DEVS-based Methodological Framework for Multi-Quality Attribute Evaluation using Software Architectures

Verónica Bogado
CIT Villa María (CONICET-UNVM),
Carlos Pellegrini 211, Villa María,
Córdoba, Argentina
Email:vbogado@frvm.utn.edu.ar

Silvio Gonnet
INGAR (CONICET – UTN)
Avellaneda 3657, S3002 GJC Santa Fe,
Argentina.
Email:sgonnet@santafe-conicet.gob.ar

Horacio Leone
INGAR (CONICET – UTN)
Avellaneda 3657, S3002 GJC Santa Fe,
Argentina.
Email:hleone@santafe-conicet.gob.ar

***Abstract*— Simulation is a powerful tool to evaluate quality attributes of complex software systems, but it is not the most comfortable environment for software engineers. They usually use to specify software architectures graphical notations like Use Case Map (UCM). This paper presents a methodological framework to apply DEVS to the software architecture evaluation problem. This framework is organized in two parts: frontend and backend. The frontend allows engineers to specify software architectures with UCM. The backend uses Discrete Event System Specification formalism (DEVS) to model and simulate the specified architectures. The proposal defines required elements following the Model-driven Development to transform UCMs into domain-specific DEVS models. Engineers can formulate UCMs and generate the corresponding simulation models to evaluate several quality attributes together with the system functionality. This methodology guides developers to a dynamic multi-attribute assessment under different usage profiles, whose results can be used for making early decisions to reduce development and maintenance costs.**

***Index Terms*—Software quality, Software architecture evaluation, DEVS**

I. Introduction

SOFTWARE is transversal to the activities of any organization including companies, government, and society in general. In the last few years, the demand from users and developers for software that delivers greater functionality, reliability, performance, usability, and a lot of other attributes spurs innovation and new paradigms. These challenges require a quick response in terms of methodologies and tools to conceive high quality software. Computer simulation has recently emerged as a way of generating innovation, minimizing costs, and removing boundaries in a wide variety of real-world problems.

The role of software quality becomes fundamental to the software lifecycle [1]. Software Architecture (SA) is a way to achieve quality attributes and to manage risks and costs in complex technological projects. The SA can restrict the level of quality attributes. The evaluation of software architectures implies analyzing different quality attributes. In this process, a distinction can be made between attributes visible at runtime and attributes that are internal focalized in static aspects. Several approaches have proposed to evaluate quality attributes based on quantitative or qualitative analysis [2][3][4][5][6]. However, many of them evaluate isolated attributes, few approaches attempt to provide a trade-off analysis between quality attributes visible at runtime, and in few proposals the software is partially analyzed considering quality without functionality. Furthermore, the methodologies to pass from a SA to a quality evaluation model, e.g., performance model using Queue Nets [5] are not clearly defined making them difficult to be applied.

In this context, aspects that make computer simulation a fundamental tool to innovate and potentially discover unused capability are applied to the software engineering field. For example, in the software architecture evaluation problem particularly, architects can experiment with different configurations of software operation (architectures) before making changes in their designs. Or, architects have the possibility of creating and modifying software behavior using simple indicators or rules related to the software quality and design. Furthermore, architects have practical ways of exploring existing quality problems, discovering unseen element and relationship problems, and improving design problems, and simulators can easily communicate with other systems.

This work presents a methodology based on Discrete Event System Specification (DEVS [7]) to give life to SA specified with Use Case Map (UCM [8]) notation with the purpose of analyzing multi-quality attribute together the system functionality. The proposed methodology defines the required elements (steps, products, metrics, and constraints) to apply DEVS-based software architecture evaluation presented in previous works [9][10] generating the basics to develop a software solution for the architects. For the transformation, the approach follows the Model-driven architecture (MDA [11]) defining rules to translate Use Case Maps into domain-specific

978-1-5386-3057-0/17/$31.00 ©2017 IEEE

DEVS models. Several quality attributes are considered in the methodology, including the definition of the most commonly used quality metrics and indicators. This methodology guides developers to a dynamic evaluation focused on the main challenge of analyzing multiple attributes under the specifications of the system usage profiles, whose results allow architects to make early decisions to reduce time and costs during the development. Other benefits that are consequence of improving quality are the reduction of documentation and maintenance costs, high productivity, and a positive corporate image.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the importance of multi-attribute evaluation. Section 4 explains the DEVS-based methodological framework. Section 5 describes two applications with common architectural patterns. Section 6 presents conclusions and future work.

II. RELATED WORK

A. Software architecture Evaluation

Software Engineering Institute (SEI) has proposed methods based on scenarios and focused on a more qualitative analysis than in quantitative aspects, e.g., ATAM, SAAM [2]. More formal approaches related to a particular quality attribute can be found. For example, proposals based on Markov Decision Process to evaluate system reliability [3][12], or performance and security [4]. Other authors have proposed Queueing Theory to measure performance [5]. Petri Nets have been another important contribution to evaluate different quality attributes such as security, performance, and reliability [6]. These formal techniques make a quantitative study of specific quality attributes with mathematical fundamentals. However, some studies have shown limitations [13] [14]. They are too restrictive from the modeling viewpoint, being difficult to represent some situations of "real-world" systems and they include the complexities of the technique into the quality model, losing SA concepts. Palladio Component Model (PCM) is a complete metamodel for the description of component-based SAs, but it is transformed into [15][16]: a Performance model (analytical solver, queueing networks, prototype, or Java skeletons); and a Reliability model (Markov chain). In the last years, empirical techniques including Simulation and Prototyping have gained an increasing importance because they have a high abstraction level to model real problems. Architectural prototyping requires extra effort and costs to prototype the software at the beginning of the development [17]. The simulation has the advantages of reduced costs and predictive approach.

In all mentioned works, the need of an integrated software model and a more formal methodology to pass from SAs (specification) to the quality models (dynamic model to evaluate the architecture) is clear. In the first case, most approaches take the architecture and its components, but they do not specify functionalities in the model being the analysis focused only on quality requirements. UCM has emerged as a good approach to define behavior in the architectural structures [8]. Since it is an informal notation, it can be combined with formal techniques to obtain a dynamic model that gives life to each UCM element. In the second case, simulation is a promising area to early capture the dynamics of software elements and to evaluate several quality attributes in the same model providing several quality indicators to make design decisions to improve the final product and, in consequence, to reduce problems in the industry, or any other organization, that operates the software.

