Enriching UML Statecharts through a Metamodel: A Model-Driven Approach for the Graphical Definition of DEVS Atomic Models

Fidel Dalmasso, María J. Blas, and Silvio Gonnet

***Abstract*— The Discrete Event System Specification (DEVS) formalism provides a set of mathematical elements for modeling time-varying systems. However, when DEVS models are implemented in an executable representation (i.e., using a general-purpose programming language), some deviation from the formalism is unavoidable. One way to bridge the gap between modeling and simulation theory and practice is to define new artifacts that support both views during the specification. When the specification is supported with a graphical representation, the formalization task is less complex and can be performed by non-expert modelers. For DEVS atomic models, most common graphical representation is through UML statecharts. In this paper, we present a theoretical and practical metamodel for the definition of atomic models structured following the Classic DEVS with Ports formalization. Such a metamodel is the core of a model-driven approach used to develop a modeling software tool that employs enriched UML statecharts for the graphical representation of the DEVS behavior. In here, the traditional UML statechart representation is enriched with a set of new components with the aim to provide a broad definition of DEVS atomic models. The final software tool is deployed as a plugin for Eclipse Platform.**

***Index Terms*—Discrete Event System Specification, Modeling and Simulation, Theory and Practice, State Diagram.**

I. Introducción

De acuerdo con la teoría de sistemas, un sistema de interés puede verse como la fuente de datos de comportamiento que posibilita su estudio dentro de un marco experimental. Cuando estos datos son usados para crear una representación abstracta del sistema, se obtiene un modelo. Por medio de un conjunto de instrucciones, reglas o ecuaciones matemáticas, dicho modelo busca replicar el comportamiento del sistema de interés bajo condiciones experimentales [1]. En este contexto, el formalismo Discrete Event System Specification (DEVS) [2] es un formalismo de modelado y simulación (M&S) basado en la teoría de sistemas que provee una metodología general para la construcción jerárquica de modelos reusables siguiendo un enfoque modular. DEVS facilita el M&S de sistemas dinámicos complejos en base a una abstracción de eventos discretos. A este nivel de abstracción, una secuencia de entrada (dada por eventos), causa cambios instantáneos en el estado de un sistema. Estos eventos pueden ser generados de forma externa (por otro modelo) o, por el contrario, de forma interna (por el propio comportamiento del sistema bajo estudio). Luego, el próximo estado del sistema queda definido en base al par *(estado previo, evento)*. Entre evento y evento, el estado del sistema se mantiene. Esto implica que los simuladores DEVS solo consideran estados en los que ocurren eventos (omitiendo los instantes de tiempo intermedios) [3].

Entre las principales ventajas de DEVS se destacan su definición formal basada en teoría de conjuntos y la capacidad de dar soporte a esquemas de composición modular [2]. En DEVS clásico, existen dos tipos de modelos: atómico y acoplado. Mientras que los modelos atómicos definen comportamiento, los modelos acoplados definen estructura. De esta forma, es posible anidar modelos DEVS (uno dentro de otro), dando lugar a una jerarquía de modelos mediante la composición de modelos acoplados. Dada su naturaleza, los diagramas de estado de UML (en inglés, *statecharts* [4]) han sido utilizados con éxito como mecanismo de soporte para la representación de modelos atómicos. Sin embargo, con los años, la brecha entre la teoría y la práctica ha dado lugar a discrepancias entre la definición formal, la implementación de modelos, y su representación gráfica basada en *statecharts*.

En este trabajo se presenta un metamodelo basado en los conceptos formales de un modelo DEVS atómico con puertos, que busca enriquecer la definición de diagramas de estado con elementos propios de su implementación práctica. Sobre la base de este metamodelo, se utiliza un enfoque de desarrollo dirigido por modelos (en inglés, Model-Driven Development -MDD), para la implementación de una herramienta basada en Java que funciona como complemento del entorno Eclipse [5]. Esta herramienta facilita la definición gráfica (y posterior validación) de modelos atómicos, con el objetivo de sentar las bases para generar (a futuro) implementaciones en lenguajes de programación de propósito general (como ser, Java y C++). Las tecnologías usadas para su implementación forman parte de Eclipse Modeling Project [6]. Entre las principales contribuciones se destacan el metamodelo como definición conceptual teórico-práctica de modelos atómicos basados en DEVS y la herramienta de software que facilita su instanciación utilizando una representación gráfica basada en diagramas de estado enriquecidos como definición de diseño.

El resto del trabajo se encuentra estructurado de la siguiente manera. La Sección II aborda los trabajos relacionados, presentando la novedad de nuestra propuesta en relación con las existentes. La Sección III describe la forma en la cual es posible

F. Dalmasso, Universidad Tecnológica Nacional –Regional Santa Fe, Lavaisse 610, Santa Fe, 3000, Argentina (e -mail: fdalmasso@frsf.utn.edu.ar).

M. Blas, INGAR (UTN-CONICET), Avellaneda 3657, Santa Fe, 3000, Argentina (e -mail: mariajuliablas@santafe-conicet.gov.ar).

S. Gonnet, INGAR (UTN-CONICET), Avellaneda 3657, Santa Fe, 3000, Argentina (e -mail: sgonnet@santafe-conicet.gov.ar).

aplicar MDD sobre la plataforma Eclipse, presentado la arquitectura de herramientas de software aplicada en nuestra propuesta. La Sección IV presenta el conjunto de modelos generados para dar soporte a la definición de *statecharts* teórico-prácticos para los modelos atómicos. Además, incluye el diseño e implementación de la herramienta de software generada. Sobre dicha herramienta, la Sección V introduce dos casos de estudio como prueba de concepto. Finalmente, la Sección VI está destinada a conclusiones y trabajos futuros.

