *procreqip*, receive the request when it is sent by *SAEG* and after finishing the causal flow in the system (a response was given), respectively. Thus, the corresponding times are verified. *SAEPS* remains in *inactive* until an external event (*start*) arrives and then changes to the phase *calculating* (Figure 11), where it continues receiving inputs from *rtaip, sentreqip*, and *procreqip* until another external event indicates the end of the simulation (*stop* value from *ssip*). At this point, the state changes to *resultant* (Figure 11), and then to *inactive* again. This entity is mainly driven by external transitions, as shown in Figure 11. *SAEPS* first records base measures of each responsibility (adding the values $tam_{i,j}$ to *rtatimes*), sent requests in *srequests* and processed requests in *prequests*, and then returns indicators. An internal transition occurs when this element is in state *resultant*, changing to *inactive* immediately (Figure 11). A port for each system indicator is defined (Figure 11), *staop* and *sthop* to emit *stam* (system turnaround time) and *sthm* (system throughput) (Table 2). These metrics are calculated by functions *computeSystemAvgTa*() and *computeSystemThr*(), respectively. Furthermore, a log tracking with information per responsibility to calculate performance metrics related to each responsibility is maintained (Table 2).

6.3 SAE reliability stat calculator: SAERS *atomic model*

This simulation element uses information related to failures with the purpose of analyzing the reliability of the software. In this way, this element calculates the total failures per responsibility and the total failures of the system. These indicators can show the system weaknesses and the level of reliability. This element is defined as an atomic model called *SAERS* (Figure 12), and its structure and dynamics are similar to *SAEPS* (specified in Appendix A, Section A.4). *SAERS* has a port that receives values of a control variable, *ssip*, and a port that receives the failures from each responsibility when they occur, *rfailip*. This simulation element initiates in *inactive*; when *start* arrives at *ssip*, it changes to *calculating*, where it keeps listening to the failure messages sent by each responsibility of the architectural view (simulation model). *SAERS* creates a register of the failures by adding these values to *rfails*. When *stop* arrives at *ssip*, this element changes to

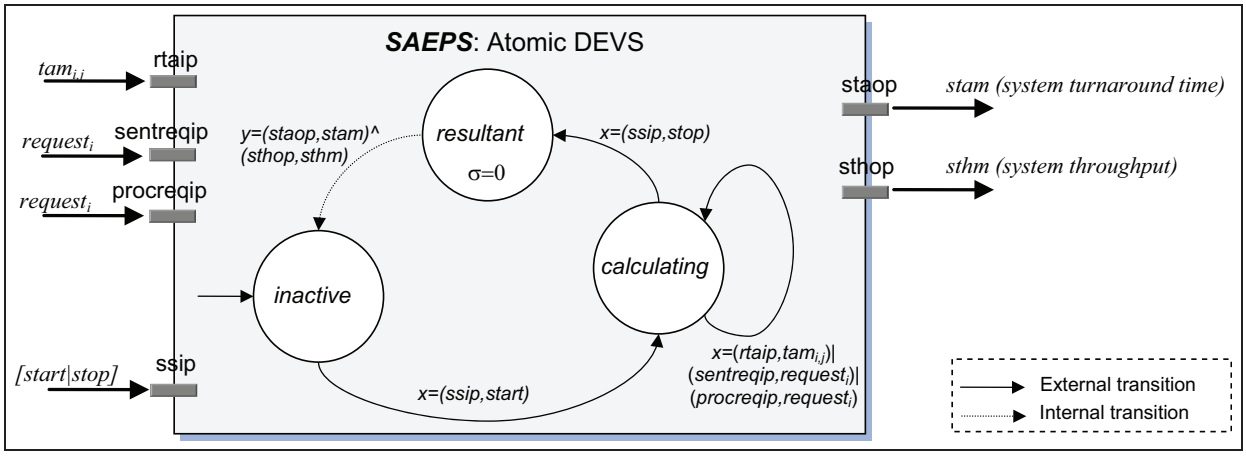

Figure 11. *SAEPS*, atomic DEVS with state transition diagram including the interface, structure, and dynamics.

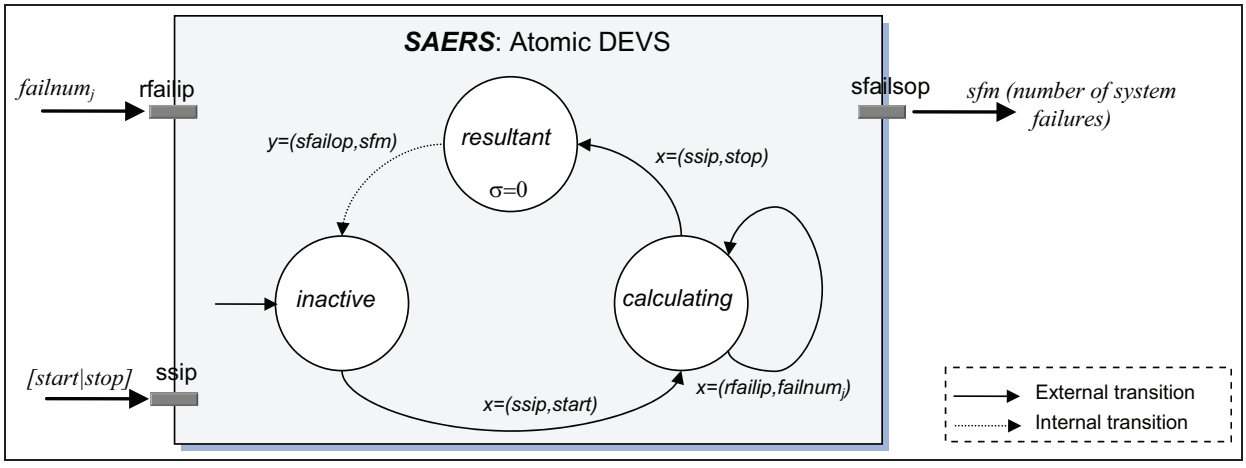

Figure 12. *SAERS*, atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

resultant and then to *inactive* again, returning the system failures (*sfm*, Table 2) using port *sfailsop*. Internally, *SAERS* has a log tracking of the reliability information per responsibility (Table 2).

6.4 *SAE availability stat calculator:* SAEAS *atomic model*

This simulation element takes information about the down-time, uptime, and recovery time of each responsibility to calculate quality indicators that show the time in which the system is available and the time in which it is unavailable. This atomic DEVS is called *SAEAS* (Figure 13, specified in Appendix A, Section A.5). The input interface has ports for receiving values of a control variable, *ssip*, and values for the calculation of availability indicators, *rdowntimeip, rrecovtimeip*, and *rfailip. rdowntimeip* receives values of the time during which a responsibility was not executing due to a failure ($dtime_{k,j}$), *rrecovtimeip* receives the time that a responsibility is needed to recover from a *failed* state ($rtime_{k,j}$), and *rfailip* receives the number of failures that occurred in a responsibility ($failnum_j$). This element works like *SAEPS* and *SAERS* and saves the values in internal records (*rfails, rdowntimes, rrecovtimes*) to be used in the calculations. When *stop* arrives at *ssip*, it changes to *resultant* and then automatically to *inactive* again, returning the indicators using *savailop* and *sunavailop* (system uptime-*sam*- and unavailable time-*suam*, Table 2). Furthermore, this element has a log tracking of availability information per responsibility (Table 2).

6.5 *SAE acceptor:* SAEA *atomic model*

The acceptor controls the beginning and the end of the simulation (Figure 14). The end condition may be set by the architect. In the basic case, the condition is the

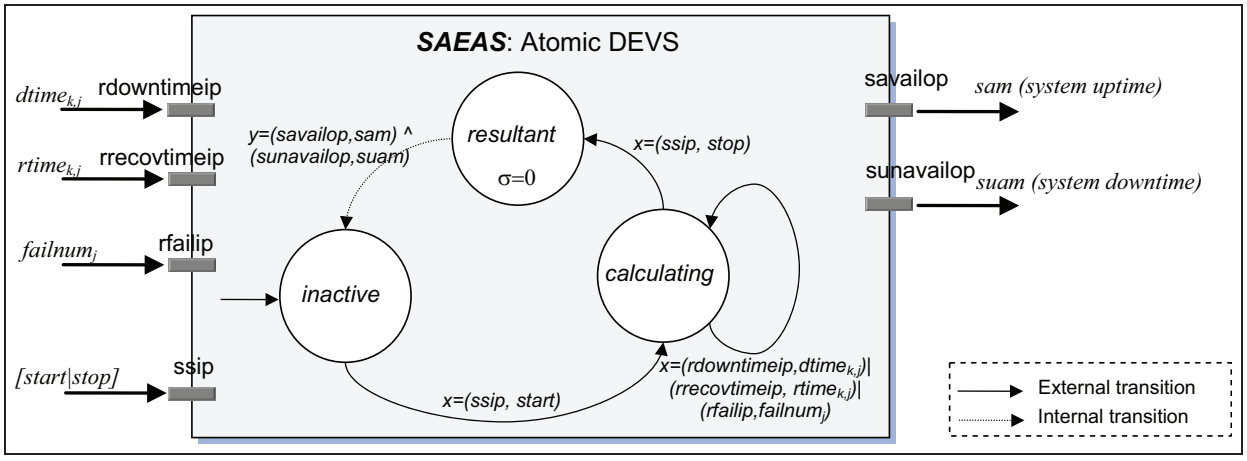

Figure 13. *SAEAS*, atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

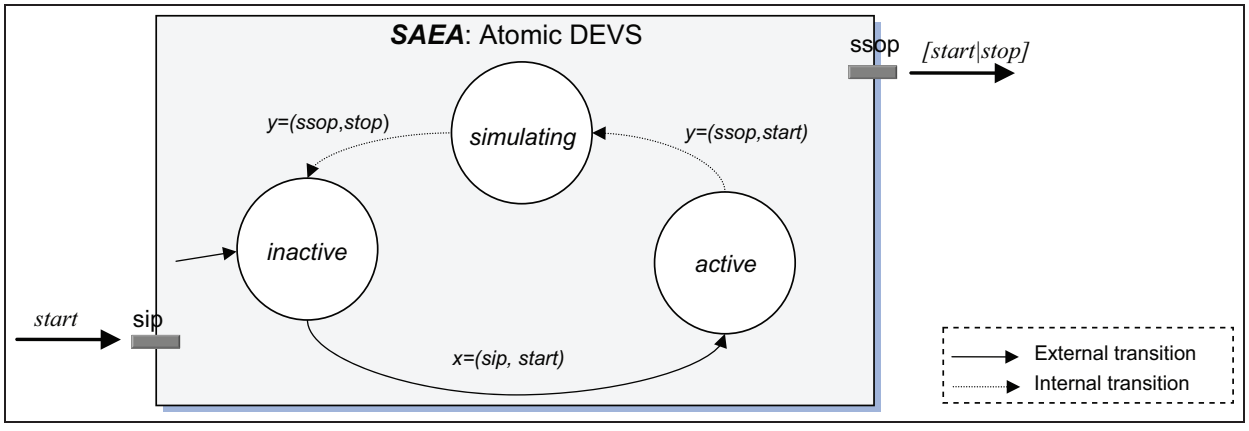

Figure 14. *SAEA*, atomic discrete event system specification (DEVS) with state transition diagram including the interface, structure, and dynamics.