B. DEVS

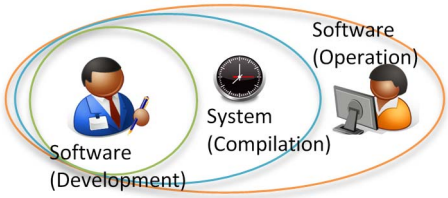
DEVS is a formalism for simulating discrete event system based on a hierarchy of atomic, coupled and hierarchical models, which represent different levels of complexity [7]. DEVS has shown to be a powerful formalism in many domains. This formalism and its framework are an adaptable and scalable simulation approach to tackle the SA evaluation problem. They provide elements to build simple and complex dynamic systems, which keep the semantics of the domain that it is being modeled [7]. Regarding to simulators, there are several tools with some particularity each one. DEVS-Suite and the embedded DEVSJAVA library are written with Java language [18]. DEVS-Suite provides support for implementing parallel DEVS, automating the design of experiments in combination with animating models and generating data trajectories at runtime.

A preliminary version of the SA evaluation simulation model has formalized the basic elements of SA and considered only performance in a simple behavioral flow [9]. In a later work [10], a more complex simulation model that considers not only performance, but also reliability and availability in the same analysis was presented. DEVS specifies timed event systems, where the time is the base upon which all events are ordered and the internal states can transition. Modeling temporal properties is an important issue in the software operation, even more in software quality evaluation from the dynamic viewpoint. These works have focused in the simulation model definition; so, it is important to formalize the methodology to apply the approach and to emphasize in the Experimental Frame (EF) and the multiple quality attribute evaluation.

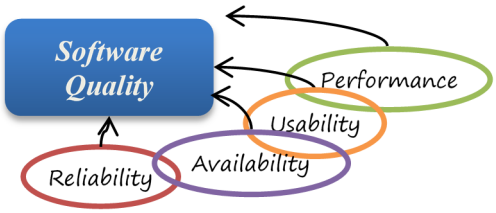
III. MULTI-QUALITY ATTRIBUTE PERSPECTIVE

A variety of qualitative and quantitative techniques were proposed for analyzing specific quality attributes. These techniques have evolved in separate communities, each with its own language and point of view, and have typically been performed in isolation [19]. However, it is known that attribute-specific analyses are inter-reliant, e.g., performance affects usability or availability is related to reliability. The architects lack a set of tools that allows evaluating SA from different perspectives in the same analysis, without a high cost in the building of the quality evaluation model.

Quality attributes can affect the software internally (not visible at runtime, Development in Figure 1(a)) such as modifiability, or externally (visible at runtime, i.e., Compilation or Operation time in Figure 1(a)) such as performance. Evaluating SAs from a dynamic view allows architects to



(a) Internal and external factors or attributes



(b) Internal and external factors or attributes

Fig. 1. Software quality perspectives.

analyze quality attributes visible at runtime. In this proposal, four quality attributes are considered (Figure 1(b)): performance, reliability, availability, and usability, but others can be added as future work.

Two kinds of factors can explain the low quality: i) factors related to the company such as market pressures to deliver software before it is totally completed and costs; ii) technical factors. Thus, the four considered attributes are important in terms of the requirements in most systems. Performance is focused on system responses and timing; it is the most common attribute required in software development. Reliability is concerned with the system failures and availability comprises the lack of reliability considering failures and their effects. Finally, usability involves the user and the easy way of system usage, focalizing on tasks and system support. This attribute is too complex and it is difficult for a measure to be objective. Moreover, there is a repudiation of the magnitude of the usability in the software development, but recently usability has gained an increasing importance due to it is a critical aspect in interactive software [19].

All these attributes have a crucial impact on the SA because solutions that satisfy them require that a big part of the source code must be rewritten, which is highly expensive during the advanced stages of the development. Although functionality and quality attributes are considered orthogonal, any level of any quality attribute is not achieved with any function [19]. So, functionality may be achieved through many possible structures and SA constrains its allocation to structure when a set of attributes is important. UCM includes the concept of scenarios into the SA elements, then in the same model can coexist structural and functional aspects [8].

IV. DEVS-BASED METHODOLOGICAL FRAMEWORK FOR SA EVALUATION

The proposed methodological framework has as main fundamentals UCM notation, DEVS formalism, and MDA for the transformation process. Figure 2 depicts the elements involved in the frontend and backend: activities, input/output products, tools and methodologies, and the role of the architect.

A. Synthesize Software System

Input:

- *FRs* (Functional Requirements): describe the system scenarios and responsibilities of each software component can be defined from them.
- *Quality Requirements*: specify the responses of the software to realize business goals [19], i.e., the quantitative values of each quality attribute. Despite definitions provided for an attribute are not operational, it is necessary to reduce ambiguity, inconsistence, and omissions in the specification of each quality requirement.

