Design of Discrete Event Simulations

María Julia Blas[1,2], Victoria Elizabeth Meichtry Regner[2], Silvio Gonnet[1,2]

[1] INGAR, CONICET-UTN, Avellaneda 3657, Santa Fe, Argentina
[2] Facultad Regional Santa Fe, UTN, Lavaisse 610, Santa Fe, Argentina
{mariajuliablas@santafe-conicet.gov.ar, vmeichtry-regner@frsf.utn.edu.ar, sgonnet@santafe-conicet.gov.ar}

Abstract. This paper presents a modeling proposal for the definition of Discrete Event Simulation (DES) models based on the set of artifacts proposed by (Robinson, 2014). It focuses on the realization of the *Model Design* artifact through the definition of a textual modeling language. This language is intended to support the constructors and logic related to the DES model prior to the implementation of the computational model in specific simulation software. As a result, a software tool is presented as an Editor of the proposed language. Such an editor has been developed as a Java-based textual modeling tool that enables the creation of DES models based on the proposed language. It also allows validating the models edited following the DES paradigm through the language's constraints.

Keywords: Modeling and Simulation, Design Model, Textual representation.

Diseño de Simulaciones basadas en Eventos Discretos

Resumen. En este trabajo se presenta una propuesta de modelado para la definición de modelos de simulación DES considerando el conjunto de artefactos propuestos por (Robinson, 2014). Se aborda la materialización del artefacto Model Design por medio de la definición de un lenguaje de modelado textual. Dicho lenguaje tiene como finalidad dar soporte a los constructores y la lógica asociada al modelo DES bajo desarrollo, en una etapa previa a la implementación del modelo computacional asociado en un software específico. Como resultado, se presenta un editor del lenguaje. Dicho editor ha sido desarrollado como una herramienta de modelado textual basada en Java que debe ser ejecutada sobre la plataforma Eclipse. Esta herramienta permite la creación de modelos DES basados en el lenguaje diseñado, validando además la integridad de dichos modelos respecto al paradigma DES por medio de las restricciones del lenguaje.

Palabras clave: Modelado y Simulación, Modelo de Diseño, Representación Textual.

1 Introducción

Con los años, el Modelado y la Simulación han definido una disciplina en sí misma. Sin embargo, el *modelado* y la *simulación* son actividades diferentes (Oren et al., 2018), las cuales apuntan a la resolución de problemas vinculados. *Modelar* es el proceso de desarrollar y utilizar abstracciones para simplificar el mundo real con vistas a un análisis específico. Por otro lado, *simular* es el proceso de desarrollar, ejecutar y analizar resultados obtenidos a partir de modelos de simulación previamente diseñados. La *simulación*, como actividad, parte de las actividades de *modelado*, poniendo el foco en la implementación y ejecución de un tipo de modelo específico (modelo de simulación) en un simulador en particular.

En la actualidad, existe una ausencia de herramientas de software que den soporte al vínculo entre las actividades de *modelado* y la posterior *construcción de modelos de Simulación basados en Eventos Discretos* (DES, por sus siglas en inglés). Esto se debe a que, en general, el desarrollo de modelos de simulación se considera una tarea práctica, donde los involucrados centran su atención en el uso de herramientas de software para la implementación (no así para el diseño).

En este trabajo se presenta una propuesta de modelado para la definición de modelos de simulación DES considerando el conjunto de artefactos propuestos por (Robinson, 2014). Puntualmente se aborda la materialización del artefacto *Model Design* por medio de la definición de un lenguaje de modelado textual. Dicho lenguaje tiene como finalidad dar soporte a los constructores y la lógica asociada al modelo DES bajo desarrollo, en una etapa previa a la implementación del modelo computacional asociado en un software específico. El lenguaje de modelado propuesto utiliza los conceptos tradicionales de DES (Banks et al., 2010) como los elementos centrales requeridos en la definición de modelos. Luego, facilita al modelador la creación de entidades, atributos, eventos, actividades, retrasos y funciones (como parte de la definición estática) junto con rutinas de eventos primarios (como parte de la definición dinámica). Estas últimas quedan definidas haciendo uso de un diagrama de actividad basado en PlantUML[1]. De esta manera, el modelador puede construir tanto la definición estática de las entidades, como así también su dinámica, dentro de un único modelo. La validación de la definición dinámica en relación con la definición estática forma parte de lenguaje. Como resultado, se presenta un editor del lenguaje. Dicho editor ha sido desarrollado como una herramienta de modelado textual basada en Java que debe ser ejecutada sobre la plataforma Eclipse. Esta herramienta permite la creación de modelos DES basados en el lenguaje diseñado, validando además la integridad de dichos modelos respecto al paradigma DES por medio de las restricciones del lenguaje.

El resto del trabajo se encuentra estructurado de la siguiente manera. La Sección 2 presenta los fundamentos teóricos referidos al modelado conceptual como parte de un proyecto de simulación y a la simulación basada en eventos discretos. La Sección 3 introduce el lenguaje de modelado propuesto como parte de un framework de modelado DES. La Sección 4 describe una prueba de concepto. Finalmente, la Sección 5 se encuentra dedicada a las conclusiones y trabajos futuros.

[1] Disponible en https://plantuml.com/es/.