simulation time during which the system behavior is observed; in other cases, system constraints could be built by more complex conditions. We have included this component in the EF to encapsulate the experimental conditions in a simulation element, which provides: flexibility (conditions may be changed or be more complex) and dynamic control (the simulation end can be set by the architect according to the needs). In DEVS, this element is specified as an atomic parallel model called *SAEA* (Appendix A, Section A.6). *SAEA* has an input port, *sip*, which receives *start* to indicate the beginning of the evaluation (simulation). Phase *inactive* is passive, while *active* is a transitory state that indicates the environment is ready for the simulation. Finally, phase *simulating* (Figure 14) indicates that the evaluation is being processed, where the system is observed and the metrics are calculated. Two possible internal transitions can occur: one when the entity immediately passes from the *active* transitory state to *simulating*, and another when the simulation time has elapsed passing from *simulating* to *inactive*. The acceptor emits events to stop the execution when the objectives are achieved in the studied system.

6.6 *SAE experimental frame:* SAEEF *coupled model*

The EF is formally defined as a coupled model called *SAEEF*, where the external interfaces, components and their relationships are shown in Figure 15. *SAEEF* is coupled with *SAVSM* to interact, providing a stimulus to the system and obtaining information that can be useful to analyze the quality of the software.

SAEEF has a set of input ports and values, where the input ports propagate the values to the corresponding components. This simulation element has six input ports (Figure 15): *procreqip* receives the output requests from *esop* (output port of *SAVSM*); *rtaip, rfailip, rdowntimeip*,

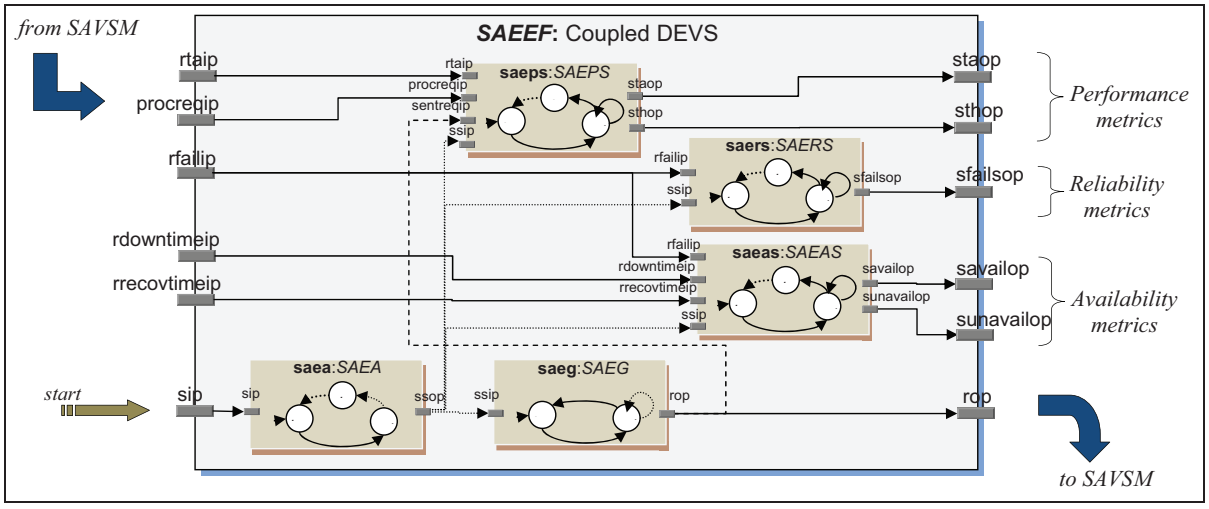

Figure 15. *SAEEF*, coupled discrete event system specification (DEVS) that represents the experimental frame.

and *rrecovetimeip* receive values required to calculate quality metrics from the corresponding ports of *SAVSM* (*rtaop, rfailop, rdtop*, and *rrtop*, respectively); and *sip* receives the initial signal to begin the evaluation. These values are sent to the corresponding components of this model, which are defined in the external input couplings (EIC_{SAEEF}). The set of components that are part of this EF are references to instances of atomic components defined in the previous sections ($M_{SAEEF,d}$, whose components are *saea, saeg, saeps, saers*, and *saeas*, as shown in Figure 15). The relationships among these components are defined as internal couplings (IC_{SAEEF}). As another part of the interface, this DEVS has a set of output ports and values, where the output ports propagate the events generated by the components of the frame to other entities (external or internal to the simulation environment). This element has six output ports (Figure 15): *staop* to emit the system turnaround time; *sthop* to emit system throughput; *sfailsop* to send a message with the total failures of the system; *savailop* to send a message with the total time that the system was available (uptime); *sunavailop* to send a message with the total downtime of the system; and *rop* to send messages ($request_i$) to the system. The external output couplings (EOC_{SAEEF}) specify which element provides the values and from which ports the values are propagated to other entities.

The entire environment has two parts that interact to fulfill the specified goals, as we have described in Section 4, Figure 4. This simulation environment is represented as a hierarchical model called *SAESE* (*SA Evaluation – Simulation Environment*), which includes an instance of *SAVSM* and an instance of *SAEEF*. The specification of interfaces, components, and corresponding couplings are defined in a similar form as previous coupled models, so we have omitted the mathematical specification here.

7. Implementation and testing of the simulation environment

As introduced in Section 4, the proposed simulation environment is modeled in two main conceptual parts: a simulation model for SA evaluation and an EF for SA evaluation. These entities are decoupled from the simulator, which is taken from the DEVSJAVA implementation that encapsulates this part from the others.[33] DEVSJAVA (DEVS-Suite 2.1) is a library that provides a set of packages to implement DEVS models. A modeling package contains class *devs*, which is a superclass of the classes *Atomic* and *Coupled*, and class *Digraph* that inherits from the last class. Furthermore, there are two specialized classes –*ViewableAtomic* (extended from class *Atomic*) and *ViewableDigraph* (extended from class *Digraph*) – which add the user interface to the basic models. Therefore, the proposed simulation elements of the *SA Evaluation – Simulation Environment* (Figure 4), *Software Architecture View – Simulation Model* and *Software Architecture Evaluation – Experimental Frame*, are implemented using this set of libraries and the JAVA programming language. The simplest elements are *responsibilities*, which are atomic DEVS in the simulation model (formal specification); therefore, this concept is implemented as a subclass of the class *ViewableAtomic*. In this way, the internal, external, and confluent transition functions, and the output function are rewritten according to the specification of *RM*, which defines the corresponding parameters. When failures are taken into account, one failure generator is coupled to each *responsibility*. Failure generator is another atomic DEVS (*FG*) implemented as a subclass of the class *ViewableAtomic*. The other elements of the simulation model (*SC, CM*, and *CC*) are more complex structures being coupled and hierarchical DEVS. Therefore,

they are implemented as subclasses of class *ViewableDigraph*, where each one is composed of instances of the corresponding elements. Finally, the SA view (*SAVSM*), which represents the highest level of the SA, is implemented using the class *ViewableDigraph*, which contains instances of the implementations of *SC, CM*, or *CC* with the corresponding couplings. In the EF, the simplest simulation elements (atomic DEVS), such as generator (*SAEG*), acceptor (*SAEA*), and stat calculators (*SAEPS, SAEAS, SAERS*) are implemented as subclasses of the class *ViewableAtomic*. The more complex simulation elements (*SAEEF* and *SAESE*) are implemented as subclasses of the class *ViewableDigraph*.

DEVS-Suite (2.1) provides a Parallel DEVS simulator, which is encapsulated from the model and the EF. In this way, the three parts of the DEVS framework allow for the realization of the simulation. Furthermore, this DEVS-Suite provides support for automating the design of experiments in combination with animated models and generating data trajectories at runtime. This environment allows for the instantiation of the implemented DEVS, the structural validation of the simulation elements, and the dynamics of each element through manual tests. Moreover, this application provides tools for designing and implementing experiments and for testing the ports (interfaces of the models), and provides graphical assistance to visualize the internal changes (states and parameters) and the events that are sent between elements. This tool allowed us to conduct unit tests (each simulation element – atomic) and integration tests (each complex element – coupled) for an internal verification of the operation, and validation tests for external results.

8. Case study

With the purpose of understanding the entire simulation environment for the evaluation of a SA and attempting not to confuse the reader with the complexity of the SA, a clear architecture was selected to illustrate the proposal. The idea was to appreciate how a SA view is translated into an executable model inside of a simulation environment and to focus on the EF, which is an important part to evaluate the software quality.

A real software system, its SA, actual setting, historical data, and expert opinion were used to analyze the simulation model (*SAVSM*) and to validate the results of the simulation. Firstly, we specified the part of the system to be analyzed (subsystem), FRs, NFRs, and quality scenarios. Then we obtained the SA view of the system, its elements, and relationships. The responsibilities were taken from FRs and were assigned to the architectural elements building the UCM. This view was translated into *SAVSM*, the experimental frame (*SAEEF*) was added to the simulation model to build the simulation environment, and the required information (setting parameters) was set to run the simulation.

8.1 Software architecture: application

The studied system is a software management tool commonly used by software factories or other organizations to control their systems licenses. The license manager software (*LM*), provided by a software factory, controls where and how their software products are able to run (clients). This system has a clear architecture, where each element is clearly defined. Moreover, this system is a traditional example of a SA design, where the two main parts are the server (*LMServer*) and client (*LMClient*). Each one has a set of components represented by a pipe and filter pattern whose responsibilities define a causal flow of execution.

The problem was limited to model the main elements required to understand the concepts presented in the simulation model (Figure 16): view, simple and composite components, and responsibility. To have a comprehensible model, connectors were omitted in this example because they only have the responsibility of passing information from one filter to the next. However, they can be easily included in a more detailed simulation model as instances of *CM*, which can be applied as *SC*. The proposed simulation model provides the required elements to introduce these entities in the evaluation for careful study.