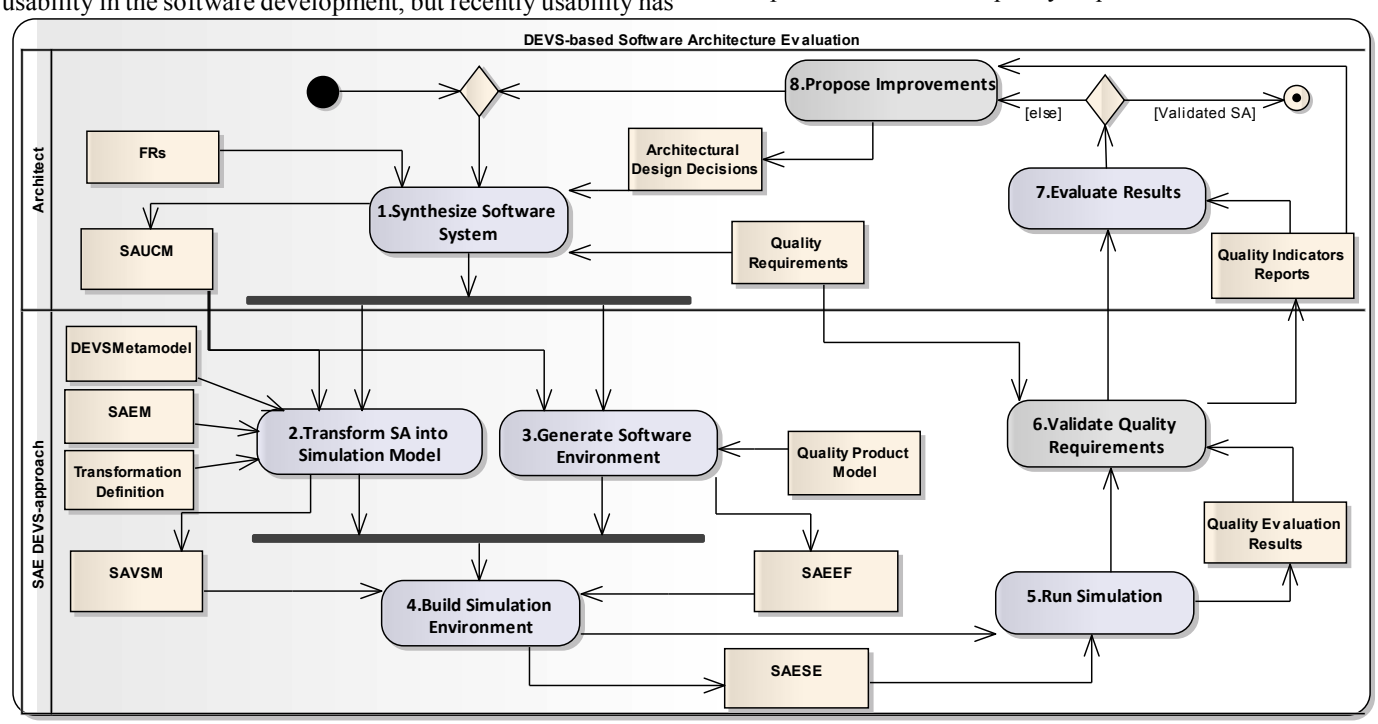


Fig. 2. DEVS-based methodology for a multi-quality attribute software architecture evaluation.

A good solution is to use a template such as Quality Attribute Scenarios [19]. Note that is important to clearly specify the value of each requirement to be validated quantitatively.

- *Architectural Design Decisions*: patterns, tactics, or other strategies used by the architect to achieve specified quality attributes, which build the SA.

Output:

- *SAUCM*: UCM that includes not only the scenarios of the system specified by means of responsibilities, OR Fork/Join, etc.; but also, the SA elements such as components, simple and composite, view, etc. Furthermore, this UCM includes the dynamic model inside by setting some parameters that represent dynamic aspects of the software, which will be evaluated.

Methodologies and tools: this activity requires the application of the UCM notation to represent the SA and the system scenarios [8]. This visual notation allows architects to model complex dynamic systems making the work of analyzing the SA easier. The main reason for choosing this notation is UCM builds functional scenarios together with software structures by means of overlapping paths composed of responsibilities and other elements, all in the same model. Then, the SA assessment has an input with a whole view of the software. UCM is a more natural language for the architect providing a rich set of elements that improves the communication between architects and other stakeholders. Furthermore, there are tools that implement this notation in a development environment commonly used by developers. In this framework, the tool employed is *jUCMNav* [20].

B. Transform SA into Simulation Model

Input:

- *SAEM*: is the *UCM Metamodel* (Figure 3) that captures the main concepts of UCM notation in the context of SA [10]. This notation is used to create the input models in a visual language that is comfortable to the architect.
- *DEVS Metamodel* (Figure 3): captures the basic concepts of DEVS language. This work uses Atomic and Coupled Parallel DEVS because these models have multiple ports to receive values at the same time (bag of inputs), allowing all imminent components to be activated and send outputs to other components [7]. DEVS models capture the software element dynamics driven by quality requirements.
- Transformation Definition: specifies the set of rules needed to translate the software elements (view, components, responsibilities, etc.) into simulation elements (atomic and coupled DEVS). The set of main rules are defined in a previous work [10].
- *SAUCM*: is the *UCM Model* (Figure 3) conforms to *UCM Metamodel*.

Output:

- *SAVSM*: simulation model, which is a hierarchy of DEVS models that represents the SA view (*SAUCM*), i. e., the software whose quality is under evaluation.

Methodologies and tools: MDA framework is applied to translate software elements into DEVS elements. Two metamodels (UCM and DEVS) and a set of rules of transformation are defined at Metamodel level. This metamodels are specified in *Ecore* to be directly implemented. The defined rules are implemented using ATL [21] developing the module *UCM2DEVS*. ATL is a means to produce a number of target models (*DEVS Model* in Figure 3) from different source models (*UCM Model* in Figure 3).

At model level, the UCM input model is conformed to *UCM Metamodel* and saved as a XMI file using *jUCMNav* tool. This file is taken by the *UCM2DEVS* module, which processes the rules and generates an XMI output file that is conformed to *DEVS Metamodel*. This file contains the DEVS models according to *SAVSM* specification. This model is integrated in the next activity manually in DEVS-Suite.

The simulation model (*SAVSM*) is one part of the output DEVS model and represents the view of the software, where its components can be simple and composite components and different elements that represent the system scenarios. The main unit of construction is the responsibility and it is the basic building block to represent the paths and architectural elements. It is defined as a coupled DEVS called *CPXRES* (Figure 4), which encapsulates two internal components that represent the actual functionality (r:*RM*) and the wrong behavior caused by failures (fg:*FG*). The first component is the main part affected by other elements, which represent aspects of the responsibility behavior that could modify its normal operation. Failures are considered to evaluate reliability, availability, and usability, but new components could be coupled to *RM* in order to model more complex behavioral aspects related to other quality attributes visible at runtime.

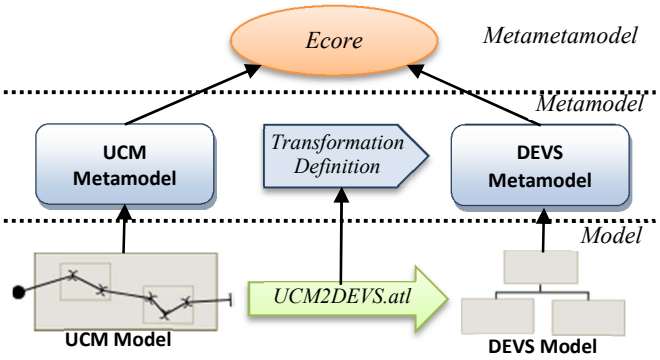


Fig. 3. MDA for DEVS-based software architecture evaluation.

