Metamodel-based transformation from UML state machines to DEVS models

Ariel Gonzalez
Universidad Nacional de Río Cuarto,
Río Cuarto, Argentina
email: agonzalez@dc.exa.unrc.edu.ar

Carlos Luna
Universidad de la República,
Montevideo, Uruguay
email: cluna@fing.edu.uy

Roque Cuello, Marcela Perez, Marcela Daniele
Universidad Nacional de Río Cuarto,
Río Cuarto, Argentina
email: cuello, perez, marcela@dc.exa.unrc.edu.ar

***Abstract*—The development of complex dynamic systems require studies and analysis prior to deployment with the goal of detecting unwanted behavior. UML is a language widely used for modeling these systems through state machines, among other mechanisms. Currently, these models do not have appropriate execution and simulation tools to analyze the real behavior of systems. Modeling and simulation for design and prototyping of systems are widely used techniques. In particular, the Discrete Event system Specification (DEVS) formalism separates the modeling and simulation; there are several tools available on the market that run and collect information from DEVS models. This paper proposes a model transformation mechanism of UML state machines to DEVS models in the context MDD, through the declarative QVT Relations language, in order to perform simulations using tools, such as PowerDEVS.**

***Keywords*– Statecharts, DEVS, UML, MDA, QVT Relations.**

I. Introducción

El Desarrollo de Software Dirigido por Modelos (*Model Driven Development* - MDD) [1], [2], [3] es una metodología de la ingeniería de software que, durante años, ha representado los artefactos de software como modelos, con el objetivo de incrementar la productividad, calidad y reducir costos en el proceso de desarrollo de software. Uno de sus principales objetivos es organizar los niveles de abstracción y las metodologías de desarrollo, promoviendo el uso de modelos como artefactos principales a ser construidos y mantenidos. Un modelo está constituido por un conjunto de elementos que proporcionan una descripción sintética y abstracta de un sistema, concreto o hipotético. De esta manera, el proceso de desarrollo se convierte en un proceso de refinamiento y transformación entre modelos, generando niveles de abstracción cada vez menores, hasta que en un último paso se genera código para una plataforma específica. Existe un interés creciente en este campo. En particular, la Arquitectura Dirigida por Modelos (*Model Driven Architecture* - MDA) [4] es la aproximación definida por el *Object Management Group* (OMG) [5].

Para el desarrollo de software dirigido por modelos, el lenguaje *Unified Modeling Language* (UML) [5], [6] se ha convertido en el estándar para el modelado de diferentes aspectos de sistemas de software, tanto en el ambiente académico como en desarrollos industriales. UML sigue el paradigma de orientación a objetos y permite la descripción de aspectos tanto estáticos como dinámicos de sistemas de software. Más que un lenguaje es un conjunto de lenguajes, en su mayoría notaciones gráficas, soportados por un número importante de herramientas propietarias y de código abierto [7]. Si bien UML es uno de los medios preferidos de comunicación entre expertos del modelado, por su poder de representación gráfica, esta capacidad se encuentra acotada en términos de ejecución de los modelos, es decir, de ejecución de simulaciones.

En el campo de la simulación, el lenguaje Especificación de Sistemas de Eventos Discretos (*Discrete Event System Specification* - DEVS) [8] es un formalismo modular y jerárquico para modelar y analizar sistemas de diversos tipos; en particular, sistemas de eventos discretos, sistemas de ecuaciones diferenciales (sistemas continuos) y sistemas híbridos (continuos y discretos). DEVS proporciona una base teórica de un sistema para ejecutar modelos usando el protocolo de simulación DEVS [8], [9], [10]. Sus modelos son de naturaleza jerárquica; se componen de modelos atómicos y modelos acoplados, con el fin de construir diseños con distintos niveles de abstracción.

Actualmente existe una necesidad importante de disponer de herramientas de simulación de los modelos dinámicos de UML, a fin de analizar el comportamiento real de sistemas complejos. Mas aún, es recomendable aplicar técnicas de modelado y simulación en etapas tempranas del desarrollo de software, dado que éstas ayudan a detectar problemas poco visibles antes de las implementaciones. UML es muy poderoso en términos de su representación gráfica, pero débil en cuanto a la ejecución de sus modelos dinámicos. Dentro de la comunidad de ingeniería de software, las *máquinas de estados* son uno de los lenguajes de modelado de sistemas dinámicos de UML más utilizados [7]. El presente trabajo propone potenciar la simulación de éstas utilizando en el formalismo DEVS. En la actualidad, DEVS es muy utilizado por ingenieros dentro del ámbito académico, que entienden de modelado de sistemas de eventos discretos y que son capaces de traducir requisitos de sistemas a códigos de modelos DEVS.

La principal contribución del trabajo es el desarrollo de un mecanismo que permita ejecutar y analizar máquinas de estados UML a través del formalismo de modelado y simulación (Modeling & Simulation - M&S) DEVS. Se define formalmente la vinculación entre los elementos de las máquinas de estados (StateCharts - SCs), propuestas por Harel [11], y los elementos del formalismo DEVS [8]. Además, se define una representación (metamodelo) de DEVS y se presenta un mapeo desde SCs de UML a modelos DEVS, a través de reglas de transformación en el lenguaje QVT Relations (QVT-R) [12]. Por último, se construye la implementación de un modelo DEVS que pueda ser importado por la herramienta de Simulación PowerDevs [13] para la ejecución y análisis del modelo.

978-1-4799-6130-6/14/$31.00

El resto del artículo está organizado de la siguiente manera. En la sección II describimos las motivaciones que dan origen a la actual propuesta, junto a los trabajos que se vinculan directamente con ésta. En la sección III introducimos los componentes teóricos que constituyen la base de la problemática planteada, tales como MDA, las SCs y el formalismo DEVS. En la sección IV presentamos formalmente el proceso de transformación de SCs a modelos DEVS. En la sección V definimos, en primer lugar, las relaciones en QVT-R que implementan el proceso, y posteriormente la generación de código C++ para ser utilizado por PowerDevs. Luego, en la sección VI incluimos un ejemplo de aplicación: el análisis de un cajero automático de un banco. Finalmente, en la sección VII detallamos las conclusiones y los trabajos futuros.

II. Motivación y Trabajos Relacionados

La transformación de componentes de UML a DEVS (y viceversa) no es una idea nueva, ni una iniciativa aislada. Integrar y unificar las comunidades que utilizan a uno u otro como método de modelado y simulación de sistemas, ha sido y es el objetivo principal de muchos trabajos de investigación que han dado origen a distintas metodologías, procesos e incluso a la definición y el desarrollo de frameworks específicos.

Hay diversas herramientas que dan soporte a UML (*UML Computer Aided Software Engineering* - UML CASE), tales como IBM Rational Rose [14], Poseidon [15], entre otras, y que proveen funcionalidades de simulación. Todas ellas tienen motores de simulación propietarios y la mayoría de los motores no son extensibles, razón por la cual no pueden ser adaptadas para cubrir necesidades específicas. Para la actual propuesta se necesita un motor de simulación especializado para dar soporte a los detalles de los procesos de simulación (administración de eventos, de tiempos, de distribuciones de probabilidad, etc.). En particular, el motor debe implementar el formalismo DEVS; por ejemplo, *PowerDevs*.

En cuanto a la representación de los modelos DEVS, existen distintos enfoques. Por ejemplo, *Scalable Entity Structure Modeler* con *Complexity Measures* (SESM/CM) [16] es adecuada para el desarrollo de modelos jerárquicos basados en componentes, ofrece una base para modelar aspectos de comportamiento de modelos atómicos, provista por una especificación estructural, y el almacenamiento de los modelos usando XML. No osbtante, este enfoque es más cercano a los expertos de simulación que a los expertos de dominios. Se trata de una herramienta que necesita mayor desarrollo para representar modelos atómicos usando XML.

En [17] y en [18] se define otro metamodelo para representar un sistema DEVS a través de XML. En ambos casos, JavaML [19] se utiliza para describir el comportamiento de un modelo atómico. Esta solución es adecuada para transformar modelos de una plataforma específica (PSM) a modelos independientes de una plataforma (PIM), pero no es un medio que permita brindar una solución gráfica para el modelado de dichos sistemas. Dentro de este contexto, ATOM3 [20] es una buena solución, ya que tiene una capa de metamodelado que permite la especificación de lenguajes de modelado específicos de un dominio, y una capa de modelado que soporta la construcción, modificación y transformación de modelos de dominio.

En el campo del modelado y simulación basados en UML, varios autores han abordado el tema desde diversas perspectivas. Choi [21] utiliza diagramas de secuencia UML para definir el comportamiento de un sistema. En [22] se introducen ocho pasos para construir modelos DEVS usando UML, pero en este caso se necesitan muchas decisiones humanas para transformar el modelo.