The server is focused on the creation of software licenses (license file generator, *LMServer* in Figure 16). This composite element was designed with a pipe and filter pattern, which is applied to represent the elements that participate in this process by implementing the transformation of a sequence of three filter components (*Creator, Codifier*, and *Encryptor*). Component *Creator* takes the input data (organization name, license date, days counter, license time, etc.) needed to generate the software license and creates a plain text file with these data. Component *Codifier* is in charge of generating a unique identifier (hash) that depends on the content of the input file, and saves this identifier with input file information in another text file. Component *Encryptor* takes as input the hash file with the private key and generates the encrypted file with the digital firm that certifies the authentication of this final document. Therefore, the source provides the data for the license and the sink receives a file that includes the data, hash (identifier) and the digital firm (Enterprise). Table 3 presents the responsibilities of the SA components of *LMServer* (*Creator, Codifier*, and *Encryptor*).

The other composite component, *LMClient* (Figure 16), represents the client subsystem, which has three components or filters (*Decryptor, Authenticator, Recorder*). *Decryptor* is in charge of decrypting the received file that was generated by the server component. This is a plain text file that contains license data and a digital firm, which is encrypted with a unique identifier inside (hash). Therefore,

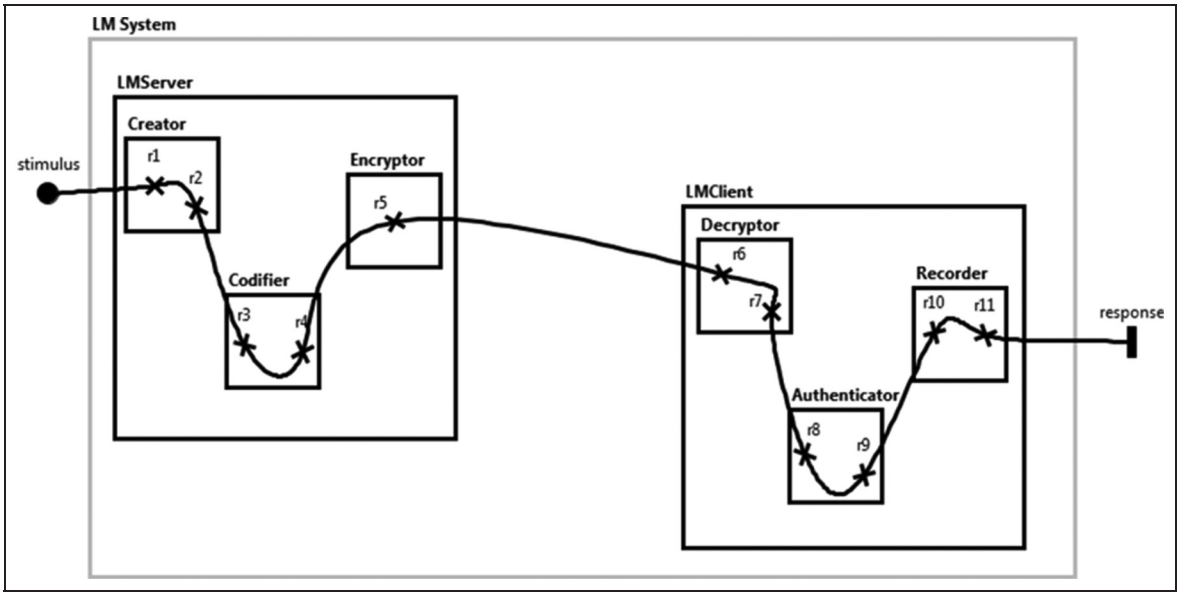

Figure 16. Use Case Map model of the software architecture view of the *LM system*.

Table 3. Elements of the software architecture (SA) of the *LM system* and their assigned responsibilities.

Elements of the SA		Responsibilities	
Composite	Simple	ID	Description
LMServer	*Creator*	*r1*	Receiving the license data.
		r2	Creating a plain text file
	Codifier	*r3*	Generating an unique identifier (hash)
		r4	Saving the identifier and input data in other plain text file
	Encryptor	*r5*	Encrypting the license file
LMClient	*Decryptor*	*r6*	Receiving the license file
		r7	Decrypting the digital firm with the public key (from client side)
	Authenticator	*r8*	Authenticating data comparing the corresponding information
		r9	Sending the data with a report
	Recorder	*r10*	Receiving the authenticated data
		r11	Saving the corresponding data in a DB (information about the correct or error data)

this component decrypts the firm with the public key and sends the file with this information to the following component. *Authenticator* receives the file and the identifier (hash). Then, this component subtracts the identifier (hash) using the content of the file by following an algorithm and then compares these two identifiers. The authenticator then sends the file and the result to the next component. *Recorder* takes the authenticated information (file and result) and updates the license information in the client database if the result is correct; otherwise, this component blocks the user system, when an error in the license data is detected. Table 3 details the responsibilities of these SA elements.

Figure 16 shows the SA view using the UCM notation. This model represents the entities that have a presence at runtime. In particular, architectural components responsible for a ''process'' and their responsibilities, which are extracted from the specified functionalities, represent executable units. The UCM represents the architectural elements with functional aspects, considering not only quality requirements, but also FRs. The responsibilities have causal relationships. Stimulus begins an execution flow and obtains a result as response.

The system is online in a server, waiting for the signal that indicates the system must start the process of validation of the license for a client. An external system controls the dates; if the customer has not paid the licenses, the system blocks the clients using another part of the system in charge of it. However, if the customer has paid the license on time, this subsystem is activated and a request to generate the updated license (license extension) is sent to this subsystem (*LM System*), which then initiates the license validation process.

Source of stimulus: *system/user (external to the system).*
Stimulus: *periodic requests.*
Artifact: *LM system (server and client components).*
Environment: *normal operation.*
Response: *authenticated license.*
Responsemeasure: *turnaround time within 55 seconds, less than 5 per service.*

Figure 17. Example of a performance scenario.

Source of stimulus: *Internal to the system.*
Stimulus: *a responsibility fails to respond to an input (omission).*
Artifact: *LM System (process).*
Environment: *normal operation.*
Response: *failure event (record it and inform).*
Response measure: *no more 5 downtime hours.*

Figure 18. Example of an availability scenario.

The *LM* subsystem is essential to the correct operation of the licenses control system. Thus, it is important to analyze its behavior to find problems that can cause more latency or errors that prevent the correct execution of the key components. Errors might block the systems under control, causing problems to the customers, especially when their systems are critical for the organization. Figures 17 and 18 show two important quality scenarios that the SA must fulfill, which have been specified following the template suggested by the SEI.[3] These scenarios allow architects to delimit the subsystem that will be evaluated, making explicit the portion of the system that is being stimulated. Figure 17 shows a scenario where an external user is a source of stimuli and the artifact is the entire system. This user requests authentication and the system should answer in a specified turnaround time. This situation represents a typical performance scenario. Figure 18 presents an availability scenario that describes a failure in the process of the *LM* system, and the time during which *LM* is unavailable.

8.2 Simulation model and experimental settings

The SA is translated into a discrete event simulation model. Four types of elements are identified in the SA of the designed system: view, composite components, simple components, and responsibilities. Following a bottom-up viewpoint, we transformed the elements from atomic to more complex structures. The smallest architectural units are the responsibilities, so each instance of *Responsibility* is translated into an instance of *RM* in the simulation model (*SAVSM*). Responsibilities are connected using their corresponding ports to represent the causal relationships between them (the UCM model). In this way, these couplings build more complex structures. Each instance of *SimpleComponent* in the SA is represented by an instance of the *SC* in the simulation model, with the assigned responsibilities (instances of *RM*). These coupled DEVS are connected using the corresponding ports to compose another level of complexity, composite components (instances of *CC*) that represent the client and server components of the architecture. Finally, the view of the *LM* system is translated into an instance of *SAVSM* with two instances of *CC* (client and server) as components, which are coupled using the ports. Finally, the concrete simulation model has four levels (Figure 19): a SA view (*LM System-SAView*), 2 composite components (*LMServer, LMClient*), 6 simple components (*Creator, Codifier, Encryptor, Decryptor, Authenticator, Recorder*), and 11 responsibilities (r_j; $j = 1..11$).

The model assumes that a responsibility can fail as we have explained in previous sections. We have extended the simulation model by associating an instance of *FG* to each responsibility (Figure 20).

The quantitative aspects are related to the parameters, which must be set before the simulation based on the study of previous systems behavior, historical data, and software development experiences as follows.

- Execution time of reference (*ref_execution_time*) was defined using empirical data and following historical information from other software systems and components. Due to the features of the responsibility, this time was set following a uniform distribution between some values.
- Recovery time of reference (*ref_recovery_time*) was set to 0 for all responsibilities, assuming that the recovery is automatic. However, if we added additional information about recovery time, we would be able to set it to another value (e.g., sometimes when a failure occurs in a functionality it might need more time to recover to a normal operation after the failure has been repaired).
- Mean time between requests (*ref_mtbr*) was used to generate the requests that are sent to the system by external entities, and the mean value was set assuming an exponential distribution. This time is related to the arrival pattern for external events (requests).
- Mean time between failures (*ref_mtbf*) was used to generate failures of each responsibility and was set assuming an exponential distribution. These values were set by analyzing the complexity and size, which can affect the probability of failure of each responsibility.
- Downtime of reference (*ref_downtime*) was set considering failures that commonly occur in this type of software and the behavior of the time when a failure occurs. We assumed that the time a responsibility can be ''failed'' can vary from 0 to a value