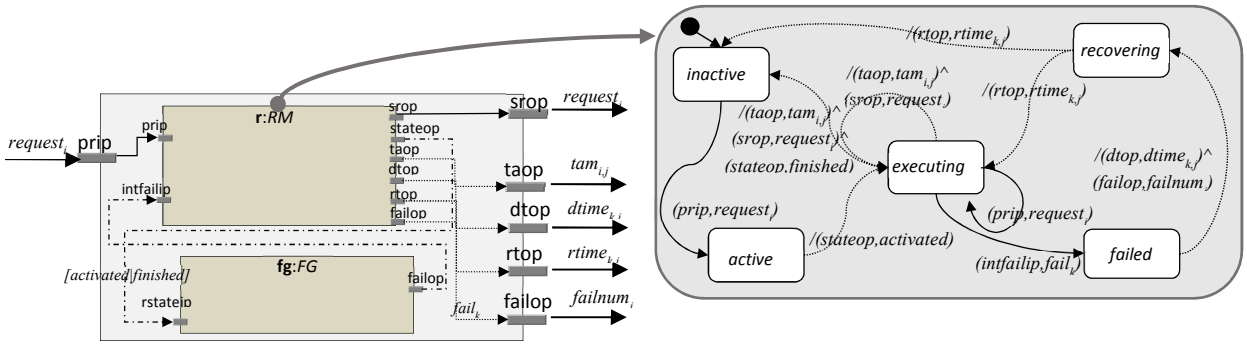


Fig. 4. Responsibility: Coupled DEVS.

Each atomic component has an interface input and output ports as it can be seen in Figure 4, and internal behavior defined by a set of states, transitions, and internal functions. Figure 4 illustrates the *RM* dynamics, where it starts in inactive and passes to 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 responsibility cannot be in normal operation, it changes to failed. Then, it moves automatically to recovering, which represents the recovery process that might be required when a failure interrupts the execution.

The other UCM concepts are mapping to atomic and coupled DEVS defined in a similar form to this. Further detailed information about DEVS specification of the simulation model and its elements can be found in [9] and [10].

C. Generate Software Environment

Input:

- *SAUCM*
- Quality Product Model: in this case, this model follows the ISO/IEC 25000:2014 [22] quality model for software products and SEI approach to select the subset of features visible at runtime and the metrics required to analyze them. The proposal is focused on: performance, reliability, availability, and usability.

Output:

- *SAEEF:* experimental frame for the SA evaluation with multiple attributes. It is a hierarchy of DEVS models that represents the software environment and captures the quality metrics for each attribute under evaluation.

Methodologies and tools: selection of the EF (*SAEEF* in Figure 5) according to the quality requirement specification. Some parameters defined in the UCM represent the dynamics of the software operation environment, e.g., the system load generated by the users, and they are the base for the simulation.

A set of metrics and indicators is defined to be included in each component of the EF responsible for each quality attribute considered in the proposal (*SAEPS*: atomic DEVS for performance, *SAERS*: atomic DEVS for reliability, *SAEAS*: atomic DEVS for availability, *SAEUS*: atomic DEVS for usability).

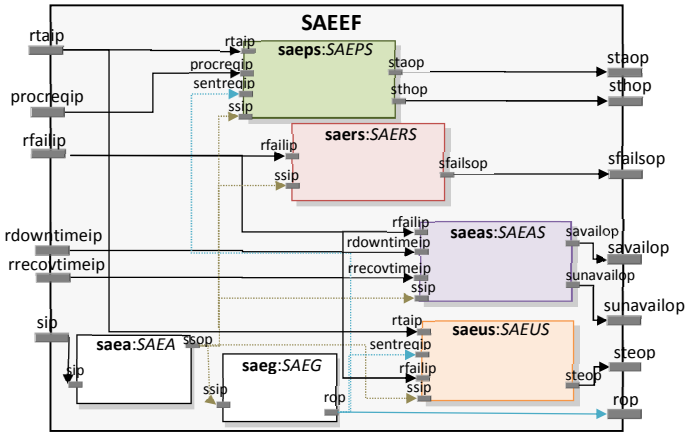


Fig. 5. EF for a multiple quality attributes evaluation.

For performance, reliability, and availability the metrics are defined in previous works [9][10].

The metrics defined for usability consider the turnaround time per request, the request sent by user to be processed (one request per user), the task (or goal represented by a system scenarios), and successful/failure scenario execution. So, we have a measure of time-based efficiency from the system viewpoint, not considering the process of understanding user interfaces because it is not an aim of SA evaluation, but this measure must be complemented with this additional time.

D. Build Simulation Environment

Input:

- *SAVSM*
- *SAEEF*

Output:

- *SAESE: SA Evaluation-Simulation Environment* that includes the simulation model representing the view of the architecture (software that is being evaluated) and the EF capturing the quality goals, metrics and the software environment (dynamics).

Methodologies and tools: the two products generated in the previous steps are integrated into the SA evaluation simulation environment (*SAESE*). This model is loaded manually in DEVS-Suite using DEVSJAVA library [18], step to be

improved in future work to automatize the whole process.

E. Run Simulation

Input:
- *SAESE*

Output:
- *Quality Evaluation Results*: quality measures and indicators obtained in the simulation.

Methodologies and tools: the simulator is executed to give life to the simulation model starting the EF. The request generator (*SAEG*: atomic model, Figure 5) is started to generate the load in the software that is being evaluated.

F. Validate Quality Requirements

Input:
- Quality Evaluation Results
- Quality Requirements

Output:
- *Quality Indicators Reports*: detailed reports of the SA and its elements (responsibilities) highlighting the required values and the quality indicators in relation to them.

Methodologies and tools: evaluation process that provides a set of quality measures (results), which are compared with the specified values in the quality requirements. If the reference values are not achieved, the architecture will not be validated.

G. Evaluate Results

Input:
- Quality Indicators Reports

Output:--

Methodologies and tools: analytical process where the architect must decide if the results fulfill the specified requirements to accept the SA without modifications. If the SA evaluation does not fulfill the initial requirements, or the requirements are adjusted, or new requirements are defined to the system, the architecture has to be evaluated again.

