# Verificación formal de un modelo de simulación DEVS de una aplicación Storm

*Formal verification of a DEVS simulation model of a Storm application*

Alonso Inostrosa-Psijas[1]* Mauricio Oyarzún-Silva[1]
Fernando Medina-Quispe[1] Francisco García-Barrera[1] Roberto Solar-Gallardo[2]

Recibido 8 de mayo de 2019, Aceptado 14 de junio de 2019
*Received: May 08, 2019 Accepted: June 14, 2019*

**RESUMEN**

Las plataformas de procesamiento de permiten la manipulación y análisis de datos en tiempo real. Un sistema reconocido para este propósito es la llamada plataforma Storm, que es un sistema para computación distribuida, de código abierto, escalable, rápido y tolerante a fallos. La plataforma Storm puede ser desplegada sobre sobre un gran número de procesadores. Sin embargo, la determinación del número apropiado de procesadores que son necesarios para ejecutar adecuadamente aplicaciones de software para Storm no es una tarea fácil ni simple, especialmente si la aplicación está diseñada para ser usada por un gran número de usuarios. Una forma de resolver este problema es mediante un modelo de simulación que permita evaluar su rendimiento usando diferentes escenarios de cargas de trabajo. En este artículo, se presenta un modelo de simulación de una aplicación Storm usando el formalismo DEVS, posteriormente se define un modelo equivalente usando Autómatas Temporizados (AT). Mediante un proceso denominado "bisimulación" se comprueba que ambos modelos sean efectivamente equivalentes. Finalmente, el modelo descrito usando AT es verificado formalmente evaluando sus propiedades, demostrando que el modelo de simulación posee el mismo comportamiento que la aplicación real.

Palabras clave: Simulación, verificación formal, DEVS, autómatas temporizados.

***ABSTRACT***

*Stream processing platforms allow processing and analyzing realtime data. A recognized system for this purpose is the so-called Storm platform, an open-source system for distributed computing, which is a scalable, fast and, fault-tolerant. The Storm platform may be deployed on a large number of processors. However, determining the proper number of processors that are suitable to execute a given Storm based software application is a challenging task, especially if the application is intended to serve a very large user community. One way of solving this is by employing a simulation model that allows the evaluation of its performance under different workload scenarios. In this paper, a simulation model of a Storm application using the DEVS formalism is presented, then, an equivalent model is described using Timed Automata (TA). Throughout a procedure called "Bisimulation", it is checked that both models are effectively equivalent. Finally, the model defined using TA is formally verified evaluating its properties, proving that the simulation model has the same behavior as the real applications.*

*Keywords: Simulation, formal verification, DEVS, timed automaton.*

[1] Universidad Arturo Prat. Facultad de Ingeniería y Arquitectura. Iquique, Chile.
E-mail: alinostros@unap.cl; moyarzunsil@unap.cl; femedina@unap.cl; francgar@unap.cl
[2] Universidad de Santiago de Chile. Departamento de Ingeniería Informática. Santiago, Chile. E-mail: roberto.solar@usach.cl
* Autor de correspondencia: alinostros@unap.cl

## INTRODUCCIÓN

Las plataformas de procesamiento de *streams* están diseñadas para manejar grandes volúmenes de datos provenientes de diversas fuentes (redes sociales, publicidad, aplicaciones científicas, etc.). Estas plataformas deben procesar estos grandes volúmenes de datos en muy poco tiempo, con una latencia muy baja y un alto *throughput*. El procesamiento de estos datos debe ser realizado con la misma velocidad con que arriban. Una de las plataformas de procesamiento de *streams* más popular es Apache Storm [(1)]. La experimentación sobre estas plataformas en ambientes de producción resulta muy costosa, y en muchos casos, imposible debido a los riesgos de aplicar estrategias que afecten su rendimiento.

Es por este motivo que las herramientas de simulación resultan muy efectivas. Sin embargo, un simulador para que sea útil como herramienta de predicción, no sólo debe estar validado, sino que también debe estar correctamente verificado. A pesar de esto, la verificación es una tarea que muchas veces se deja de lado, aun cuando de esta forma se puede asegurar que el modelo de simulación se comporta de manera símil al sistema real.

En este trabajo, se presenta un modelo formalizado utilizando DEVS, el que luego es transformado a un modelo equivalente de Autómatas Temporizados (AT) mediante un proceso conocido como bisimulación. Luego, al modelo AT se lo ha verificado formalmente para garantizar un modelado adecuado.

El resto de este trabajo se organiza de la siguiente manera. Se presentan antecedentes relacionados con la plataforma de procesamiento de *streams* Apache Storm, luego se presentan los formalismos DEVS, y luego el formalismo de AT. Después, se presenta el proceso de bisimulación, que permite asegurar la correspondencia entre un mismo modelo descrito en dos formalismos diferentes. Luego, se presenta un caso de estudio, al cual se le aplica el proceso de bisimulación. A continuación, al modelo AT equivalente se lo verifica formalmente analizando las propiedades de *alcanzabilidad*, *seguridad*, *vivacidad* y *de ausencia de interbloqueos*. Finalmente, se presentan las conclusiones de este trabajo.

## ANTECEDENTES

### Plataforma Apache Storm

Apache Storm es un sistema de procesamiento de *stream* en tiempo real, tolerante a fallos y distribuido [1]. Este sistema provee un set de primitivas para realizar computación en tiempo real. Storm es utilizado por muchas compañías, incluyendo a Twitter para personalizaciones y búsqueda; Flipboard para la generación de información personalizada; Weather Channel, WebMD.com, entre otros.

La API además soporta múltiples lenguajes como Java, Python y Ruby. Storm está formado por cinco componentes principales: *tuplas*, *streams*, *spouts*, *bolts* y *topologías*. Una tupla es una lista ordenada de elementos. Por ejemplo, para datos provenientes de Twitter, una tupla puede ser el nombre de usuario seguido del *tweet* (<"INFONOR 2018", "Plazo por finalizar.">). Un *stream* es una secuencia de tuplas potencialmente infinita. En esencia, se tiene una secuencia de tuplas de entrada (que forman un *stream*), y estas tuplas son procesadas una a la vez.

Las aplicaciones en Storm son llamadas topologías. Una topología es un grafo dirigido, donde sus vértices son operadores (*spouts* o *bolts*) y sus aristas representan el flujo de los datos a través de ellos. Las topologías pueden contener ciclos, pero en este caso, el programador debe ser cuidadoso de no crear ciclos infinitos, donde las tuplas recorran el sistema en un procesamiento sin término.