2 Modelado en DES

2.1 Modelado Conceptual según (Robinson, 2014)

De acuerdo con (Robinson, 2008), un *Modelo Conceptual* es una especificación no basada en software de un modelo de simulación computacional que será, es, o ya ha sido desarrollado; la cual describe los objetivos, entradas, salidas, contenido, suposiciones y simplificaciones realizadas en torno a dicho modelo. El *Modelo Conceptual* (*Conceptual Model*) surge como consecuencia de la actividad de *Modelado Conceptual* (*Conceptual Modeling*), la cual se encarga de generar una abstracción de una parte del mundo real a ser representado (lo que normalmente se conoce como "el sistema real") en un modelo de simulación.

Para entender el *Modelado Conceptual* como una parte de los proyectos de Modelado y Simulación (M&S), Robinson (2014) propone un conjunto de artefactos vinculados (Fig. 1), a saber:

1. *Descripción del Sistema*: Enunciado definiendo la situación/problema a resolver, que incluye todos los elementos del mundo real relacionados con dicha situación/problema.
2. *Modelo Conceptual*: Especificación no basada en software de un modelo de simulación que contiene sus objetivos, entradas, salidas, contenido, suposiciones y simplificaciones.
3. *Modelo de Diseño*: Diseño de los constructores y la lógica del modelo computacional en términos del software que será utilizado para su desarrollo (Fishwick, 1995).
4. *Modelo Computacional*: Representación basada en software del modelo conceptual.

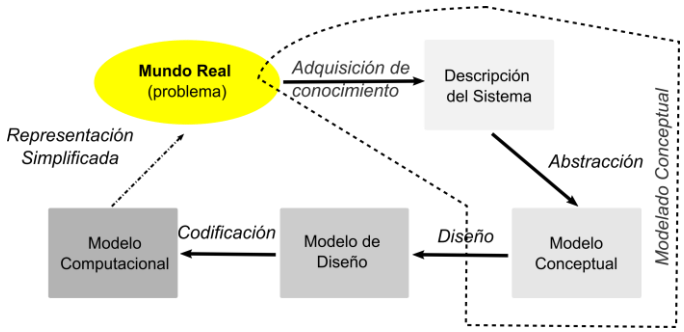


Fig. 1. Conjunto de actividades y artefactos vinculados a la etapa de modelado conceptual (adaptado de (Robinson et al., 2011)). La figura amarilla representa el mundo real en el cual reside el problema a ser resuelto. Los rectángulos representan los artefactos resultantes de la tarea de modelado conceptual, mientras que las flechas simbolizan las actividades que transforman un artefacto en el siguiente.

Cada uno de estos artefactos tiene su propio objetivo, surgiendo cada uno de ellos como una evolución del artefacto previo. Dicha evolución queda expresada en la Fig.

1 por medio de las flechas que unen los diferentes artefactos. Aun cuando el *Modelo de Diseño* y el *Modelo Computacional* no forman parte del *Modelado Conceptual* (limitado por la línea de puntos en la Fig. 1), ambos artefactos permiten la materialización de la conceptualización en un diseño propio del tipo de simulación elegida, junto con su posterior implementación.

Usualmente, el *Modelo Computacional* es el único artefacto explícitamente definido (quedando los artefactos restantes en la mente del modelador a cargo del proyecto y/o de los stakeholders). Sin embargo, documentar todos estos artefactos es una buena práctica de M&S y, como ocurre con cualquier artefacto que se utiliza en un proceso de desarrollo, puede facilitar la comunicación con todos los involucrados en el proyecto. Aún así, no existe un consenso respecto a la forma en la cual estos artefactos deben ser documentados.

En un trabajo previo, (Alvarez et al., 2023) han propuesto una estrategia para la documentación de los artefactos *Descripción del Sistema* y *Modelo Conceptual*. Dicha estrategia se basa en el uso de una planilla diseñada para capturar el conocimiento obtenido durante el proceso de M&S en base a los siguientes pasos: *a)* entender el problema, *b)* determinar los objetivos de M&S, *c)* identificar salidas, *d)* identificar entradas, *e)* determinar el contenido del modelo (tanto en términos de sus componentes como del nivel de abstracción asociado a cada uno de ellos), y *f)* definir suposiciones y simplificaciones. Luego, este trabajo continúa esta línea de investigación abordando la definición del *Modelo de Diseño*, en primera instancia, en base a un lenguaje de modelado textual.

2.2 Simulación basada en Eventos Discretos: Conceptos Principales

El concepto de modelado basado en DES nace de la idea de representar sistemas dinámicos estocásticos (es decir, sistemas que evolucionan en el tiempo y contienen componentes aleatorias) en los cuales los cambios se producen de forma discreta. La Tabla 1 resume los principales conceptos utilizados en modelado DES según (Banks et al., 2010) indicando, para cada uno de ellos, la relación que existe con el artefacto previo (*Conceptual Model*). De esta manera, se muestra la trazabilidad existente entre un artefacto de modelado y el siguiente.

Los componentes listados en la Tabla 1 corresponden a la definición estática del modelo. Para la definición dinámica en un modelo DES, es necesario definir un mecanismo de avance de tiempo que indique la forma en la cual los eventos futuros son planificados. Uno de estos mecanismos es el denominado *Lista de Eventos Futuros* (*Future Event List*) el cual, según (Banks et al., 2010), requiere para su implementación dos nuevos elementos:

1. *Aviso de Evento* (*Event Notice*): Entrada asociada a un *Evento* que ocurrirá en el instante actual o en un instante futuro, junto con toda la información requerida para su ejecución[2].
2. *Lista de Eventos* (*Event List*): Lista que contiene todos los *Avisos de Eventos* ordenados por tiempo de ocurrencia[3].

[2] Como mínimo, la entrada debe almacenar el tipo de evento y el tiempo de ocurrencia.
[3] Esta lista también es conocida como la *Lista de Eventos Futuros* (*Future Event List*).

