Generación Automática de Modelos DEVS a partir de UCM en el Contexto de la Evaluación de Arquitecturas de Software

Bogado, Verónica[1]; Lazaroni, Esteban[2]; Leone, Horacio[2,3]; Gonnet, Silvio[2,3]
[1]Departamento Ingeniería en Sistemas de Información, Facultad Regional Villa María, UTN
[2]Departamento Ingeniería en Sistemas de Información, Facultad Regional Santa Fe, UTN
[3]INGAR (CONICET – UTN)

Abstract
La calidad de los productos de software en la industria es crítica hoy en día, ya que afecta en forma transversal a las actividades de las organizaciones u otras entidades que emplean sistemas de software. En este contexto, adquiere importancia la evaluación de la calidad en etapas tempranas del desarrollo. En este trabajo, se describe un entorno de simulación basado en el formalismo DEVS y se propone una herramienta para la generación automática de dicho entorno a partir de arquitecturas de software especificadas mediante mapas de casos de uso (o Use Case Maps, UCM). El entorno propuesto permite integrar aspectos funcionales, no funcionales y cuantitativos relacionados a la calidad en el análisis dinámico de arquitecturas de software, asistiendo al diseñador en la verificación de la solución arquitectónica según los requerimientos trazados.

Palabras Clave
Use Case Maps, DEVS, Generación Automática de Modelos, Evaluación de Arquitecturas.

Introducción

La arquitectura de software se define como la estructura o estructuras del sistema, las cuales contienen componentes de software, propiedades visibles externamente de esos componentes y las relaciones entre ellos [1]. Las decisiones de diseño plasmadas en la arquitectura tienen un impacto directo sobre cumplimiento o no de los requerimientos de calidad y, por ende, en la calidad del producto final. De aquí el gran valor que tiene su evaluación.

Actualmente, existen trabajos centrado en el análisis de la arquitectura con distintos propósitos, los cuales van desde enfoques informales hasta modelos matemáticos complejos.

Existen métodos que sugieren el uso de escenarios y especificaciones de la arquitectura acorde a las vistas propuestas por el SEI [2]. ATAM es el más conocido en este grupo de métodos y tiene como propósito descubrir los riesgos creados por decisiones arquitectónicas mas que proveer un análisis preciso. El mismo se centra en la interacción entre los diferentes atributos de calidad basándose en los escenarios previamente especificados. Los principales problemas de éstas técnicas consisten en la burocracia que generan y la necesidad de un equipo de evaluación dedicado.

También existen propuestas más formales enfocadas en la medición de atributos de calidad específicos. Algunas de ellas, emplean Procesos de Decisión de Markov para evaluar la confiabilidad de un sistema [3], performance y seguridad [4]. Otros autores han desarrollado propuestas basadas en la Teoría de Colas para medir performance [5]. Las redes de Petri son otra técnica propuesta para evaluar diferentes tipos de atributos de calidad tales como seguridad, performance y confiabilidad [6]. Las tres técnicas mencionadas son formales, permitiendo un estudio cuantitativo de atributos de calidad específicos con fundamentos matemáticos. Sin embargo, la aplicabilidad de estas técnicas es limitada, ya que son restrictivas desde el punto de vista del modelado [7]. Además, al momento de analizar la dinámica del sistema en etapas tempranas, se carece de herramientas que muestren cómo ocurren los cambios en el sistema.

La simulación es un instrumento poderoso para analizar los estados por los que puede pasar un sistema y obtener valores para evaluar diferentes escenarios. Las variables

que caracterizan una entidad, como ser valores que permitan medir la performance pueden ser modificados y estudiados, viendo el impacto que ello provoca sin necesidad de implementar el sistema. Propuestas recientes ([8], [9], [10]) plantean la especificación genérica de un entorno siguiendo el *framework* DEVS [11] adaptado para evaluar distintas arquitecturas especificadas con la notación de mapas de casos de uso (UCM) [12]. Los elementos son construidos siguiendo los principios de modularidad y jerarquía, donde el alto nivel de abstracción permite representar los conceptos del dominio arquitectónico adecuadamente.

En la Figura 1, se ilustra la idea básica de la propuesta para evaluar arquitecturas de software empleando el formalismo DEVS. El proceso de evaluación comienza con la construcción de un modelo UCM de la arquitectura de software, el cual representa su estructura y la forma en que será utilizado, es decir, ciertas condiciones de operación. A partir del modelo UCM, se deriva el entorno de simulación DEVS para realizar la evaluación de la arquitectura. La ejecución del entorno de simulación brindará estimaciones de indicadores de calidad relacionados a atributos del software. Algunos ejemplos de indicadores que pueden ser obtenidos a partir de la simulación del sistema en ejecución son el tiempo de respuesta del sistema, la cantidad total de fallas ocurridas en el sistema, el tiempo de recuperación, entre otros, donde cada uno se encuentra relacionado a un atributo de calidad.

Figura 1. Evaluación de Arquitecturas de Software empleando DEVS [8].

A partir de las contribuciones mencionadas ([8],[10]), se propone en este trabajo la generación automática del entorno de simulación a partir de una especificación de la arquitectura de software con la notación UCM. En primer lugar, el arquitecto especifica los elementos arquitectónicos empleando UCM. Luego, se aplican las reglas de transformación especificadas en [8] para relacionar los conceptos capturados con los conceptos del lenguaje DEVS, obteniendo así un modelo de simulación para la arquitectura dada y un marco experimental para evaluar tres atributos de calidad visibles en tiempo de ejecución: performance, disponibilidad y confiabilidad, con la posibilidad de incorporar otros atributos en el análisis. Finalmente, se genera el código Java que implementa el entorno de simulación proporcionando una herramienta para realizar experimentos que permitan obtener indicadores de calidad concretos. El entorno de simulación para la evaluación de arquitecturas propuesto se implementó empleando el simulador *DEVS-Suite* [13].