Los *spouts* son la fuente de los datos de un *stream* en la topología de Storm. Usualmente, los datos que conforman las tuplas son leídas desde una fuente externa como un *crawler* o una base de datos. Un *spout* puede generar simultáneamente múltiples *streams*.

En la topología de Storm, los *bolts* son los encargados del procesamiento de los *streams* a través de una tarea u operación predefinida para cada tupla. Los *bolts* procesan una gran cantidad de datos de manera rápida y eficiente. Usualmente un *bolt* procesa información proveniente de un solo *stream* de entrada, sin embargo, puede también procesar múltiples *streams* de entrada, generando un *stream* de salida que puede ser usado para alimentar para alimentar otros *bolts*. Algunas de las operaciones que realiza un *bolt* son:

- Filtrar: reenvía una tupla solo si esta satisfice una condición.
- Unir: cuando se reciben múltiples *streams*, digamos A y B, se calcula el producto cruz para todas las tuplas en el *stream* A y el *stream* B, y para cada par de tuplas (una en A y otra en B), se retorna el resultado de este proceso si el par de tuplas satisfacen una condición dada. Este proceso es muy similar a la noción de unión (join) de las bases de datos.
- Transformar: modifica la tupla de acuerdo a una función dada.
- Emisor: controla la emisión de tuplas. Las tuplas son emitidas en intervalos regulares de tiempo.

La Figura 1 muestra la capa lógica de una topología en Storm. En este ejemplo, tenemos una fuente de datos o *spout*, y cuatro unidades de proceso *bolt*. El *spout* recolecta *streams* desde diferentes fuentes. Las tuplas entrantes forman un *stream* que es enviado a diferentes *bolts*. Los enlaces entre las diferentes unidades (*spouts* y *bolts*) representan el flujo de datos del *stream*.

Para que un *bolt* sea más rápido, pueden ser desplegadas múltiples réplicas de él, ejecutándose en paralelo en múltiples procesos o tareas, permitiendo la reducción de la carga general de trabajo del *bolt*. Para esto, el *stream* de entrada es dividido entre estas tareas. Usualmente, cada tupla de entrada es asignada a una tarea del *bolt*, pero esta puede ir a múltiples tareas. La asignación de las tuplas a cada tarea es decidida por una estrategia de agrupamiento. Algunas de las estrategias de agrupamiento más populares soportadas por Storm son: el agrupamiento por mezclas, el agrupamiento por campos y el *broadcast*. El agrupamiento por mezclas distribuye las tuplas de un *stream* en las tareas del *bolt*, utilizando *Round-Robin*. El agrupamiento por campos agrupa las tuplas de un *stream* por un subgrupo de sus campos enviando cada tupla al grupo que le corresponde utilizando una función *hash*. En el *broadcast*, cada tarea en el *bolt* recibe todas las tuplas del *stream*, siendo esto útil para realizar la unión.

## FORMALISMO DEVS

La Especificación de Sistemas por Eventos Discretos (*Discrete-Event System Specification* o DEVS) [2] es el modelamiento y formalismo de simulación para sistemas dinámicos de eventos discretos. DEVS describe sistemas mediante modelos atómicos y modelos acoplados. Los modelos atómicos son las entidades elementales y básicas para representar sistemas. Los modelos atómicos pueden reaccionar a eventos internos y externos.

Estos definen una manera de especificar sistemas donde los estados cambian durante la recepción de un evento de entrada o la expiración de un periodo de tiempo. En DEVS un modelo atómico se define como [2]:

$$DEVS = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >,$$

donde $X$ es el conjunto de eventos *externos*;
$Y$ es el conjunto de eventos de *salida*;
$S$ es el conjunto de estados *secuenciales*;
$\delta_{ext}\ Q \times X \rightarrow S$ es la función de transición *externa*, donde $Q = \{s, e\} \mid s\ e\ S,\ 0 \leq e \leq ta(s)\}$ y es el tiempo transcurrido desde la última transición de estado;

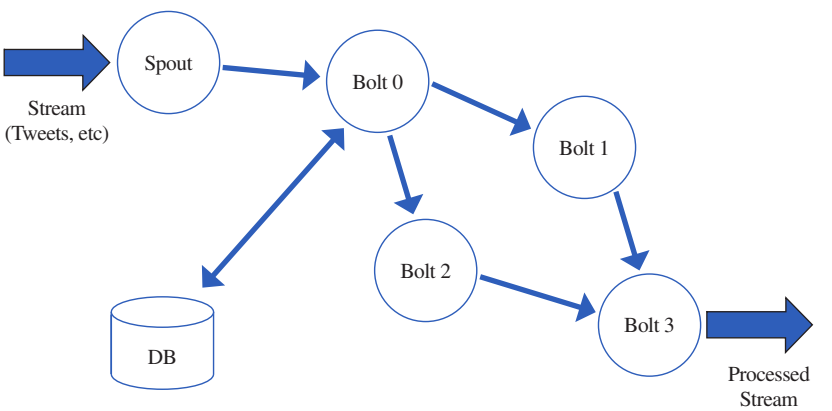


Figura 1. Topología de una aplicación Storm.

$\delta_{int}$: $S \rightarrow S$ es la función de transición de estado *interna*;
$\lambda$: es la función de *salida*;
*ta:* es la función de *avance de tiempo*.

Su semántica es como sigue: Para cualquier tiempo dado, un modelo DEVS está en un estado $s \in S$ y en la ausencia de eventos externos, se quedará en este estado por un periodo de tiempo definido por *ta(s)*. La función *ta(s)* puede tomar cualquier número real entre 0 e ∞. Un estado para el cual $ta(s) = 0$ es llamado un estado transiente. Por el contrario, si $ta(s) = \infty$, el sistema debe permanecer en ese estado de forma permanente, a menos que un evento externo sea recibido. En este caso, s es llamado un estado pasivo.