En [23], Zinoviev presenta un mapeo formal de DEVS a UML. Dentro de esta técnica, puertos de entrada y salida son mapeados a eventos UML. Variables de estado DEVS no continuas son mapeadas a estados UML y aquellas que si son continuas son mapeadas a atributos de un estado UML. El mapeo presentado es elegante, sin embargo su representación en UML no tiende a proveer una representación unificada sobre el formalismo de modelado.

Huang y Sarjoughian definen en [24] un mapeo para modelos acoplados en diagramas de estructura UML-RT [25], [26], pero el uso de perfiles UML para planificación, rendimiento y especificación temporal (OMG 2005) es innecesario para el mapeo de DEVS a UML. Concluyen que UML-RT no es adecuado para un entorno de simulación, y afirman que el diseño de software y la simulación son inherentemente distintos.

En [27] se introduce un mapeo informal de DEVS a una *statechart* STATEMATE equivalente. Shulz indica que DEVS posee un mayor poder expresivo que los *statechart* [28] y que cualquier modelo DEVS puede ser representado por un diagrama de actividades STATEMATE, junto con una apropiada convención de identificadores para los eventos.

En el contexto de MDA, Tolk y Muguira [29] muestran de que manera las ideas complementarias y los métodos de *High Level Architecture* (HLA) y DEVS pueden ser integrados en un dominio de aplicación M&S bien definido, dentro del framework MDS. HLA tiene arquitectura de simulación distribuida, independiente de la plataforma de computación. Provee una intefaz en la cual cada motor de simulación debe conformarse para participar en un ejercicio de simulación distribuida. Mientras que es ampliamente utilizado en la industria de defensa, su adopción a la industria comercial ha sido restringida debido a la falta de poder de expresividad.

El trabajo que más se relaciona con nuestra propuesta es [30]. Aquí, Mittal y Risco-Martín presentan *eUDEVS (Executable UML with DEVS Theory of Modeling and Simulation)*. Los autores analizan no sólo la especificación de la estructura y el comportamiento de los modelos DEVS en UML, sino también un procedimiento de modelado de propósito general para modelos DEVS, que soporta la especificación, el análisis, el diseño, la verificación y la validación de una amplia variedad de sistemas basados en DEVS. Se propone una metodología de M&S basada en UML que consiste en tres pasos. Primero, se sintetiza la estructura estática de la máquina de estados UML y se genera su correspondiente representación en XML (SCXML). Segundo, se transforma el archivo SCXML a un modelo de máquina de estados DEVS finita y determinística (*Finte Deterministic DEVS* - FD-DEVS), definida por Mittal en [31], que especifica su comportamiento. En esta etapa el modelo es totalmente independiente de plataforma (PIM). Finalmente, a partir del modelo XML FD-DEVS se genera un modelo de simulación (PSM, específico de la plataforma)

utilizando una serie de transformaciones basadas en XML para ser ejecutado por el motor de simulación DEVSJAVA [32]. El método propuesto es interesante y ambicioso, pero la utlización de transformaciones XSL (*eXtensible Stylesheet Language Transformations* - XSLT) lo hacen poco claro.

El presente trabajo se diferencia de los anteriores, y en especial de [30], en los siguientes aspectos: (1) se presenta una propuesta en el contexto de MDA; (2) se define un mapeo formal entre las SCs definidas por Harel y los modelos DEVS; (3) se define una transformación en QVT-R que implementa el mapeo desde máquinas de estados de UML a modelos DEVS. Se utiliza un lenguaje declarativo y estandarizado por el OMG en lugar de un lenguaje imperativo o herramientas de transformación de XML. La instrumentación de reglas de transformación a través de definiciones XML conduce a desarrollos, en general, poco claros y difíciles de analizar; por último, (4) se genera código C++ de modelos DEVS para ser importado por la herramienta (open source) de simulación PowerDEVS.

III. Nociones Preliminares

A. Model Driver Architecture (MDA)

MDA es una iniciativa del OMG, promovida a partir del año 2000, que propone un conjunto de estándares no propietarios que especifican tecnologías interoperables con las cuales implementar los principios de MDD, con transformaciones automatizadas. MDA define un proceso de construcción de software basado en la producción y transformación de modelos. Los principios en los cuales se fundamenta MDA son: abstracción, automatización y estandarización. El proceso central de MDA es la transformación de modelos. La idea principal subyacente es utilizar modelos, de manera que las propiedades y características de los sistemas queden contenidas en un modelo abstracto independiente de los cambios producidos en la tecnología. MDA proporciona una serie de guías o patrones expresadas como modelos y establece cuatro niveles de abstracción: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM), y la aplicación final. Los modelos CIM describen el entorno en el que se utilizará el sistema, sin referencia directa a su implementación. Los PIM modelan funcionalidad y estructura de un sistema de información sin considerar detalles tecnológicos de la plataforma en la cual se implementará el sistema. Los PSM describen los modelos específicos de plataforma donde se ejecutará el sistema. MDA plantea el siguiente proceso de desarrollo: de los requisitos se obtiene un PIM, luego este modelo es transformado con la ayuda de herramientas en uno o más PSM, y finalmente cada PSM es transformado en el código. Por lo tanto, MDA incorpora la idea de transformación de modelos (CIM a PIM, PIM a PSM, PSM a código), siendo necesaria la utilización de herramientas para la automatización de estas actividades. La figura 1 muestra el proceso y los roles definidos por MDA.

El principio de abstracción utilizado por MDA centra su atención en el dominio del problema más que en la tecnología. Los diferentes modelos tienen por finalidad la definición de una semántica que separe aspectos relevantes del problema de los vinculados a la tecnología. Respecto a la automatización, MDA favoreció al surgimiento de nuevas herramientas CASE con funcionalidades específicas destinadas al intercambio de modelos, verificación de consistencia, transformación de modelos y manejo de metamodelos, entre otras.

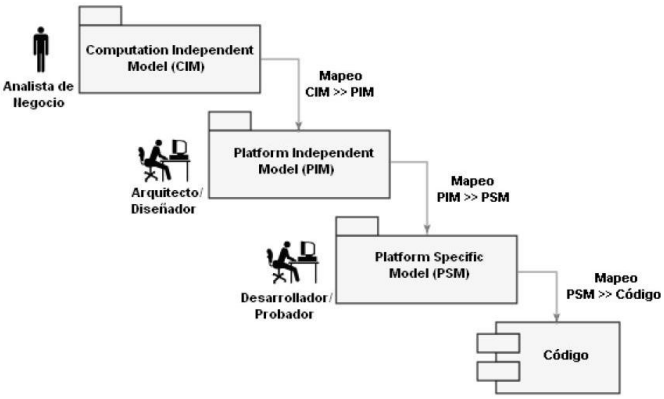

Fig. 1: Proceso y roles en MDA.

Mapeo y Transformación de Modelos

En la figura 1 se establecen una serie de mapeos entre modelos pertenecientes a diferentes niveles de abstracción. También es posible definirlos entre modelos pertenecientes a un mismo nivel. A continuación se muestran algunas de las transformaciones posibles:

- Mapeo de PIM a PIM. Es aplicado en modelos con el fin de optimizarlos durante el transcurso del proceso de desarrollo. Este mapeo no requiere de información alguna de la plataforma de implementación.
- Mapeo de PIM a PSM. Es aplicado en aquellos casos donde se cuenta con un PIM resultante de una serie de refinamientos de PIM a PIM. Este mapeo será implantado en una arquitectura dependiente de una plataforma específica.
- Mapeo de PSM a PSM. Es aplicado durante la codificación y el desarrollo de componentes. Este mapeo está vinculado al refinamiento propio de modelos PSM.
- Mapeo de PSM a PIM. Esta transformación puede ser requerida para generar modelos abstractos a partir de implementaciones existentes. Puede ser utilizado con la finalidad de extraer en modelos genéricos propiedades deseables de los sistemas.
- Mapeo de PSM a Código. Permite la generación del código fuente específico para una plataforma particular a partir de un PSM.

De una variedad de lenguajes existentes para describir transformaciones modelo a modelo, los más usados actualmente son QVT, en su versión imperativa (*Operational*) o declarativa (*Relations*), y ATL (Atlas Transformation Language) [33]. En el actual trabajo se optó por QVT-R por su simplicidad y claridad para definir declarativamente relaciones entre los elementos de los metamodelos origen y destino.