Representación de la Arquitectura de Software empleando UCM

Como se indicó previamente, el primer modelo que se debe construir en el proceso propuesto de evaluación de arquitecturas es una especificación de la arquitectura de software empleando la notación UCM (Figura 1). En un modelo UCM, se representan la estructura y los escenarios relevantes del sistema. La descripción de cada escenario incluye el estímulo de entrada que activa al sistema iniciando la ejecución del escenario. En la Figura 2, se muestra un modelo que integra los elementos estructurales, funcionales de UCM con información cuantitativa para la obtención de indicadores de calidad a partir de una arquitectura de software desde un punto de vista dinámico. En dicha vista participan diferentes elementos arquitectónicos (*ArchitecturalElement*, Figura 2), los cuales son entidades de software que tienen presencia en tiempo de ejecución, diferenciándose entre componente simple (*SimpleComponent*, Figura 2) y compuesto (*CompositeComponent*, Figura 2). El

componente simple no contiene otros componentes y es la unidad más pequeña dentro de los elementos arquitectónicos mientras que el compuesto está conformado de otros componentes, simples o compuestos, siendo una unidad más compleja, la cual delega sus responsabilidades en las partes.

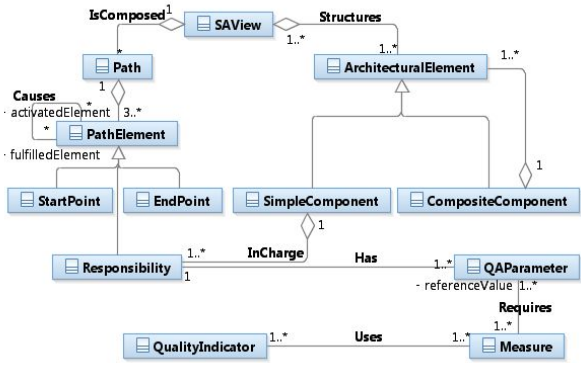

Figura 2. Conceptos para la evaluación de arquitecturas de software especificadas en UCM.

Una responsabilidad (*Responsibility*, Figura 2) es una declaración general sobre un objeto de software [12]: una acción que realiza el elemento, un conocimiento que mantiene sobre algo o una decisión importante que afecta a otro objeto de software. Una responsabilidad tiene asociado un conjunto de parámetros (*QAParameter*, Figura 2) que son empleados para modelar el comportamiento de diferentes aspectos dinámicos de la ejecución del software, por ejemplo, el tiempo de ejecución (*ref_execution_time*) o el tiempo de recuperación, en caso de ocurrir una falla (*ref_recovery_time*). Estos parámetros son necesarios para la obtención de distintas medidas (*Measure*, Figura 2) empleadas para la obtención de indicadores de calidad (*QualityIndicator*, Figura 2), como ser *performance* o *disponibilidad*.
Un elemento arquitectónico puede tener asignado responsabilidades, las cuales se relacionan con otras del mismo elemento o de otros para realizar acciones requeridas. El tipo de relación es causa-efecto (asociación *Causes*, Figura 2), donde el cumplimiento de una responsabilidad (*fulfilledElement*, Figura 2) implica la ejecución de las consecutivas (*activatedElement*, Figura 2) [12].
En UCM, se especifican los distintos escenarios mediante caminos (*Path*, Figura 2). Los caminos vinculan por medio de asociaciones de causa-efecto las responsabilidades contenidas en los componentes de la arquitectura. El inicio de un camino es un elemento que especifica la espera de un estímulo (*StartPoint*, Figura 2). Su efecto inmediato es la ejecución de la primera responsabilidad del camino. Esta, una vez ejecutada, habilita la ejecución del siguiente elemento del camino (*PathElement*, vinculados por la asociación *Causes*, Figura 2). El camino finaliza cuando se emite una respuesta, final del escenario es alcanzado (*EndPoint*, Figura 2). El camino es progresivo, en el sentido que cada punto (responsabilidad) a través de él avanza hacia el final del mismo [14].

Especificación DEVS de una Arquitectura de Software

La dinámica del software modelado con UCM se especifica en DEVS. DEVS es un formalismo para la simulación de sistemas de eventos discretos [11]. A partir del modelo planteado en la Figura 2, se construye un entorno de simulación, compuesto por modelos DEVS estructurados en forma modular y jerárquica, donde los elementos (bloques de construcción) se determinan en base a ciertas características, como ser que el elemento sea auto-contenido (información y procesos locales), interoperable (cooperación entre bloques), reusable (instanciación múltiple), reemplazable (intercambiable en el modelo), respetando el encapsulamiento de la estructura interna mediante interfaces bien definidas [15].
En la Figura 3, se muestra cómo se estructura el entorno de simulación, *SAESE* (*SAE - Simulation Environment*), el cual es una jerarquía de modelos DEVS. Los dos elementos que lo componen son la vista de la arquitectura (*SAVSM*, *SAView - Simulation Model*) y el marco experimental (*SAEEF*, *SA Evaluation - Experimental*

Frame). El modelo de simulación (*SAVSM*) representa la entidad bajo estudio, es decir, el software representado por su arquitectura y los escenarios funcionales. Por otro lado, el marco experimental (*SAEEF*) define el ambiente con el cual el software simulado interactúa, resume las medidas tomadas en los elementos del modelo y calcula los indicadores de calidad.

La Figura 3 presenta la jerarquía de modelos DEVS y sus relaciones con los elementos de representación de una arquitectura de software siguiendo las propuestas [8] y [10].