Las transiciones que ocurren debido a la caducidad de *ta(s)*, son llamadas transiciones internas. Cuando ocurre una transición interna, la salida del sistema es el valor $\lambda(s)$, y el estado $\delta_{int}(s)$ es cambiado. Un estado de transición puede también ocurrir por la incidencia de un evento externo. En este caso, el nuevo estado está dado por $\delta_{ext}$ basado en el valor de entrada, el estado actual y el tiempo transcurrido. El formalismo DEVS incluye modelos atómicos y acoplados. Los modelos atómicos permiten representar el comportamiento de un sistema, mientras que un modelo acoplado representa su estructura. Un modelo acoplado agrupa múltiples modelos DEVS, juntos en un modelo compuesto que puede ser considerado, debido a la propiedad de cierre, como otro modelo DEVS. Un modelo acoplado es definido como una estructura de la forma [3]:

$$DN = X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >,$$

donde $D$ es un conjunto de componentes, y para cada $I \in D$, $M_i$, es un componente con la restricción de que $M_i = < X_i, Y_i, S_i, \delta_{ext_i}, \delta_{int_i}, \lambda_i, ta_i >$ es un modelo DEVS;
para cada $i \in D \cup \{self\}$, $I_i$, es el conjunto de influencias de $i$;
para cada $j \in I_i$, $Z_{i,j}$, es una función de mapeo salida-entrada $i - a - j$;
*select* es una función que, de un grupo de eventos que deben ejecutarse en el mismo tiempo, permite decidir cuál de ellos se ejecutará primero;
$I_i$ es un subconjunto de $D \cup \{self\}$, $i$ no está en $I_i$;
$Z_{self,j}$: $X_{self,j} \rightarrow X_j$;
$Z_{i,self}$: $Y_i \rightarrow Y_{self}$;
$Z_{i,j}$: $Y_i \rightarrow X_j$;
*select:* subconjunto de $D \rightarrow D$ tal que, para cualquier subconjunto no vacío de $E$, $select(E) \in E$.

Un modelo acoplado puede tener sus propios eventos de entrada y salida, definidas por los sets $X_{self}$ e $Y_{self}$. Cuando llega un evento externo, el modelo acoplado debe redirigir la entrada a uno o más de sus componentes. Adicionalmente, cuando uno de sus componentes produce una salida, esta debe ser redirigida a la entrada de otro componente o a la salida del modelo acoplado. La función $Z$ define las redirecciones de estas entradas/salidas.

En DEVS, los modelos acoplados pueden ser integrados para formar una jerarquía de modelos, lo que permite su reutilización. Esto, gracias a la propiedad de *acoplamiento bajo clausura*, que garantiza que los modelos acoplados pueden ser considerados como modelos atómicos [2].

**Aplicaciones de DEVS**

Actualmente, DEVS y sus extensiones, son ampliamente utilizadas en una gran variedad de aplicaciones. En [4] se presenta un modelo Cell-DEVS de un sistema de tráfico aéreo para evaluar el riesgo de colisión de aeronaves no tripuladas con otros aviones. Los autores de [5] se desarrollaron un modelo de simulación de un motor de búsqueda web de gran escala utilizando DEVS. En [6] Cell-DEVS es utilizado para estudiar el proceso de *handover* en redes móviles 5G. DEVS también ha sido usado para modelar la forma en que los *malware* se propagan en diferentes tipos de redes [7]. Otro tipo de aplicaciones de Cell-DEVS ocurre en el diseño de infraestructuras de servicios de salud que requieren de coordinación de actividades [8].

**Extensiones de DEVS**

Existen muchas extensiones de DEVS, una de ellas es *Parallel DEVS* o PDEVS [9], que define una función adicional –llamada función confluente. –que mejora el mecanismo que maneja eventos simultáneos. *Cell-DEVS* [10] es otra extensión que permite soportar la descripción de modelos de eventos discretos mediante espacios de células n-dimensionales, donde cada célula corresponde a un modelo DEVS básico. El espacio n-dimensional es entonces un modelo acoplado DEVS. *Parallel Cell-DEVS* es una extensión a

Cell-DEVS que permite soportar un nivel mayor de paralelismo, permitiendo múltiples eventos en los puertos de salida, entregando una manera de superar los problemas que surgen en simulaciones secuenciales cuando dos o más eventos poseen el mismo tiempo [11].

## AUTÓMATAS TEMPORIZADOS

Los autómatas temporizados (AT) [12] son una extensión a los autómatas finitos que permiten representar el comportamiento de sistemas con restricciones de tiempo. Un AT se define como una tupla [12]: A = *(N, $l_0$, E, l)* donde N es un conjunto finito de nodos o ubicaciones, $l_0 \in N$ es la ubicación inicial, $E \subseteq N \times \beta(C) \times \Sigma \times 2^c \times N$ es un conjunto de aristas, y $I : N \rightarrow \beta(C)$ asigna invariantes a las ubicaciones. Donde $C$ corresponde a un conjunto de variables de reloj, $\Sigma$ es un conjunto de acciones y $2^c$ es una selección de relojes a ser reiniciados a cero. Existen restricciones aplicadas a las variables de reloj llamadas "guardas" (*guards*) que pueden ser usadas en las transiciones (arcos), o pueden ser usadas en los nodos (donde son llamadas invariantes). Las guardas son restricciones que deben ser satisfechas para poder activar una determinada transición, y los invariantes restringen el tiempo que se puede permanecer en un determinado nodo del AT.

En AT, los estados son pares de forma $<L, u>$, donde $L$ es una ubicación y $u$ es un valor de reloj [13]. Una expresión de la forma $l \xrightarrow{g,a,r} l'$ cuando $(l, g, a, r, l') \in E$ es una transición desde la ubicación l a la ubicación l' donde g es una restricción de reloj, a es una acción y r es un conjunto de relojes ajustados a cero. Dos tipos de transiciones son usadas en AT: transiciones con retardo y transiciones de acción. En una transición con retardo de la forma $< L,u > \xrightarrow{d} < L,u+d >$, el avance del tiempo $d$, provoca una transición desde una ubicación de inicio a una ubicación de destino. En una transición de acción $< L,u > \xrightarrow{a} < L',u' >$, una acción desencadena una transición desde la ubicación $L$ a la ubicación $L'$.

## BISIMULACIÓN

Para verificar modelos DEVS mediante cualquiera de las herramientas de software para AT, se debe construir modelos AT que sean equivalentes en cuanto a su comportamiento al modelo DEVS original. Para esto, se utiliza un método denominado "bisimulación" [14] que permite su comprobación. La bisimulación entre dos sistemas $A$ y $B$, establece la relación entre cada estado de $A$ y su correspondiente en el sistema $B$, y relaciona las transiciones observables de $A$ y su correspondencia en $B$. La metodología que se sigue en este trabajo [15] se basa en una equivalencia de bisimilaridad temporizada débil.