H. Propose Improvements

Input:
- Quality Indicators Reports

Output:
- Architectural Design Decisions

Methodologies and tools: the architect proposes alternatives to improve and evolve the architecture incorporating them in the SA specification.

V. CASE STUDY

Traditional architectural patterns illustrate the proposal, describing how the framework is applied. In this context, two systems are evaluated and the results shows how architects can use this information to make design decisions.

A. Pipe and filter System

The first case is the license manager software (*LM*) provided by a software factory, which controls where and how their software products can be operated.

Step 1. Synthesize Software System

From the analysis of FRs, the main system scenarios and responsibilities of each software component can be defined as follow, identifying the responsibilities of the scenario as r_i (i: 1..*scenario responsibilities*). So, the server (*LMServer* composite component) is focused on the creation of software licenses and is designed with a pipe and filter pattern, which implements the transformation by a sequence of three filters. *Creator* takes (r_1) the input data (organization name, license date, etc.) needed to generate the software license (r_2). *Codifier* is in charge of generating (r_3) a unique identifier (hash) saving (r_4) it with the information of the input file in another text file. *Encryptor* takes as input the hash file with the private key and generates (r_5) the encrypted file with the digital firm that certificates the authentication of this final document.

LMClient applies the pipe and filter pattern with three filters too. *Decryptor* is in charge of receiving (r_6) and decrypting (r_7) the generated file by the server. *Authenticator* receives the file and the identifier (hash) and subtracts the identifier using the content of the file and following an algorithm and compares these two identifiers (r_8), and finally sends (r_9) the data with a report. *Recorder* takes the authenticated information (file and result) (r_{10}) and updates (r_{11}) the license information in the client database if the result is right, otherwise it blocks the user system, in case of an error in the license data.

Two quality requirements are specified using the Quality Attribute Scenarios templates for performance and availability (Figures 6(a) and 6(b) respectively).

QR001: Performance Requirement
Source of **stimulus**: *system/user*
Stimulus: *periodic requests*
Artifact: *LM system*
Environment: *normal operation*
Response: *authenticated license*
Response measure: *turnaround time* within 55 seconds per request.

(a) Example of performance requirement

QR002: Availability Requirement
Source of **stimulus**: *Internal to the system*
Stimulus: *a responsibility fails to respond to an input*
Artifact: *LM System*
Environment: *normal operation*
Response: *failure event (record it and inform)*
Response measure: *no more than 5 hours of downtime (3mo)*

(b)Example of availability requirement

Fig. 6. Concrete Quality Attribute Scenarios.

This software is a pipeline that consists of a chain of filters (architectural components), where the output of each component is the input of the next. In the synthesis of the software, the UCM is built as it can be seen in Figure 7, taking as input the FRs briefly descripted in the previous paragraphs, the architectural patterns selected by the architect, and the quality requirements.

Furthermore, the tool *jUCMNav* allows architect to add metadata to the UCM model; so, quality historical information (means, probability distributions of the execution times, failures, etc.; they are detailed in Table I) related to the responsibilities is defined to be used in the simulation to represent the behavior of this elements.

Steps 2 and 3. Transform SA into Simulation Model and Generate Software Environment

The transformation module (*UCM2DEVS*) is executed to translate the UCM into a simulation model (*SAVSM*), generating a *xmi* file. In this file the system view, composite components such *LMServer* and *LMClient*, simple components such as Creator, Codifier, etc., and each responsibility and causal relationships are represented with the corresponding coupled and atomic DEVS models (such as the illustrated in this paper *CPXRES*) defined in [9][10]. The EF with three atomic components for evaluating performance and availability (related to reliability) is selected and coupled with the simulation model of the architecture.

TABLE I
INPUT PARAMETERS PER RESPONSIBILITY

Resp ID	Execution Time (sec)	Failure Prob. (hr/mo)	Downtime Mean (hr)
1	Uniform [0.1-5]	Poisson: 2	Uniform [0.000278-1]
2	Uniform [0.1-10]	Poisson: 3	Uniform [0.000278-3]
3	Uniform [0.1-15]	Poisson: 5	Uniform [0.000278-8]
4	Uniform [0.1-8]	Poisson: 2.4	Uniform [0.000278-1]
5	Uniform [0.1-30]	Poisson: 10	Uniform [0.000278-16]
6	Uniform [0.1-5]	Poisson: 2	Uniform [0.000278-1]
7	Uniform [0.1-20]	Poisson: 6	Uniform [0.000278-8]
8	Uniform [0.1-10]	Poisson: 3	Uniform [0.000278-3]
9	Uniform [0.1-5]	Poisson: 2	Uniform [0.000278-1]
10	Uniform [0.1-5]	Poisson: 2	Uniform [0.000278-1]
11	Uniform [0.1-10]	Poisson: 3	Uniform [0.000278-3]

Step 4. Build Simulation Environment

The integration of the two parts, simulation model and EF, results in the simulation environment in DEVS-Suite in Figure 7, where the simulation model and experimental framework components are implemented using Java Language and DEVSJAVA libraries. The simulation environment is integrated in DEVS-Suite, which provides the simulator to give life to the simulation components in DEVS.

Steps 5 and 6. Run Simulation and Validate Quality Requirements

The data required to set the parameters for the simulation is taken from the historical information of the development of the system. Part of this historical data is considered to validate the outputs of the simulation, i.e., results of the software architecture evaluation. The simulation is run for 3 months of operation of the system. After running the simulation, quality evaluation results are processed validating the quality requirements; then, the indicator reports are generated. The Figures 8 show a report per quality requirement, where the specified value is the red line and the values of the software operation (simulation) for 10 runs with different seeds are in blue or green depending on the measure (System Turnaround Time and Downtime respectively).