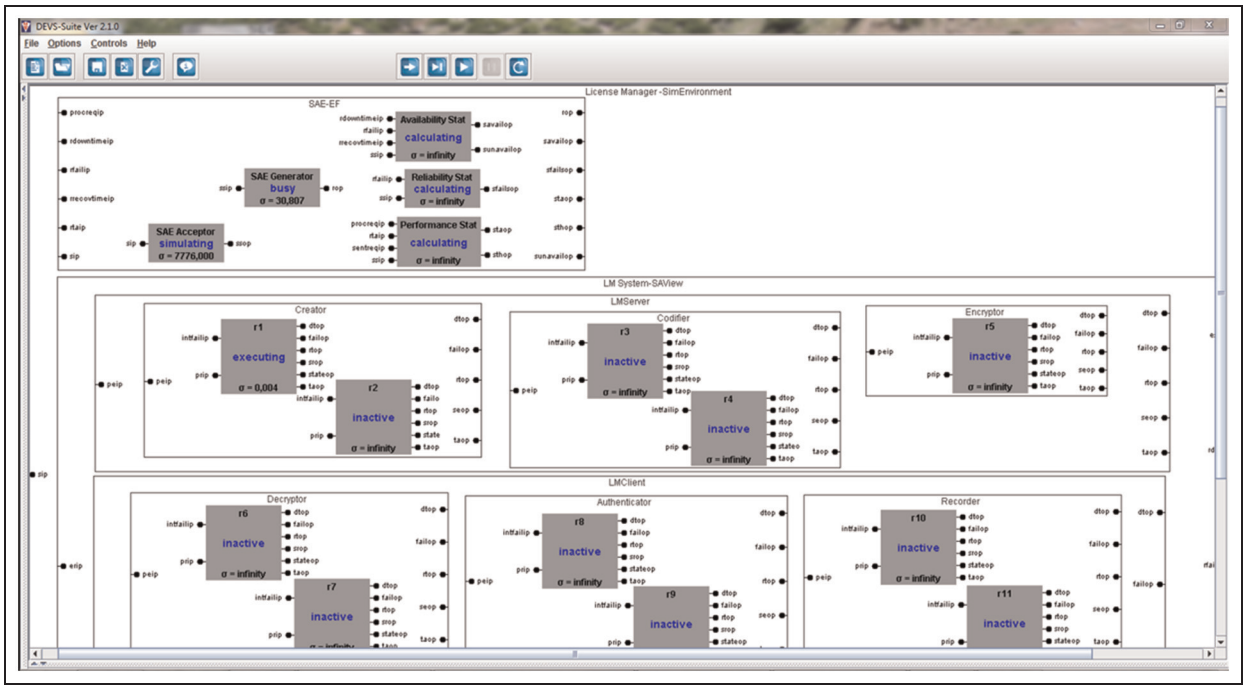

Figure 19. *Simulation Model* for the software architecture of the system (without considering failures).

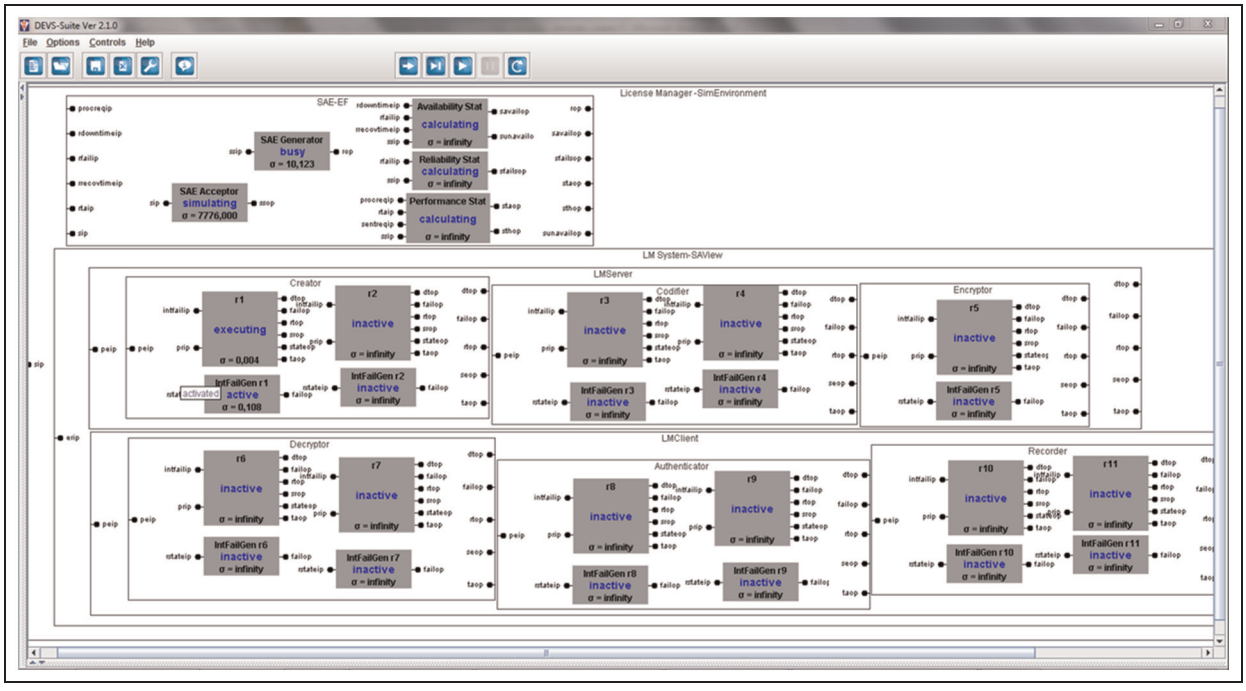

Figure 20. Simulation *Model* for the software architecture of the system considering failures.

that depends on the type of the failure and the complexity and size of the responsibility.

- Simulation time (*sim_time*) was set to three months due to the features of the system.

Users or other loading factors are modeled in the generator in the EF, where we can vary the arrival pattern for the events (request). Therefore, we have assumed that the requests respond to Poisson arrivals; they arrive more

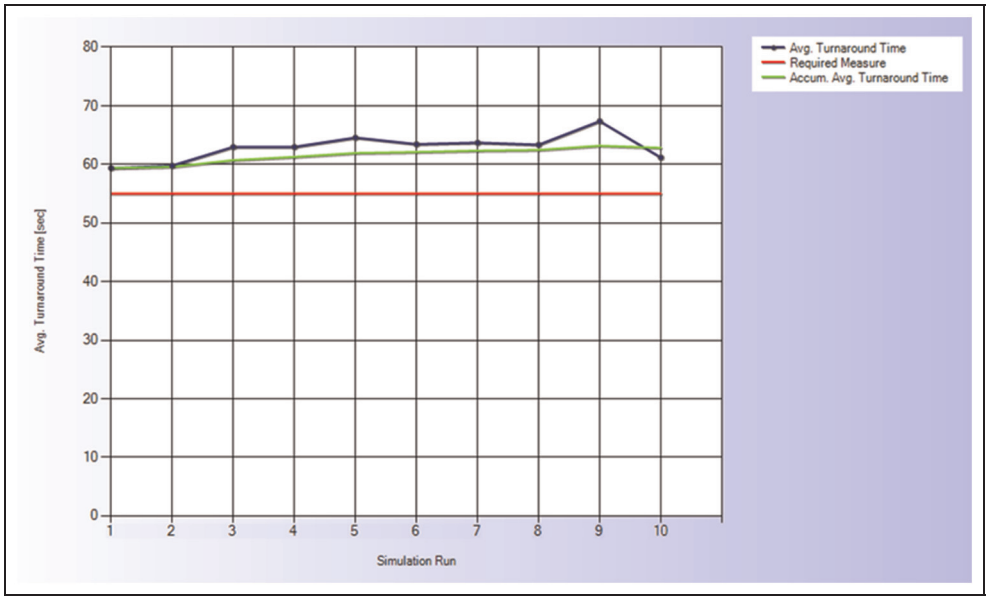

Figure 21. Average turnaround time of the system.

frequently in some periods and the load is lower in others. The load depends on the day, hour, and type of client licenses. Finally, we have made several replications for each simulation run using different seeds for the random generators.

8.3 Simulation results

The simulation of the software system execution provided information that is needed to make decisions about the design of the system. This study focused on two concrete scenarios related to the performance and availability attributes; the simulation shows the behavior of the system and calculates quantitative information to validate these quality attributes and to find the possible causes that can affect the measures. The reports show information not only to validate the quality scenarios specified in the previous section but also information to analyze each responsibility (atomic element). This information helps the architects to determine if the architecture fulfills the quality requirements and which are the critical software elements. System measures are used to decide between different alternatives, with the better situation being architecture whose measures are closer to the response measure of the quality scenario.

The first system measure is the average turnaround time in 3 months, which was 62 seconds. This value indicates the average time that the system needs to emit a response to an external stimulus. In the case of the performance scenario, the stimuli are the requests sent by the users/external systems, during a normal operation of the system, where the turnaround time is greater than the value defined in the performance scenario. Figure 21 shows the value specified in the quality attribute scenario, the average turnaround time of the system for 10 simulation runs and the accumulated average that indicates the tendency of the system turnaround time. As can be seen, the system did not achieve the performance requirement.

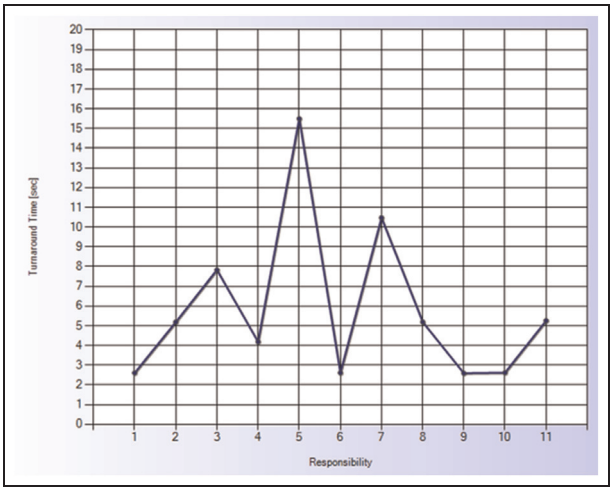

Figure 22. Average turnaround time per responsibility obtained in an execution.

Figure 22 shows a detailed report describing the average turnaround time per responsibility. As can be seen, responsibility 5 was a critical artifact of the SA because its turnaround time was greater than the time for the others. Therefore, responsibility 5 could be a complex functionality. The same situation could be true for *r3* and *r7*, and these responsibilities could also be critical functionalities.