Sin embargo, dado que la bisimulación corresponde a una relación binaria sobre un conjunto de estados, lo primero es definirlos a partir del modelo DEVS. Luego, se define la equivalencia de comportamiento para después determinar el AT para los elementos básicos de comportamiento de DEVS (las transiciones internas y externas). Finalmente, se deducen los valores constantes del modelo AT para completar la equivalencia. La verificación se realiza mediante el software UPPAAL [16] que permite validar las propiedades de modelos AT.

## CASO DE ESTUDIO

### Aplicación Storm

La Figura 2, muestra la topología correspondiente a la aplicación en estudio[1]. Consta de un *spout* y siete *bolts*, los que realizan las siguientes operaciones:

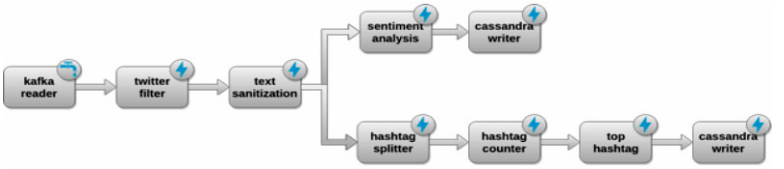


Figura 2. Topología de la aplicación Storm en estudio.

[1] https://github.com/mserrate/twitter-streaming-app (último acceso - 7 de enero, 2019)

- *Spout Kafka Reader*: Obtiene *tweets* desde *Kafka* y los inyecta en la topología, enviándolos al *bolt Twitter Filter.*
- *Bolt Twitter Filter*: Realiza un filtrado de los *tweets* que se reciben desde el *spout*. Solamente los *tweets* escritos en idioma inglés son enviados al siguiente *bolt* en la topología.
- *Bolt Text Sanitization*: aplica un operador de normalización al texto del *tweet*. Este *bolt* envía el *tweet* a dos *bolts* diferentes, según se indica en la topología.
- *Sentiment Analysis*: Aplica un algoritmo de análisis de sentimiento (*SentiWordNet Classifier* [17]), dando puntuación a cada *tweet* de acuerdo con las palabras que contenga.
- *Hashtag Splitter*: Separa los diferentes hashtags del *tweet* y emite una tupla por cada hashtag al siguiente *bolt*.
- *Hashtag Counter*: Realiza un conteo de las ocurrencias de cada *hashtag*.
- *Top Hashtag*: realiza un ranking de los top-*K hashtags*.
- *Cassandra Writer*: Realiza dos acciones: i) almacena *tweets* y su puntaje de análisis de sentimiento (en base a lo que recibe desde el *bolt sentiment analysis*), y ii) almacena los top-*K hashtags* (en base a tuplas que recibe desde el *bolt Top Hashtag*). Ambos son almacenados en una base de datos *Cassandra* [18].

**Modelo DEVS de la aplicación Storm**

El modelo DEVS está compuesto de un modelo acoplado correspondiente a la topología la aplicación. Sus componentes son modelos atómicos que corresponden al *spout* y a cada uno de los *bolts*. Por motivos de espacio sólo se describe en DEVS el *bolt Text Sanitization*. Su elección se debe a que los comportamientos internos de los *bolts* de la aplicación son muy similares entre sí, y por su ubicación en la topología de la aplicación, dado que su salida alimenta dos *bolts* diferentes.

De acuerdo a la descripción de la funcionalidad del *bolt Text Sanitization*, se observa lo siguiente:

1. Inicialmente, el *bolt* se encuentra en estado "Disponible". Revisando periódicamente la cola en busca de *tweets*. El *bolt* deberá permanecer en el estado "Disponible" mientras no arriben nuevos *tweets*.
2. Si en la cola encuentra un *tweet*, cambia su estado a "Ocupado" y aplica un operador (algoritmo de análisis de sentimiento) al elemento recibido. Una vez terminada la aplicación del operador, el *bolt* revisa la cola e intenta extraer un nuevo *tweet*.
   a. Si existe un nuevo *tweet*, entonces vuelve a (2), de modo que su estado continúa como "Ocupado".
   b. Si no hay un nuevo *tweet*, entonces vuelve a (1) y su estado cambia a "Disponible".

**Modelo DEVS**

De la descripción anterior, se deriva que el *bolt* sólo posee dos estados: "Disponible" y "Ocupado". Además, de esta descripción, se infieren las reglas de transición internas y externas para el modelo DEVS del *bolt*. En la Figura 3, se puede observar la definición del modelo acoplado DEVS que representa la aplicación completa.

En consecuencia, el *bolt Text Sanitization* se representa como un modelo atómico DEVS. Su descripción formal se aprecia en la Figura 4.

El conjunto de entrada *($X_{ts}$)* corresponde al conjunto de *tweets (T)* que son recibidas en el puerto de entrada "in''. El conjunto de salida *($Y_{ts}$)* corresponde al conjunto de *tweets* () que son emitidas por alguno de los puertos de salida del conjunto *OUT*. Los estados son "Disponible" y "Ocupado", y se utiliza la variable de estado que representa la cantidad de tweets encolados que serán procesados por el *bolt* en cuestión. Se tienen dos transiciones internas. La primera es recursiva en el estado "Ocupado", que restaura el tiempo transcurrido a 0 y decrementa en 1 el valor de $q_{ts}$. La segunda se ejecuta si en la cola no hay *tweets* ($q_{ts} = 0$) y cambia del estado "Ocupado" a "Disponible" por tiempo indeterminado o hasta que se active alguna transición externa.

Si estando en el estado "Ocupado" se recibe una consulta, se restaura el tiempo transcurrido al tiempo que se debe esperar unidades de tiempo, y se incrementa en 1 el valor de $q_{ts}$.