Query/View/Transformation (QVT): El OMG definió el estándar QVT [12] para trabajar con modelos de software. QVT consta de tres partes: consulta, vista y transformaciones. En particular, una transformación describe relaciones entre un metamodelo fuente *F* y un metamodelo objetivo *O*, especificados en MOF [34]. La especificación de QVT 1.1 tiene una naturaleza híbrida declarativa/imperativa. En este trabajo se utiliza el lenguaje *Relations* que tiene naturaleza declarativa. La herramienta utilizada para la definición de las transformaciones se denomina MediniQVT [35]. Esta herramienta implementa la especificación QVT-R del OMG en un poderoso motor QVT. Su interface está basada en Eclipse y utiliza Eclipse Modeling Framework (EMF) [36] para la representación de los modelos. MediniQVT exige que los metamodelos (y modelos) sean escritos en una variante simplificada del estándar MOF, llamada Ecore [37], la cual ha sido definida en el EMF.

Características de Eclipse Modeling Framework y Ecore: Los metamodelos y modelos usados por EMF se representan con documentos XML; EMF posee un editor gráfico de modelos Ecore que facilita la creación de los mismos. Existen herramientas que tratan automáticamente estos metamodelos y modelos, entre ellas, el plug-in de Eclipse para transformar este tipo de modelos, con la definición de transformaciones QVT. Las características y elementos del lenguaje de modelado Ecore son los siguientes: el elemento aglutinador (raíz) es el paquete `EPackage`, que contiene físicamente a sus elementos (especificación de `containment`) y, a su vez, éstos pueden estar contenidos en otros paquetes. Hay una fábrica (`EFactory`) por paquete que permite la creación de los elementos del modelo. Las construcciones que describen a un conjunto de elementos (instancias) son clasificadores (`EClassifiers`): `EClass` y `EDataType`. Ecore especifica las características de las clases (`EClass`), sus características estructurales (`EStructuralFeatures`), atributos (`EAttributes`), operaciones y relaciones (herencia, referencia (`EReference`)). Las `EClasses` tienen superclase y están compuestas por características estructurales (`EStructuralFeatures`: `EReference` y `EAttribute`). Tanto las `EReferences` como los `EAttributes` pueden estar dotados de multiplicidad. Los `EDataTypes` modelan tipos básicos o indivisibles del modelo de datos, y los `EReferences` pueden estar contenidos o ser referencias (punteros). Las `Operations` modelan operaciones de la interfaz (aunque no se provee implementación para ellas). Todos los elementos heredan de `ENamedElement` (tienen nombre), y de `EModelElement` (elemento del modelo). Además, todo elemento del modelo puede tener asociadas anotaciones (`EAnnotation`): pares nombre/valor para especificaciones extra; por ejemplo, restricciones OCL o cadenas de documentación.

B. Máquinas de Estados

Una SC es una representación visual de un autómata de estados finito con jerarquía de estados. Estas máquinas fueron introducidas por Harel [11] e incorporadas a las diferentes versiones de UML con algunas variaciones. La principal característica de las SCs es que sus estados pueden refinarse, definiendo así una jerarquía de estados. La descomposición de un estado puede ser secuencial o paralela, en la primera, un estado se descompone en un autómata, mientras que en la segunda se descompone en dos o más autómatas que se ejecutan concurrentemente. Aunque permiten esta jerarquía, una SC con estados compuestos tiene su equivalente utlizando únicamente estados simples. En este trabajo, nos basaremos en la definición y aplicación de SCs que contienen sólo estados simples. Existen muchos algoritmos y herramientas que convierten una SC con estados compuestos a su equivalente SC con estados simples [11], [38], [39].

Las transiciones son dirigidas. Una transición (*t*) está formada por su nombre, el estado origen, el evento que la "dispara" (*e*), la condición de disparo (*c*), las acciones a ejecutar (α) y el estado destino. La notación gráfica utilizada es $t : e, c/\alpha$.

C. M&S y DEVS

Durante las últimas décadas, la rápida evolución de la tecnología ha producido una proliferación de nuevos sistemas dinámicos, de gran complejidad. Ejemplos de ellos son las redes de computadoras, sistemas de producción automatizados, de control de tráfico aéreo, ; y sistemas en general de comando, de control, de comunicaciones y de información. Todas las actividades en estos sistemas se deben a la ocurrencia asincrónica de eventos discretos, algunos controlados (tales como el pulsado de una tecla) y otros no (como la falla espontánea de un equipo). Esta característica es la que lleva a definir el término: Sistemas de Eventos Discretos (Discrete-Event Systems - DES) [40]. Dentro de los formalismos más populares de representación de DES están las Redes de Petri, las SC, Grafcet, Grafos de Eventos, y muchas generalizaciones y particularizaciones de los mismos.

Orientado a los problemas de modelización y simulación de DES, en la década del '70 Bernard Zeigler propuso un marco teórico y una metodología para M&S de sistemas. Es aquí que surge DEVS (Discrete Event System Specification) [8], un formalismo de modelización con una sólida semántica, fundado en una base teórica de sistemas. Este formalismo, a veces caracterizado como universal, permite describir sistemas dinámicos a eventos discretos. Su universalidad se refiere a que cualquier otro formalismo para estos sistemas puede ser ajustado a DEVS. Por su gran adaptación para modelizar sistemas complejos y a la simplicidad y eficiencia de la implementación de simulaciones, DEVS es actualmente una de las herramientas más utilizadas en modelado y simulación por eventos discretos. En la sección IV-B se presentan los detalles del formalismo.

IV. De Máquinas de Estados a Modelos DEVS

El objetivo principal de este trabajo es proveer un mecanismo que logre ejecutar SCs UML, mediante un proceso que las transforme a modelos atómicos DEVS, permitiendo así llevar a cabo simulaciones sobre ellas utilizando herramietas para tal fin. Para elaborar dicho proceso de transformación es necesario hacer una revisión previa de los elementos de cada formalismo. Posteriormente, se define un mapeo entre ellos a través de un conjunto de reglas que denotan como la información de un dominio se traduce a otro. Estas reglas son especialmente necesarias dado que si bien el dominio de elementos de las SC tiene aspectos similares a DEVS, la especificación de DEVS es más estricta.

A. Definición Formal de las SC

Las SCs UML están basadas en las SCs definidas por Harel. Están compuestas por un conjuntos de estados, transiciones y eventos. Formalmente, siguiendo [41], una SC se define como una tupla $\langle S, \Sigma, T, A \rangle$, donde:

- S: es un conjunto finito de estados (no vacío).
- Σ: es un conjunto finito de eventos.
- $(A \subseteq \Sigma)$: es el conjunto de acciones, donde $\tau \in A$ representa la acción nula.
- T es un conjunto finito de transiciones que se representan mediante la tupla $t = (t', s_o, e, c, a, s_d)$, donde:
  - t' es el nombre de la transición.
  - $s_o \in S$ es el estado origien.
  - $s_d \in S$ es el estado destino.
  - $e \in \Sigma$ es el evento que dispara la transición.
  - c es una condición de disparo sobre la transición.
  - $a \in A$ es una acción que se ejecuta cuando se efectúa la transición.

Por practicidad, se puede asociar a un estado un tiempo de permanencia. Estos son utilizados para describir un nuevo comportamiento, cuya semántica define el tiempo en que el sistema puede permanecer en un estado. Al transcurrir dicho tiempo debe dispararse una transición definida para tal situación. Este comportamiento también se podría implementar asociando un evento de tiempo a una transición, la cual se dispara cuando el estado origen alcanza dicho tiempo. Por un lado, se formaliza la asociación de un tiempo a cada estado, y por otro, se distingue un evento especial $\gamma \in \Sigma$ denominado *evento de tiempo* para ser utilizado en aquellas transiciones que se dispararán al alcanzar el tiempo de permanencia en su estado origen. Denominamos a éstas *transiciones de tiempo*. Sólo una *transición de tiempo* debe definirse por cada estado, cuyo tiempo de permanencia es distinto de ∞.

Se extiende la definición de una SC con los siguientes componentes: $\langle S, \Sigma, T, A, \Pi \rangle$, donde:

- $\Pi : S \times \Re^{+}_{0,\infty}$ es un conjunto de pares (función) que asocia a cada estado un valor en $\Re^{+}_{0,\infty}$, que representa el tiempo de permanencia.
- $\gamma \in \Sigma$ es el *evento de tiempo*.
- Si $\Pi(s) \neq \infty$ entonces debe existir una única *transición de tiempo* $(t', s, \gamma, true, a, s_d)$ con origen s.

La figura 4 muestra un ejemplo de una SC. Notar que las transiciones que se disparan cuando se agota el tiempo de permanencia en su estado origen (*transiciones de tiempo*), se dibujan simplemente asociandole el tiempo de permanencia de su estado origen y la acción de salida; su evento (evento de tiempo tau) y condición ($true$) no se grafican.