II. Trabajos Relacionados

Dada la naturaleza matemática de los modelos atómicos DEVS, es posible definir un mismo modelo haciendo uso de múltiples representaciones [7]. Formalmente, un modelo atómico DEVS se define haciendo uso de teoría de conjuntos (es decir, se adopta una representación matemática). Esta representación, luego, es implementada en una representación computacional que facilita su procesamiento (ejecución) en un simulador particular. Este enfoque ha sido tomado en [8] y [2].

Adicionalmente a las representaciones matemáticas y de implementación, se han diseñado representaciones gráficas. El objetivo de estas representaciones es asistir a los modeladores que no necesariamente son especialistas en la implementación computacional de modelos de simulación [9]. En estos casos, se busca modelar el conjunto de estados y transiciones que definen el comportamiento del modelo en una representación gráfica que, luego, puede ser llevada a una representación computacional (modelo ejecutable). Los diagramas *statecharts* de UML [4] han sido utilizados con éxito como soporte de estas representaciones al trabajar con Finite-Deterministic DEVS[1] (FD-DEVS).

Tales diagramas permiten describir el comportamiento de sistemas, exigiendo que el mismo esté compuesto por un número finito de elementos. En principio, es posible pensar en abstraer cada uno de los estados del modelo atómico en un estado del diagrama y, luego, especificar las transiciones entre estados usando los elementos de modelado. Sin embargo, aun cuando la cantidad de estados es finita, su definición formal (representación matemática) es un producto de conjuntos. Tanto la cantidad de conjuntos a considerar como su tamaño, dependen del comportamiento bajo definición. Luego, para una cantidad de conjuntos reducida (en donde cada conjunto posea relativamente pocos elementos no continuos), es posible dar con un conjunto de estados y formular una representación gráfica para el mismo. Sin embargo, ante un aumento en la cantidad de conjuntos y/o gran tamaño y/o continuidad, es difícil determinar una cantidad finita de estados para formular una definición gráfica. Por este motivo, las representaciones gráficas de [10], [11], [12], [13], [14] se basan en FD-DEVS.

Los autores de [10] presentan un enfoque integrado para la transformación de máquinas de estado UML a modelos DEVS. El mecanismo de transformación se basa en XML y, con el objetivo de incorporar las capacidades de los modelos atómicos FD-DEVS, se incorpora nueva información a la definición de la máquina durante dicho proceso. Por su parte, en [11] se presenta un metamodelo (denominado MetaDEVS) que permite la creación de modelos DEVS independientes de la plataforma de ejecución. En este caso, los autores aplican la propuesta para la transformación de modelos de máquinas de estado finitas en modelos MetaDEVS usando un lenguaje híbrido (declarativo e imperativo, más cercano a una representación de implementación que a la representación matemática de DEVS). Los modelos de máquinas de estado también se presentan haciendo uso de un metamodelo. En este contexto, se proponen transformaciones entre modelos para lograr el objetivo. Sin embargo, el grado de expresividad de MetaDEVS es limitado ya que, por ejemplo, no permite la especificación de funciones y condiciones complejas. En la misma línea, en [12] se propone la transformación de máquinas de estado UML a modelos DEVS siguiendo el enfoque MDD según relaciones Query/View/Transformation.

Aun cuando sea posible determinar el conjunto finito de estados (como ocurre en FD-DEVS), la disposición gráfica de las máquinas de estado se complejiza cuando aumenta la cantidad de estados a visualizar (es decir, debe presentarse un gran volumen de información). En este sentido, el uso de una fase (en inglés, *phase*) como identificador global de un subconjunto de estados facilita la representación gráfica de modelos atómicos. En este caso, se utiliza la fase como una variable de estado adicional que engloba un subconjunto de valores asociados a las variables de estado restantes. Luego, las decisiones referidas a la forma en la cual se presentan las transiciones dependen del valor de la fase actual. Este es el enfoque de representación gráfica que adoptan la mayoría de las herramientas de M&S como ser, por ejemplo, MS4Me [13]. En esta herramienta, los *statecharts* son usados para definir FD-DEVS, donde la definición de los modelos atómicos se encuentra restringida a las fases que definen subgrupos de estados. Luego, las transiciones siguen el mismo patrón. Es posible ampliar la definición de estados y de efecto de transiciones, pero el modelador debe escribir código Java para adaptar su FD-DEVS a DEVS. En sí, la herramienta no brinda soporte a la definición a nivel de modelo.

Una alternativa de representación con alto poder de expresividad es la denominada DEVS Graphs, la cual fue originalmente propuesta en [14]. Bajo esta representación, los modelos atómicos se definen como un conjunto de objetos simbólicos, cada uno con múltiples atributos, a fin de definir estados y transiciones entre estados. La base de esta notación son los diagramas *statecharts*, los cuales son enriquecidos con nuevos símbolos para admitir la definición de los DEVS atómicos. Sin embargo, aun ajustándose a la teoría, esta notación no admite dentro de la simbología las convenciones prácticas exhibidas al desarrollar modelos de simulación. Desde esta perspectiva, es importante evidenciar la brecha existente entre teoría y práctica.