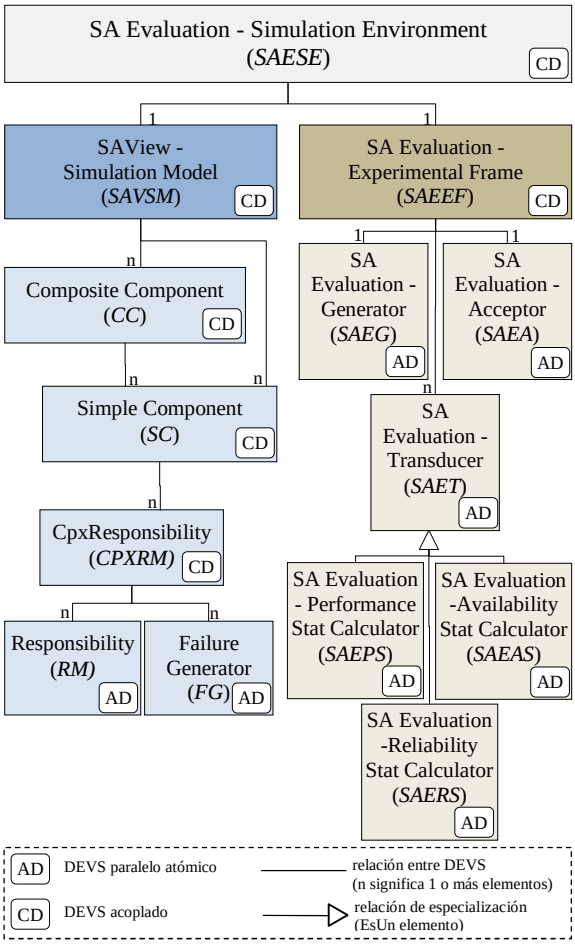

Figura 3. Jerarquía de elementos de simulación y su relación con los elementos conceptuales de la arquitectura de software.

En la construcción de la simulación, se emplean modelos DEVS atómicos y acoplados para la representación de las entidades de software modeladas en la Figura 2. El concepto responsabilidad (*Responsibility*, Figura 2) es especificado mediante un DEVS atómico paralelo (*RM*, Figuras 3 y 4). La Figura 4 describe el modelo *RM* con sus puertos de entrada (*prip*, *intfailip*), puertos de salida (*srop*, *stateop*, *taop*, *dtop*, *rtop*, *failop*), estados representados por una fase (*inactive*, *active*, *executing*, *failed*, *recovering*) y un valor de sigma, transiciones internas y externas. Una explicación detallada de este modelo está disponible en [8].

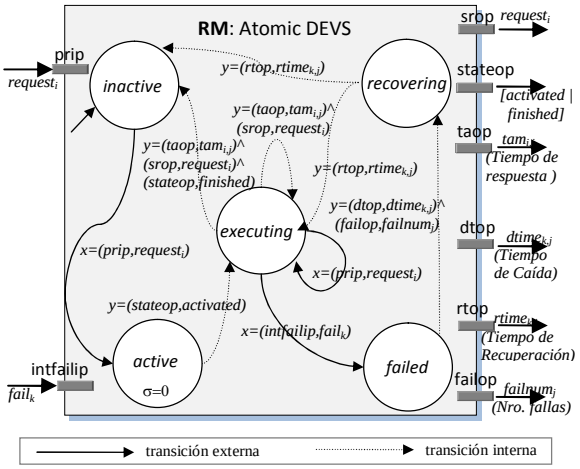

Figura 4. *RM* (*Responsibility*), DEVS atómico con diagrama de transición de estados.

En la Figura 3, se puede observar el agregado de un modelo DEVS adicional, *FG*, el cual es introducido para representar problemas en la ejecución de la funcionalidad dada por la responsabilidad. Los modelos DEVS *RM* y *FG* son acoplados y encapsulados por un modelo DEVS acoplado denominado *CPXRM* (Figura 3). Los parámetros que posee una responsabilidad (*QAParamenter*, Figura 2) son representados en los modelos *RM* y *FG* como parámetros fijos de cada modelo DEVS, y son empleados en la simulación para el cálculo de los indicadores de calidad (*QualityIndicator*, Figura 2). El cálculo de estas métricas se modela en componentes específicos DEVS incluidos para tal fin (*SAEPS*, *SAEAS* y *SAERS* en Figura 3), partes del *SAEEF*.

El componente simple (*SimpleComponent*, Figura 2) es especificado como un modelo DEVS acoplado (*SC*, Figura 3), cuyos elementos son una o más responsabilidades (multiplicidad *n* en Figura 3). Estos

elementos pueden ser parte de un componente compuesto (*CompositeComponent*), el cual es representado por un DEVS acoplado denominado *CC*. Un modelo *CC* puede estar compuesto por instancias de *CC* y *SC* (Figura 3). Estos elementos arquitectónicos conforman una vista arquitectónica (*SAView*, Figura 2), la cual es traducida a un DEVS acoplado (*SAVSM*) que representa el modelo de simulación. Una especificación formal de estos modelos está disponible en [8].

Diseño de *UCM2DEVS*

La evaluación de arquitecturas de software empleando DEVS (Figura 1) requiere de herramientas computacionales que permitan la generación automática del entorno de simulación a partir de una especificación de la arquitectura de software en la notación UCM. En consecuencia, en esta sección se especifica el diseño de una herramienta, *UCM2DEVS*, que permite la generación del entorno de simulación en DEVS.

En la propuesta, se considera el empleo de las herramientas *jUCMNav* [16], un editor gráfico para la Notación de Requerimientos de Usuario, la cual incluye UCM, y *DEVS-Suite*, un simulador para DEVS paralelo [13]. Ambas herramientas corren bajo el entorno de desarrollo *Eclipse*.

UCM2DEVS es la herramienta encargada de tomar un modelo de entrada, una vista de una arquitectura especificada con la notación UCM y transformar los elementos de la misma a elementos de simulación de forma automática, obteniendo como salida el entorno de simulación en DEVS. *UCM2DEVS* está conformado por tres componentes implementando una arquitectura "*Pipe & Filters*". El primer elemento es un *Parser*, el cuál toma como entrada la especificación de la arquitectura (un archivo en formato *xml*) y genera los objetos correspondientes al modelo incluido en la Figura 2. Las instancias de este modelo son tomadas por el segundo elemento, el *Transformador*, que traduce los distintos elementos arquitectónicos en los modelos DEVS incluidos en la Figura 3. Por último, el tercer elemento de *UCM2DEVS* es el encargado de generar el entorno de simulación, obteniendo la especificación DEVS empleando el lenguaje Java.