B. Definición del Formalismo DEVS

Un modelo atómico DEVS clásico se define como una tupla $\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$, donde:

- X es el conjunto de eventos de entrada (posiblemente infinito).
- Y es el conjunto de eventos de salida (posiblemente infinito).
- S es el conjunto de estados (posiblemente infinito).
- $\delta_{int} : S \to S$ es la función de *transición interna* que define como un estado del sistema cambia internamente, cuando el tiempo de simulación transcurrido alcanza el tiempo de vida del estado.
- $\delta_{ext} : Q \times X \to S$ es la función de *transición externa*, con: $Q = \{(s, e) \,|\, s \in S, 0 \leq e \leq ta(s)\}$ el conjunto de estados total y e el *tiempo transcurrido* desde la última transición.
- $\lambda : S \to Y$es la función de salida. Esta función define como un estado del sistema genera un evento de salida, cuando el tiempo de simulación transcurrido alcanza el tiempo de permanencia en el estado.
- $ta : S \to \Re^{+}_{0,\infty}$ es la función de *avance de tiempo*, la cual se usa para determinar el tiempo de permanencia en un estado.

Un sistema que está en un estado $s \in S$, permanecerá en el mismo por un tiempo $ta(s)$ salvo que, habiendo transcurrido un tiempo e menor o igual que $ta(s)$ en ese estado, ocurra un evento de entrada con valor $x \in X$. En tal caso, el sistema experimentará una *transición externa* transicionando al estado $s' = \delta_{ext}(s, e, x)$. En cambio, si el tiempo transcurrido e es igual a $ta(s)$ sin que se hayan producido eventos externos, un evento interno ocurrirá dando lugar a una *transición interna*. Esto producirá un evento de salida con valor $y = \lambda(s)$ y un cambio a un nuevo estado $s'' = \delta_{int}(s)$. En ambos casos, el sistema permanecerá en el nuevo estado, por el tiempo que ta lo determine o hasta que otra vez ocurra un evento externo.

Notar que en la definición de DEVS presentada, la función de *avance de tiempo* admite asociar un tiempo 0 o ∞ a un estado $s \in S$. En el primer caso, el sistema sólo podrá permanecer 0 instantes de tiempo en s. Por lo tanto, cuando se alcance s, inmediatamente sucederá una transición interna generando un evento de salida y un cambio de estado. Este tipo de estado es llamado *estado transitorio*. Por otra parte, cuando el sistema alcance s, cuyo tiempo asociado sea ∞, no existirán transiciones internas y permanecerá por siempre en este estado, salvo que, un evento externo ocurra. Este tipo de estado es llamado *estado pasivo*.

C. Mapeo de Elementos

Las transformaciones de modelos se definen como programas que toman un modelo (o más de uno) como entrada y devuelven otro modelo (o más de uno) como salida. Están formadas por un conjunto de reglas que describen cómo uno o más elementos del modelo origen son transformados en uno o más elementos del modelo destino. A continuación se define formalmente el conjunto de reglas que describen de que manera los elementos del modelo origen se transforman en elementos del modelo destino, extrayendo la información necesaria de una SC para construir un modelo atómico DEVS.

Seas ME = $\langle S_{uml}, \Sigma, T, A, \Pi \rangle$ una SC, y MA = $\langle X, Y, S_{devs}, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$ un modelo atómico DEVS, las reglas de transformación se definen como sigue:

1) *Estados*: los estados de S_{uml} y sus respectivos tiempo de permanencia, se transforman a estados DEVS, donde: $S_{devs} = \{(s, \sigma) | s \in S_{uml}, \Pi(s) = \sigma\}$. Por otro lado, la función ta se define como: $ta = \Pi$
2) *Eventos de entrada*: un evento $e \in \Sigma$ se transforma en un evento de entrada DEVS con el mismo nombre; $X = \Sigma - \gamma$
3) *Eventos de salida*: una acción $a \in A$ se transforma a un evento de salida DEVS con el mismo nombre; $Y = A$
4) *Transiciones*: para definir la transformación de las transiciones es necesario disntinguir casos particulares de éstas, ya que en ME toda la información se encuentra contenida en la tupla que define a la transición, mientras que en un modelo DEVS la misma es especificada en distintos elementos del modelo atómico. Esta clasificación consiste en diferenciar a las trasiciones según su disparador (evento de tiempo y el resto), y según contenga definida una determinada acción a realizar o bien la acción nula.
   - Una transición $t \in T$ que se dispara por la ocurrencia de un evento de tiempo (γ), se transforma en una transición interna en MA.
     Sea $t = (t', s_o, \gamma, true, a, s_d) \in T$, entonces: $\delta_{int}((s_o, \sigma)) = (s_d, \Pi(s_d))$, y $\lambda((s_o, \sigma)) = a$.
   - Una transición $t \in T$ que se dispara por la ocurrencia de un evento $e \neq \gamma$ y que contiene la acción nula τ, se transforma en una transición externa en MA.
     Sea $t = (t', s_o, e, c, \tau, s_d) \in T$, donde $e \neq \gamma$,
     if c then $\delta_{ext}(((s_o, \sigma), t_e, e) = (s_d, \Pi(s_d))$
     else $\delta_{ext}(((s_o, \sigma), t_e, e) = (s_o, \sigma - t_e))$.
   - Una transición $t \in T$ que se dispara por la ocurrencia de un evento $e \neq \gamma$ y que contiene la acción no nula $a \neq \tau$, produce los siguientes elementos en MA.
     Sea $t = (t', s_o, e, c, a, s_d) \in T$, donde $e \neq \gamma$ y $a \neq \tau$, entonces:
     a) Se define un estado intermedio $s_o - s_d$ con tiempo de vida 0, con el objetivo de producir como salida del modelo MA la acción a.
        $(s_o - s_d, 0) \in S_{devs}$
     b) Una transición externa de s_o a $s_o - s_d$.
        $\delta_{ext}(((s_o, \sigma), t_e), e) = (s_o - s_d, 0)$
     c) Una transición interna de $s_o - s_d$ a s_d.
        $\delta_{int}(s_o - s_d, \sigma) = (s_d, \Pi(s_d))$
     d) Un elemento de la funcion de salida que asocia $s_o - s_d$ con el evento de salida (acción) a.
        $\lambda(s_o - s_d) = a$

V. Implementacion de la Transformacion con QVT y exportación a PowerDevs

En esta sección se propone una implementación de las reglas de transformación de SCs UML a modelos atómicos DEVS, basada en el lenguaje QVT-R. Posteriormente, se describe la exportación del modelo resultante de tal manera que pueda ser interpretado por PowerDevs.

Se define un conjunto de relaciones QVT que implementan las reglas detalladas en la sección IV-C. Se utiliza en esta implementación el metamodelo de UML 2.4.1 descripto en el sitio oficial del OMG [6]. En función de dicha especificación, se describen previamente los elementos de UML que son utilzados para dar soporte al formalismo de las SCs definidas en IV-A. Luego, se define el metamodelo del formalismo DEVS presentado en IV-B, en formato Ecore.

Una ejecución de las reglas en QVT toma como parámetros un modelo (o instancia) de una SC UML que satisface el metamodelo UML en Ecore [36], y construye un modelo atómico DEVS que satisface el respectivo metamodelo en formato Ecore. La tabla I refleja brevemente cuales elementos de UML son utilizados para modelar las SCs definidas en IV-A.

TABLA I: Vinculación entre la definición formal de una SC y los elementos de UML 2.4.1.

Definición	Descripción	Ecore (EClass)
ME	Máquina de estados	StateMachine
S	Estado	State
Σ	Evento	SignalEvent
A	Acción	FunctionBehavior
T	Transición	Transition
Π	Tiempo de vida	Constraint

Metamodelo DEVS

La definición del metamodelo DEVS se basa en la especificación descripta en IV-B y se construye con la herramienta MediniQVT. La figura 2 muestra gráficamente dicho metamodelo.

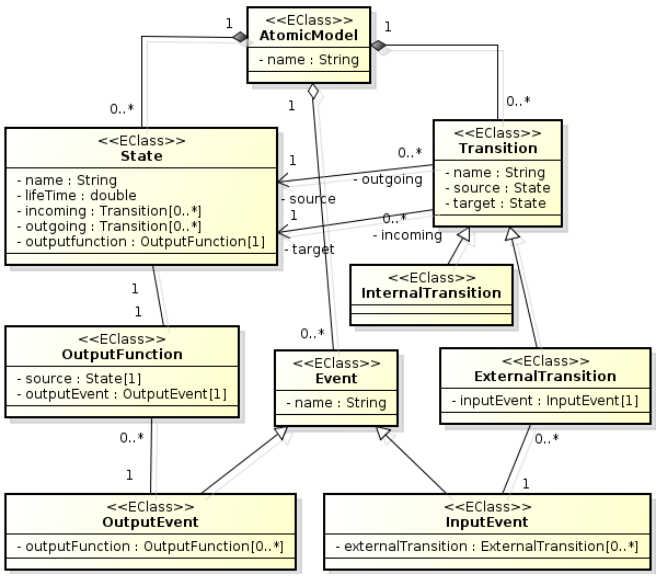

Fig. 2: Metamodelo del formalismo DEVS.