Tabla 1. Elementos asociados a modelos DES (parte estática).

Elemento	Definición	Vinculado a
Modelo (*Model*)	Representación abstracta de un *Sistema* que usualmente contiene relaciones estructurales, lógicas y/o matemáticas que describen dicho sistema en términos de su *Estado* (*State*) y sus *Entidades* (*Entities*).	*Modelo Conceptual* (*Conceptual Model*) en sí mismo.
Estado del Sistema (*System State*)	Colección de *Variables de Estado* (*State Variables*) que contiene toda la información necesaria para describir el *Sistema* en cualquier punto en el tiempo.	N/A[4]
Entidad (*Entity*)	Componente del Sistema que requiere una representación explícita en el *Modelo* (*Model)*, la cual queda expresada en términos de sus *Atributos* (*Attributes*), *Actividades* (*Activities*), *Retrasos* (*Delays*), y *Eventos* (*Events*).	*Componente* (*Components*) incluido en el *Modelo Conceptual* (*Conceptual Model*).
Atributo (*Attribute*)	Propiedad de una *Entidad* (*Entity*).	*Nivel de Detalle* (*Level of Detail*) definido para un *Componente* (*Component*) en el *Modelo Conceptual* (*Conceptual Model*).
Actividad (*Activity*)	Período de tiempo de duración predefinida (conocida o determinada en el momento en que inicia), que empieza y termina por un Evento (*Event*).	N/A[4]
Retraso (*Delay*)	Período de tiempo de duración desconocida, que empieza y termina por un *Evento* (*Event*).	N/A[4]
Evento (*Event*)	Ocurrencia instantánea que altera el *Estado del Sistema* (*System State*).	N/A[4]

La aplicación del mecanismo *Lista de Eventos Futuros* involucra la definición del *Efecto* (*Effect*) de los eventos a ser ejecutados (es decir, aquellos incluidos en la lista). Como se muestra en la Tabla 1, toda *Actividad* debe finalizar por un *Evento*. Este evento se denomina *Evento Primario* (*Primary Event*) y, según el mecanismo de avance de tiempo elegido, debe manejarse colocando un *Aviso de Evento* en la *Lista de Eventos* cada vez que el evento en cuestión debe tener lugar. Por otro lado, si bien los *Retrasos* también finalizan por medio de un *Evento*, estos eventos se denominan

[4] No Aplica (N/A): Refiere a un nuevo elemento que se da a partir de una mejora de la definición del artefacto previo como consecuencia de su evolución en un nuevo artefacto.

Eventos Secundarios (*Secondary Event*) y no deben ser incluidos en la *Lista de Eventos* debido a que no poseen un *Efecto* asociado. Comúnmente, los *Retrasos* se denominan "esperas condicionales". Esto implica que dicha espera finalizará según el cumplimiento de alguna condición asociada (esto es, un evento condicional o secundario) que deberá cumplir el sistema. Por este motivo, el manejo de los *Retrasos* se asocia con la colocación de la *Entidad* asociada en una lista de espera hasta el momento en que se sabe que la condición de salida de dicha espera es verdadera. Luego, la definición de *Efectos* sólo es requerida para *Eventos Primarios*.

2.3 PlantUML

PlantUML es una herramienta de código abierto que permite generar diagramas basados en el lenguaje UML (Unified Modeling Language) a partir de un modelo textual especificado en un lenguaje específico. Incluye formatos de especificación para diagrama de clases, diagrama de secuencia, diagrama de casos de uso, diagrama de actividades, y diagrama de componentes. Debido a su formato basado en texto, es muy útil para los diseñadores que buscan especificar diagramas UML de forma rápida y sencilla (y, sobre todo, sin necesidad de usar herramientas gráficas).

Tomando en consideración que *(a)* la propuesta previa, basada en el trabajo de (Alvarez et al., 2023), hacía uso de diagramas UML para la definición de la estructura y la dinámica requerida en el artefacto *Descripción del Sistema* (diagrama de clase y diagrama de estados, respectivamente), y *(b)* el lenguaje de modelado DES propuesto se basa en una sintaxis concreta textual; se decidió representar los flujos de los *Efectos* asociados a *Eventos Primarios* haciendo uso de diagramas de actividad especificados en PlantUML. De esta manera, el lenguaje permite la definición completa de modelos DES, facilitando la especificación tanto de la parte estática como dinámica de las entidades de manera textual.

3 Lenguaje de Modelado

3.1 Sintaxis Abstracta del Lenguaje

La Fig. 2 presenta el metamodelo que da soporte al lenguaje diseñado. Como puede observarse, cada uno de los conceptos descriptos en la Tabla 1 ha sido materializado como un constructor del lenguaje.

Se tiene entonces que un *Model* queda especificado por la composición de dos elementos: *SystemState* y *Entity*. Para *SystemState* se indica una composición basada en una asociación de carácter mandatorio, mientras que para *Entity* se hace uso del mismo tipo de vinculo ampliándolo a un posible conjunto de elementos (es decir, un modelo debe tener al menos una entidad). Luego, por ejemplo, para el constructor *Attribute* presentado en la Tabla 1, se define que una *Entity* queda compuesta de un conjunto denominado *AttributeDefinitions*. En este conjunto, cada uno de sus elementos corresponde a un *Attribute* caracterizado por su nombre (*name*) y tipo (*type*). Esta estrategia de modelado ha sido aplicada a cada uno de los constructores requeridos para el modelado DES.