Una vez procesado el archivo de entrada, es posible ejecutar el entorno de simulación en la *DEVS-Suite* para simular una arquitectura particular.

Para la implementación del *Parser* se empleó el analizador léxico *JLex* [17] y el generador de *parser CUP* [18], ambos basados en Java. Se especificó la gramática libre de contexto para representar la sintaxis de los archivos generados por *jUCMNav*. En la Figura 5, se incluye parcialmente la gramática especificada junto al comportamiento vinculado a las distintas reglas de producción. Se detalla el símbolo inicial de la misma, *UCMSpecification*, y los símbolos no terminales *responsibilities*, *responsibility* y *pathElements*. Una especificación está dada por un conjunto de *Responsibility*; la especificación de los *PathElement*, la cual incluye las relaciones entre los mismos (relación *Causes* en Figura 2); las relaciones entre la vista y los elementos arquitectónicos (asociaciones de agregación *SAView – ArchitecturalElement*, *CompositeComponent–ArchitecturalElement*, Figura 2) representadas como un conjunto de referencias; los elementos de conexión que representan los fragmentos del *Path*; y una lista de componentes (*ArquitecturalElements*, Figura 2). Una responsabilidad es especificada mediante la secuencia "<responsibilities" (B_RESP, Figura 5), seguida del identificador (ID), nombre (NAME), número del nodo (NUMB_NODE) donde se define información específica del *PathElement*, y se finaliza con la secuencia "/>" (E_RESP). Estos elementos son identificados por el analizador léxico. En la Figura 9 (en la Sección *Caso de Estudio: Resultados y Discusión*), se incluye un ejemplo de la especificación de responsabilidades en el archivo en *jUCMNav* y, en la Figura 6, se

detallan las reglas del analizador léxico que generan los distintos *token* definidos para responsabilidad en la Figura 5.

```
UCMSpecification::= responsibilities:r pathElements:ps
references:refs connectionElements:c
componentsList:comps
  {:
      saucm.responsibilities=r;
      saucm.components=comps;
      saucm.conections=c;
      saucm.nodes=ps;
      saucm.references=refs;
      saucm.associatesResponsibilities();
      saucm.setCauses();
      saucm.aggregatesComponents();
      ucm2devs(saucm);
  :}
                        ;
responsibilities::=   responsibility:r
  {:
     Vector<Responsibility> resps =
                              new Vector<Responsibility>();
     resps.add(r);
     RESULT=resps;
  :}
                        | responsibilities:l responsibility:s
  {:
      l.add(s);
      RESULT=l;
  :}
                        ;
responsibility::=     B_RESP     ID:id     NAME:name
NUMB_NODE:nodo E_RESP
  {:
     Responsibility r=new Responsibility(id,name,nodo);
     RESULT=r;
  :}
                        ;
pathElements::= endPoint:r                {: … :}
              | startPoint:r              {: … :}
              | nodeResp                  {: … :}
              | pathElements:l nodeResp:n {: … :}
              | pathElements:l endPoint:s {: … :}
              | pathElements:l startPoint:s {: … :}
                        ;
```

Figura 5. Especificación parcial de la gramática empleada en el *Parser*.

```
<YYINITIAL>"<responsibilities"
   {
      yybegin(RESPSTATE);
      return new Symbol(sym.B_RESP);
   }
<RESPSTATE>"id=\""[0-9]+\"
   {
      return new Symbol(sym.ID,tomarInt(yytext(),4,1));
   }
<RESPSTATE>"@nodes."[0-9]+\"
   {
       return new Symbol(sym.NUMB_NODE,
toInt(yytext(),7,1));
   }
<RESPSTATE>"/>"
   {
       yybegin(YYINITIAL);
       return new Symbol(sym.E_RESP);
   }
```

Figura 6. Especificación parcial del analizador léxico.

A partir de estos analizadores, se generan las instancias del modelo presentado en la Figura 2. El código para la generación está representado entre {: :} en la Figura 5. Una vez recuperado el modelo de la especificación de la arquitectura, se invoca desde el comportamiento asociado a la regla *UCMSpecification* (Figura 5) la aplicación de las reglas de conversión de UCM a DEVS (ucm2devs(saucm);).

Como se indicó previamente, la especificación del modelo de simulación DEVS se implementa en la herramienta *DEVS-Suite* [13], la cual provee el conjunto de paquetes, incluyendo el paquete original *DEVSJAVA* para implementar los modelos DEVS usando el lenguaje de programación Java [19].

En este trabajo, se emplearon las clases *devs*, *atomic* y *digraph* de *DEVSJAVA* y sus especializaciones *ViewableAtomic* y *ViewableDigraph*. Estas dos últimas clases extienden el modelo DEVS con operaciones para el manejo gráfico de estos elementos. En consecuencia, los elementos de simulación propuestos (Figura 3) son implementados en DEVS empleando el lenguaje de programación Java. Los DEVS atómicos, tales como *RM* y *FG* (Figura 3), son implementados como subclases de la clase *ViewableAtomic*. En la Figura 7, se incluye la declaración y el constructor de la clase *ResponsibilityDEVS* que implementa *RM* según la especificación dada en la Figura 4. Los DEVS acoplados (Figura 3) son implementados como subclase de *ViewableDigraph*.