Brevemente, la tabla II describe la relación entre los elementos de la definición formal de un modelo atómico DEVS y los elementos del metamodelo en Ecore.

A. Reglas de transformación en QVT-R

La definición de las relaciones (reglas) en QVT-R implementan las reglas especificadas en IV-C. Dichas reglas de transformación se escriben de forma declarativa y describen

TABLA II: Vinculación entre la definición formal de un modelo atómico DEVS y su representación en Ecore.

Definición	Descripción	Ecore (EClass)
MA	Modelo atómico	AtomicModel
S	Estado	State
X	Evento de entrada	InputEvent
Y	Evento de salida	OutputEvent
δ_{int}	Transición interna	InternalTransition
δ_{ext}	Transición externa	ExternalTransition
λ	Función de salida	OutputFunction
ta	Tiempo de vida	lifeTime(Double)

relaciones consistentes entre el conjunto de elementos de cada modelo. Esta consistencia se puede verificar ejecutando la transformación en modo *checkonly* (sólo lectura), siendo un resultado satisfactorio si ambos modelos son consistentes según las relaciones. De la misma manera, se puede ejecutar en modo *enforce* para modificar o construir uno de los modelos, de tal manera que ambos satisfagan las relaciones al final de la ejecución. En este trabajo, las relaciones QVT se construyen marcando *checkonly* el modelo UML, y con *enforce* el modelo DEVS, mapeando así los elementos del primero a elementos del segundo.

Por cuestiones de espacio, se muestran las partes más relevantes del código de las relaciones. En particular, sólo se muestra la construcción de los estados, de los eventos de entrada y parcialmente la construcción de las transiciones internas y externas del modelo atómico DEVS. En https://www.dropbox.com/sh/4e0zorfyvgdmdg3/dnfWO_ija9 se puede descargar la definición completa de las reglas. Notar que una regla definida en IV-C puede estar implementada por más de una relación en QVT-R.

La relación `state2state` construye los estados del modelo DEVS según el siguiente mapeo: el nombre del estado DEVS se corresponde con el nombre que posee en el modelo UML y su tiempo de vida (lifeTime) con el tiempo especificado en la restriccion UML (Constraint).

```
top relation state2state {
  nameS: String;
  checkonly domain smUml s_source :uml::State {
    container = regionsource :uml::Region{
        stateMachine = sm :uml::StateMachine{} },
    stateInvariant = s_stateInvariant :uml::Constraint{
      specification = spec :uml::ValueSpecification{} },
    name = nameS
  };
  enforce domain smDevs s_target :devs::State {
    atomicModel = am :devs::AtomicModel{},
    lifeTime = spec.oclAsType(uml::LiteralDouble).value,
    name = nameS
  };
  when {
    statemachine2atomicmodel(sm,am);
  }
}
```

Los eventos de UML `SignalEvent` son mapeados a elementos `InputEvent` del metamodelo DEVS. Esto es definido por la relación `signalEvent2inputEvent`.

```
top relation signalEvent2inputEvent {
  nameT : String;
  checkonly domain smUml s_source :uml::SignalEvent {
    name = nameT
  };
  enforce domain smDevs s_target :devs::InputEvent {
    name = nameT
  };
}
```

Una transición de UML que se dispara por un evento externo y contiene una acción asociada produce la creación de varios elementos del modelo atómico DEVS resultante, acorde a la última regla definida en IV-C.

```
-- Transición origen.
top relation transition2mediatorState {
  nameT : String;
  checkonly domain smUml s_source :uml::Transition {
  container = regionsource :uml::Region{
    stateMachine = sm :uml::StateMachine{} },
  source = ss_uml :uml::State{},
  target = st_uml :uml::State{},
  trigger = t_uml :uml::Trigger{
    event = se_uml :uml::SignalEvent{} },
  effect = a_uml :uml::Activity{
    classifierBehavior = fb_uml :uml::FunctionBehavior{} },
  name = nameT
};

-- Estado Intermedio
  enforce domain smDevs s_target_MS :devs::State {
  atomicModel = am :devs::AtomicModel{},
  lifeTime = 0.0,
  name = ss_uml.name + '-' + st_uml.name
  };

-- Transición externa
  enforce domain smDevs s_target_ET :devs::
  ExternalTransition {
    atomicModel = am :devs::AtomicModel{},
    source = ss_devs :devs::State{},
    target = s_target_MS,
    inputEvent = ie_devs :devs::InputEvent{},
    name = ss_uml.name + 'to' + s_target_MS.name +'(?' +
      t_uml.name+')'
  };

- Transición interna
  enforce domain smDevs s_target_ITTarget :devs::
  InternalTransition {
    atomicModel = am :devs::AtomicModel{},
    source = s_target_MS,
    target = st_devs :devs::State{},
    name = s_target_MS.name + 'to' + st_uml.name
  };

-- Función de salida
  enforce domain smDevs s_target_OF :devs::OutputFunction {
    atomicModel = am :devs::AtomicModel{},
    state = s_target_MS,
    outputEvent = oe_devs :devs::OutputEvent{},
    name = s_target_MS.name + '(!' + fb_uml.name + ')'
  };

-- Precondiciones
  when {
    state2state(ss_uml,ss_devs);
    state2state(st_uml,st_devs);
    signalEvent2inputEvent(se_uml, ie_devs);
    function2outputEvent(fb_uml,oe_devs);
    isUmlExternalTransition(s_source);
    statemachine2atomicmodel(sm,am);
  }
}
```

B. Exportación a PowerDevs

El modelo DEVS resultante de la transformación puede ser exportado a distintas herramientas para realizar simulaciones; entre otras, PowerDevs, DEVSJAVA [32], DEVS-C++ [42], DEVSim++ [43], CD++ [44] y JDEVS [45]. Estas herramientas de software ofrecen diferentes características, las cuales incluyen interfaces gráficas y funcionalidades de simulación avanzadas de propósito general y de dominio

específico para modelos DEVS. En este trabajo hemos elegido PowerDevs, por ser una herramienta de código abierto (open source) y sencilla para implementar estos modelos; además, posee versiones tanto para *windows* como para *linux* y es utilizada ampliamente en el ámbito académico para la enseñanza de M&S.

PowerDevs está implementado en C++ (con librerías gráficas *QT*) y sus modelos DEVS deben ser definidos también en C++. Se compone de varios programas independientes:

- *Model Editor*: contiene la interfaz gráfica que permite, entre otras definiciones de alto nivel, la construcción jerárquica de la estructura de los modelos DEVS (archivos .pdm), los cuales tienen una sintaxis especial.
- *Atomic Editor*: permite definir en C++ (archivos .h y .cpp) el comportamiento de los modelos atómicos DEVS, es decir, las funciones de transición, la función de salida, el tiempo de vida y demás elementos que componen dichos modelos.
- *Preprocessor*: traduce los archivos del *Model Editor* en estructuras con información necesaria para construir el código de simulación, y enlaza los archivos creados con el *Atomic Editor* compilando a un archivo *stand-alone* ejecutable.
- *Simulation Interface*: ejecuta los archivos *stand-alone* ejecutables, permitiendo variar los parámetros de la simulación.
- Una instancia de ejecución de *Scilab* donde los parámetros de la simulación son leídos y los resultados pueden ser exportados.

La exportación de un modelo atómico DEVS en Ecore produce, por un lado, un archivo (.pdm) que define la estructura del modelo y puede ser abierto por el *Model Editor*; y por otro lado, los archivos que definen el comportamiento (archivos .h y .cpp) de tal manera que puedan ser interpretados por el *Atomic Model*. El algoritmo que produce la generación de código (*XEP: XML Ecore to PowerDEVS*) fue implementado por los autores del actual trabajo y está basado en la librería *Saxon XSLT and XQuery Processor* [46], la cual es una colección de herramientas para procesar documentos XML[1].

El proceso de exportación descripto es representado en la figura 3. En la sección VI se puede observar su aplicación a través de un ejemplo del análisis de un cajero automático.

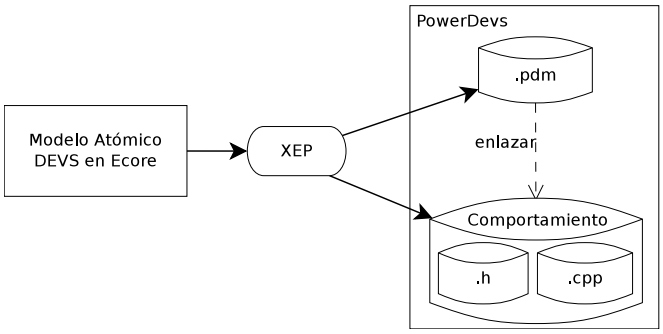

Fig. 3: Exportación de modelos atómicos DEVS en Ecore a PowerDevs.