En [15], los autores discuten acerca de las desviaciones que tienen lugar en relación con el formalismo DEVS (teoría) y las

[1] FD-DEVS aborda un subconjunto de modelos de simulación DEVS que poseen un conjunto de características que los hacen finitos.

consideraciones prácticas que motivan a los desarrolladores a adoptar convenciones informales. Algunas de estas diferencias son una cuestión de necesidad. Cuando los modelos se definen en representaciones matemáticas, se observa una diferencia entre estructura y comportamiento. Sin embargo, en las representaciones computacionales, los modelos deben ser ejecutables en un simulador. Entonces, por ejemplo, la formalización de un modelo atómico incluye tres conjuntos matemáticos que no pueden representarse en un lenguaje de programación [15]. Así, las representaciones computacionales construidas en herramientas específicas difieren de las representaciones matemáticas del formalismo subyacente. Sumado a esto, existen desviaciones que obedecen a la comodidad, eficiencia, reproducibilidad o algún otro beneficio práctico del modelador. Para DEVS atómicos, se destacan [15]: descripción del modelo (documentación), lista de puertos (entrada/salida), lista de parámetros y variables de estado, estado inicial, función de transición externa que lea/modifique variables de estado según entradas, y función de transición interna que lea/modifique variables de estado, entre otros.

En este contexto, en este trabajo se propone un metamodelo teórico-practico basado en la definición formal de un modelo atómico y, al mismo tiempo, en las convenciones enunciadas en [15]. El metamodelo propuesto constituye un artefacto independiente del simulador DEVS subyacente. En nuestro caso, es tomado como entrada en un proceso de MDD basado en Java (más específicamente, en la plataforma Eclipse [5]) para la construcción de una definición gráfica que utiliza *statecharts* enriquecidos. Al incluir aspectos prácticos sobre los elementos teóricos de DEVS, la representación gráfica diseñada permite generar instancias del metamodelo que corresponden a modelos válidos que contemplan todos los aspectos requeridos en un contexto práctico. El grado de cobertura aumenta en relación con las propuestas basadas en FD-DEVS, ya que los modelos DEVS son más generales que los modelos FD-DEVS. Además, la introducción de elementos prácticos (como ser, por ejemplo, la definición de parámetros), presenta una ventaja significativa sobre otros enfoques teóricos. La novedad de nuestra propuesta radica no solo en la combinación teórico-práctica, sino también en la aplicación de metamodelado como base para enriquecer la representación gráfica clásica de DEVS atómicos.

III. Desarrollo Dirigido por Modelos en Eclipse

Eclipse Modeling Framework (EMF) [16] es un entorno de trabajo que permite definir un modelo a través de código Java, un Schema XML o un diagrama UML. Luego, a partir de este modelo, permite generar cualquiera de los otros equivalentes. En este contexto, se usó EMF para generar código Java partiendo de un metamodelo definido como diagrama UML. Dicho metamodelo tiene como objetivo la definición teórico-práctica de un DEVS atómico. Sobre este metamodelo, se continuó con la obtención de los proyectos Java asociados a su instanciación. Para estos fines, EMF genera automáticamente tres plugins sobre un modelo dado [16]: *i) model*: proyecto que contiene las entidades requeridas para crear instancias del metamodelo, *ii) edit*: proyecto que contiene las entidades requeridas para mostrar el modelo en una interfaz gráfica de usuario, *iii) editor*: proyecto que define un editor de ejemplo para la creación y modificación de instancias del metamodelo.

En el diagrama UML que da soporte al metamodelo, se incorporaron restricciones Object Constraint Language (OCL) [17]. El estándar OCL define un lenguaje formal para escribir expresiones en modelos UML. Estas expresiones típicamente corresponden a dos categorías: *i)* condiciones invariantes que deben cumplirse por el sistema modelado, y *ii)* consultas sobre los objetos descriptos en el modelo. En nuestro caso, se usaron invariantes OCL para garantizar una correcta instanciación. Tales invariantes se especificaron utilizando [18].

Sobre la base del metamodelo EMF complementado con OCL, se usó Sirius [19]. Esta herramienta facilita la creación de entornos gráficos de modelado a través del uso de las tecnologías de modelado de [6]. Esto es, ofrece una solución de software para el desarrollo rápido de herramientas gráficas para lenguajes específicos de dominio sin necesidad de comprender los procesos internos de EMF y Eclipse. En este sentido, Sirius se usa para crear, visualizar y editar modelos usando editores interactivos llamados "modelers". Según el tipo de representación visual, Sirius soporta tres tipos de dialectos: diagrama, tabla o árbol. Cada una de estas representaciones se denomina "viewpoint"; y para un mismo modelo, pueden disponerse de múltiples representaciones en simultáneo [20]. En este trabajo usamos Sirius para la construcción de una herramienta gráfica que brinde soporte a los diagramas de estado definidos a través de la instanciación del metamodelo. Para esto, se utilizó el proyecto *model* como soporte a la definición de elementos en un nuevo proyecto *design* (que corresponde a un Viewpoint Specification Project de Sirius). Tal proyecto contiene la descripción gráfica de los componentes del modelo, junto con la especificación de un entorno gráfico que el usuario usa para especificar sus modelos. Además, con el objetivo de generar un complemento de Eclipse que facilite la instanciación del metamodelo, se modificó el proyecto *editor* generado por EMF.