Caso de Estudio: Resultados y Discusión

En esta sección, se aplica la propuesta en la evaluación de la arquitectura de un software para la gestión de licencias de software. Se presenta un sistema simple con el fin de que se comprenda el ambiente de simulación propuesto, intentando no introducir la complejidad adicional relacionada a la arquitectura de software o al ambiente de operación del mismo.

```
public class ResponsibilityDEVS extends
ViewableAtomic {
        …
    public ResponsibilityDEVS(int id, String  name, double
refExecutionTime, double refRecoveryTime){
        super(name);
        this.setResponsibilityId(id);
        //queue
        requests = new Queue();
        clockIns = new Queue();
        //initialize the generators
        rndExecutionTimeGen = new rand(1);
        rndRecoveryTimeGen = new rand(1);
        //Define the ports for the model:
        //Input ports
        addInport("prip");
        addInport("intfailip");
        //output ports
        addOutport("srop");
        addOutport("stateop");
        addOutport("taop");
        addOutport("dtop");
        addOutport("rtop");
        addOutport("failop");
        //Initialize parameters
        this.setExecutionTime(refExecutionTime);
        this.setRefRecoveryTime(refRecoveryTime);
        taLastR=0;
        procRequestsCount=0;
        this.clockLastFail=0;
    }
        …
}
```

Figura 7. Declaración y constructor de la clase *ResponsibilityDEVS* que implementa *RM*.

El software de gestión de licencias (*LM System*) realiza un control automático de las licencias de los productos de software instalados. El mismo es comúnmente empleado por empresas dedicadas al desarrollo de software o por organizaciones que deben llevar un control sobre las licencias de software que se encuentran en operación. Un software para la gestión de licencias controla dónde y cómo los productos de software estarán disponibles para su operación por parte de los clientes.

En este caso, el estudio se centra en la parte del sistema (subsistema) que se encarga de validar o extender las licencias de los clientes a partir del pago correspondiente.

Las dos partes principales de *LM System* son: el servidor, denominado *LMServer*, y el cliente, denominado *LMClient*. Cada uno tiene un conjunto de componentes modelados aplicando el patrón "*Pipe & Filter*", cuyas responsabilidades definen un flujo causal simple de ejecución.

La Figura 8 muestra la vista de la arquitectura de software usando la notación UCM y el software *jUCMNav*.

El servidor (*LMServer*) es el responsable por la creación de las licencias de software, siendo el generador del archivo de licencia (Figura 8). Este elemento está conformado por una secuencia de tres filtros denominados: *Creator*, *Codifier* y *Encryptor*. El componente *Creator* toma los datos de entrada (nombre de la organización, fecha de la licencia, contador de días, tiempo de la licencia, etc.) requeridos para generar la licencia del software en cuestión y crear un archivo de texto plano (responsabilidades *r1* y *r2* respectivamente). El componente *Codifier* está a cargo de generar un identificador único (valor *hash*) que depende del contexto de la entrada (*r3*), guardando este identificador junto con la información del archivo de entrada en otro archivo de texto plano (*r4*). El componente *Encryptor* toma como entrada el archivo *hash* con la clave privada y genera un archivo encriptado con la firma digital que certifica la autentificación del documento final (*r5*). Entonces, la fuente provee datos para la licencia y el destino (*sink*) recibe un archivo que incluye los datos, el identificador (*hash*) y la firma digital de la empresa.

El otro componente compuesto, *LMClient* (Figura 8), representa el subsistema cliente con tres elementos modelados como filtros: *Decryptor*, *Authenticator* y *Recorder*. El primer componente (*Decryptor*) está a cargo de recibir y desencriptar el archivo recibido generado por el servidor (*r6* y *r7*). Este archivo de texto plano contiene los datos de la licencia y la firma digital. Entonces, este componente desencripta la firma con la clave pública enviando el archivo con esta información al siguiente componente. El segundo componente (*Authenticator*) recibe el archivo y el identificador (*hash*). Luego, calcula el identificador usando el contenido del archivo y siguiendo un algoritmo específico para tal fin y, finalmente, compara estos dos

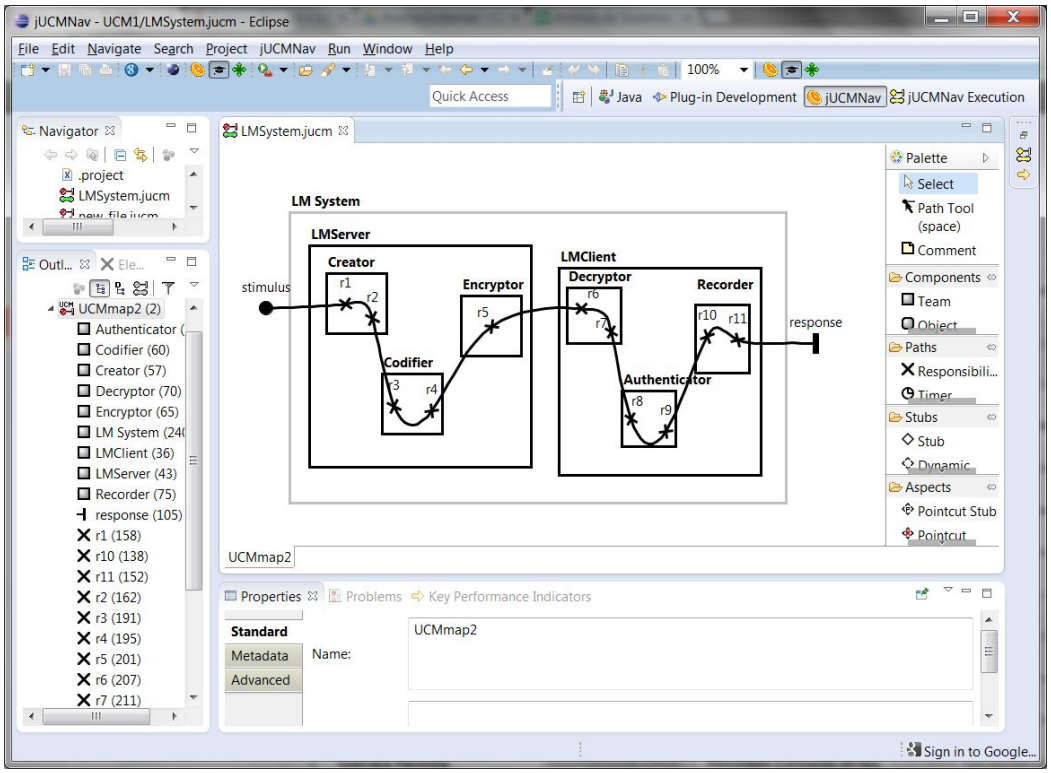

Figura 8. UCM de la vista de la arquitectura de *LM System.*

identificadores (*r8*), enviando el archivo y el resultado al siguiente componente (*r9*).