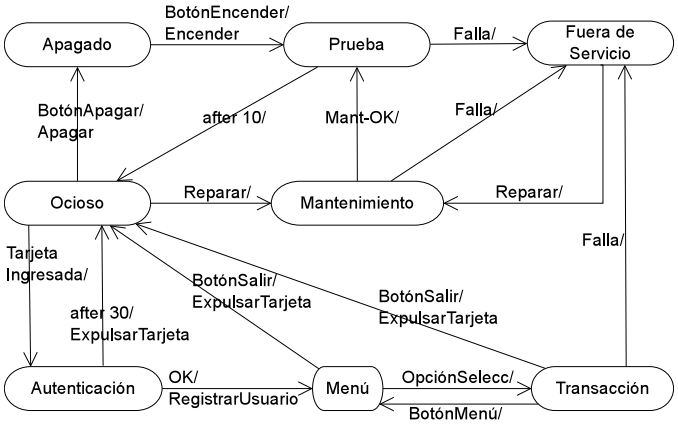

Fig. 4: SC UML de un Cajero Automático de un Banco.

VI. Ejemplo de Aplicación

El ejemplo muestra el funcionamiento simplificado de un *cajero automático* de un banco (CAB), siendo éstos máquinas que le permiten a un usuario seleccionar distintos tipos de transacciones bancarias mediante el intercambio de datos, a través de una tarjeta magnética. Además, la máquina dispone de mecanismos para diagnosticar fallas y proveer información al personal de mantenimiento para llevar a cabo la reparación.

Por limitaciones de espacio, a continuación se describe sólo el comportamiento más relevante del CAB. Se considera que el CAB comienza apagado y, una vez conectado a la alimentación eléctrica, inicia un proceso de prueba para la puesta en marcha, quel dura unos 10 segundos. En caso de falla queda fuera de servicio, de lo contrario estará listo para realizar operaciones bancarias. Cuando el usuario ingresa su tarjeta debe autenticarse. En caso de fallar la autenticación, a los 30 segundos el CAB vuelve a estar listo para operar expulsando la tarjeta ingresada; en caso contrario, presenta un menú de opciones para comenzar una transacción. Al finalizar la transacción el usuario decide si volver al menú de opciones o salir del sistema. En caso de fallas, el CAB dispone de un proceso de diagnóstico y reparación que el personal de mantenimiento se encarga de configurar y ejecutar. Este procedimiento se puede realizar cuando el CAB se encuentra ocioso o fuera de servicio.

La figura 4 ilustra la SC del CAB utilizando la representación gráfica de UML 2.4.1.

La especificación formal de la SC del CAB (*Sc_Cab*) es la siguiente: $Sc\_Cab = \langle S, \Sigma, T, A, \Pi \rangle$, donde:

S:{*Apagado, Prueba, Fuera de Servicio, Ocioso, Mantenimiento, Autenticación, Menú, Transacción*} ;

Σ:{*BotónEncender, BotónApagar, BotónSalir, Ok, BotónMenú, TarjetaIngresada, OpciónSelecc, Falla, Reparar, Mant-OK*} ;

A:{*Apagar, Encender, ExpulsarTarjeta, RegistrarUsuario*} ;

[1]Los metamodelos de Ecore son archivos XML.

T:{*(Apagado→Prueba, Apagado, BotónEncender, true, Encender, Prueba), (Prueba→Fuera de Servicio, Falla, true, τ, Fuera de Servicio), (Ocioso→Apagado, Ocioso, BotónApagar, true, Apagar, Apagado), (Ocioso→Autenticación, Ocioso, TarjetaIngresada, true, τ, Autenticación), (Autenticación→Ocioso, Autenticación, γ, true, τ, Ocioso), (Prueba→Ocioso, Prueba, γ, true, τ, Ocioso), (Ocioso→Mantenimiento, Ocioso, Reparar, true, τ, Mantenimiento), (Mantenimiento→Fuera de Servicio, Mantenimiento, Falla, true, τ, Fuera de Servicio), (Mantenimiento→Prueba, Mantenimiento, Mant-OK , true, τ, Prueba), (Fuera de Servicio→Mantenimiento, Fuera de Servicio, Reparar, true, τ, Mantenimiento), (Autenticación→Menu, Autenticación, Ok, true, RegistrarUsuario, Menú), (Menú→Ocioso,Menú, BotónSalir, true, ExpulsarTarjeta, Ocioso), (Menú→Transacción, Menú, OpciónSeleccionada, true, τ, Transacción), (Transacción→Menú, Transacción, BotónMenú, true,τ, Menú), (Transacción→Fuera de Servicio, Transacción, Falla, true,τ, Fuera de Servicio), (Transacción→Ocioso, Transacción, BotónSalir, true, ExpulsarTarjeta, Ocioso)*} ;

Π:{*(Apagado, ∞), (Prueba, 10), (Fuera de Servicio, ∞), (Ocioso, ∞), (Mantenimiento, ∞), (Autenticación, 30), (Menú, ∞), (Transacción, ∞)*} .

De CAB UML a CAB DEVS

La ejecución de las reglas de transformación en QVT-R toma como argumento el modelo del CAB UML en formato Ecore y crea un modelo atómico DEVS. La figura 5 muestra un diagrama del modelo atómico DEVS resultante. Si bien los modelos DEVS no tienen una notación gráfica estandarizada debido a que sus conjuntos pueden ser infinitos, en este caso es posible graficar el modelo, ya que la cantidad de elementos transformados son finitos. Por convención, las transiciones internas, están dibujadas con líneas punteadas y la función de salida $lambda$ es representada implicitamente sobre las transiciones internas especificando la salida en función del estado origen, por ejemplo, $!ExpulsarTarjeta$. Los demás elementos son intuitivos en el diagrama.

Notar que la transformación de las transiciones que se disparan por la ocurrencia de eventos externos, y que tienen acciones como salida, generan un estado intermedio con tiempo de permanencia 0 $(cero)$, con el fin de producir una salida inmediata de la acción. Por ejemplo, la transición que va desde el estado $Apagado$ al estado $Prueba$ produce el estado intermedio $Apagado\text{-}Prueba$; cuando éste es alcanzado se dispara inmediantamente (debido a que el tiempo de vida es cero) una transición interna que cambia el CAB al estado $Prueba$, y además, lanza una salida de la acción $!Encender$ ($\lambda(Apagado\text{-}Prueba) = Encender$).

La especificación formal del modelo atómico DEVS del CAB (*Devs_Cab*) obtenido es como sigue:

Devs_Cab = $\langle X, Y, S, ta, \delta_{ext}, \delta_{int}, \lambda \rangle$, donde:

X:{*BotónApagar, BotónSalir, Ok, BotónMenú, TarjetaIngresada, OpciónSelecc, BotónEncender, Falla, Reparar*}

Y:{*Apagar, ExpulsarTarjeta, RegistrarUsuario, Encender*}

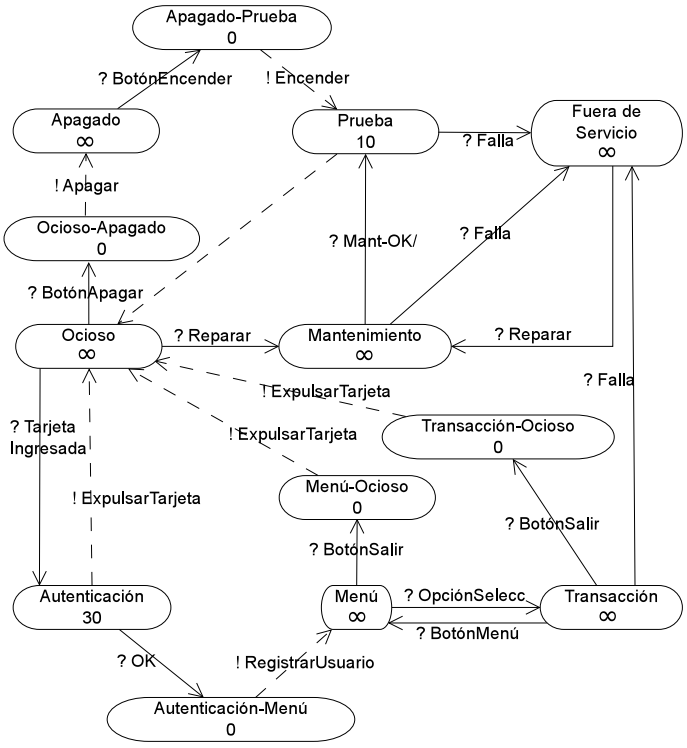

Fig. 5: Modelo DEVS del Cajero Automático de un Banco.