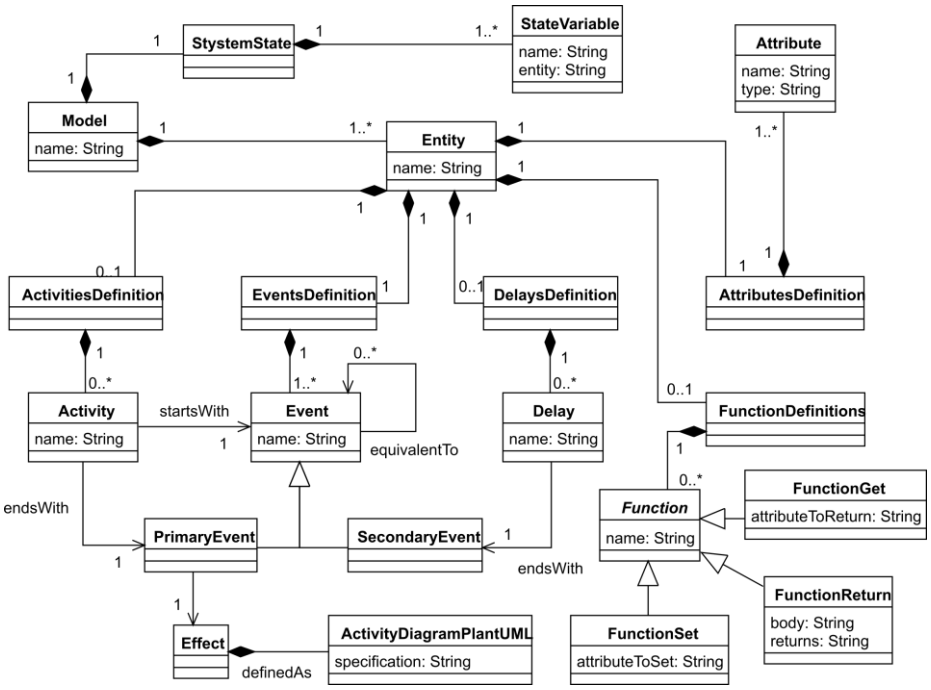


Fig. 2. Metamodelo utilizado para dar soporte a la sintaxis abstracta del lenguaje a partir de los elementos definidos en (Banks et al., 2010). Este metamodelo se complementa con un conjunto de restricciones (ver Tabla 2) que permiten validar la conformación de un modelo creado a partir del mismo.

Adicionalmente, se observa en la Fig. 2 la inclusión del concepto *Función* como constructor adicional del lenguaje (por medio de los constructores *FunctionsDefinition* y *Function*). Aunque el término "función" no se define explícitamente en (Banks et al. 2010), se hace lugar a su inclusión dentro del lenguaje por los siguientes motivos:

1. De (Banks et al. 2010), se puede inferir que la noción de "función" refiere al resultado de operaciones lógicas o matemáticas que determinan el valor de nuevos atributos a partir de los atributos existentes[5].
2. En el contexto de modelado, las funciones pueden ser utilizadas como métodos de acceso a atributos (setters y getters en el paradigma orientado a objetos), a fin de asegurar que una entidad no tenga permiso de modificar/consultar atributos de otras entidades de forma directa. De esta manera, el lenguaje permite controlar el acceso a atributos desde el modelado, a fin de garantizar el encapsulamiento deseado por el modelador.

[5] En modelado conceptual, este tipo de atributos corresponde a *atributos derivados*. Un *atributo derivado* es un atributo cuyo valor no se almacena en la clase, sino que se calcula a partir de otros atributos o relaciones.

Validaciones del Lenguaje. La Tabla 2 presenta un listado coloquial del conjunto de restricciones incluidas en el lenguaje a fin de determinar si un modelo es válido o no. Este conjunto de restricciones se implementó haciendo uso de las multiplicidades propias en el metamodelo, como así también haciendo uso de restricciones OCL (Object Constraint Language). De esta manera, se garantiza que una especificación textual basada en el lenguaje define correctamente un modelo DES.

Tabla 2. Conjunto de validaciones realizadas en el lenguaje a fin de garantizar la validez de un modelo DES. Los conceptos en cursiva refieren a elementos definidos en la sintaxis abstracta (Fig. 2). La columna "Tipo" refiere a la forma en la cual se ha implementado cada validación dentro del lenguaje. La columna "Nivel" refiere al elemento de contexto sobre el que se está realizando la validación.

N°	Definición	Tipo	Nivel
1	Un Modelo no puede estar vacío.	Metamodelo	Archivo
2	Un Modelo debe tener solo una definición *Model*.	Metamodelo	Archivo
3	Un *Model* debe tener al menos un *SystemState*.	Metamodelo	Modelo
4	Un *Model* debe tener al menos un *Entity*.	Metamodelo	Modelo
5	Un *Entity* debe tener al menos un *Attribute* en su *AttributesDefinition*.	Metamodelo	Entidad
6	Un *Entity* debe tener al menos un *Event* en su *EventDefinition.*	Metamodelo	Entidad
7	Un *Entity* puede tener solo *DelaysDefinition* o solo *ActivitiesDefinition*, pero no pueden faltar ambas. Si existen, cada uno de estos apartados (*DelaysDefinition* y/o *ActivitiesDefinition*) debe tener al menos un componente definido (*Delay* o *Activity*, respectivamente).	OCL	Entidad
8	Un *Entity* puede no tener *FunctionsDefinition* (pero no puede tenerlo vacío).	OCL	Entidad
9	El nombre de un *Event* es único dentro de un *Entity*.	OCL	Definición de Eventos
10	Un *Event* solo puede ser equivalente a otro *Event*, si este último pertenece a otra *Entity* que, a su vez, pertenece al mismo *Model*.	OCL	Definición de Eventos
11	Dos *Event* pueden ser equivalentes solo si son del mismo tipo.	OCL	Definición de Eventos
12	Toda equivalencia entre *Event* es única.	OCL	Definición de Eventos
13	Toda equivalencia entre *Event* requiere estar expresada en todas las *Entity* involucradas.	OCL	Definición de Eventos
14	Cuando hay equivalencias en un *PrimaryEvent*, su *Effect* debe estar especificado una única vez.	OCL	Definición de Eventos
15	El nombre de un *Activity* es único dentro de un *Entity*.	OCL	Definición de Actividad