Luego, si estando en el estado "Disponible" se recibe una consulta, el estado cambia a "Ocupado" y se ajusta el tamaño de la cola al valor 1. La función de salida retorna un tweet y es enviada por el puerto

$$App = < X_{app}, Y_{app}, D_{app}, \{M_{d_{app}}\}, \{I_{app}\}, \{Z_{app}\}, select_{app} >$$
$$X_{app} =< \emptyset >$$
$$Y_{app} =< \emptyset >$$
$$D_{app} = \{\text{kafkaReader, twitterFilter, textSanitization, sentimentAnalysis, hashtagSplitter, hashtagCounter, topHashtag, cassandraWritter}\}$$
$$M_{d_{app}} = \{\text{kafkaReader, twitterFilter, textSanitization, sentimentAnalysis, hashtagSplitter, hashtagCounter, topHashtag, cassandraWriter}\}$$
$$I_{app} = \{((\text{self, in}), (\text{kafkaReader, in}))\}$$
$$Z_{app} = \{((\text{kafkaReader, out}), (\text{twitterFilter, in})); ((\text{twitterFilter, out}), (\text{textSanitization, in})); ((\text{textSanitization, out}), (\text{sentimentAnalysis, in}), (\text{hashtagSplitter,in})); ((\text{sentimentAnalysis, out}), (\text{cassandraWritter,in})); ((\text{hashtagSplitter,out}), (\text{hashtagCounter, in})); ((\text{hashtagCounter,out}), (\text{topHashtag, in})); ((\text{topHashtag, out}), (\text{cassandraWritter, in}))\}$$
$$select_{App} = \{\text{kafkaReader, twitterFilter, textSanitization, sentimentAnalysis, hashtagSplitter, hashtagCounter, topHashtag, cassandraWritter}\}$$

Figura 3. Modelo DEVS acoplado que describe la aplicación Storm.

$TS = \{X_{ts}, S_{ts}, Y_{ts}, \delta_{int_{ts}}, \delta_{ext_{ts}}, \lambda, ta\}$
$IN = \{\text{"in"}\}$ es el conjunto de puertos de entrada, con valores $X_{ts-in} = T$
Donde:
$T$ es el conjunto de objetos de tipo *tweet* (tupla).
$X_{ts} = \{(\text{"}in\text{"})|t \in T\}$ es el conjunto de puertos y sus valores de entrada (objetos de tipo *tweet*).
$q_{ts}$ corresponde a la cantidad de elementos enviados al *bolt* que se encuentran encolados.
$S_{ts} = \{\text{"Disponible", "Ocupado"}\} \times T \times q_{ts}$
donde:
$\sigma \in \mathbb{R}_0^+$ contiene el valor de avance de tiempo según la definición de la función $ta$ y $q_{ts} \in \mathbb{Z}_0$.
$OUT = \{\text{"out"}\}$ es el conjunto de puertos de salida con valores $Y_{ts-out} = T$;
donde:
$out$ es el puerto de salida que conecta con el puerto $in$ de los modelos correspondientes a los *bolts*: i) *sentiment analysis* y ii) *hashtag splitter*.
$Y_{ts} = \{(p, t)|p \in OUT, t \in T\}$
$$\sigma_{int_{ts}}(\text{"Ocupado"}, \sigma, q_{ts}) = \begin{cases} (\text{"Ocupado"}, 0, q_{ts} - 1) \text{ si } q_{ts} > 0 \\ (\text{"Disponible"}, \infty) \quad \text{e.o.c} \end{cases}$$
$$\sigma_{ext_{ts}}(\text{"Disponible"}, \sigma, q_{ts}) = (\text{"Ocupado"}, \sigma, 1)$$
$$\sigma_{ext_{ts}}(\text{"Ocupado"}, \sigma, q_{ts}) = (\text{"Ocupado"}, \sigma - e, q_{ts} + 1)$$
$$\lambda(\text{"Ocupado"}, \sigma, tweet) = (\text{out}, tweet)$$
$$ta(\text{"Ocupado"}, \sigma, tweet) = \sigma$$

Figura 4. Descripción formal en DEVS del *bolt Text Sanitization* como un modelo atómico.

de salida a los *bolts sentiment analysis* y *hashtag splitter*.

## VERIFICACIÓN FORMAL

La verificación formal es un método que permite evaluar la correctitud de un modelo. Una técnica de verificación formal es el *Model Checking* [19, 20], esta permite comprobar el comportamiento de un sistema mediante la exploración sistemática de todos los estados posibles de un modelo formal [21].

En general, los AT y Redes de Petri con Tiempo (RPT) [22] son los principales formalismos utilizados para modelar sistemas reactivos que evolucionan en el tiempo, es decir, que cambian de estado a medida que el tiempo transcurre. Las RPT son una extensión a las Redes de Petri convencionales [23],

con la añadidura de que en RPT se asocian dos marcas de tiempo (*min* y *max*) a cada transición, de modo que si el último disparo de una transición ocurrió en el tiempo $\theta$, entonces *t* debe volver a ser disparada en el intervalo $[\theta + min, \theta + max]$ a no ser que otra transición la desactive [24].

En AT, un estado es definido por su ubicación actual y el valor de sus variables de reloj. En cambio, en un modelo de RPT, su estado está determinado por su marcado (cantidad no negativa de *tokens* ubicados en cada *place* o nodo de la RPT) y el intervalo temporal asociado a cada transición activada.

En [25] se demuestra que el problema de la alcanzabilidad de estados en RPT es *indecidible*. Sin embargo, para AT, en [12] se demuestra que es *decidible*, por lo que han surgido nuevos enfoques para la exploración de estados, como el descrito en [26] que comienza a partir de un estado conocido y calcula sucesivamente el estado predecesor.

En [27] los autores demostraron que AT y RPT son equivalentes con respecto a una equivalencia de lenguaje temporizado, y proponen un método que permite realizar una conversión estructural de un modelo AT a un modelo de RPT equivalente.

La teoría de AT provee muchas aplicaciones prácticas que permiten la verificación formal de modelos mediante el uso de fórmulas de lógica temporal, y su uso se ha incrementado gracias a herramientas de software como Kronos [28], WUppal [29] y UPPAAL [16]. Esta última, ampliamente utilizada en proyectos industriales [30]. Por otro lado, en [24] se presenta TINA, una herramienta para la verificación y análisis de modelos de RPT.

A pesar de los avances en herramientas de verificación, aún hay una brecha entre un modelo verificable y el código de programación que lo implementa, el cual es susceptible a errores en la etapa de implementación cuando los requerimientos son transformados a código. A la fecha, de acuerdo con la revisión bibliográfica recopilada, no existen herramientas que permitan hacer una conversión completa de forma automática.

Se han realizado variados esfuerzos para efectuar verificación formal de modelos DEVS (como los revisados en [31]), pero hasta el presente no existen herramientas que permitan evaluar propiedades relevantes (como alcanzabilidad, vivacidad, etc.) directamente sobre ellos. Por este motivo, en este trabajo se utiliza la metodología propuesta en [15] que permite realizar chequeos de modelos DEVS para garantizar su correctitud a través de AT.