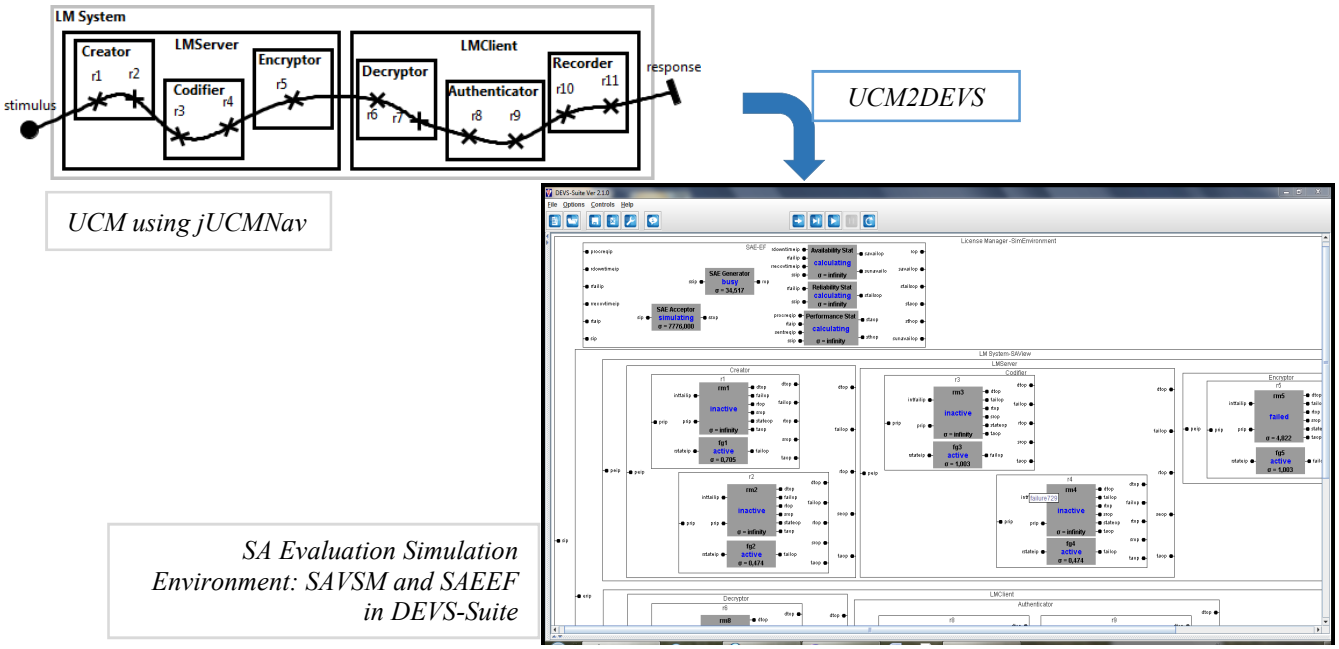
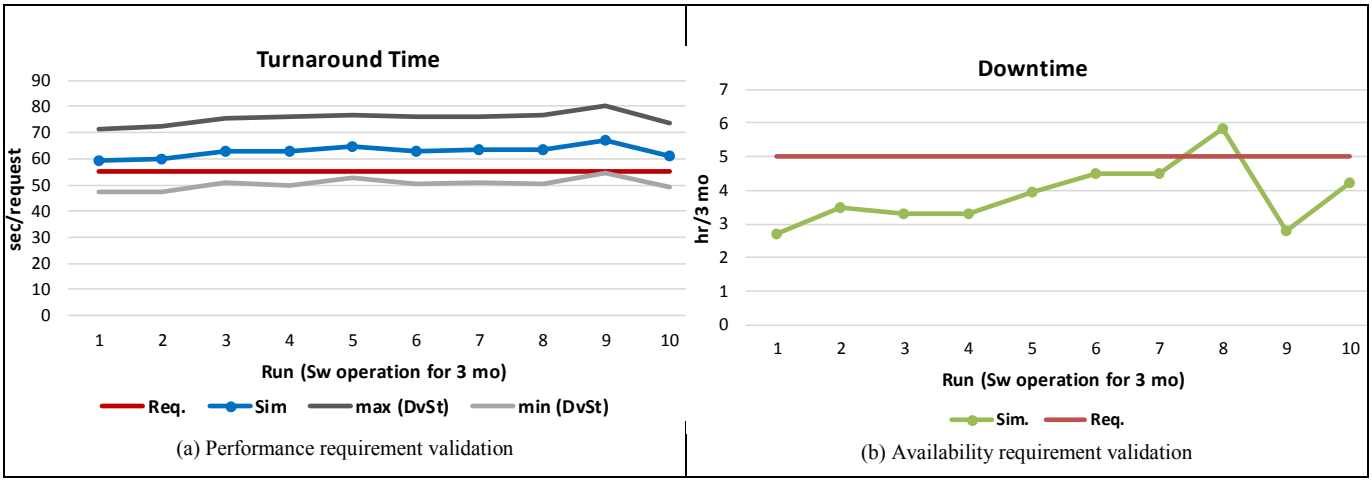


Fig. 7. UCM: pipe and filter pattern and the corresponding DEVS model.



(a) Performance requirement validation

(b) Availability requirement validation

Fig. 8. Example of a Performance and Availability indicator reports.

The variability of the System Turnaround Time is defined with the grey lines in Figure 8(a), considering the standard deviation calculated from the turnaround time values obtained during the simulation of the software operation (requests sent to the system) in each run.

Steps 7 and 8. Evaluate Results and Propose Improvements

The quality indicator reports (Figures 8(a) and 8(b)) show that the specified quality requirement related to performance is no achieved with this architecture and the assignment of responsibilities in the components. But, the availability requirement is fulfilled. So, the architect must decide whether or not to apply some architectural patterns that improve the performance of the system, or another architectural decision. Otherwise, the architect can decide not to modify the design following some criterion.

It is important to note the simulation time was 2 min in average, executing on an individual notebook (Intel Core i5, 8 GB RAM) to represent a typical operation environment for an architect during the development of the software.

B. *Client-Server System*

This second case illustrates a Client-Server pattern in a web software for ticket reservation (*TRSystem)*.

Step 1. Synthesize Software System

The causal flow starts when the user or other external entity sends a request to the client component (*Client*) to demand a connection to the server (*TRSServer*). The server has four components. *Webserver* confirms the connection to client (r_1) opening a new session and selecting one of the three options: to browse and check information about an event (r_2), to buy a ticket via on line beginning a transaction (r_3), or to close the session and release the resources (r_9). With the first option, it sends the request to the database server (r_2) to retrieve the information required by the user (r_6). *Netware* is a mediator between the web server and database server; it forwards the request from the first one to the database (r_4).