16	Los *Event* que definen un *Activity* deben estar definidos en la *Entity* a la cual pertenece el *Activity*.	OCL	Definición de Actividad
17	El par de *Event* que define un *Activity* debe ser único.	OCL	Definición de Actividad
18	Un *Activity* no puede usar como inicio y fin el mismo *Event*.	OCL	Definición de Actividad
19	Un *Activity* debe terminar con un *PrimaryEvent*.	Metamodelo	Definición de Actividad
20	El nombre de un *Delay* es único dentro de un *Entity*.	OCL	Definición de Retraso
21	Los *Event* que definen un *Delay* deben estar definidos en la *Entity* a la cual pertenece el *Delay*.	OCL	Definición de Retraso
22	El par de *Event* que define un *Delay* debe ser único.	OCL	Definición de Retraso
23	Un *Delay* no puede usar como inicio y fin el mismo *Event*.	OCL	Definición de Retraso
24	Un *Delay* debe terminar con un *SecondaryEvent*.	Metamodelo	Definición de Retraso
25	El nombre de un *Attribute* es único dentro de un *Entity*.	OCL	Definición de Atributo
26	El *Type* de un *Attribute* debe ser un tipo de dato básico o referir al nombre de un *Entity* del *Model* donde está definido el *Attribute*. Si es un *Entity*, no puede ser la misma donde está definido el *Attribute*.	OCL	Definición de Atributo
27	El nombre de un *Function* es único dentro de un *Entity*.	OCL	Definición de Función
28	Un *FunctionSet* solo puede indicar como *AttributeToSet* a un *Attribute* de la *Entity* donde la *FunctionSet* está definida.	OCL	Definición de Función
29	Un *FunctionGet* solo puede indicar como *AttributeTo-Return* a un *Attribute* de la *Entity* donde la *FunctionGet* está definida.	OCL	Definición de Función
30	Un *FunctionReturn* solo puede tener como *Return* a un tipo de dato básico o el nombre de un *Entity* del *Model* (diferente al *Entity* que define el *Function*).	OCL	Definición de Función
31	El *Body* de un *FunctionReturn* debe contener al menos un *Attribute* de la *Entity* donde está definida la *Function*.	OCL	Definición de Función
32	El *SystemState* debe tener al menos un elemento (es decir, no puede estar vacío).	Metamodelo	Estado del Sistema
33	Una *StateVariable* es única dentro de un *SystemState*.	OCL	Estado del Sistema
34	Una *StateVariable* del *SystemState* debe corresponder a un *Attribute*, *FunctionReturn* o *FunctionGet* de alguna *Entity* del *Model*.	OCL	Estado del Sistema

3.2 Sintaxis Concreta del Lenguaje

El diseño de la sintaxis concreta (en este caso, textual) se definió haciendo uso de Eclipse Xtext™[6]. Esta herramienta provee un framework para el desarrollo de lenguajes (tanto de programación como específicos de dominio) que facilita el diseño y la implementación de lenguajes textuales usando una gramática específica basada en Xtext. Tanto Xtext como el producto especificado con Xtext corren sobre la plataforma Eclipse.

La sintaxis propuesta consta de múltiples reglas que permiten definir cada uno de los constructores especificados en la sintaxis abstracta haciendo uso de un conjunto de palabras reservadas (por ejemplo, `model`, `attribute`, `activity`, `delay`, entre otras) y delimitadores (como ser, `:;(){}`, entre otros). Luego, la definición de un *Modelo* del cual se obtiene un *Estado del Sistema* en base a un conjunto de *Entidades* se estableció de la siguiente manera:

```
model: <nombre>
systemState: (<at1> <- <ent>,..., <atN> <- < ent>)
<entidad1>
...
<entidadM>
```

donde los elementos definidos en `rojo` corresponden a elementos String y los elementos definidos en `verde` corresponden a una *Entidad*.

En este sentido, tomando como base la notación de los lenguajes estructurados, se buscó definir una sintaxis textual para el concepto *Entidad* que ilustre las relaciones entre constructores a nivel abstracto (es decir, que facilite el entendimiento de una entidad como un contenedor de atributos, eventos, actividades, retrasos, y funciones). Por este motivo, se decidió definir una *Entidad* como:

```
entity -> <nombre> = {
   attributes = { <atributo1>; ... ; <atributoA>; }
   activities = { <actividad1>; ... ; <actividadM>; }
   delays = { <retraso1>; ...; <retrasoD>; }
   events = { <evento1>; ... ; <eventoE>; }
   functions = { <funcion1>; ... ; <funcionF>; }
}
```

Luego, para cada uno de los constructores internos, se definió una sintaxis que permite listar la cantidad de componentes deseados por el modelador. Por ejemplo, las *Actividades* quedan definidas como:

```
activity: <nombre> [startsWith <eventoA> | endsWith <eventoB>];
```

[6] Disponible en https://eclipse.dev/Xtext/.