La Fig. 1 presenta la dependencia de modelos propuesta, indicando el nivel de abstracción abordado en este trabajo. En base a la arquitectura de cuatro capas de UML [4], el meta-metamodelo Ecore (capa M3) define el lenguaje con el que se especifica nuestro metamodelo (capa M2). Sobre este metamodelo, el usuario genera (usando nuestra herramienta) un modelo atómico en sí mismo (capa M1). Finalmente, para obtener una implementación de dicho modelo, se genera una instancia ejecutable en un simulador DEVS específico, como ser, por ejemplo, DEVSJAVA [21] (capa M0).

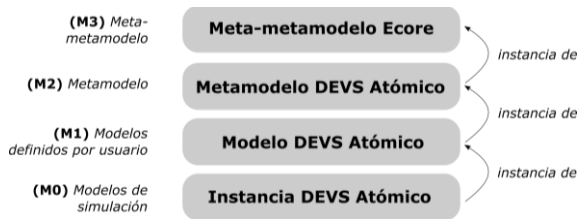

Fig. 1. Dependencia de modelos y enfoque.

IV. Modelos Teórico-Prácticos de los DEVS Atómicos

A. Modelo Formal (Definición Matemática)

La definición matemática de un modelo atómico en el formalismo DEVS clásico con puertos es la siguiente [2]:

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

donde

$X=\{(p,v)\mid p \in InPorts, v \in X_p\}$ es el conjunto de puertos de entrada y sus valores asociados,
$Y=\{(p,v)\mid p \in OutPorts, v \in Y_p\}$ es el conjunto de puertos de salida y sus valores asociados,
S es el conjunto de estados secuenciales,
$\delta_{ext}: Q \times X \rightarrow S$ es la función de transición externa, siendo $Q=\{(s,e)\mid s \in S, 0 \leq e \leq ta(s)\}$ el conjunto de estados totales,
$\delta_{int}: S \rightarrow S$ es la función de transición interna,
$\lambda: S \rightarrow Y$ es la función de salida,
$ta: S \rightarrow R_0^+ \cup \infty$ es la función de avance en el tiempo.

Sobre la base de esta definición formal, la siguiente interpretación es válida. Sea M un modelo atómico que representa un sistema cuyo estado inicial es s. Si no se da ningún evento externo que altere su estado, el modelo permanece en el estado s durante una cantidad de instantes de tiempo definida por $ta(s)$. Cuando el tiempo transcurrido en el estado s (definido por e) es igual a la cantidad de instantes de tiempo que el modelo debe mantenerse en dicho estado (es decir, $e = ta(s)$), el sistema produce un evento de salida con valor $\lambda(s)$. Seguidamente, cambia de estado (de s a s') por medio de la ejecución de la función $\delta_{int}(s)$ (habiéndose efectuado una transición interna). Por el contrario, si antes de que se alcance la condición de transición interna se presenta un evento externo con valor x, el modelo debe atenderlo. Dado que este evento ocurre cuando el sistema aún se encuentra en estado s habiendo transcurrido e instantes de tiempo en dicho estado (es decir, se cumple que $e \leq ta(s)$), el estado total del sistema queda definido como (s,e). En este caso, el sistema no produce ningún evento de salida pero modifica su estado interno de acuerdo a la función $\delta_{ext}(s,e,x)$. Esto se conoce como la ocurrencia de una transición externa.

B. Modelo Conceptual (Definición Abstracta)

1) Metamodelo Ecore

Como se explicó anteriormente, el metamodelo propuesto se definió a través de un diagrama UML (usando EMF) al cual se le impusieron restricciones mediante invariantes OCL. Este metamodelo está formado por seis componentes principales que se agrupan en dos categorías: *componentes estructurales* y *componentes funcionales (o de comportamiento)*.

Los *componentes estructurales* se usan para definir la estructura general del modelo. Estos componentes incluyen:

- La estructura de estado (*StateStructure*): Elemento que contiene la lista de variables que forman parte del estado del modelo (en el modelo formal, *S*). Cada variable de estado tiene un nombre y un tipo de dato asociado. Para todo modelo, es mandatorio que las variables *Phase* y *Sigma* formen parte del *StateStructure*. El usuario tiene la posibilidad de definir nuevas variables (*CustomVariable*).
- El estado inicial (*InitialState*): Elemento que contiene los valores iniciales de las variables de estado definidas. Por cada *StateVariable*, debe existir un *StateValue* asociado que establezca su valor inicial. Las especializaciones de *StateValue* guardan el valor en el atributo *variable*.
- La lista de parámetros -definida como una característica opcional- (*ParameterList*): Elemento que contiene un conjunto de variables estáticas (es decir, constantes) que pueden usarse en distintos tipos de cálculos al momento de definir el comportamiento del modelo. Cada parámetro (*Parameter*) está identificado por un nombre (que debe comenzar con @) y, en forma similar a los *StateValue*, un valor definido por *ParameterValue*. Al igual que en el caso previo, el valor es almacenado en el atributo *parameter* de la especialización correspondiente.
- Los puertos de entrada/salida (*InputPort* y *OutputPort*): Tal como se ha presentado, los puertos son el punto de contacto de un modelo con el mundo exterior. A través de ellos se reciben entradas y se despachan las salidas. Cada puerto es identificado por un nombre. A su vez, tiene asociado tanto un tipo de dato como una variable de control. El nombre de dicha variable de control se forma automáticamente como *x[port name]* si el puerto es de entrada, o bien, *y[port name]* si el puerto es de salida.