S:{*(d, σ) | d ∈ {Apagado, Prueba, Fuera de Servicio, Ocioso, Mantenimiento, Autenticación, Menú, Transformación*} ∧ $\sigma \in \Re^{+}_{0,\infty}$}

Las transiciones internas son:

δ_{int}*(Apagado-Prueba, σ) = (Prueba, 10)*

δ_{int}*(Prueba, σ) = (Ocioso, ∞)*

δ_{int}*(Ocioso-Apagado, σ) = (Apagado, ∞)*

δ_{int}*(Autenticación, σ) = (Ocioso, ∞)*

δ_{int}*(Autenticación-Menú, σ) = (Menú, ∞)*

δ_{int}*(Menú-Ocioso, σ) = (Ocioso, ∞)*

δ_{int}*(Transacción-Ocioso, σ) = (Ocioso, ∞)*

Las transiciones externas son:

δ_{ext}*(((Apagado, σ), t_e), ?BotónEncender) = (Apagado-Prueba, 0)*

δ_{ext}*(((Prueba, σ), t_e), ?Falla) = (Fuera de Servicio, ∞)*

δ_{ext}*(((Fuera de Servicio, σ), t_e), ?Reparar) = (Mantenimiento, ∞)*

δ_{ext}*(((Mantenimiento, σ), t_e), ?Falla) = (Fuera de Servicio, ∞)*

δ_{ext}*(((Ocioso, σ), t_e), ?Reparar) = (Mantenimiento, ∞)*

δ_{ext}*(((Ocioso, σ), t_e), ?BotónApagar) = (Ocioso-Apagado, 0)*

δ_{ext}(((Ocioso, σ), t_e), ?TarjetaIngresada) = (Autenticación, 30)

δ_{ext}(((Autenticación, σ), t_e), ?Ok) = (Autenticación-Menú, 0)

δ_{ext}(((Menó, σ), t_e), ?BotónSalir) = (Menú-Ocioso, 0)

δ_{ext}(((Menú, σ), t_e), ?OpciónSelecc) = (Transacción, ∞)

δ_{ext}(((Transacción, σ), t_e), ?BotónMenú) = (Menú, ∞)

δ_{ext}(((Transacción, σ), t_e), ?Falla) = (Fuera de Servicio, ∞)

δ_{ext}(((Transacción, σ), t_e), ?BotónSalir) = (Transacción-Ocioso, 0)

La función de salida es:

λ(Apagado-Prueba, σ) = !Encender

λ(Ocioso-Apagado, σ) = !Apagar

λ(Autenticación-Menú, σ) = !RegistrarUsuario

λ(Menú-Ocioso, σ) = !ExpulsarTarjeta

λ(Transacción-Ocioso, σ) = !ExpulsarTarjeta

De CAB DEVS en Ecore a PowerDevs

El modelo atómico DEVS en Ecore del CAB que resulta de la transformación, es traducido a código C++ por el algoritmo XEP desarrollado por los autores con el fin de ser interpretado y simulado por PowerDevs. Acorde a lo descripto en la sección V-B, se generan los archivos que definen la estructura (.pdm) y el comportamiento del modelo (.h y .cpp).

El siguiente código muestra el archivo CAB.pdm generado:

```
/*CAB.pdm*/
Coupled {
  Type = Root
  Name = CAB
  Ports = 0; 0
  Description =
  Graphic {
    Position = 0; 0
    Dimension = 600; 600
    Direction = Right
    Color = 15
    Icon = Window = 5000; 5000; 5000; 5000
  }
  Parameters {
  }
  System {
    Atomic {
      Name = CAB
      Ports = 1 ; 1
      Path = discrete\CAB.h
      Description = Atomic DEVS model
      Graphic {
        Position = -6105 ; -2610
        Dimension = 675 ; 720
        Direction = Right
        Color = 15
        Icon = None
       }
      Parameters {
      }
    }
  }
}
```

A continuación se exhibe el código C++ generado correspondiente al archivo de cabecera CAB.h:

```
/* File: CAB.h */
#if !defined CAB
#define CAB
#include "simulator.h"
#include "event.h"
#include "stdarg.h"

class CAB: public Simulator {
  // Declare the state,
  // output variables
  // and parameters

  char *s;
  double sigma;
  char *y;
  #define INF 1e20

  public:
    CAB(const char *n): Simulator(n) {};
    void init(double, ...);
    double ta(double t);
    void dint(double);
    void dext(Event , double );
    Event lambda(double);
    void exit();
};
#endif
```

Finalmente, por razones de espacio, a continuación sólo se muestran las partes más relevantes del código C++ generado (CAB.cpp) correspondiente al comportamiento del CAB DEVS en Ecore :

```
/* File: CAB.cpp */
#include "CAB.h"
void CAB::init(double t,...) {
  //The 'parameters' variable contains the parameters
  // transferred from the editor.
  va_list parameters;
  va_start(parameters,t);
  //To get a parameter: %Name% = va_arg(parameters,%Type%)
  //where:
  //  %Name% is the parameter name
  //  %Type% is the parameter type
  s = "Apagado";
  sigma = INF;
}

double CAB::ta(double t) {
  //This function returns a double.
  return sigma;
}

void CAB::dint(double t) {
  if (strcmp(s,"Prueba")== 0) {
      s = "Ocioso";
      sigma = INF
  };
  if (strcmp(s,"Autenticacion")== 0) {
    s = "Ocioso";
    sigma = INF
  };
  if (strcmp(s,"Apagado-Prueba")== 0) {
    s = "Prueba";
    sigma = 10.0;
  };
  ...
}

void CAB::dext(Event x, double t) {
  //The input event is in the 'x' variable.
  //where:
  //     'x.value' is the value (pointer to void)
  //     'x.port' is the port number
  //     'e' is the time elapsed since last transition
  if ((strcmp(s,"Prueba")== 0) && (strcmp((char*)x.value,
      "Falla")== 0)) {
    s = "FueraDeServicio";
    sigma = INF;
```

```
  };
  if ( (strcmp(s,"Ocioso")== 0) && (strcmp((char*)x.value,
      "TarjetaIngresada")== 0)) {
    s = "Autenticacion";
    sigma = 30.0;
  };
  ...
}

Event CAB::lambda(double t) {
  //This function returns an Event:
  // Event(%Value%, %NroPort%)
  //where:
  //%Value% points to the variable which contains the value.
  //%NroPort% is the port number (from 0 to n-1)
  if (strcmp(s,"Apagado-Prueba")== 0) {
    y = "Encender";
    return Event(y,0);
  };
  if (strcmp(s,"Ocioso-Apagado")== 0) {
    y = "Apagar";
    return Event(y,0);
  };
  ...
}

void CAB::exit() {
  //Code executed at the end of the simulation.
}
```

Todos los archivos que implementan el ejemplo de aplicación del CAB pueden descargarse desde https://www.dropbox.com/sh/4e0zorfyvgdmdg3/dnfWO_ija9. La herramienta MediniQVT puede descargarse en el enlace indicado en [35].

VII. Conclusiones y Trabajos Futuros

MDD es un enfoque con el potencial de hacer más eficiente el desarrollo y más confiables los resultados del mismo, ya que, entre otras cosas, habilita la verificación de los sistemas en etapas tempranas del desarrollo, ofreciendo mayor control. Gran parte de las técnicas de MDD utilizan UML, lenguaje incorporado como estándar de facto a nivel académico e industrial, que permite la descripción de múltiples aspectos de un sistema. En particular, las SCs de UML constituyen un mecanismo para especificar el comportamiento de sistemas mediante una representación gráfica. Estos diagramas son compactos, expresivos y proveen la capacidad de modelar no sólo sistemas simples sino también complejos sistemas reactivos. Son múltiples las herramientas desarrolladas que dan soporte a las últimas versiones de UML, las cuales proveen generalmente prestaciones gráficas que faciltan el modelado de los sistemas. Sin embargo, carecen de la capacidad de ejecución y simulación de los modelos dinámicos. Esto limita el análisis del comportamiento de los sistemas en escenarios reales. DEVS está fundado sobre los principios de teoría de sistemas y ha procurado su desarrollo mediante la ingeniería basada en componentes, desde sus inicios. Con los avances de UML en los últimos años, la comunidad entorno a DEVS ha dedicado esfuerzos para definir un mapeo entre los elementos de DEVS y los de UML. Es prominente la conclusión de que DEVS es más riguroso y expresivo que UML, pero debido a la manipulación de conjuntos posiblemente infinitos que puede contener un modelo, carece de una notación gráfica, la cual es muy requerida por profesionales en la industria. La combinación de las virtudes gráficas de UML con las poderosas herramientas de simulación de modelos DEVS ha dado lugar a la actual propuesta. Algunos grupos de investigación han encarado distintas propuestas de este importante vínculo entre DEVS y UML, pero ninguno presenta una solución formal y una implementación basada en la definición de metamodelos haciendo uso de estándares.