Por razones de espacio, no es posible mostrar todos los constructores diseñados para dar soporte al lenguaje. Sin embargo, en el siguiente apartado se presenta una prueba de concepto donde se visualiza el desarrollo de un ejemplo sencillo en el editor Java implementado. En dicho ejemplo se visualizan todos los constructores disponibles junto con las alternativas de modelado.

En este punto, es importante aclarar que, si bien el lenguaje de modelado ha sido diseñado desde cero, el mismo toma los constructores de PlantUML para la validación de los *Effect* asociados a *PrimaryEvents*. Por este motivo, se realizan dos validaciones adicionales. La primera valida que el *String* que se detalla como *Effect* cuyo tipo corresponde a *PlantUML::ActivityDiagram* debe corresponder a un diagrama de actividad bien definido según PlantUML[7]. Por su parte, la segunda restricción valida que dicho *String* (ya habiendo sido validado como diagrama de actividad en PlantUML), cumpla con lo siguiente: "Un diagrama de actividad UML válido será correcto como efecto de un evento si los nodos de acción y de decisión dependen, al menos, de: *(a)* un atributo o función de la entidad donde el efecto tiene lugar, *(b)* un atributo o función de una entidad que posee un evento equivalente al evento que produce el efecto descripto en el diagrama, o *(c)* corresponde a la programación de un nuevo evento similar al que está siendo definido por el efecto".

4 Editor y Prueba de Concepto

A fin de mostrar la herramienta de software que da soporte a la edición y validación de modelos según el lenguaje descripto en la Sección 3, en este apartado utilizaremos el caso de estudio propuesto en (Alvarez et al., 2013). Este caso de estudio, si bien es sencillo, permite afianzar los conceptos DES asociados a partir de un *Conceptual Model*. Para mayores detalles de este artefacto, consultar (Alvarez et al., 2013).

El *Modelo Conceptual* tomado como punto de partida para la definición DES abstrae la situación de Despacho de Equipaje en una Aerolínea, teniendo como objetivo analizar el proceso de atención al cliente. Para esto, el modelador ha identificado tres componentes: *Pasajero*, *Mostrador*, y *Cola*. En el artefacto previo, se han identificado además entradas y salidas, como ser: *a) entradas*: tiempo entre arribos de pasajeros a cola, tiempo de atención de empleado en mostrador, y *b) salidas*: porcentaje de ocupación del empleado en mostrador, cantidad de pasajeros en cola. Adicionalmente, para cada componente se identifica un nivel de detalle requerido a fin de dar respuesta a las salidas planteadas. Luego, por ejemplo, para el caso de *Mostrador*, se determina que es necesario incluir el tiempo total de ocupación, el tiempo total que ha estado libre, y, en caso de estar ocupado, el pasajero que está siendo atendido por el empleado. Un nivel de detalle similar es requerido para los otros componentes.

Partiendo de la información recopilada en el modelo previo, en las Fig. 3 a 5 se presenta el modelo DES (válido) asociado a dicho escenario. Debido a la magnitud de la definición completa, las figuras muestran una construcción parcial de las entidades definidas. Esto se observa en los apartados colapsados (los cuales pueden visualizarse siguiendo los números de línea en el margen izquierdo).

[7] Esto se logra integrando el plugin de PlantUMLcomo parte del editor del lenguaje propuesto.

despachodeequipaje.desml

```
model: DespachoEquipaje

systemState: (porcentajeDeOcupacion<-mostrador, cantidadPasajeros<-cola)

entity -> mostrador = {
     attributes = {
        attribute: tiempoTotalOcupado type: Float;
        attribute: tiempoTotalLibre type: Float;
        attribute: tiempoFinUltimaAtencion type: Float;
        attribute: DuracionAtencion type: Float;
        attribute: Empleado type: pasajero;
    }
    activities = {
        activity: atender [ startsWith comienzaAtencion | endsWith finalizaAtencion];
    }
    delays = {
        delay: esperar [ startsWith finalizaAtencion | endsWith comienzaAtencion];
    }
    events = {...
    functions = {...
}
```

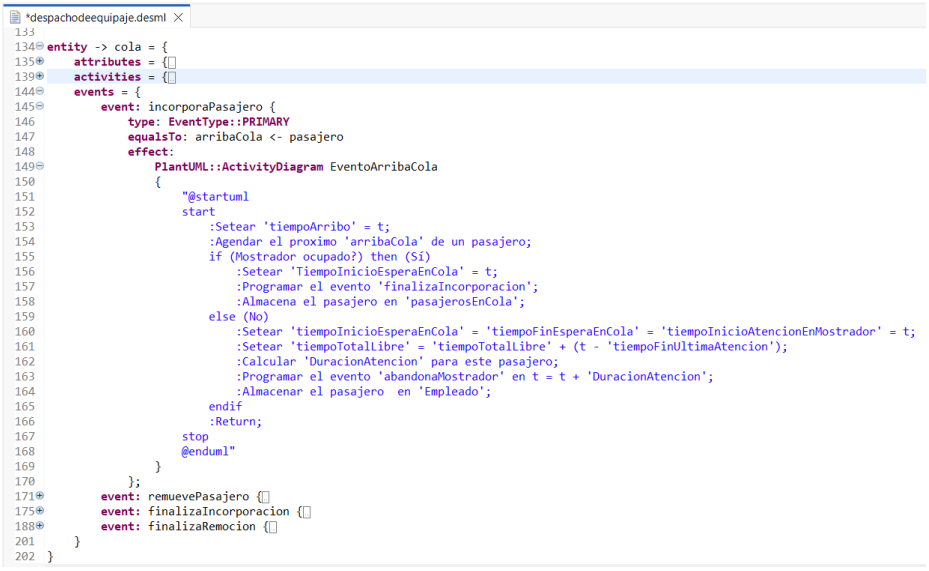
Fig. 3. Definición del modelo, estado del sistema y entidad “mostrador”.

```
entity -> pasajero = {
    attributes = {
        attribute: tiempoInicioEsperaEnCola type: Float;
        attribute: tiempoFinEsperaEnCola type: Float;
        attribute: tiempoInicioAtencionEnMostrador type: Float;
        attribute: tiempoFinAtencionEnMostrador type: Float;
        attribute: tiempoArribo type: Float;
    }
    activities = {
        activity: despacharEquipaje [startsWith arribaMostrador | endsWith abandonaMostrador ];
    }
    delays = {
        delay: esperar [startsWith arribaCola|endsWith abandonaCola];
    }
    events = {...
}
```