Por su parte, los *componentes funcionales* se usan para describir el funcionamiento dinámico del modelo en base a los componentes estructurales. Para esto, el metamodelo incluye:

- Las transiciones (*Transition*): Estos elementos indican un cambio de estado del modelo. Una transición es disparada por la llegada de un evento externo a través de un puerto de entrada (transición externa, *ExternalTransition*), o bien, por el mismo avance del tiempo (transición interna, *InternalTransition*). Una transición vincula un *StatePhase* (*source*) a otro *StatePhase* (*target*), o bien, un *StatePhase* consigo mismo (formando un bucle). Al desencadenarse un cambio de estado, las transiciones indican los nuevos valores que deben tomar las variables de estado. Es decir que, por cada *StateVariable* del modelo, debe existir el correspondiente *ValueData* asociado a la transición. Además, las transiciones tienen información sobre el puerto al que están asociadas. En el caso de transiciones externas, se requiere un puerto de entrada. Por su parte, en el caso de una transición interna, además de un puerto de salida se incluye el valor de salida a ser producido. En ambos casos, la clase *Condition* permite al usuario definir condiciones a satisfacer para ejecutar la transición.
- Las fases (*StatePhase*): Este elemento representa los posibles "nombres" que pueden tomar los estados del modelo. A fin de generar una representación consistente con las propuestas de otros autores, cada *StatePhase* se corresponde con una "clase" en nuestra representación gráfica. Sin embargo, se debe tener en cuenta que las fases no representan el estado del modelo, sino que éste último está dado por el conjunto de valores de todas las variables de estado definidas en el *StateStructure* (siendo una de

ellas, el *StatePhase*).

Además de los *componentes estructurales* y *funcionales*, para dar soporte a múltiples tipos de datos, el metamodelo incluye *tipos de datos* definidos a partir de la clase *Type*. Esta clase provee un conjunto de especializaciones que diferencian datos primitivos (*Integer*, *Double*, *String*, *Boolean*) de datos complejos (definidos por el usuario, *UserDefinedType*).

2) Restricciones Impuestas al Metamodelo

Un subconjunto de restricciones se definió directamente con EMF configurando parámetros de generación de código. Tal configuración impide que se eliminen y/o modifiquen componentes esenciales una vez superada su instanciación. Por ejemplo, en el *InitialState* se configuró el campo *Property Type* en modo "ReadOnly" de forma tal que una vez que se ha indicado su tipo, no pueda modificarse (ya que sienta las bases para la correctitud del resto del modelo).

Las restricciones de mayor complejidad se definieron a través de invariantes OCL [17] implementadas sobre el metamodelo Ecore. Por ejemplo, por medio de la restricción:

```
class InitialState{
   invariant everyVariableMustHaveAValue:
   self.atomicdevs.definition.statevariable->
   forAll(s: StateVariable |
   self.value->one(sv:StateValue|sv.statevariable = s));
}
```

se define una invariante OCL que verifica que, por cada *StateVariable*, exista un único *StateValue* asociado a la instancia de *InitialState*.

C. Diagrama de Estado Enriquecido (Definición Gráfica)

1) Componentes Estructurales: Especificación Guiada

Para definir los elementos básicos del modelo se diseñó un proceso de instanciación implementado un wizard de Eclipse. El usuario es guiado a través de una serie de pasos que le permiten definir la estructura de su modelo (es decir, los componentes estructurales). El aspecto visual del wizard fue desarrollado con Standard Widget Toolkit (SWT).

El wizard contiene siete páginas en las que se le pide al usuario que ingrese información acerca del modelo, a saber: *i)* pág. 1: nombre del modelo y directorio de almacenamiento, *ii)* pág. 2 y 3: puertos de entrada y salida (para cada puerto, nombre y tipo de dato), *iii)* pág. 4 y 5: variables de estado (nombre, tipo de dato y valores iniciales), y *iv)* pág. 6 y 7: parámetros (nombre, tipo de dato y valor). Este proceso guido se dispara cuando, desde la plataforma Eclipse, el usuario elige crear una nueva especificación para un DEVS atómico.

Una vez finalizado el proceso de creación, la herramienta genera automáticamente un nuevo modelo. Dicho modelo contiene instancias de las clases *AtomicDevs*, *StateStructure*, *StatePhase*, *InitialState*, *Type*, *StateVariable*, *StateValue*, *Parameter* y *Port* que modelizan lo definido por el usuario. Luego, es tarea del usuario completar la definición de este modelo agregando los componentes funcionales asociados a la especificación estructural. Para esto, deberá hacer uso de la interfaz gráfica que se describe en la siguiente sección.

2) Componentes Funcionales: Interfaz de Diseño

En el proyecto Design, se implementaron tres módulos: una descripción gráfica basada en diagramas de estados UML enriquecidos, una paleta de herramientas estilo "drag&drop" que incluye los elementos disponibles para especificar el modelo, y una pestaña de configuración de propiedades.