**Bisimulación**

A continuación, se determina la equivalencia entre los modelos DEVS y AT. La descripción formal del *bolt Text Sanitization* (usando DEVS) se muestra en la Figura 4. En la Figura 5, se presenta un esquema que representa sus transiciones y estados. Los estados corresponden a los nodos, las transiciones internas se describen mediante líneas segmentadas y las externas por medio de líneas continuas.

De acuerdo con la descripción DEVS del *bolt* (Figura 4), este se puede encontrar en los estados "Disponible" u "Ocupado". Por motivos prácticos, la cantidad de tweets asignados al *bolt* que se encuentran encolados se representa mediante la variable *Q_TS*.

Para determinar la equivalencia entre un modelo DEVS atómico y el correspondiente modelo de AT, se comienza por el análisis de las diferentes transiciones (internas y externas) y por los estados posibles de cada elemento atómico que lo compone. En el caso de este trabajo corresponde a los estados posibles de cada uno de los *bolts* que conforman la aplicación. En las Figuras 5 y 6, se muestra el modelo DEVS atómico del *bolt Text Sanitization* y su equivalente desarrollado utilizando AT.

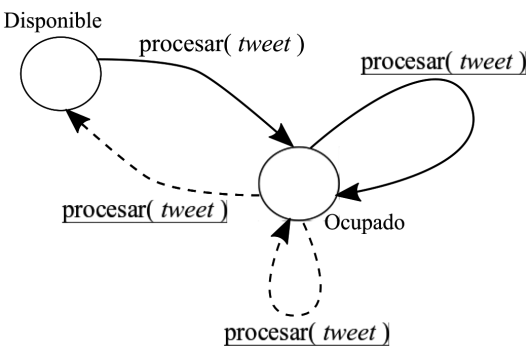


Figura 5. Estados y transiciones (internas y externas) del *bolt Text Sanitization* de acuerdo con su definición en DEVS.

**Transformación de DEVS a AT (transiciones internas)**

Se comprueba la bisimilaridad de sus transiciones internas (incluyendo estados de origen y destino).

A) *($Ocupado_D$, 0* ***R*** *$Ocupado_T$)*: Existen dos transiciones internas, ambas en el estado $Ocupado_D$. Una de ellas es una transición recursiva y la otra apunta hacia el estado , como se aprecia en la Figura 4.

A.1) ***Transición Recursiva:*** En el modelo DEVS (Figura 5) la transición recursiva se representa como procesamiento de un *tweet* (procesar(*tweet*)), y se utiliza para indicar que quedan más *tweets* encolados (por procesar). La transición equivalente en AT (Figura 6) corresponde a la observada en la parte inferior del nodo "ocupado". Esta transición ocurre cuando se finaliza el procesamiento de un *tweet*, razón por la que se descuenta en 1 la cantidad de *tweets* encolados ($Q_{TS} = Q_{TS} - 1$) y el tiempo de procesamiento de un *tweet* cualquiera se restaura a 0 (*elapsedTime* = 0). Si se considera la ejecución de la transición interna DEVS: ($Ocupado_D$, $e$) $\xrightarrow{d}$ ($Ocupado_D$, 0) si $q_{ts} > 0$, como una transición con retardo $d$, donde $0 \leq e < ta$ ($Ocupado_D$) y $0 \leq d <$ ($Ocupado_D$) $- e$, restaurando el tiempo transcurrido a $e = 0$ en el estado de destino. Por lo tanto, para satisfacer la bisimulación se debe cumplir: ($Ocupado_T$, *elapsedTime* = $e$) $\xrightarrow{d}$ ($Ocupado_T$, *elapsedTime* = $e + d$) si $Q\_TS > 0$ para el mismo valor de $d$. Del invariante *elapsedTime* < *hold* en el estado $Ocupado_T$ y sustituyendo *elapsedTime* y $d$, se tiene:

$$ta\ (Ocupado_D) \leq hold \qquad (1)$$

Es decir, mientras la desigualdad sea verdadera, se permanecerá en el estado $Ocupado_D$, y la transición se activará cuando deje de cumplirse y se cumpla además que $Q\_TS > 0$, en dicho caso se restaura el reloj (*elapsedTime* = 0). Cabe señalar que $Q\_TS$ (del modelo AT) y $q_{ts}$ (del modelo DEVS) son equivalentes, por lo que pueden utilizarse indistintamente.

A.2) ***Transición desde Ocupado a Disponible****:* Se chequea la bisimilaridad entre $Ocupado_D$ y $Ocupado_T$ para la transición temporizada desde $Ocupado_D$ hacia $Disponible_D$. Considerando la transición interna DEVS: ($Ocupado_D$, $e$) $\xrightarrow{d}$ ($Disponible_D$, $e$) donde $0 \leq e < ta$ ($Ocupado_D$) y d = ($Ocupado_D$) $- e$, se tiene también que la ejecución de la transición ($Ocupado_T$, *elapsedTime* = $e$) $\xrightarrow{d}$ ($Disponible_T$, *elapsedTime* = $e + d$) del AT.

Para que estas dos transiciones sean equivalentes en términos de comportamiento, se necesita que comiencen en estados bisimilares, y que luego de transcurrido el mismo tiempo alcancen dos estados destino bisimilares. Para ello, se utiliza el mismo valor de retardo $d$ en ambas transiciones lo cual permite deducir el valor de la constante *hold* en la restricción de tiempo de la transición del AT.