En este trabajo se presenta un mecanismo que permite cerrar la brecha entre dos formalismos con distintas herramientas y tecnologías, distintas bases teóricas, pero unidos por un mismo propósito, que es el dar soluciones a problemas reales a través de la creación y el análisis de modelos abstractos. Se define un proceso de transformación de modelos que mapea elementos de SCs UML a elementos de modelos DEVS, explotando las cualidades gráficas de uno y la especificidad y rigurosidad del otro, resultando en un modelo de simulación que puede ser ejecutado y analizado por un gran número de herramientas específicas. La transformación se definió como un conjunto de reglas por casos y se implementó utilizando el lenguaje estandarizado por el OMG, QVT-R. Para ello, se utilizó el metamodelo en formato Ecore de UML 2.4.1 y se definió en este trabajo el metamodelo del formalismo DEVS. El resultado de la transformación es un modelo DEVS en formato Ecore que satisface su correspondiente metamodelo. Este modelo provee un conjunto de información suficiente para ser exportado y ejecutado por distintas herramientas y motores de simulacion DEVS ya existentes, en particular, se definió aquí la exportación a PowerDevs. Además, se desarrolló un ejemplo parcial de un caso de estudio referido al comportamiento de un cajero automático de un banco, que muestra la aplicación del enfoque propuesto.

Esta investigación forma parte de un proyecto integral que en el futuro abordará las siguientes extensiones: (a) el desarrollo de una transformación directa de SCs UML con estados compuestos a modelos DEVS; (b) la incorporación de más elementos de UML a las SCs, incluyendo sus transformaciones a elementos de un modelo DEVS; (c) la extensión del formalismo DEVS con transiciones condicionales, con el objetivo de permitir agregar elementos de decisión al flujo del sistema; (d) la exportación del modelo DEVS en Ecore resultante a otras herramientas de simulación; y (e) la construcción de una herramienta que abarque el proceso completo de transformación, ejecución y análisis.

Referencias

[1] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: Model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[2] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, 2003.

[3] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[4] J. Miller and J. Mukerji, "Mda guide version 1.0.1," Object Management Group (OMG), Tech. Rep., 2003.

[5] *Object Management Group*, Object Management Group Std., Último acceso: Abril de 2014. [Online]. Available: http://www.omg.org

[6] *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*, Object Management Group Std., Rev. 2.4.1, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[7] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[8] B. P. Zeigler and S. Vahie, "Devs formalism and methodology: Unity of conception/diversity of application," in *Proceedings of the 25th Conference on Winter Simulation*, ser. WSC '93. New

York, NY, USA: ACM, 1993, pp. 573–579. [Online]. Available: http://doi.acm.org/10.1145/256563.256724

[9] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[10] H. G. Molter, "Discrete event system specification," in *SynDEVS Co-Design Flow*. Springer Fachmedien Wiesbaden, 2012, pp. 9–42. [Online]. Available: http://dx.doi.org/10.1007/978-3-658-00397-5_2

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[12] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, Object Management Group Std., Rev. 1.1, 2011. [Online]. Available: http://www.omg.org/spec/QVT/1.1/

[13] F. Bergero and E. Kofman, "Powerdevs: A tool for hybrid system modeling and real-time simulation," *Simulation*, vol. 87, no. 1-2, pp. 113–132, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1177/0037549710368029

[14] *IBM Rational Rose*, IBM, Último acceso: Abril de 2014. [Online]. Available: http://www.ibm.com/developerworks/rational/products/rose

[15] *Poseidon for UML*, Último acceso: Abril de 2014. [Online]. Available: http://www.gentleware.com/new-poseidon-for-uml-8-0.html

[16] B. S. and S. H. S., "Discrete-event behavioral modeling in sesm: Software design and implementation," in *Advanced Simulation Technology Conference*, 2005, pp. 23–28.

[17] P. P. Vladimir and P. Slavícek, "Towards devs meta language," in *Industrial Simulation Conference*, 2006, pp. 69–73.

[18] S. Mittal, J. L. Risco-Martín, and B. P. Zeigler, "Devsml: Automating devs execution over soa towards transparent simulators," in *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, ser. SpringSim '07. San Diego, CA, USA: Society for Computer Simulation International, 2007, pp. 287–295. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404680.1404725

[19] G. J. Badros, "Javaml: a markup language for java source code." *Computer Networks*, vol. 33, no. 1-6, pp. 159–177, 2000. [Online]. Available: http://dblp.uni-trier.de/db/journals/cn/cn33.html#Badros00

[20] J. de Lara and H. Vangheluwe, "Using atom3 as a meta-case tool." in *ICEIS*, 2002, pp. 642–649. [Online]. Available: http://dblp.uni-trier.de/db/conf/iceis/iceis2002.html#LaraV02

[21] K. Choi, S. Jung, H. Kim, D.-H. Bae, and D. Lee, "Uml-based modeling and simulation method for mission-critical real-time embedded system development," in *IASTED Conf. on Software Engineering*, P. Kokol, Ed. IASTED/ACTA Press, 2006, pp. 160–165.

[22] S. Y. Hong and T. G. Kim, "Embedding uml subset into object-oriented devs modeling process," in *Society of Modeling and Computer Simulation International*, 2004, pp. 161 – 166.

[23] D. Zinoviev, "Mapping devs models onto uml models," *CoRR*, vol. abs/cs/0508128, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0508.html#abs-cs-0508128

[24] D. Huang and H. S. Sarjoughian, "Software and simulation modeling for real-time software-intensive systems," in *DS-RT*. IEEE Computer Society, 2004, pp. 196–203.

[25] B. P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[26] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[27] S. S. II, T. C. Ewing, and J. W. Rozenblit, "Discrete event system specification (devs) and statemate statecharts equivalence for embedded systems modeling," in *ECBS*. IEEE Computer Society, 2000, pp. 308–.

[28] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1998.

[29] A. Tolk and J. A. Muguira, "M&s within the model driven architecture," in *Interservice/Industry Training, Simulation, and Education Conference*, 2004.

[30] J. L. Risco-Martín, J. M. de la Cruz, S. Mittal, and B. P. Zeigler, "eudevs: Executable uml with devs theory of modeling and simulation." *Simulation*, vol. 85, no. 11-12, pp. 750–777, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/simulation/simulation85.html#Risco-MartinCMZ09

[31] Z. B. P. Mittal S. and H. M. H., *W3C XML Schema for Finite Deterministic (FD) DEVS Models*, Std., Último acceso: Abril de 2014. [Online]. Available: http://www.saurabh-mittal.com/fddevs/

[32] B. P. Zeigler and H. Sarjoughian, "Introduction to devs modeling and simulation with java: A simplified approach to hla-compliant distributed simulations," *Arizona Center for Integrative Modeling and Simulation*, 2000.

[33] *ATL Transformation Language*, Último acceso: Abril de 2014. [Online]. Available: http://www.eclipse.org/atl

[34] *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*, Object Management Group Std., Rev. 2.4.1, 2011. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1

[35] *Medini QVT*, ikv++ Technologies, Último acceso: Abril de 2014. [Online]. Available: http://www.ikv.de/

[36] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[37] *Especificación del metamodelo Ecore*, Object Management Group, Último acceso: Abril de 2014. [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

[38] *MOLA Tool Architecture*, Último acceso: Abril de 2014. [Online]. Available: http://mola.mii.lu.lv

[39] A. Wasowski, "Flattening statecharts without explosions," *SIGPLAN Not.*, vol. 39, no. 7, pp. 257–266, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/998300.997200

[40] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd ed. Orlando, FL, USA: Academic Press, Inc., 2000.

[41] M. von der Beeck, "A structured operational semantics for uml-statecharts," *Software and Systems Modeling*, vol. V1, no. 2, pp. 130–141, December 2002. [Online]. Available: http://dx.doi.org/10.1007/s10270-002-0012-8

[42] H. Cho and Y. Cho, *DEVSC++ Reference Guide*, The University of Arizona, 1997.

[43] T. G. Kim, *DEVSim++ Users Manual. C++ Based Simulation with Hierarchical Modular DEVS Models*, Korea Advance Institute of Science and Technology, 1994.

[44] G. Wainer, "Cd++: A toolkit to develop devs models," *Softw. Pract. Exper.*, vol. 32, no. 13, pp. 1261–1306, 2002. [Online]. Available: http://dx.doi.org/10.1002/spe.482

[45] J.-B. Filippi and P. Bisgambiglia, "Jdevs: an implementation of a devs based formal framework for environmental modelling," in *Environmental Modelling and Software*, 2004, pp. 261–274.

[46] M. Kay, *SAXON. The XSLT and XQuery Processor*, Saxonica, Último acceso: Abril de 2014. [Online]. Available: http://saxon.sourceforge.net