La descripción gráfica propuesta fue diseñada en base a los siguientes lineamientos:

- La estructura de estado (*StateStructure*) no tiene una representación en el diagrama, sino que puede obtenerse a partir de los demás elementos (como ser, por ejemplo, el *InitialState* y los *StatePhase*).
- El estado inicial (*InitialState*) se visualiza, al igual que los diagramas de estados de UML, como un pequeño círculo relleno de color negro (Fig. 2). De dicho elemento se desprende una transición que apunta al primer *StatePhase* definido en la estructura del modelo. Conectado a esta transición, se presenta una flecha (color gris, línea discontinua) que apunta a un cuadro que contiene la lista de variables de estado junto con su tipo de dato y su valor inicial (es decir, la definición formal del *InitialState*).
- La lista de parámetros (*ParameterList*) se representa de forma similar al *InitialState* (Fig. 2). En este caso, para cada parámetro se indica nombre, tipo de dato y valor.
- Los puertos (*InputPort* y *OutputPort*) no disponen de una representación específica, sino que se los exhibe como información asociada a la transición que los involucra.
- Las transiciones (*Transition*) se representan como flechas continuas. El color rojo refiere a transiciones externas, mientras que el color azul a transiciones internas (Fig. 3). Desde la flecha, se extiende una línea discontinua que conecta la transición con un cuadro de información. Para ambas transiciones se presenta: *i)* la lista de cambios de variables de estado (nombre de variable y nuevo valor)[2], *ii)* la condición que debe resultar verdadera antes de ejecutar la transición[2], y *iii)* el puerto asociado (nombre y variable de control)[2]. Para el caso de las transiciones externas, se incluye el campo *ElapsedTimeVariable*. Por su parte, las transiciones internas incorporan el valor de salida (*OutputValue*).
- Las fases (*StatePhase*) son equivalentes a los *Simple State* de los diagramas de estado UML. Se representan con un rectángulo gris (Fig. 3) en donde se define el "nombre" de la fase asociada.

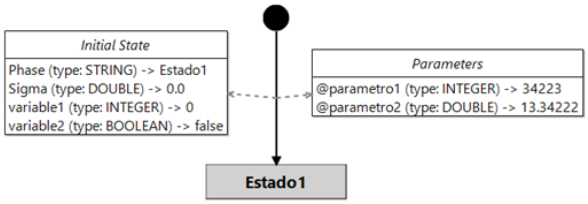

Fig. 2. Representación gráfica del estado inicial y los parámetros.

[2] En Fig. 3: *i)* la *ExternalTransition* indica que *Sigma* tomará el valor *3.0*, *variable1* tomará el valor *42*, etc.; *ii)* la condición de ejecución de la *InternalTransition* es que *Sigma* sea menor a *1.0*; y *iii)* la *ExternalTransition* recibe su entrada por el puerto definido como *(puerto1, $x_{puerto1}$)*.

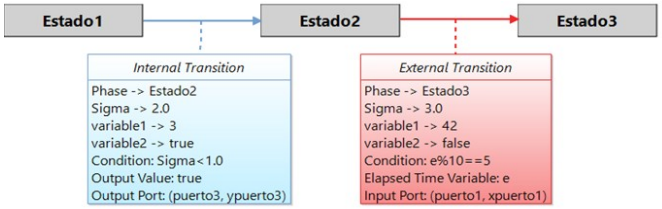

Fig. 3. Representación gráfica de las transiciones y las fases.

Para cada una de estas representaciones, el segundo módulo incluye una entrada en una paleta de herramientas. Esta paleta permite agregar nuevas fases y crear transiciones (tanto internas como externas) entre ellas. A través de un mecanismo drag&drop (es decir, arrastrando y soltando los distintos componentes sobre el diagrama), el usuario define los componentes funcionales y construye la dinámica de su modelo de simulación en forma gráfica. Internamente, la herramienta se encarga de instanciar y configurar todas clases del metamodelo que sean necesarias para dar soporte a las nuevas definiciones, garantizando la consistencia del modelo gráfico con el formalismo DEVS.

Finalmente, la pestaña de configuración de propiedades se usa para configurar los atributos de los diferentes elementos. El contenido de esta pestaña varía de acuerdo con el elemento gráfico seleccionado en el diagrama. Si no existen elementos seleccionados, se muestran propiedades generales del modelo. Si se selecciona un *StatePhase*, la pestaña permite definir el nombre de la fase en un campo específico. Si se selecciona una *Transition*, la pestaña muestra un conjunto de campos y listas desplegables que facilitan la configuración de los valores de las variables de estado, el puerto asociado y la condición de transición. La Fig. 4 presenta el contenido de la pestaña cuando se selecciona una *InternalTransition*.

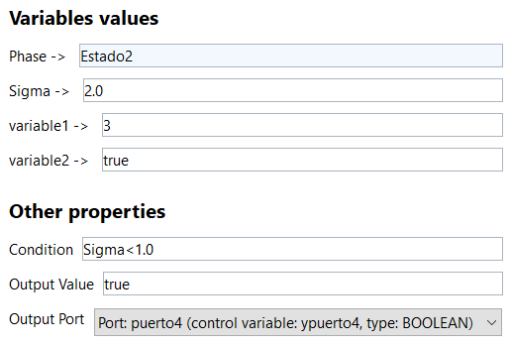

Fig. 4. Pestaña de configuración de propiedades de una *InternalTransition*.

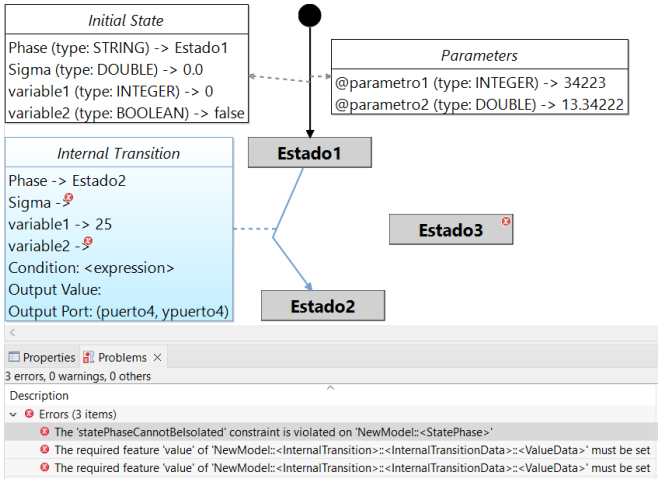

Fig. 5. Validación de instancia. La fase *Estado3* viola la restricción de no existencia de fases aisladas; mientras que la transición interna entre *Estado1* y *Estado2* tiene dos campos obligatorios sin configurar.