La transición del AT comienza en $Ocupado_T$, con un valor de *elapsedTime* igual al tiempo transcurrido e de la transición del modelo DEVS. Luego, después del mismo retardo de la transición DEVS, el AT cambia al estado $Disponible_T$ y el valor de *elapsedTime* se incremente en $d$ unidades de tiempo. De esto se

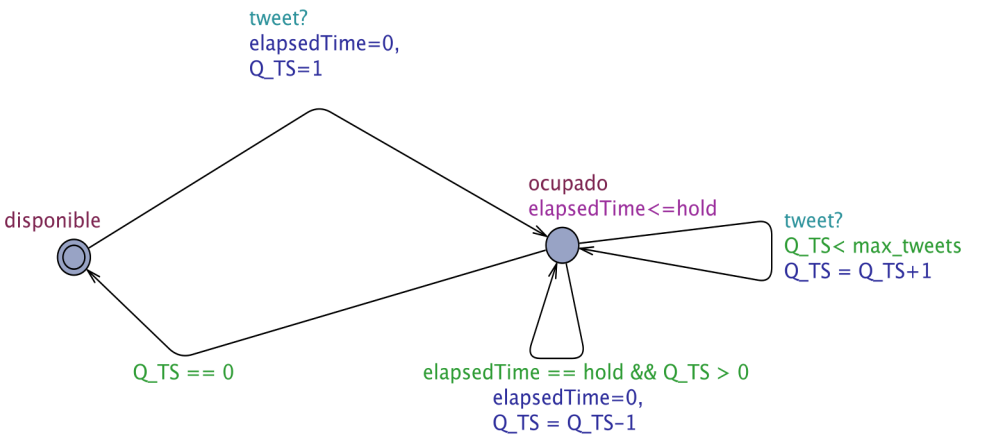


Figura 6. Modelo AT del bolt Text Sanitization.

tiene que *elapsedTime* = *e* + *d*, y con la condición de guarda de la transición del AT (*elapsedTime* = *hold*) se tiene que:

$$ta\ (Ocupado_D) = hold \quad (2)$$

Esto indica que mientras se utilice una constante *hold* en la guarda de la transición del AT con un valor igual o mayor a la función de avance de tiempo de $Ocupado_D$, la transición del AT se ejecutará como la transición DEVS. La condición *Q_TS* = 0 de la transición del AT que lleva desde $Ocupado_R$ hacia $Disponible_T$ es igual a la transición interna DEVS, que finalmente la distingue de la transición recursiva anterior, dado que los estados de origen poseen el mismo invariante (*elapsedTime* ≤ *hold*).

Finalmente, de las ecuaciones (1) y (2) se tiene que: *ta* ($Ocupado_D$) = *hold*, lo que garantiza la relación de bisimulación de las transiciones internas del modelo DEVS con la transición con retardo del modelo AT.

**Transformación de AT a DEVS (transiciones internas)**

Ahora se debe cumplir con la otra dirección de la bisimulación, en donde se transforma el modelo AT (Figura 6) en un modelo DEVS (Figura 5).

B) ($Ocupado_T$ ***R*** ($Ocupado_D$): Existen dos transiciones internas en el AT. Una de ellas es recursiva y la otra apunta hacia el estado $Ocupado_T$. La otra transición tiene como origen el estado $Ocupado_T$ y como destino $Disponible_T$ (Figura 6).

B.1) ***Transición Recursiva:*** Esta corresponde a una transición con retardo en el AT: ($Ocupado_T$, *elapsedTime* = *e*) $\xrightarrow{d}$ ($Ocupado_T$, *elapsedTime* = *e* + *d*). Se necesita que el valor del invariante (*elapsedTime* < *hold*) en $Ocupado_T$ sea verdadero para que el AT se mantenga en dicho estado. Por lo que se tiene que:

$$e + d < hold \quad (3)$$

Ahora, en la transición con retardo *d* ($Ocupado_D$, *e*) $\xrightarrow{d}$ ($Ocupado_D$, *e* + *d*) del modelo DEVS, se tiene que para permanecer en el estado $Ocupado_D$ luego de *d* unidades de tiempo, la suma del tiempo transcurrido y *d* no debe superar ta ($Ocupado_D$). Luego, se obtiene que:

$$e + d < ta\ (Ocupado_D) \quad (4)$$

B.2) ***Transición desde Ocupado a Disponible:*** Corresponde a la transición del AT: ($Ocupado_T$, *elapsedTime* = *e*) $\xrightarrow{d}$ ($Disponible_T$, *elapsedTime* = 0). Esta expresión indica que comienza en el estado $Ocupado_T$ con un tiempo transcurrido *e*. Luego de *d* unidades de tiempo, el AT cambia su estado a $Disponible_T$ y la variable de tiempo *elapsedTime* se fija en cero. Para cambiar de estado, se necesita que el invariante *elapsedTime* < *hold* sea falso, y la condición de guarda de la transición (*elapsedTime* = *hold*) debe ser verdadera, lo que resulta en:

$$e + d = hold \quad (5)$$

Por otro lado, en DEVS esta transición corresponde a ($Ocupado_D$, *e*) $\xrightarrow{d}$ ($Disponible_D$, 0) e indica que para activarla se necesita que el tiempo transcurrido más el retardo sean iguales a la función de avance de tiempo de $Ocupado_D$, por lo tanto, se tiene que:

$$e + d = ta\ (Ocupado_D) \quad (6)$$

De las ecuaciones (5) y (6) se deduce que *hold* = *ta* ($Ocupado_D$), razón por lo que las ecuaciones (3) y (4) resultan ser equivalentes. Entonces, se tiene una relación de bisimulación desde el modelo AT hacia el modelo DEVS. Por lo tanto, la transición interna del modelo DEVS y la transición temporizada del modelo AT son bisimilares respecto al tiempo y equivalentes en comportamiento cuando se tiene una constante *hold* igual al periodo de permanencia correspondiente al estado $Ocupado_D$ en el modelo DEVS. De esto se concluye que ($Disponible_D$ ***R*** $Disponible_T$) y ($Ocupado_D$ ***R*** $Ocupado_T$) mediante la relación de bisimulación ***R***.

**Transiciones externas DEVS**

Se verifica que las transiciones externas del modelo DEVS (con líneas continuas en la Figura 4) sean bisimilares a las del modelo AT equivalente. El modelo DEVS presenta dos transiciones externas – una que va desde $Disponible_D$ hacia $Ocupado_D$ y otra que es recursiva en $Ocupado_D$, las que pueden ser expresadas como: $\delta_{ex_{tts}}$ (“Disponible”, σ, $q_{ts}$) = (“Ocupado”, σ, 1), que indica que estando el *bolt* disponible – sin importar el tiempo transcurrido – al recibir un *tweet*, la variable de estado $q_{ts}$ toma el valor 1 (indicando un tweet encolado), y como: $\delta_{ex_{tts}}$ (“Ocupado”, σ, $q_{ts}$) = (“Ocupado”, σ – *e*, $q_{ts}$ + 1) que indica que mientras se está procesando al menos un *tweet*, y se recibe otro, el nuevo *tweet* será encolado (incrementando el valor de en 1)

y se proseguirá con el procesamiento del *tweet* no finalizado. El tiempo para la activación de la transición interna corresponde al retardo menos el tiempo ya transcurrido. Como ambas transiciones no involucran restricciones temporizadas, pueden ser expresadas como transiciones de la siguiente forma:

1. $Disponible_D \xrightarrow{a} Ocupado_D$
2. $Ocupado_D \xrightarrow{a} Ocupado_D$

Con estas expresiones es posible representar las transiciones externas como transiciones del AT:

1. $(Disponible_T, elapsedTime) \xrightarrow{a} (Ocupado_T, elapsedTime = 0)$ donde se restaura el valor de *elapsedTime* a cero.
2. $(Ocupado_T, elapsedTime) \xrightarrow{a} (Ocupado_T, elapsedTime)$.