Fig. 4. Definición de la entidad “pasajero”.

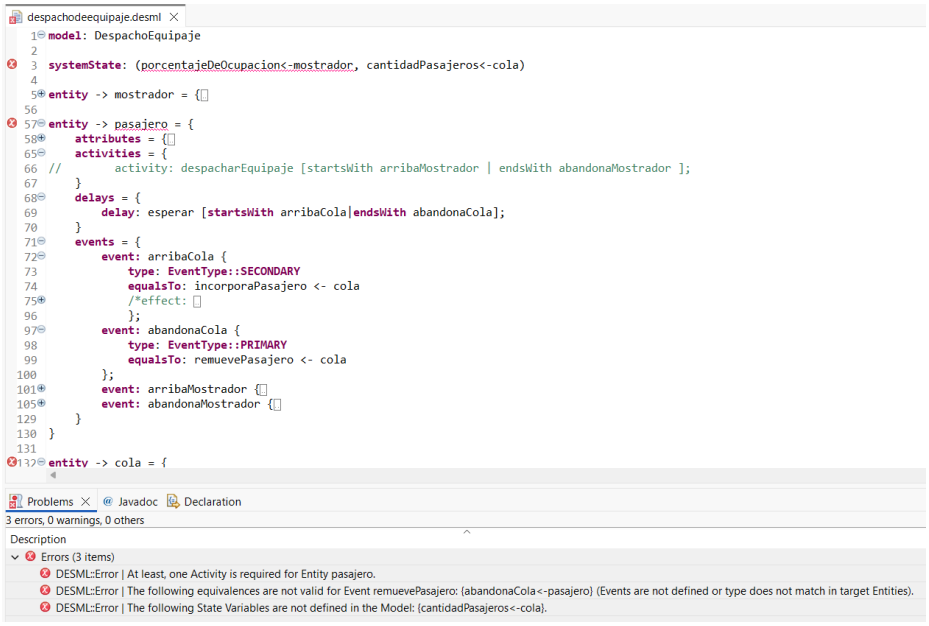
La Fig. 3 presenta la primera parte del modelo, definiendo su nombre y el estado del sistema. Este estado corresponde a la definición de salidas propuestas en el Modelo Conceptual. Luego, se visualiza la definición de la entidad *Mostrador*, la cual queda compuesta de atributos, actividades, retrasos, eventos y funciones (una única función llamada *porcentajeDeOcupacion*). La Fig. 4 muestra la conformación de la entidad *Pasajero*, donde se observa que esta entidad no tiene apartado de funciones (opcional). Finalmente, la Fig. 5 presenta la entidad *Cola*, ilustrando la forma en la cual un evento primario (*incorporaPasajero*) es definido en PlantUML. Aquí se observa que dicho evento es equivalente al evento *arribarCola* de la entidad pasajero (el cual no se observa en la Fig. 4 ya que el bloque *events* se encuentra colapsado).

Si bien las Fig. 3 a 5 presentan el modelo “valido”, en caso de no pasar una validación el editor provee una descripción detallada de los errores encontrados. Un ejemplo de este comportamiento se visualiza en la Fig. 6, donde se ha comentado la línea 66 (haciendo que la restricción 7 deje de ser correcta) y se ha cambiado el *type* del evento *arribaCola* a *Secondary* (haciendo que la equivalencia con *incorporaPasajero* deje de ser válida, ya que viola la restricción 11).



```
*despachodeequipaje.desml
entity -> cola = {
    attributes = {
    activities = {
    events = {
        event: incorporaPasajero {
            type: EventType::PRIMARY
            equalsTo: arribaCola <- pasajero
            effect:
                PlantUML::ActivityDiagram EventoArribaCola
                {
                    "@startuml
                    start
                        :Setear 'tiempoArribo' = t;
                        :Agendar el proximo 'arribaCola' de un pasajero;
                        if (Mostrador ocupado?) then (Sí)
                            :Setear 'TiempoInicioEsperaEnCola' = t;
                            :Programar el evento 'finalizaIncorporacion';
                            :Almacena el pasajero en 'pasajerosEnCola';
                        else (No)
                            :Setear 'tiempoInicioEsperaEnCola' = 'tiempoFinEsperaEnCola' = 'tiempoInicioAtencionEnMostrador' = t;
                            :Setear 'tiempoTotalLibre' = 'tiempoTotalLibre' + (t - 'tiempoFinUltimaAtencion');
                            :Calcular 'DuracionAtencion' para este pasajero;
                            :Programar el evento 'abandonaMostrador' en t = t + 'DuracionAtencion';
                            :Almacenar el pasajero  en 'Empleado';
                        endif
                        :Return;
                    stop
                    @enduml"
                }
            };
        event: remuevePasajero {
        event: finalizaIncorporacion {
        event: finalizaRemocion {
    }
}
```