D. *Instancia de Modelo (Validación de Definición)*

Como se mencionó anteriormente, se usaron invariantes OCL para restringir el metamodelo. Tales invariantes pueden validarse desde el entorno gráfico seleccionado la opción "Validate Diagram". Al hacer esto, la herramienta marca las partes del diagrama que presentan fallas y, en la solapa "Problems", presenta información detallada respecto a las restricciones con problemas. De esta manera, el modelador puede volver a modificar su diagrama para que cumpla con la representación propuesta. La Fig. 5 muestra una instancia sobre la cual se ejecutó el proceso de validación. Para que un modelo DEVS atómico esté bien definido, debe validarse con éxito.

V. Prueba de Concepto

A modo de prueba de concepto, en este apartado se presenta el modelado gráfico de dos DEVS atómicos clásicos. Ambos ejemplos corresponden a modelos DEVS propuestos tanto en la literatura clásica de DEVS [2] como en tutoriales básicos dictados por reconocidos académicos [3].

En [2], como parte de la introducción a DEVS con puertos, se formaliza un modelo Switch. Este modelo consta de 2 puertos de entrada (*in* e *in1*) y 2 puertos de salida (*out* y *out1*) entre los cuales alterna el envío de eventos con contenido *store*. La alternancia se da según el puerto por el que ha entrado el evento (almacenado en *inport*) y valor de la variable de estado *Sw*. Por ejemplo, si *store* almacena un valor que ha entrado por el puerto *in* y *Sw* es *true*, se envía *store* por *out*. En el mismo escenario, si *Sw* es *false* se envía por *out1*. La especificación formal de este modelo es la siguiente:

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

donde

$X=\{(p,v)|\ p \in InPorts,\ v \in Xp\}$ con $InPorts = \{"in","in1"\}$ y $X_{in}=X_{in1}=V$ (un set arbitrario),

$Y=\{(p,v)|\ p \in OutPorts,\ v \in Y_p\}$ con $OutPorts=\{"out","out1"\}$ e $Y_{out}=Y_{out1}=V$ (un set arbitrario),

$S=\{"passive","active"\}\times R_0^+\times\{"in","in1"\}\times V\times\{true,false\}$,

$\delta_{ext}(phase,\sigma,inport,store,Sw,e,(p,v)) =$

("active",processing_time,p,v,!Sw) si *phase="passive"* y *p∈ in,in1*
(phase, σ-e, inport, store, Sw) en otro caso

δ_{int}*(phase,σ,inport,store,Sw,e,(p,v)) =*
("passive", ∞, inport, store, Sw)

λ(phase,σ,inport,store,Sw) =
(out, store) si *phase="active"*, *Sw=true* y *inport=in*,
(out1, store) si *phase="active"*, *Sw=true* y *inport=in1*,
(out1, store) si *phase="active"*, *Sw=false* y *inport=in*,
(out, store) si *phase="active"*, *Sw=false* y *inport=in1*,

ta(phase,σ,inport,store,Sw) = σ

Sobre la base de este modelo formal, la construcción paso a paso de su representación gráfica con la herramienta propuesta puede visualizarse aquí.

Un ejemplo práctico diferente se presenta en [3], donde los autores proponen la definición de un modelo DEVS atómico para un semáforo cuyo accionar puede verse interrumpido, por ejemplo, por tareas de mantenimiento. Este caso práctico se denomina "interruptible traffic light", y ha sido modelado con la herramienta propuesta en el siguiente video. Por cuestiones de espacio, la definición formal no ha sido incluida.

VI. Conclusiones y Trabajos Futuros

En este trabajo se ha presentado una herramienta basada en un metamodelo que busca enriquecer la representación de DEVS atómicos con puertos usando diagramas *statechart*. El metamodelo final se compone de 42 clases, 28 asociaciones (simples y de composición), 22 especializaciones, 32 atributos y 19 restricciones OCL (formuladas como invariantes) que cubren todos los aspectos requeridos para la definición de modelos DEVS atómicos. La principal contribución es la inclusión de conceptos prácticos vinculados a los DEVS teóricos como parte de la representación elegida, lo que ayuda al modelador a generar modelos teóricos válidos con utilidad práctica. La herramienta propuesta ha sido diseñada siguiendo los principios de MDD usando tecnología de modelado de Eclipse. El uso de un metamodelo como núcleo del proceso de desarrollo posibilita, a futuro, vincular los conceptos de este nivel de diseño con otros niveles de M&S como los propuestos en [7] (como ser, por ejemplo, los modelos independientes de la plataforma). Esto forma parte del trabajo futuro asociado a la línea de investigación abordada.