Finally, the system gives a response displaying the corresponding information to the user. With the second option, the user sends a request to the web server to confirm the purchase (r_3), which requires to verify the credit card (r_7). So, it sends the request to the credit card verification component (*CCVerificationServer*) by the *Netware*, which forwards the request (r_5). Once the credit card is checked, the ticket purchase request is sent to database server to update its records with the ticket purchase information (r_8).

In the software architecture view, a composite component (*TRServer*), and five simple components (*Client*, *WebServer*, *Netware*, *CCVerificationServer*, *DBServer*) with their responsibilities are identified from the FRs.

In this case, two quality attributes are important, performance and usability. So, two requirements are specified as examples of the application of the methodology, one for each attribute (Figures 9(a) and 9(b)).

In this SA, the architect decides to apply a Client-Server pattern. The UCM is built using *jUCMNav* tool as it can be seen in Figure 10, where the path that represents the functional scenario of this system overlaps the software components identified for this view of the system.

QR001: Performance Requirement
Source of **stimulus**: *client subsystem*
Stimulus: *requests*
Artifact: *TRSServer*
Environment: *normal operation*
Response: *displayed information | purchased ticket*
Response measure: *turnaround time within 30 seconds*

(a) Example of performance requirement

QR002: Usability Requirement
Source of **stimulus**: *User*
Stimulus: *wants to display/purchase a ticket (task)*
Artifact: *System*
Environment: *at runtime*
Response: *displayed information | purchased ticket*
Response measure: *less than 1 task per 60 seconds*

(b) Example of usability requirement

Fig. 9. Concrete Quality Attribute Scenario.

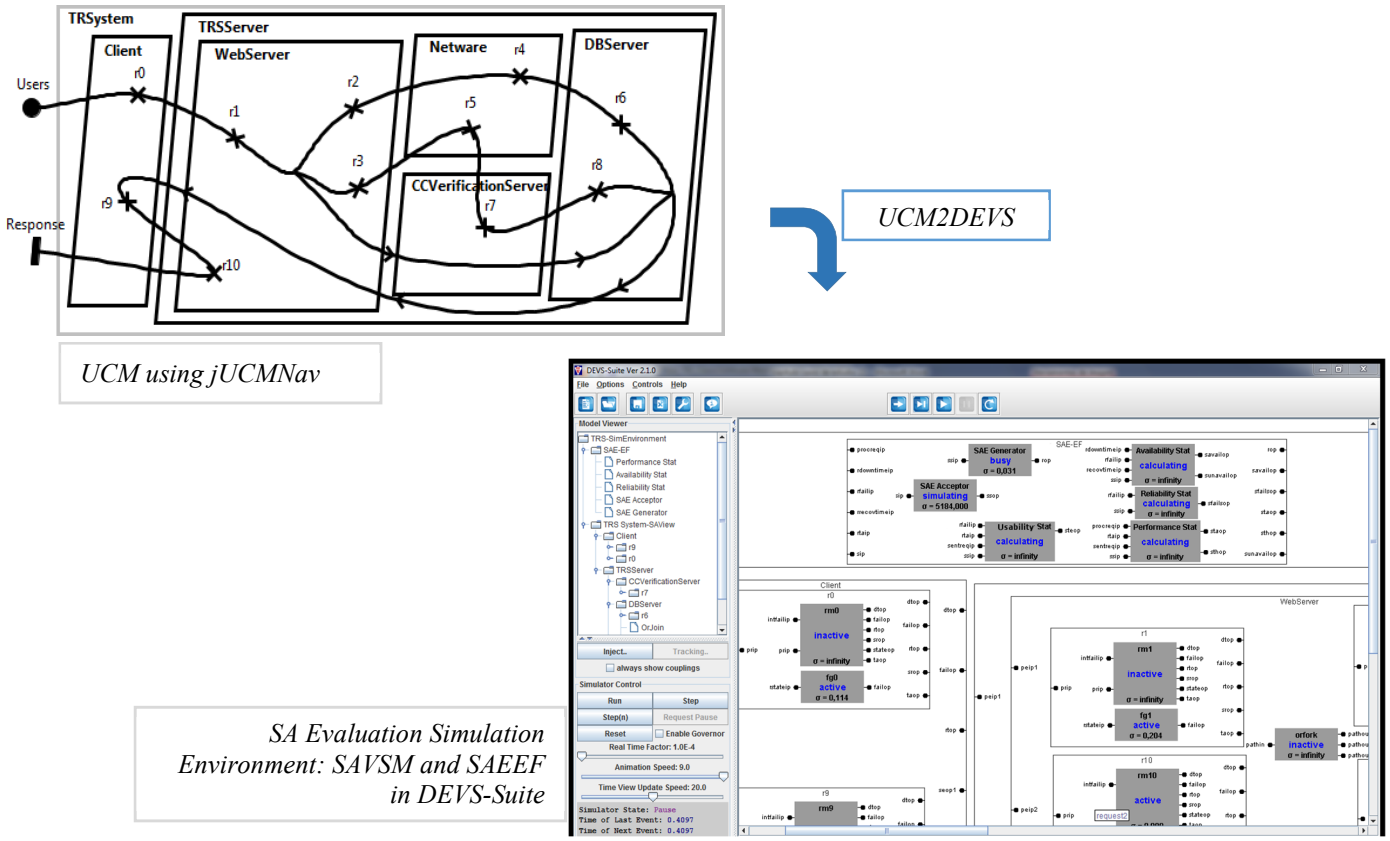


Fig. 10. UCM: client-server pattern and the corresponding DEVS model.

Some metadata required to simulate the behavior of each responsibility from the quality viewpoint are set in this step too, using reported information of similar functionality from other projects. For each responsibility, the same probability distribution set in the first case was used for the execution time and downtime, and the probability of failure occurrence is defined assuming Poisson distribution where the time between failures responds to an exponential distribution. The reference limits and mean rates were calculated from previous studies as in the first case for each responsibility (functionality).

Steps 2, 3 and 4. Transform SA into Simulation Model, Generate Software Environment, and Build Simulation Environment

Figure 10 shows the UCM for this system, including the architectural pattern and other components and the ticket reservation scenario. This UCM is mapped to DEVS hierarchy in DEVS-Suite (Figure 10) that represents the model and EF with four atomic components: performance, usability that requires measures that are related to reliability and availability.