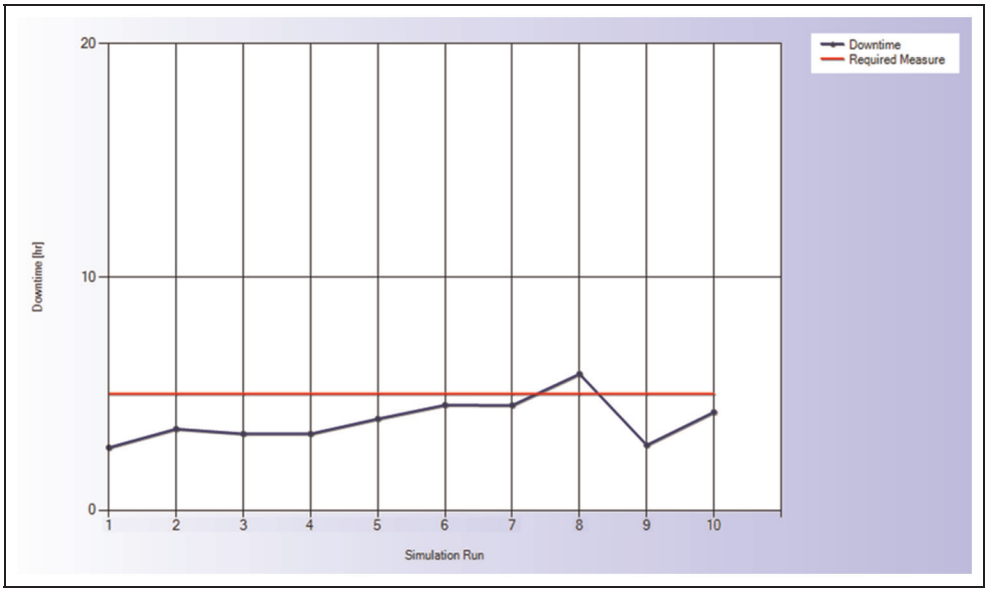

Figure 23. System downtime.

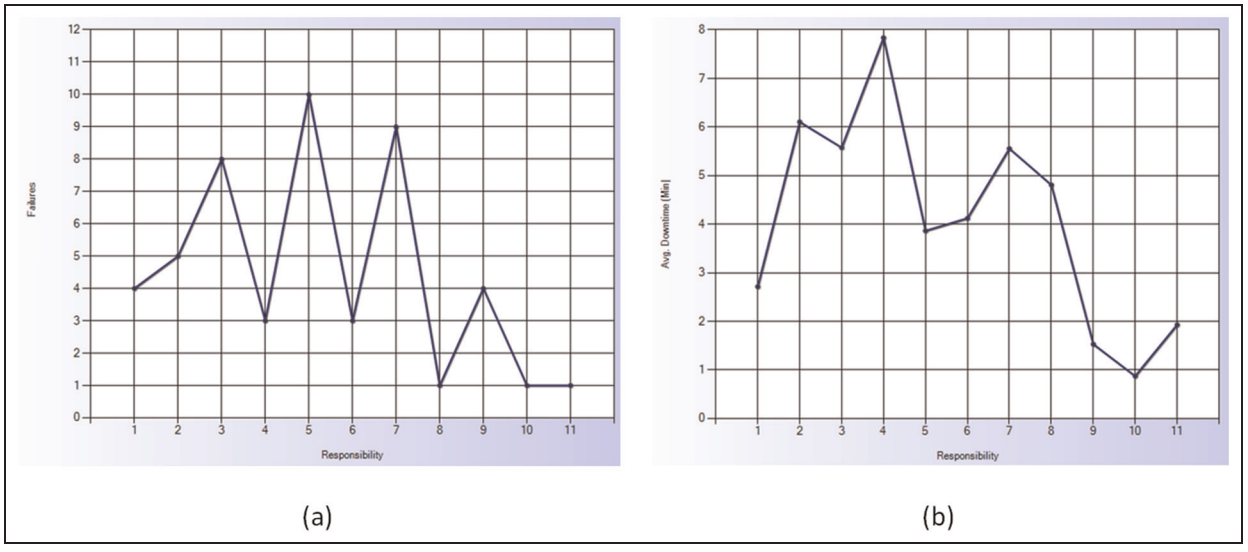

Figure 24. (a) Failures per responsibility obtained in an execution. **(b)** Average downtime per responsibility obtained in an execution.

The other important metric was the average downtime of the system (using total downtime of the system); this value was used to validate the second quality scenario. After several simulation runs, this indicator value was 3 hours and 30 minutes, indicating the system was unavailable for an average of 0.1% of the total time (3 months). This result indicates that the system was not available for this amount of time under the defined conditions, normal operation, and failure probability of each responsibility. Each time a responsibility has a failure, an inform event will be sent as a response of this stimulus. Figure 23 shows the measure defined in the scenario and the values of this metric obtained in 10 simulation runs. As can be seen, the scenario was not achieved in only one simulation run; for the other simulations, the downtime was less than 5 hours over the 3-month period. Other system metrics that can complete the information to analyze the availability scenario include the average number of failures of the system, 37.5 (using the total number of failures of the system after several runs), and the average uptime of the system (using the total uptime of the system after several runs) that provides

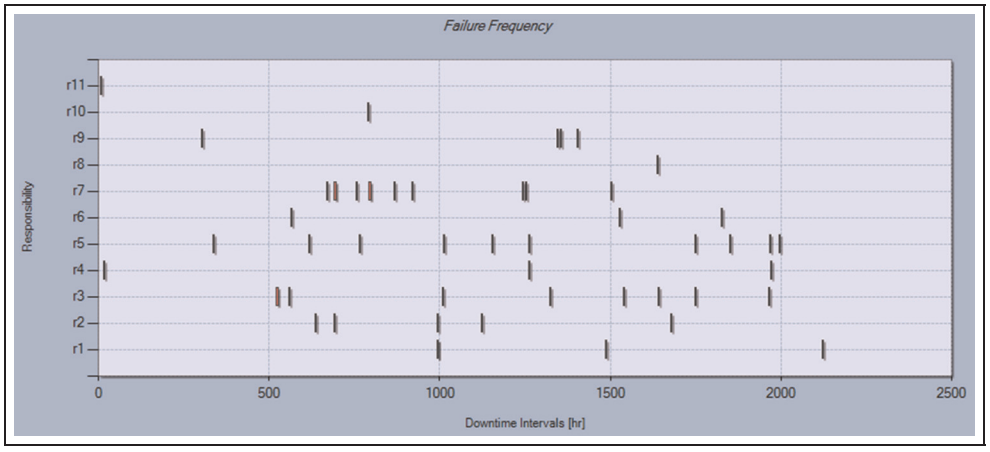

Figure 25. Failure frequency per responsibility and downtime intervals.

information to determine the availability (99% of the total time for this example).

The next reports shows simulation results that can be used to study information related to this scenario, but at the responsibility level. Therefore, we can detect problems in specific elements. The report in Figure 24(a) shows the failures per responsibility that have occurred during the 3 months the system was operating. This information could be useful to find the responsibility that has more problems. Figure 24(b) describes the average downtime per responsibility, indicating the average amount of time (minutes) that each responsibility was ''failed'' because of a failure.

The last report in Figure 25 shows the behavior of the failures during the studied time (3 months of operation) per responsibility and the time intervals during which each responsibility was not available. Note that responsibility 1 had two failures (the first two failures), which occurred in quick succession, making it difficult to visualize the interval between the failures in the figure.

8.4 Discussion

The architect can study the critical elements that might be causing problems in the design from the results of the simulation. In the case study, *r5* and *r7* appeared to be the most critical responsibilities in the model. These responsibilities had a greater response time than the others, and they were more likely to have failures. Therefore, to achieve the specified performance scenario, these responsibilities might be candidates for redesign (e.g., *r5* could be overloaded having the additional functionality of encrypting and sending the license file). Thus, the architect must decide if a tactic or pattern can be applied or divided into two or more responsibilities. Other performance scenarios could be validated in a more detailed study (e.g., to analyze the turnaround time when the system is overloaded). The second scenario (availability) with this design was achieved. The SEI has proposed several approaches to design decisions. The performance tactics help to control the time within which a response is generated.[3] On the other hand, if it was necessary to reduce the failures and downtimes, availability tactics or patterns could be applied.[3,35] Although the specified scenarios are related to two measures (turnaround time and downtime), other scenarios can be added to evaluate the entire system, including a scenario for each new aspect that we want to analyze.

DEVS provides flexibility and scalability advantages with mathematical fundamentals to model the simulation environment and allows us to specify domain-specific simulation elements. Currently, frameworks for the SA evaluation must be adapted to the new requirements of the industry. In this way, there are some features that need to be analyzed.

i) Simulation model expressiveness. The main approaches proposed to evaluate SA in a quantitative form use a low-level notation.[10–13] This low level of abstraction is more limited than notations used in the SA modeling, for example UCM, UML, etc. Although there is an implicit mapping from architectural concepts to the elements of the particular notation, some concepts and features of the SA language are lost in the transformation due to a lack of semantics. Techniques such as Queueing Theory represent components as activity, relationships between components as a call with a probability associated, Markov chain models components as states, and relationships between components as transitions probabilities between states.[21] Therefore, these techniques operate at the component (high granularity level) and do not consider the complex structure and interface where the internal behaviors of the component are not modeled. Moreover, these techniques are focused only on quality aspects and exclude other aspects, such as functional components (the responsibility of each component). In our proposal, the simulation model was close to the domain and followed the elements of the UCM

notation and the SA semantic.[14] The level of abstraction, modularity, and hierarchical representation provided by DEVS provides a potential powerful approach for modeling SAs. The elements of SA and the metrics for the quantitative evaluation are represented in DEVS, which builds domain-specific elements to make structural, behavioral, and quality analysis of the software.

ii) Analysis objectives. Currently, software products must be adapted to the dynamics of the domain where they are applied. Software systems are becoming more complex in order to respond to the new requirements of the organizations that employ these systems. To fulfill these requirements, the software industry needs tools that integrate different perspectives into a more comprehensive evaluation of the designs; these tools must be adaptable to different contexts and quality objectives of an evaluation. Traditional techniques are useful in a particular analysis objective, such as performance[12] or reliability,[10] among others. These approaches often have a model per analysis and include only one quality attribute. However, a few works have integrated multiple quality objectives with a functional view in the same model. The DEVS approach for the SA evaluation has introduced a support for a trade-off analysis that contains different quality objectives and several aspects (functional and non-functional) in the same analysis. Although it is not focused on complex algorithms and metrics, this approach proposes a simulation model that represents the SA concepts with more semantics, thereby obtaining an integral view of the quality, metrics, and functional aspects of the software to give support to the architects to make design decisions.

iii) Scalability. The DEVS framework follows the object-oriented principles to simplify the implementation of the formal specification.[17] While other approaches are focused on specific calculus or models (such as those in Spitznagel and Garlan,[12]Fukuzawa and Saeki,[13] Singh et al.,[21] and Luckham and Vera[27]), DEVS formalism provides fundamentals to build the simulation elements in a modular and hierarchical way, which enables us to suitably represent the concepts of the SA domain that have the same principles. This type of modeling allows the proposal to evolve into more specific or/and complex SAs and quality attribute evaluation. Although this approach considers three quality attributes and basic elements of the SA and UCM notation, both the model and the EF can be extended. New simulation elements can be added to the simulation model, other components of the type ''stat'' could be included into the EF (one for each quality attribute visible at runtime that we need to analyze), or new metrics could be defined to analyze other indicators of the current quality attributes. Due to these considerations, this approach is focused on the model and on how to represent elements of the specific domain of SA evaluation but not on complex algorithms or calculus.