Se han presentado dos ejemplos prácticos acompañados de videos que muestran la forma en la cual la herramienta da soporte a la tarea de modelado con el objetivo de crear una instancia del metamodelo propuesto. De esta manera, se observa la principal ventaja de nuestro trabajo: *la definición de instancias del metamodelo partiendo de su definición formal (matemática) y haciendo uso de la representación gráfica implementada*. A futuro, se abordará el proceso de transformación desde el metamodelo hacia simuladores DEVS basados en Java a fin de obtener el conjunto de clases que represente el modelo computacional ejecutable asociado al modelo formal descripto. Aún más, aunque se ha utilizado la tecnología de modelado de Eclipse como soporte para la herramienta, el metamodelo propuesto puede ser utilizado como base en nuevas plataformas a fin de generar mapeos a otros lenguajes de programación. Tales mapeos requerirán de la implementación de las transformaciones requeridas para obtener código ejecutable en distintos tipos de simuladores DEVS. En este sentido, el metamodelo propuesto en este trabajo es independiente de la herramienta de M&S elegida.

Referencias

[1] G. A. Wainer, *Discrete-event modeling and simulation: A practitioner's approach*. CRC Press, 2009.

[2] B. Zeigler, A. Muzy, and E. Kofman, *Theory of modeling and simulation: discrete event & iterative system computational foundations*, 3rd ed. Academic Press, 2018.

[3] Y. Van Tendeloo, and H. Vangheluwe, "Classic DEVS modelling and simulation", in WSC, Las Vegas, NV, USA, 2017, pp. 644-658). DOI: 10.1109/WSC.2017.8247822.

[4] OMG unified modeling language version 2.5.1, OMG, 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[5] Eclipse, Eclipse foundation. [Online]. Available: https://www.eclipse.org/, Accessed on: Feb. 6, 2021

[6] Eclipse modeling project, Eclipse foundation. [Online]. Available: https://www.eclipse.org/modeling/, Accessed on: Feb. 6, 2021

[7] M. J. Blas, S. Gonnet, and B. Zeigler, "Towards a universal representation of DEVS: a metamodel-based definition of DEVS formal specification", in ANNSIM, Fairfax, VA, USA, 2021. DOI: 10.23919/ANNSIM52504.2021.9552162.

[8] M. J. Blas, S. Gonnet, and H. Leone, "Modeling User Temporal Behaviors Using Hybrid Simulation Models", *IEEE Latin America Transactions*, vol. 15, no. 1, pp. 341-348, 2017, DOI: 10.1109/TLA.2017.7854631.

[9] G. Wainer, and Q. Liu, "Tools for graphical specification and visualization of DEVS models", *Simulation*, vol. 85, no. 3, pp. 131-158, 2009. DOI: 10.1177/0037549708101182.

[10] J. L. Risco-Martın, S. Mittal, B. P. Zeigler, and J. de la Cruz, "From UML state charts to DEVS state machines using XML", in MODELS, Nashville, TN, USA, 2007, pp. 35-48.

[11] S. Garredu, E. Vittori, J. F. Santucci, and P. A. Bisgambiglia, "From state-transition models to DEVS models. Improving DEVS external interoperability using MetaDEVS: a MDE approach", In Simultech, Reykjavík, Iceland, 2013, pp. 186-196. DOI: 10.5220/0004494401860196.

[12] A. Gonzalez, C. Luna, R. Cuello, M. Perez, and M. Daniele, "Towards an automatic model transformation mechanism from UML state machines to DEVS models", *CLEI electronic journal*, vol. 18, no. 2, paper 3, 2015. DOI: 10.19153/cleiej.18.2.3.

[13] MS4 Me. Discrete event systems specification (DEVS) modeling environment. [Online]. Available: http://www.rtsync.com/pages/products/ms4me.html, Accessed on: Aug. 10, 2021

[14] H. Praehofer, and D. Pree, "Visual modeling of DEVS-based multiformalism systems based on higraphs", In WSC, Los Angeles, CA, USA, 1993, pp. 595-603. DOI: 10.1145/256563.256737.

[15] R. Goldstein, S. Breslav, and A. Khan, "Informal DEVS conventions motivated by practical considerations", in SpringSIM, San Diego, CA, USA, 2013, article 10.

[16] Eclipse modeling framework, Eclipse foundation. [Online]. Available: https://www.eclipse.org/modeling/emf/, Accessed on: Jun. 5, 2021

[17] Object constraint language specification version 2.4, OMG, 2014. [Online]. Available: https://www.omg.org/spec/OCL/2.4

[18] Eclipse OCL, Eclipse foundation. [Online]. Available: https://www.eclipse.org/ocl, Accessed on: Jul. 21, 2021

[19] Sirius, Eclipse foundation. [Online]. Available: https://www.eclipse.org/sirius, Accessed on: Jul. 21, 2021

[20] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: a rapid development of DSM graphical editor", in INES, Tihany, Hungary, 2014. DOI: 10.1109/INES.2014.6909375.

[21] DEVSJAVA, Arizona center for integrative modeling and simulation. [Online]. Available: https://acims.asu.edu/software/devsjava/, Accessed on: Jul. 21, 2021

Fidel Dalmasso is a fifth-year Information Systems Engineering student at Universidad Tecnológica Nacional. In 2020, he received an undergraduate scholarship to work in the research project "Evaluación de Arquitecturas de Software de Sistemas Auto-Adaptativos mediante Simulación".

Maria J. Blas is a Postdoctoral Fellow at INGAR and an Assistant Professor in the Information Systems Department at Universidad Tecnológica Nacional. She received her PhD degree in Engineering from Universidad Tecnológica Nacional in 2019. Her research interests include discrete-event M&S.

Silvio Gonnet received his PhD degree in Engineering from Universidad Nacional del Litoral in 2003. He currently holds a Researcher position at CONICET. His research interests are models to support design processes, software engineering and conceptual modeling.