El último componente (*Recorder*) toma la información autenticada (archivo y resultado) (*r10*) y actualiza la información de la licencia en la base de datos si el resultado es correcto; en caso contrario, bloquea al usuario (cliente), registrando el error junto con los datos de la licencia en la base de datos (*r11*). Así, la empresa se asegura de que no se haya modificado el contenido del archivo y que es ella la que generó la licencia.

En la Tabla 1, se detallan dos escenarios de atributo de calidad típicos, donde la arquitectura propuesta debe cumplirlos alcanzando los valores correspondientes para las medidas de respuesta especificadas. Los escenarios de calidad fueron especificados aplicando la plantilla propuesta por el SEI [1]. Estos escenarios permiten al arquitecto delimitar el sistema a ser evaluado, haciendo explícita la porción del sistema que está por ser simulada en este caso. En la segunda columna, se describe un escenario típico de performance, donde el usuario es la fuente de estímulo generando solicitudes de autenticación y el artefacto es el sistema completo (*LM System*). De este modo, el usuario solicita la autenticación y el sistema debería responder en el tiempo especificado en el escenario. En la tercera, se presenta un escenario relacionado al atributo de calidad disponibilidad que describe una falla en el proceso del sistema (*LM System*) y especifica el tiempo tolerable de caída, en el cual se podría encontrar no disponible para el caso de uso planteado.

Tabla 1. Escenarios de calidad de *LM System*.

Atributos	***Performance***	***Disponibilidad***
Fuente de estímulo	Externa al sistema (Sistema/usuario)	Interna al sistema
Estímulo	Solicitud de extensión de licencia	Responsabilidad falla al responder ante una entrada
Artefacto	*LM System* (Sistema: servidor y cliente)	*LM System* (Sistema como proceso)
Ambiente	Operación normal	Operación normal
Respuesta	Licencia autenticada	Evento de falla registrado
Medida de Respuesta	Tiempo de respuesta menor a 55 segundos, menos de 5 segundos por servicio.	Menos de 5 horas de caída en 3 meses

El modelo visualizado en la Figura 8 es almacenado por la aplicación *jUCMNav* como un archivo en formato *xml*, *LMSystem.jucm* (Figura 9). Este archivo es la entrada a la herramienta *UCM2DEVS*, la cual aplica en primer lugar el *parser* para luego traducir la arquitectura a un modelo de simulación, donde los elementos de software pasan a ser elementos de simulación. La salida de la herramienta es el código Java que extiende a la biblioteca DEVSJAVA con las clases necesarias y la generación de sus instancias para implementar el modelo particular.

La Figura 10 ilustra la implementación del entorno de simulación (*License Manager - SimEnvironment*, instancia de *SAESE*), usando la herramienta *DEVS-Suite*, donde se configura para su ejecución. Los datos para la configuración fueron estimados a partir de información de sistemas de similares características, componentes y funcionalidades. En particular, se especificaron: tiempo de ejecución y tiempo de recuperación en cada responsabilidad, tiempo medio entre fallas y tiempo de caída en cada generador de fallas, tiempo medio entre solicitudes y tiempo de simulación en el marco experimental.

La simulación de la operación del sistema de software, para un período de 3 meses, proporciona información sobre diferentes aspectos del mismo, la cual es importante para la toma de decisiones relacionadas al diseño del sistema.

Las medidas del sistema indican que el tiempo de respuesta promedio del sistema durante los 3 meses de operación es de 62 segundos. Este valor indica el tiempo promedio que requiere el sistema para emitir una respuesta a un estímulo externo. En el caso del escenario de performance, los estímulos son las solicitudes que envían los usuarios/sistemas externos, en una operación normal del sistema, donde el tiempo de respuesta se encuentra sobre el valor definido en el escenario de calidad (Tabla 1). En la Figura 11, se detalla el comportamiento de esta variable bajo las condiciones dadas para el sistema en 10 corridas del entorno de simulación. Como se puede ver, el tiempo de respuesta promedio para cada corrida se encuentra por encima del tiempo especificado. Además, la tendencia del tiempo de respuesta del sistema es en aumento, teniendo en cuenta el promedio acumulado, alejándose del valor requerido, es decir, que dada esta arquitectura el comportamiento de este tiempo es creciente.

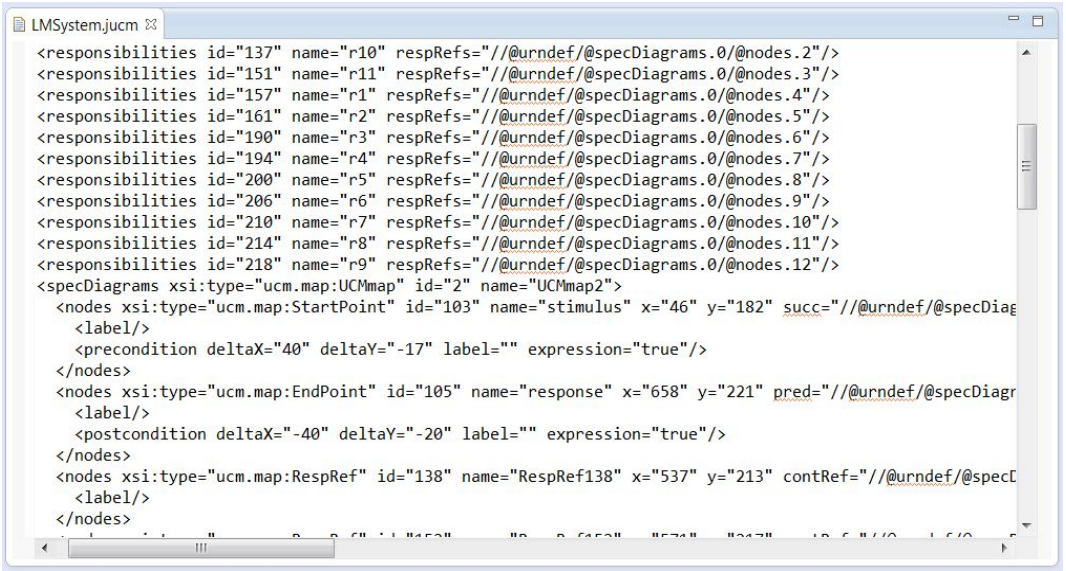

Figura 9. Archivo generado por la aplicación *jUCMNav* de la vista de la arquitectura de *LM System*.