Fig. 5. Definición de la entidad “cola”.



```
despachodeequipaje.desml
model: DespachoEquipaje

systemState: (porcentajeDeOcupacion<-mostrador, cantidadPasajeros<-cola)

entity -> mostrador = {

entity -> pasajero = {
    attributes = {
    activities = {
//        activity: despacharEquipaje [startsWith arribaMostrador | endsWith abandonaMostrador ];
    }
    delays = {
        delay: esperar [startsWith arribaCola|endsWith abandonaCola];
    }
    events = {
        event: arribaCola {
            type: EventType::SECONDARY
            equalsTo: incorporaPasajero <- cola
            /*effect: {
            };
        event: abandonaCola {
            type: EventType::PRIMARY
            equalsTo: remuevePasajero <- cola
        };
        event: arribaMostrador {
        event: abandonaMostrador {
    }
}

entity -> cola = {

Problems  Javadoc  Declaration
3 errors, 0 warnings, 0 others
Description
Errors (3 items)
    DESML::Error | At least, one Activity is required for Entity pasajero.
    DESML::Error | The following equivalences are not valid for Event remuevePasajero: (abandonaCola<-pasajero) (Events are not defined or type does not match in target Entities).
    DESML::Error | The following State Variables are not defined in the Model: {cantidadPasajeros<-cola}.
```

Fig. 6. Visualización de errores encontrados en el proceso de validación.

Adicionalmente, puede observarse que se ha cambiado el *type* del evento *abandonaCola* que define el fin del *Delay* denominado *esperar*. Esto resultaría en un error debido a la restricción 24. Sin embargo, este error no se visualiza en el panel inferior. Esto se debe a que el orden de las validaciones sigue la secuencia indicada en

la Tabla 2, siendo bloqueante el no cumplimiento de una restricción en algunos casos. Esto implica que, como se observa en el ejemplo, el error detectado en relación con el cambio en *abandonaCola* refiere al no cumplimiento de la equivalencia con *remuevePasajero* de la entidad *Cola* (los eventos son de diferente tipo por lo que violan la restricción 11). Esto da como resultado que la entidad *Cola* no esté bien definida en el *Model*, lo que hace que el validador determine que no existe la *StateVariable* denominada *cantidadPasajeros* (ya que no existe la *Entity* denominada *Cola*). Aquí se observa como la restricción 24 es bloqueada por la restricción 11. Si se resuelve la violación de la restricción 11, también se resolverá el conflicto en la restricción 24 (como consecuencia de tener eventos equivalentes).

Si bien el orden no forma parte de la correcta definición de un modelo, ayuda en el proceso de modelado DES, ya que el modelador puede centrar su atención en los conceptos DES elementales e ir mejorando su especificación a medida que el artefacto evoluciona con bloques correctamente definidos.

5 Conclusiones y Trabajos Futuros

En este trabajo se ha presentado un lenguaje de modelado textual para la especificación del artefacto *Model Design* como un modelo DES. Dicho lenguaje ha sido diseñado en base a un metamodelo (sintaxis abstracta) que, junto con una sintaxis concreta estructurada en base a una representación textual, ha sido implementada en el entorno de desarrollo Eclipse haciendo uso de la herramienta Xtext.

Nuestra propuesta busca fortalecer las habilidades de modelado vinculado a DES, proporcionando un enfoque claro y estructurado para que los estudiantes de cualquier disciplina puedan abordar de forma sencilla el diseño de modelos de simulación basados en eventos discretos. Como resultado, se presenta un editor del lenguaje propuesto, el cual forma parte de un framework de modelado DES en proceso de desarrollo.

El lenguaje propuesto constituye el primer componente (finalizado) de dicho framework, el cual brindará la posibilidad de transformar modelos DES en modelos computacionales. Además, permitirá representar la abstracción diseñada en términos del metamodelo propuesto; junto con la obtención de los diagramas asociados tanto a la estructura como a la dinámica definida. La trazabilidad requerida entre artefactos forma parte de los trabajos futuros.

Referencias

Alvarez, G., Sarli, J., Blas, M. J. (2023). Una Propuesta de Extensión de un Framework para el Desarrollo de Modelos Conceptuales para Simulación. Actas del 2023 Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI).

Banks, J., Carson, J., Nelson, B., Nicol., D. (2010). Discrete-Event System Simulation. 5th ed. Prentice-Hall.

Fishwick, P.A. (1995). Simulation Model Design and Execution: Building Digital Worlds. Prentice-Hall, Upper Saddle River, New Jersey.

Ören, T., Mittal, S., Durak, U. (2018). A Shift from Model-Based to Simulation-Based Paradigm: Timeliness and Usefulness for Many Disciplines. International Journal of Computer & Software Engineering 3(2018),126.

Robinson, S. (2008). Conceptual Modelling for Simulation Part I: Definition and Requirements. Journal of the Operational Research Society, 59(3), 278–290.

Robinson, S., Brooks, R.J., Kotiadis, K. Van Der Zee, D-J. (2011). Conceptual Modeling for Discrete-Event Simulation. Chapman and Hall/CRC.

Robinson, S. (2014). Simulation: The Practice of Model Development and Use. 2nd ed. Palgrave MacMillan.