iv) Reusability. The DEVS framework is based on the system theory, so the three basic elements (model, EF, and simulator) and their internal simulation elements can be formulated in the same manner. These elements are manipulated as objects, from simple to complex elements, encapsulating the internal behavior (dynamics) and communicating through their interfaces. These features allow simulation elements to be interchangeable, and the elements can be reused in different contexts and under several conditions of evaluation. Other formalisms, such as Petri Nets,[13] Queuing Theory,[12] or the Markov process,[10] are not modular and are more specific than DEVS; consequently, the reuse of components becomes more complex.

v) Usability. Low-level notations sometimes obscure the original model and the SA semantic, making it potentially difficult for the architects to understand and to work during the system development. In this approach, the simulator encapsulates the complexity of the technique (simulation) and keeps the model in a higher level, while the EF captures the architect's objectives and how they impact the SA model (simulation model). These features and graphical representation tools could make this approach easier to be learned and used by architects.

vi) Input parameters. The model needs the estimation of a set of parameters; more accurate parameters will produce more precise results. These parameters can be better adjusted to the features of the system. Therefore, it is important to have a good record of historical data that must be updated with every new project. In this way, our approach assumes that architects or experts are able to specify these initial parameters. This requirement may represent the main limitation of our approach because this information may not always be available. However, specific information can be estimated from historical data, similar software projects, expert judgment, etc. In the literature, several approaches oriented to one quality attribute (e.g., performance, reliability, etc.) suggest different tools to define this usage mode of the system. For example, Musa suggests the construction of an operational profile from users/customers, system mode, and a functional profile[36] focused on the reliability. Nevertheless, our approach has avoided including very detailed information related to each quality attribute, such as dependencies between individual points of failure (reliability) and hardware resources (performance), to simplify this task.

9. Conclusions and future work

Because most systems do not suitably respond to users' requirements, the early evaluation of software systems is an important issue in software engineering. In this context, software quality plays an important role in system development. Early evaluation of software systems is

challenging; software companies require early information to make decisions that improve designs, thereby reducing development and maintenance costs.

In this paper, a whole simulation environment based on the novel DEVS approach to evaluate SAs is presented. This approach has many advantages: it models the software system with a high level of abstraction, it provides a modular and hierarchical way to build blocks in a simulation model, which naturally fits architectural concepts, and it decouples the simulation model from the simulator, unlike other simulation techniques. Another important aspect is the implementation of the EF as a module decoupled from the other two elements of the frame (simulation model and simulator), which provides a flexible design to specify the conditions under which the system (software) is observed. The importance of the EF has been enhanced here because it implements the quality objectives and represents the environment that interacts with the system, which consequently affects simulation results directly. We also incorporate an element per quality attribute for the calculation of metrics following the quality objectives. Finally, another advantage is the homogeneous representation of all simulation elements, including both model and EF components.

Despite being initially necessary to understand certain mathematical principles involved in DEVS formalism and the underlying M&S framework, adaptability, scalability, and reusability, as well as the automation of the measurement of indicators, are incorporated in the simulation. These advantages provide an easier evolution of the simulation environment to incorporate either new structures (*SAVSM* or *SAEEF*) or new indicators (new metrics in the ''stat'' components in the EF).

Experiments have been automatically executed to test the simulation model that was obtained from a particular SA (structure and behavior using the tools provided by *DEVS-Suite*). In this work, we presented several examples of the information that can be calculated from the elements that compose the architecture, simple and complex metrics from atomic and coupled elements, while possessing the capability of adding specific metrics for specific systems. We believe that the quantitative information (quality metrics) is notably important but has been given scant attention in the software industry. Thus, we plan to perform an exploratory study of particular systems, designing and implementing specific experiments according to different quality goals to validate the model under different situations. Furthermore, this approach can be complemented with other techniques for specific studies of quality attributes.

Several related issues for future work remain open. Firstly, we will formalize the process of transformation following the MDA (*Model-driven Architecture*) framework.[37] Transformation rules must be defined formally following standards that are widely used, such as QVT (*Query, Views, and Transformations*) and MOF (*Meta Object Facility*), as proposed by OMG (*Object Management Group*). In this way, we will introduce the generation of DEVS models from UCM notation in the specific domain of SA evaluation, such as works that apply MDA to DEVS model generation from UML (state machines), but with general purposes.[38,39] Concerning this topic, novel approaches could be analyzed to implement these rules according to the current technology employed in the area of MDD (*Model-driven Development*).[40] Secondly, the use of the framework for modeling, simulation, and DEVS formalism has the advantage of scalability. Thus, other quality attributes that are visible at runtime will be studied, adding components to the proposed EF and incorporating parameters and ports into the elements of the simulation model, thereby enabling the evaluation of other quality aspects and defining new metrics. Moreover, architectural patterns and other UCM elements required to model more complex systems should be studied to include these new dynamic structures in the simulation model.

Funding

This work was supported by the *Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Universidad Tecnológica Nacional* (25/O144 – UTI1748), and *Agencia Nacional de Promoción Científica y Tecnológica* (PAE-PICT 02315).

References

1. Humphrey W. *The software quality profile*. Technical report. Software Engineering Institute, http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=29088 (2009, accessed 3 November 2013).
2. ISO/IEC 9126-1:2001. Software engineering. Product quality. Part 1: quality model.
3. Bass L, Clements P and Kazman R. *Software architecture in practice*. 3rd ed. Westford, MA: Pearson Education, 2012.
4. Hofmeister C, Nord R and Soni D. *Applied software architecture*. Stoughton, MA Addison-Wesley, 2000.
5. ISO/IEC/IEEE 42010:2011. *Systems and software engineering - architecture description, ISO/IEC IEEE*. 1st ed.
6. Eeles P and Cripps P. *The process of software architecting*. Westford, MA: Pearson Education, 2009.
7. Clements P, Kazman R and Klein M. *Evaluating software architectures: methods and case studies*. Westford, MA: Addison-Wesley, 2002.
8. Wojcik R, Bachmann F, Bass L, et al. *Attribute-Driven Design (ADD) version 2.0*. Technical report CMU/SEI-2006-TR-023 ESC-TR-2006-023. SEI, Carnegie Mellon University, USA, http://www.sei.cmu.edu/reports/06tr023.pdf (2006, accessed 3 November 2013).
9. Bass L, Klein M and Bachmann F. *Quality attribute design primitives*. Technical Note CMU/SEI-2000-TN-017. SEI, Carnegie Mellon University, USA, http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=5139 (2000, accessed 3 November 2013).

10. Wang W, Pan D and Chen MH. Architecture-based software reliability modeling. *J Syst Software* 2006; 79: 132–146.
11. Sharma V and Trivedi K. Quantifying software performance, reliability and security: an architecture-based approach. *J Syst Software* 2007; 80: 493-509.
12. Spitznagel B and Garlan D. Architecture-based performance analysis. In: *proceedings of the tenth international conference on software engineering and knowledge engineering*, San Francisco Bay, USA, 1998, pp.146–151.
13. Fukuzawa K and Saeki M. Evaluating Software Architecture by coloured Petri Nets. In: *proceedings of 14th international conference on software engineering and knowledge engineering*, Ischio, Italy, 2002, pp.263-270.
14. Amyot D. Introduction to the user requirement notation: learning by example. *Comput Network* 2003; 42: 285-301.
15. Buhr R. Making behaviour a concrete architectural concept. In *proceedings of the 32nd Hawaii international conference on system sciences*, Island of Maui, Hawaii, 1999, pp.1-5.
16. Amyot D and Mussbacher G. User requirements notation: the first ten years, the next ten years. *J Software* 2011; 6: 747-768.
17. Zeigler BP, Praehofer H and Kim T. *Theory of modeling and simulation–integrating discrete event and continuous complex dynamic systems*. 2nd ed. San Diego, CA: Academic Press, 2000.
18. Bogado V, Gonnet S and Leone H. An approach based on DEVS for evaluating quality attributes. In: *proceedings of the XXIX international conference of Chilean Computer Science Society*, Antofagasta, Chile, 2010, pp.110–118.
19. Bogado V, Gonnet S and Leone H. A discrete event simulation model for the analysis of software quality attributes. *CLEI Electron J* 2011; 14: paper 3, http://www.clei.cl/cleiej/paper.php?id=225 (accessed 3 November 2013).
20. Lyu M. Software reliability engineering: a roadmap. In: *proceedings of the conference of future of software engineering*, Minneapolis, USA, 2007, pp.153–170.
21. Singh LK, Tripathi AK and Vinod G. Software reliability early prediction in architectural design phase: overview and limitations. *J Software Eng Appl* 2011; 4: 181–186.
22. Christensen H and Hansen K. An empirical investigation of architectural prototyping. *J Syst Software* 2010; 83: 133–142.
23. Galster M, Eberlein A and Moussavi M. Early assessment of software architecture qualities. In: *proceedings of the 2nd international conference on research challenges in information science*, Marrakech, Morocco, 2008, pp.81–86.
24. Diaz-Pace A, Kim H, Bass L, et al. Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: *proceedings of the 4th international conference on quality of software-architectures: models and architectures*, Karlsruhe, Germany, 2008, pp.171–188.
25. Bachmann F, Bass L and Klein M. Experience using an expert system to assist an architect in designing for modifiability. In: *proceedings of the fourth working IEEE/IFIP conference on software architecture*, Oslo, Norway, 2004, pp.281–284.
26. Petriu D and Woodside M. Software performance models from system scenarios in use case maps. In: *proceedings of the 12th international conference on computer performance evaluation, modelling techniques and tools*, 2002, pp.141–158.
27. Luckham DC and Vera J. An event-based architecture definition language. *IEEE Trans Software Eng* 1995; 21: 717-734.
28. Ntaimo L, Hu X and Sun Y. DEVS-FIRE: Towards an integrated simulation environment for surface wildfire spread and containment. *Simulation* 2008; 84: 137–156.
29. Byon E, Pérez E, Ding Y, et al. Simulation of wind farm maintenance operations using DEVS. *Simulation* 2011; 87: 1091–1115.
30. Ferayorni AE and Sarjoughian HS. Domain driven simulation modeling for software design. In: *proceedings of the 2007 summer computer simulation conference*, San Diego, USA, 2007, pp.113–121.
31. Wirfs-Brock R and McKean A. *Object design: roles, responsibilities and collaborations*. Boston, MA: Addison-Wesley, 2002.
32. Zeigler B and Sarjoughian H. *Introduction to DEVS modeling and simulation with JAVA: developing component-based simulation models*. Lecture, University of Arizona, USA, 2005.
33. Verbraeck A and Valentin E. Design guidelines for simulation building blocks. In: *proceedings of the 2008 winter simulation conference*, Miami, USA, 2008, pp.923–932.
34. Scott J and Kazman R. *Realizing and refining architectural tactics: availability*. Technical Report CMU/SEI-2009-TR-006. SEI, Carnegie Mellon University, USA, http://repository.cmu.edu/sei/50 (2009, accessed 3 November 2013).
35. Musa J. Operational profiles in software-reliability engineering. *IEEE Software* 1993; 10: 14–32.
36. Diaz M. *Petri nets: fundamental models, verification and applications*. London, UK: ISTE-Wiley, 2009.
37. Kleppe A, Warmer J and Bast W. *MDA explained: the model driven architecture- practice and promise*. Boston, MA: Addison-Wesley, 2003.
38. Mittal S, de la Cruz JM, Risco-Martín JL, et al. eUDEVS: executable UML with DEVS theory of modeling and simulation. *Simulation* 2009; 85: 750–777.
39. Cetinkaya D, Verbraeck A and Seck MD. MDD4MS: a model driven development framework for modeling and simulation. In: *proceedings of the 2011 summer computer simulation conference*, The Hague, Netherlands, 2011, pp.113–121.
40. Sarjoughian HS and Mahmoodi Markid A. EMF-DEVS modeling. In: *proceedings of the 2012 symposium on theory of modeling and simulation - DEVS integrative M&S symposium*, Orlando, USA, 2012, Issue 19.