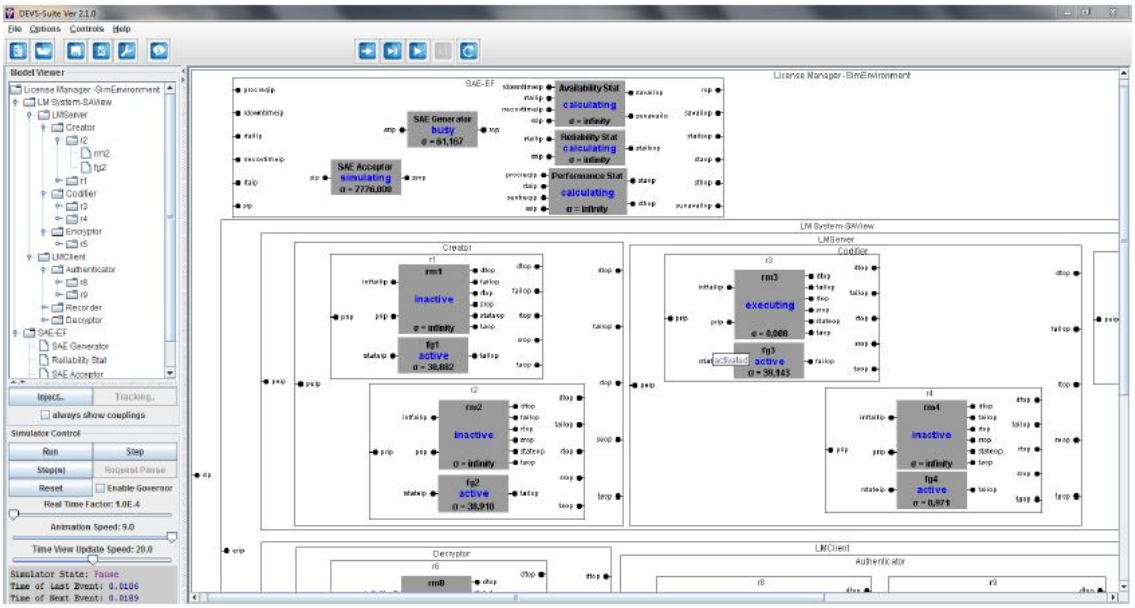

Figura 10. Entorno de simulación para *LM System*.

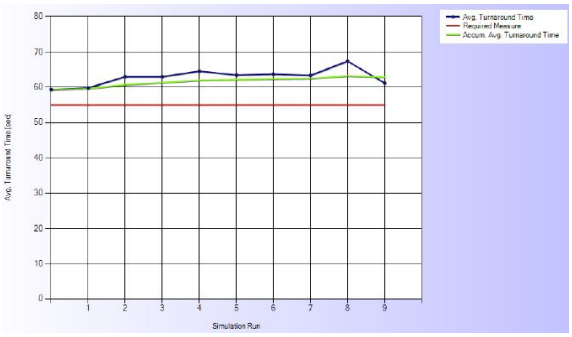

Figura 11. Tiempo de respuesta promedio del sistema simulado.

El tiempo promedio de caída del sistema, es usado para validar el segundo escenario de calidad. Luego de varias simulaciones, este indicador permitió inferir un tiempo de caída promedio de 3 horas y media, donde el sistema no se encuentra disponible 0.1% del tiempo total de operación (3 meses). En la Figura 12, se puede ver que, para 10 corridas de la simulación, el tiempo de caída del sistema solo superó el límite definido en el escenario una vez. Esta situación indica que el sistema, en promedio, no tiene sus servicios disponibles un tiempo menor a 5 horas bajo las condiciones definidas, operación normal del sistema y probabilidades de fallas de cada responsabilidad.

Otras métricas del sistema que pueden completar la información para analizar el escenario de calidad relacionado a la disponibilidad son: el número promedio de fallas del sistema, el cual fue 37.5 empleando el número total de fallas del sistema luego de varias corridas de la simulación, y el tiempo de operación del sistema, el cual determina el tiempo total en que el sistema estuvo disponible en las diferentes ejecuciones de la simulación resultando en 99% del tiempo total en operación (3 meses).

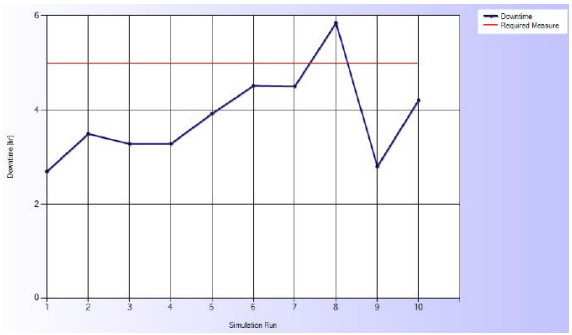

Figura 12. Tiempo de caída promedio del sistema simulado.

Adicionalmente, se puede analizar cada responsabilidad, nivel más bajo en la jerarquía de elementos de software, para poder determinar elementos causantes de problemas. Esto permite detectar problemas, localizando elementos puntuales del software a partir del diseño de entrada. De este análisis surge que la responsabilidad *r5* es una unidad de software crítica de la arquitectura,

superando la medida especificada en el escenario. Además, el tiempo de respuesta ante una solicitud generada internamente es mayor al de otras responsabilidades. Esta situación podría ser un indicador de una necesidad de rediseño. Por ejemplo, podría estar indicando una complejidad crítica en la funcionalidad, mayor a las demás. Lo mismo se podría inferir para las responsabilidades *r3* y *r7*, siendo funcionalidades críticas del escenario.