Steps 5 and 6. Run Simulation and Validate Quality Requirements

The software operation is simulated for 2 months, time required to evaluate this kind of system.

The data required to set the parameters for this case is taken from reports of measures of similar systems, which apply this pattern. The simulation implies several executions, 10 runs with different seeds, obtaining quality information of the system to validate the requirements. Table II details the quality indicators obtained during the simulation and other measures that complete the analysis. The indicators are highlighted (light grey) because the corresponding requirements are achieved. Detailed information is omitted here, but with the proposal the architect can obtain reports of responsibility quality indicators or measures.

Steps 7 and 8. Evaluate Results and Propose Improvements

In this case, the quality indicator report is presented as a summary in Table II, where the architecture for the validated functional scenario is fulfilled. So, the architect can use this architecture to start the detailed design of the system. Other functional scenarios can be validated in the same form, in each iteration of the development cycle used to develop the system (for example, *Rational Unified Process* -RUP, etc.).

TABLE II
SA EVALUATION RESULTS FOR DIFFERENT USAGES

Sent requests	Turnaround time	Failures	Completed successfully	Time-based efficiency
8671	15,51 sec	48	8623	0,067 task/sec

In these more complex system scenario and architecture, the simulation time was 3:30 min in average, executing the evaluation on the same hardware used in the first case study.

VI. Conclusion and Future Work

This work defines a methodological framework based on DEVS for the multi-quality attribute evaluation of SA. In this approach, the architect not only can analyze several quality attributes with the same evaluation model, but also has an integrated view including functional aspects in the SA evaluation. This approach includes MDA framework for the transformation from SA to quality evaluation model proposing a formal mapping from software concepts to DEVS concepts. Finally, two case studies showed how the proposal can be applied and highlighted the consideration of several quality attributes in the same analysis. In the first case, e.g., the architect can analyze performance and availability, where the reports show that increasing turnaround time has an impact on downtime. Attributes such as usability are more difficult to estimate, but in the second case the architect can see the turnaround time and how it affects the usability by means of the Time-based efficiency. So, patterns or tactics to improve performance values can affect in the improvement of usability.

As future work, it is interesting to add usability metrics including some user behaviors in the simulation to better represent the real environment of the software and its usage. Also, new quality attributes and new metrics for performance, availability, and usability are under study to be included in the proposal. Finally, the development of a tool commonly used by architects is another future issue.

Acknowledgment

This work was supported by the Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) (PIP 112 20110100906), Universidad Tecnológica Nacional (PID UTI3803TC and PID UTN3581).

The authors gratefully acknowledge the contributions of the training program “Prácticas Profesionalizantes” of Tecnicatura Superior en Traducción, Instituto Técnico Superior Victor Mercante, carried out at Universidad Tecnológica Nacional, Facultad Regional Villa María, for their linguistic revision.

References

[1] P. D. Nielsen, “Software Engineering and the Persistent Pursuit of Software Quality”, *CrossTalk-Journal of Defense Software Engineering*, May/Jun. 2015.

[2] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, MA, 2002.

[3] Wen-Li Wang, Dai Pan, and Mei-Hwa Chen, “Architecture-based Software reliability modeling”. *J. Sys. Software* 79(1):132-146, 2006.

[4] V. S. Sharma and K. S. Trivedi, “Quantifying software performance, reliability and security: an architecture-based approach”. *J. Sys. Software* 80(4): 493-509, 2007.

[5] B. Spitznagel and D. Garlan, “Architecture-based performance analysis”. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering* (*SEKE’98*), 146-151, 1998.

[6] K. Fukuzawa and M. Saeki, “Evaluating software architecture by coloured Petri Nets”. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (*SEKE’02*), 263- 270, 2002.

[7] B. Zeigler, H. Praehofer, and T. Kim, *Theory of Modeling and Simulation–Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press, Orlando, FL, 2000.

[8] ITU-T Z.151, Series Z: Languages And General Software Aspects for Telecommunication Systems, Formal description techniques (FDT)–User Requirements Notation (URN)-Language definition, 2012.

[9] V. Bogado, S. Gonnet, and H. Leone, “A Discrete Event Simulation Model for the Analysis of Software Quality Attributes”, *CLEI Electronic Journal* 14(3), paper 3, 2011.

[10] V. Bogado, S. Gonnet, and H. Leone, “Modeling and simulation of software architecture in discrete event system specification for quality evaluation”, *SIMULATION* 90(3): 290–319, 2014.

[11] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture- Practice and Promise*, Addison-Wesley, Inc. Boston, MA, 2003.

[12] J. Franco, R. Barbosa, and M. Zenha-Rela, “Reliability analysis of software architecture evolution”. In *Proceedings of 6th Latin-American Symposium on Dependable Computing* (*LADC'2013*), 11–20, 2013.

[13] L. Singh, A. Tripathi, and G. Vinod, “Software reliability early prediction in architectural design phase: Overview and Limitations”. *Journal of Software Engineering and Applications* 4(3): 181-186, 2011.

[14] M. A. Isa, M. Z. M Zaki, and D. Jawawi, “A survey of design model for quality analysis: from a performance and reliability perspective”. *Computer and Information Science* 6(2), 2013.

[15] H. Christensen and K. Hansen. 2010. “An empirical investigation of architectural prototyping”. *J. Syst. Software* 83(1): 133-142.

[16] C. Rathfelder, B. Klatt, K. Sachs, and S. Kounev, “Modelling event-based communication in component-based software architectures for performance predictions”, *Softw. Sys. Model*, 2013.

[17] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, “Architecture-based reliability prediction with the Palladio Component Model”, *IEEE T. Software Eng.* 38(6), 2012.

[18] The Arizona Center for Integrative Modeling and Simulation (ACIMS). (March, 2016). DEVS-Suite. [Online]. Available: http://acims.asu.edu/software/devs-suite.

[19] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, MA, 2013.

[20] D. Amyot. (September, 2015). jUCMNav. [Online]. Available: http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome.

[21] OBEO and AtlanMod, (March, 2016). ATL. [Online]. Available: https://eclipse.org/atl/.

[22] ISO/IEC 25000, Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE), 2014.