Author biographies

Verónica Bogado received a PhD degree in Engineering with Information Systems Engineering (2013) from the Universidad Tecnológica Nacional, Facultad Regional Santa Fe. She is currently working at the Department of Information Systems Engineering of the Facultad Regional Villa María, Universidad Tecnológica Nacional. Her current research interests are related to software quality evaluation, software architecture design, and M&S of complex systems, particularly DEVS formalism and its application to software problems.

Silvio Gonnet holds a researcher position at the National Council for Scientific and Technical Research of

Argentina (CONICET), working at ''Instituto de Desarrollo y Diseño'' (INGAR). In addition, he works as an assistant professor at ''Universidad Tecnológica Nacional'' (UTN). He received an Engineering degree in Information Systems from UTN, Santa Fe, Argentina, in 1998, and also obtained his PhD degree in Engineering from ''Universidad Nacional del Litoral'' (UNL) in 2003. His research interests are in models to support the engineering design process, software architectures, and semantic web.

Horacio Leone is Full Professor and Dean of the Department of Information Systems Engineering (Facultad Regional Santa Fe) at Universidad Tecnológica Nacional, where he has been since 1989. He also currently is Research Fellow of the National Council for Scientific and Technical Research (CONICET) of Argentina at the Instituto de Desarrollo y Diseño. He received a Chemical Engineer Degree from Universidad Tecnológica Nacional in 1981. He obtained his PhD in Chemical Engineering from the Universidad Nacional del Litoral in 1986. During 1986–1989 he was a Posdoctoral Fellow at the Laboratory for Intelligent Systems in Process Engineering (LISPE) at Massachusetts Institute of Technology.

His research interests center on improving the understanding of the engineering design process, mainly through the application of different modeling frameworks to diverse engineering domains. In the software architecture design process, he has worked on characterizing the design decisions and the rationale behind them. He has also explored semantic web applications to supply chain information systems and enterprise modeling.

Appendix A: Discrete event system specification of the simulation environment elements

In this section, the mathematical definition of the components of the simulation environment is provided.

A.1. Failure generator: *FG*

The failure generator *FG* is defined as an atomic parallel model as follows:

$$FG = (X_{FG}, Y_{FG}, S_{FG}, \delta_{int,FG}, \delta_{ext,FG}, \delta_{con,FG}, \lambda_{FG}, ta_{FG}) \quad (7)$$

where

$X_{FG} = \{(p,v) \mid p \in FGIP, v \in X_{FG,p}\}$ is the set of input ports and values. $FGIP = \{rstateip\}$ is the set of input ports, $X_{FG,rstateip} = \{activated, finished\}$ is the set of responsibility macro states.

$Y_{FG} = \{(p,v) \mid p \in FGOP, v \in Y_{FG,p}\}$ is the set of output ports and values. $FGOP = \{failop\}$ is the set of output ports, $Y_{FG,failop} = \{fail_k \mid k = 1..q\}$ is the set of failures (k is the failure ID).

$S_{FG} = \{inactive, active\} \times R_0^+$ is the set of sequential states.

External transition function ($\delta_{ext,FG}$):

$\delta_{ext,FG}((inactive, \sigma), e, (rstateip, activated)) = (activated, timebf)$

where $timebf = timeBetweenFail()$ is a function that calculates time between failures following a probability distribution. The expert has to define the parameter *ref_mtbf* used by this function.

$\delta_{ext,FG}((active, \sigma), e, (rstateip, finished)) = (inactive, \infty)$

Internal transition function ($\delta_{int,FG}$):

$\delta_{int,FG}(active, \sigma) = (active, timebf)$

where $timebf = timeBetweenFail()$.

Confluent transition function ($\delta_{con,FG}$):

$\delta_{con,FG}(s, ta_{FG}(s), x) = \delta_{ext,FG}(\delta_{int,FG}(s), 0, x)$

Output function (λ_{FG}):

$\lambda_{FG}(active, \sigma) = (failop, fail_k)$

Time advance function (ta_{FG}):

$ta_{FG}(s) = \sigma$.

A.2. Software architecture evaluation – generator: *SAEG*

The requests generator *SAEG* (*SA Evaluation – Generator*) is defined as an atomic parallel model as follows:

$$SAEG = (X_{SAEG}, Y_{SAEG}, S_{SAEG}, \delta_{ext,SAEG}, \delta_{int,SAEG}, \delta_{con,SAEG}, \lambda_{SAEG}, ta_{SAEG}) \quad (8)$$

where

$X_{SAEG} = \{(p,v) \mid p \in SAEGIP, v \in X_{SAEG,p}\}$ is the set of input ports and values, $SAEGIP = \{ssip\}$ is the set of input ports, $X_{SAEG,ssip} = \{start, stop\}$ is a set of control values of the simulation.

$Y_{SAEG} = \{(p,v) \mid p \in SAEGOP, v \in Y_{SAEG,p}\}$is the set of output ports and values, $SAEGOP = \{rop\}$ is the set of output ports, $Y_{SAEG,rop} = \{request_i \mid i = 1..n\}$.

$S_{SAEG} = \{inactive, busy\} \times R_0{}^+$ is the set of states.

External transition function ($\delta_{ext,SAEG}$):

$\delta_{ext,SAEG}((inactive, \sigma), e, (ssip, start)) = (busy, timebr)$

$\delta_{ext,SAEG}((busy, \sigma), e, (ssip, stop)) = (inactive, \infty)$

where $timebr = timeBetweenRequest()$, function that calculates random numbers that follow a probability distribution given by an expert and using an internal fixed parameter: *ref_mtbr*.

Internal transition function ($\delta_{int,SAEG}$):

$\delta_{int,SAEG}(busy, \sigma) = (busy, timebr)$

where *timebr* = *timeBetweenRequest*().

Confluent transition function ($\delta_{con,SAEG}$):
$\delta_{con,SAEG}(s, ta_{SAEG}(s), x) = \delta_{ext,SAEG}(\delta_{int,SAEG}(s), 0, x)$

Output function (λ_{SAEG}):
$\lambda_{SAEG}(busy, \sigma) = (rop, request_i)$

Time advance function (ta_{SAEG}):
$ta_{SAEG}(s) = \sigma$.

A.3. Software architecture evaluation – performance stat: *SAEPS*

The transducer *SAEPS* (*SA Evaluation – Performance Stat*) is specified as an atomic parallel model as follows:

$$SAEPS = (X_{SAEPS}, Y_{SAEPS}, S_{SAEPS}, \delta_{ext,\,SAEPS}, \delta_{int,\,SAEPS}, \delta_{con,\,SAEPS}, \lambda_{SAEPS}, ta_{SAEPS}) \quad (9)$$

where
$X_{SAEPS} = \{(p,v) \mid p \in SAPSIP, v \in X_{SAEPS,p}\}$ is the set of input ports and values. $SAPSIP = \{rtaip, sentreqip, procreqip, ssip\}$ is the set of input ports, $X_{SAEPS,rtaip} = \{tam_{i,j} \mid i = 1..n, j = 1..m, tam_{i,j} \in R_0^+\}$ is the set of turnaround times for requests i by the responsibility j (m is the number of responsibilities), $X_{SAEPS,sentreqip} = X_{SAEPS,procreqip} = \{request_i \mid i = 1..n\}$ is the set of requests, $X_{SAEPS,ssip} = \{start, stop\}$ is the set of the simulation control values.

$Y_{SAEPS} = \{(p,v) \mid p \in SAPSOP, v \in Y_{SAEPS,p}\}$ is the set of output ports and values. $SAPSOP = \{staop, sthop\}$ is the set of output ports, $Y_{SAEPS,staop} = stam$, $Y_{SAEPS,sthop} = sthm$; $stam, sthm \in R_0^+$, where $stam = computeSystemAvgTa()$ (system turnaround time) and $sthm = computeSystemThr()$ (system throughput).

$S_{SAEPS} = \{inactive, calculating, resultant\} \times R_0^+ \times X_{SAEPS,rtaip}^* \times X_{SAEPS,sentreqip}^* \times X_{SAEPS,procreqip}^*$ is the set of states.