El formalismo DEVS provee flexibilidad y escalabilidad con fundamentos matemáticos tanto al modelo de simulación como al marco experimental, permitiendo la especificación de elementos de simulación específicos del dominio. Estas ventajas proporcionan un entorno de simulación mucho más rico a diferencias de otros formalismos [5][6][7]. La automatización de la generación de los modelos DEVS contribuye a facilitar el empleo del mismo en el contexto del diseño arquitectónico, ya que evita que el arquitecto deba comprender los detalles técnicos. En las propuestas existentes, se pierde la importancia de este proceso de transformación, haciendo que las mismas no sean ampliamente empleadas debido a esta limitación técnica [3][4][5][6].

Conclusiones y Trabajos Futuros

En el presente trabajo, se propuso una herramienta de software que permite generar un entorno de simulación DEVS a partir de la especificación de una arquitectura de software en el lenguaje UCM. El entorno de simulación generado brinda soporte al arquitecto en el análisis de atributos de calidad, proporcionando información necesaria como para validar diferentes escenarios de calidad en un contexto de operación del sistema dado.

La generación automática de modelos DEVS a partir de un UCM que especifica la arquitectura del sistema así como sus escenarios resulta de vital importancia en el proceso de aplicación de la propuesta. De esta forma, el arquitecto no necesita invertir tiempo en la comprensión de los fundamentos del formalismo sino que se centra en los resultados que arroja la simulación y en la toma de decisiones que mejoren el diseño y, en consecuencia, se focaliza en la calidad del producto final.

Como trabajo futuro, se mejorará este proceso de generación automática de modelos desarrollando interfaces gráficas adecuadas al proceso de diseño de arquitecturas. Además, se pretende complementar la herramienta con un generador de reportes que provea diferentes presentaciones gráficas de los resultados para facilitar el análisis al arquitecto, así como la toma de decisiones sobre el diseño del sistema.

Referencias

[1] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice, 2012.

[2] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P; Nord, R.; Stafford, J.: Documenting Software Architecture-Views and Beyond, 2nd edition. Addison-Wesley, 2010.

[3] Wang, W.; Pan, D.; Chen, M.: Architecture-based Software Reliability Modeling, Journal of Systems and Software 79:1, 132–146, 2006.

[4] Sharma, V.; Trivedi, K.: Quantifying Software Performance, Reliability and Security: An Architecture-based Approach. Journal of Systems and Software 80:4, 493–509, 2007.

[5] Spitznagel, B.; Garlan, D.: Architecture-Based Performance Analysis. Proc. of SEKE, 1998.

[6] Fukuzawa, K..; Saeki, M.: Evaluating Software Architecture by Coloured Petri Nets. Proc. of SEKE, 2002.

[7] Singh, L.; Tripathi, A.; Vinod, G.: Software Reliability Early Prediction in Architectural Design Phase: Overview and Limitations. Journal of Software Engineering and Applications 4:3, 181–186, 2011.

[8] Bogado, V.: Un Modelo de Soporte al Análisis de Arquitecturas de Software Mediante Simulación de Eventos Discretos, Tesis Doctoral, Universidad Tecnológica Nacional, 2013.

[9] Bogado, V.; Gonnet, S.; Leone, H.: An Approach based on DEVS for Evaluating Quality Attributes, Proc. of International Conference of Chilean Computer Science Society, 110-118, 2010.

[10] Bogado, V.; Gonnet, S.; Leone, H.: A Discrete Event Simulation Model for the Analysis of

Software Quality Attributes, CLEI Electronic Journal 14:3, Paper 3, 2011.

[11] Zeigler, B., Praehofer, H., Kim,. T.: Theory of Modeling and Simulation–Integrating Discrete Event and Continuous Complex Dynamic Systems, 2000.

[12] Buhr, R.: Use Case Maps as Architectural Entities for Complex Systems, IEEE Transactions on Software Engineering, 24(12):1131–1155, 1998.

[13] DEVSSuite. Arizona Center for Integrative Modeling and Simulation- DEVS-Suite, 2011, disponible en: http://acims.asu.edu/software/devs-suite

[14] Buhr, R.: Making behaviour a concrete architectural concept, Proc. 32nd Hawaii International Conference on System Sciences, pp. 1-5, 1999.

[15] Verbraeck, A.; Valentin, E.: Design guidelines for simulation building blocks, in Proc. 2008 Winter Simulation Conference, pp. 923–932, Miami, USA, 2008.

[16] jUCMNav. jUCMNav-Eclipse Plugin for the User Requiriments Notation, disponible en: http://marketplace.eclipse.org/content/jucmnav#.UgvqJ6z-VrU

[17] Berk, E.; Ananian, C.S.: JLex: A Lexical Analyzer Generator for Java, 2003, disponible en: http://www.cs.princeton.edu/~appel/modern/java/JLex/

[18] Hudson, S.; Flannery, F.; Ananian, C.S.: CUP Parser Generator for Java, 1999, disponible en: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[19] Zeigler, B.; Sarjoughian, H.: Introduction to DEVS Modeling a Simulation with JAVA: Developing Component-Based Simulation Models. Technical report, University of Arizona, USA, 2003.

Datos de Contacto:

Verónica Bogado. Departamento Ingeniería en Sistemas de Información, Facultad Regional Villa María, UTN. Av. Universidad 450, X5900 HLR Villa María, Córdoba, Argentina. vbogado@frvm.utn.edu.ar

Esteban Lazaroni. Departamento Ingeniería en Sistemas de Información, Facultad Regional Santa Fe, UTN. Lavaisse 610, S3004EWB Santa Fe, Argentina. esteban.lazzaroni@gmail.com

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

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