External transition function ($\delta_{ext,SAEPS}$):
$\delta_{ext,SAEPS}((inactive, \sigma, rtatimes, srequests, prequests), e, (ssip, start)) = (calculating, \infty, rtatimes, srequests, prequests)$
$\delta_{ext,SAEPS}((calculating, \sigma, rtatimes, srequests, prequests), e, ((rtaip, tam_{i.j.1}),\ldots,(rtaip, tam_{i,j,c1}))) = (calculating, \infty, add(rtatimes, tam_{i.j}), srequests, prequests)\ \forall tam_{i.j} \in \{tam_{i,j,1}, \ldots, tam_{i,j,c1}\}$
$\delta_{ext,SAEPS}((calculating, \sigma, rtatimes, srequests, prequests), e, (sentreqip, request_i)) = (calculating, \infty, rtatimes, add(srequests, request_i), prequests)$
$\delta_{ext,SAEPS}((calculating, \sigma, rtatimes, srequests, prequests), e, ((sentreqip, request_{i,1}),\ldots, (sentreqip, request_{i,c2}))) = (calculating, \infty, rtatimes, srequests, add(prequests, request_i)\ \forall request_i \in \{request_{i,1}, \ldots, request_{i,c2}\}$
$\delta_{ext,SAEPS}((calculating, \sigma, rtatimes, srequests, prequests), e, (ssip, stop)) = (resultant, 0, rtatimes, srequests, prequests)$

Internal transition function ($\delta_{int,SAEPS}$):
$\delta_{int,SAEPS}(resultant, \sigma, rtatimes, srequests, prequests) = (inactive, \infty, rtatimes, srequests, prequests)$

Confluent transition function ($\delta_{con,SAEPS}$):
$\delta_{con,SAEPS}(s, ta_{SAEPS}(s), x) = \delta_{ext,SAEPS}(\delta_{int,SAEPS}(s), 0, x)$

Output function (λ_{SAEPS}):
$\lambda_{SAEPS}(resultant, \sigma, rtatimes, srequests, prequests) = (staop, stam) \wedge (sthop, sthm)$

Time advance function (ta_{SAEPS}):
$ta_{SAEPS}(s) = \sigma$.

A.4. Software architecture evaluation – reliability stat: *SAERS*

The transducer *SAERS* (*SA Evaluation – Reliability Stat*) is specified as an atomic parallel model as follows:

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

where
$X_{SAERS} = \{(p,v) \mid p \in SARSIP, v \in X_{SAERS,p}\}$ is the set of input ports and values. $SARSIP = \{rfailip, ssip\}$ is the set of input ports, $X_{SAERS,rfailip}\{failnum_j \mid j = 1..m, failnum_j \in Z_0^+\}$ is the set of number of failures in the responsibility j (m is the number of responsibilities), $X_{SAERS,ssip} = \{start, stop\}$ is the set of the simulation control values.

$Y_{SAERS} = \{(p,v) \mid p \in SARSOP, v \in Y_{SAERS,p}\}$ is the set of output ports and values. $SARSOP = \{failsop\}$ is the set of output ports, $Y_{SAERS,sfailsop} = sfm$, $sfm \in Z_0^+$, $sfm = computeSystemFailures()$, being the number of failures for the whole system.

$S_{SAERS} = \{inactive, calculating, resultant\} \times R_0{}^+ \times X_{SAERS,rfailip}^*$ is the set of states.

External transition function ($\delta_{ext,SAERS}$):
$\delta_{ext,SAERS}((inactive, \sigma, rfails), e, (ssip, start)) = (calculating, \infty, rfails)$
$\delta_{ext,SAERS}((calculating, \sigma, rfails), e, ((rfailip, failnum_{j.1}),\ldots,(rfailip, failnum_{j,c3}))) = (calculating, \infty, add(rfails, failnum_j))\ \forall failnum_j \in \{failnum_{j,1},\ldots, failnum_{j,c3}\}$
$\delta_{ext,SAERS}((calculating, \sigma, rfails), e, (ssip, stop)) = (resultant, 0, rfails)$

Internal transition function ($\delta_{int,SAERS}$):
$\delta_{int,SAERS}(resultant, \sigma, rfails) = (inactive, \infty, rfails)$

Confluent transition function ($\delta_{con,SAERS}$):
$\delta_{con,SAERS}(s, ta_{SAERS}(s), x) = \delta_{ext,SAERS}(\delta_{int,SAERS}(s), 0, x)$

Output function (λ_{SAERS}):
$\lambda_{SAERS}(resultant, \sigma, rfails) = (sfailsop, sfm)$

Time advance function (ta_{SAERS}):
$ta_{SAERS}(s) = \sigma$.

A.5. Software architecture evaluation – availability stat: *SAEAS*

The transducer *SAEAS* (*SA Evaluation – Availability Stat*) is specified as an atomic parallel model as follows:

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

where
$X_{SAEAS} = \{(p,v) \mid p \in SAASIP, v \in X_{SAEAS,p}\}$ is the set of input ports and values. $SAASIP = \{rfailip, rdowntimeip, rrecovtimeip, ssip\}$ is the set of input ports, $X_{SAEAS,rfailip} = \{failnum_j \mid j = 1..m, failnum_j \in Z_0^+\}$ being j the responsibility ID, $X_{SAEAS,rdowntimeip} = \{dtime_{k,j} \mid k = 1..q, j = 1..m, dtime_{k,j} \in R_0^+\}$ is the set of times that the responsibility j was failed due to a failure k, $X_{SAEAS,rrecovtimeip} = \{rtime_{k,j} \mid k = 1..q, j = 1..m, rtime_{k,j} \in R_0^+\}$ is the set of times needed by the responsibility j to recover from a failure k, $X_{SAERS,ssip} = \{start, stop\}$ is the set of the simulation control values.
$Y_{SAEAS} = \{(p,v) \mid p \in SAASOP, v \in Y_{SAEAS,p}\}$ is the set of output ports and values. $SAASOP = \{savailop, snoavailop\}$ is the set of output ports, $Y_{SAEAS,savailop} = sam$, $Y_{SAEAS,sunavailop} = suam$; $sam, suam \in R_0^+$ being the calculated availability metrics (system uptime and downtime), where $sam = computeSystemUptime()$ and $suam = computeSystemDowntime()$.

$S_{SAEAS} = \{inactive, calculating, resultant\} \times R_0^+$

$X_{SAEAS,rfailip}^{*} \times X_{SAEAS,rdowntimeip}^{*} \times X_{SAEAS,rrecovtimeip}^{*}$ is the set of states.

External transition function ($\delta_{ext,SAEAS}$):
$\delta_{ext,SAEAS}((inactive, \sigma, rfails, rdowntimes, rrecovtimes), e, (ssip, start)) = (calculating, \infty, rfails, rdowntimes, rrecovtimes)$
$\delta_{ext,SAEAS}((calculating, \sigma, rfails, rdowntimes, rrecovtimes), e, ((rfailip, failnum_{j,1}),\ldots,(rfailip, failnum_{j,c4}))) = (calculating, \infty, add(rfails, failnum_j), rdowntimes, rrecovtimes)$ $\forall failnum_j \in \{failnum_{j,1}, \ldots, failnum_{j,c4}\}$
$\delta_{ext,SAEAS}((calculating, \sigma, rfails, rdowntimes, rrecovtimes), e, ((rdowntimeip, dtime_{k,j,1}),\ldots, (rdowntimeip, dtime_{k,j,c5})) = (calculating, \infty, rfails, add(rdowntimes, dtime_{k,j}), rrecovtimes)$ $\forall dtime_{k,j} \in \{dtime_{k,j,1}, \ldots, dtime_{k,j,c5}\}$
$\delta_{ext,SAEAS}((calculating, \sigma, rfails, rdowntimes, rrecovtimes), e, ((rrecovtimeip, rtime_{k,j,1}),\ldots, (rrecovtimeip, rtime_{k,j,c6})) = (calculating, \infty, rfails, rdowntimes, add(rrecovtimes, rtime_{k,j})$ $\forall rtime_{k,j} \in \{rtime_{k,j,1}, \ldots, rtime_{k,j,c6}\}$
$\delta_{ext,SAEAS}((calculating, \sigma, rfails, rdowntimes, rrecovtimes), e, (ssip, stop)) = (resultant, 0, rfails, rdowntimes, rrecovtimes)$

Internal transition function ($\delta_{int,SAEAS}$):
$\delta_{int,SAEAS}(resultant, \sigma, rfails, rdowntimes, rrecovtimes) = (inactive, \infty, rfails, rdowntimes, rrecovtimes)$

Confluent transition function ($\delta_{con,SAEAS}$):
$\delta_{con,SAEAS}(s, ta_{SAEAS}(s), x) = \delta_{ext,\,SAEAS}(\delta_{int,SAEAS}(s), 0, x)$

Output function (λ_{SAEAS}):
$\lambda_{SAEAS}(resultant, \sigma, rfails, rdowntimes, rrecovtimes) = (savailop, sam) \wedge (snoavailop, suam)$

Time advance function (ta_{SAEAS}):
$ta_{SAEAS}(s) = \sigma$.

A.6. Software architecture evaluation – acceptor: *SAEA*

The acceptor *SAEA* (*SA Evaluation – Acceptor*) is defined as an atomic parallel model as follows:

$$SAEA = (X_{SAEA}, Y_{SAEA}, S_{SAEA}, \delta_{ext,\,SAEA}, \delta_{int,\,SAEA}, \delta_{con,\,SAEA}, \lambda_{SAEA}, ta_{SAEA}) \quad (12)$$

where

$X_{SAEA} = \{(p,v) \mid p \in SAEAIP, v \in X_{SAEA,p}\}$ is the set of input ports and values, $SAEAIP = \{sip\}$ is the set of input ports, $X_{SAEA,sip} = \{start\}$.
$Y_{SAEA} = \{(p,v) \mid p \in SAEAOP, v \in Y_{SAEA,p}\}$ is the set of output ports and values, $SAEAOP = \{ssop\}$ is the set of output ports, $Y_{SAEA,ssop} = \{start, stop\}$ is the set of the simulation control values.

$S_{SAEA} = \{active, simulating, inactive\} \times R_0^+$ is the set of sequential states.

External transition function ($\delta_{ext,SAEA}$):
$\delta_{ext,SAEA}((inactive, \sigma), e, (sip, start)) = (active, 0)$

Internal transition function ($\delta_{int,SAEA}$):
$\delta_{int,SAEA}(active, \sigma) = (simulating, sim\_time)$

where $sim\_time \in R_0^+$ is the simulation time.
$\delta_{int,SAEA}(simulating, \sigma) = (inactive, \infty)$

Confluent transition function ($\delta_{con,\,SAEA}$):
$\delta_{con,SAEA}(s, ta_{SAEA}(s), x) = \delta_{ext,SAEA}(\delta_{int,SAEA}(s), 0, x)$

Output function (λ_{SAEA}):
$\lambda_{SAEA}(active, \sigma) = (ssop, start)$
$\lambda_{SAEA}(simulating, \sigma) = (ssop, stop)$

Time advance function (ta_{SAEA}):
$ta_{SAEA}(s) = \sigma$.