Con lo anterior se obtiene una relación ***R*** entre los modelos DEVS y AT que muestra que tanto como $(Disponible_D \; \boldsymbol{R} \; Disponible_T)$ como $(Ocupado_D \; \boldsymbol{R} \; Ocupado_T)$ se cumplen. De forma similar se puede mostrar la relación en el otro sentido (desde AT hacia DEVS), demostrándose que existe una relación de bisimulación entre los modelos DEVS y AT.

**Verificación**

Las propiedades que se analizan sobre el modelo del WSE son:

- Alcanzabilidad (*Reachability*).
- Seguridad (*Safety*).
- Vivacidad (*Liveness*).
- Ausencia de interbloqueos (*Deadlock free*).

Para verificar estas propiedades se utiliza el software UPPALL Model Checker Tool version 4.1.19 (rev. 5648, November 2014) [16]. Estas propiedades se deben verifican en el modelo del bolt Text Sanitization desarrollado con el AT.

**Alcanzabilidad**: Esta propiedad se utiliza para determinar que todos los estados de un modelo son alcanzables a partir de los demás. Para verificar esta propiedad, se aplica una consulta sobre el modelo desarrollado en AT. En el caso del modelo que representa al *bolt* en estudio, las consultas tienen la forma: E<> TS.disponible and E<> TS.ocupado.

Al finalizar la ejecución de estas consultas, UPPAAL entrega la respuesta *propiedad es satisfecha*, lo que significa que la propiedad se satisface y que los estados son alcanzables. En otras palabras, el *bolt* –en algún momento de la simulación– estará disponible y en otros estará ocupado.

**Seguridad**: Esta propiedad se utiliza para verificar que en el *bolt* no ocurren comportamientos anómalos o no deseados. Por esta razón, se definen comportamientos (combinaciones de estados alcanzados y valores de variables de estado) que no debieran ocurrir en el modelo AT del *bolt*.

En el estado *"Disponible"* no debe haber ningún *tweet* destinado al *bolt* en la cola, lo que se traduce en la siguiente consulta: E[] (TS.disponible and not (TS.Q_TS>0).

Al ejecutar esta consulta con la herramienta UPPAAL, en todos los casos se obtiene como respuesta que la *propiedad es satisfecha*, por lo tanto, dicho comportamientos no ocurre en el modelo.

**Vivacidad**: Esta propiedad se utiliza para verificar si algún estado se alcanza eventualmente a partir de otro. Para ello, se define un estado inicial y uno final. Primero se realiza el chequeo considerando como estado inicial a *"Disponible"* y como estado de destino a *"Ocupado"*. Esto se realiza mediante la consulta: TS.disponible --> TS.ocupado.

Luego de ejecutar la consulta con la herramienta UPPAAL, se obtiene el mensaje propiedad no satisfecha. La propiedad de vivacidad no se satisface debido a que cuando el *bolt* se encuentra en el estado inicial *"Disponible"*, no hay condiciones definidas que limiten el tiempo en que se puede permanecer en dicho estado y, por tanto, forzar el cambio al estado *"Ocupado"*. Efectivamente, la única forma en que el *bolt* cambie del estado *"Disponible"* a *"Ocupado"* es cuando se recibe un *tweet*. Es por esta razón que no corresponde la añadidura de un invariante que permita satisfacer la propiedad de vivacidad. En la práctica podría ocurrir que el *bolt* no reciba *tweets* y permanezca indefinidamente en reposo. En otras palabras, el no cumplimiento de esta propiedad indica que el *bolt* podría eventualmente nunca recibir *tweets*, permaneciendo indefinidamente en estado *"Disponible"*. Obviamente, este comportamiento es previsto y es correcto.

A continuación, se verifica la propiedad considerando como estado inicial a *"Ocupado"* y como destino

a *"Disponible"*. Para ello se define la consulta: TS.ocupado --> TS.disponible.

En este caso se obtiene que la propiedad sí se satisface. Esto indica que después de procesar todos los *tweets* encolados, el *bolt* cambiará su estado a *"Disponible"*. El invariante *elapsedTimes* < = *hold* definido para el estado *"Ocupado"* de la Figura 6, representa el hecho de que es necesario que transcurran *hold* unidades de tiempo para terminar de procesar un *tweet*, y es lo que permite que se satisfaga la consulta.

Se concluye que a pesar de que la propiedad no siempre se satisface, es esperable que cuando los nodos (de cualquier servicio) se encuentren disponibles (estado *"Disponible"*) recibirán consultas en algún momento, produciendo un cambio al estado *"Ocupado"*. Por lo tanto, no satisfacer esta propiedad no representa un fallo del *bolt*.

**Presencia de Interbloqueos**: Se analiza la presencia de *deadlocks* en el modelo de AT del *bolt*. Es decir, se comprueba que el *bolt* no permanezca fijo en un estado sin poder cambiar a otro. Esto se verifica mediante la consulta: A[] not deadlock.

La condición *Q_TS < max_tweets* –que limita el tamaño de la cola de *tweets* recibidos– permite satisfacer esta propiedad. De lo contrario, el AT presentaría *deadlocks* ya que la cola podría crecer de forma infinita. Esta condición no es arbitraria, dado que en la práctica ninguna cola (implementada computacionalmente) puede almacenar un número infinito de elementos.

## CONCLUSIONES

La verificación de un modelo de simulación es un proceso que en ocasiones se deja de lado, principalmente debido a su complejidad y tiempo que se debe invertir en él. Por esta razón, es que generalmente sólo se realiza la validación del modelo. Sin embargo, la verificación de un modelo es un modo de asegurar que el comportamiento del modelo de simulación tiene una correspondencia con el comportamiento del sistema en estudio.

En este trabajo se realizó el modelado en formalismo DEVS de una aplicación desarrollada para ser ejecutada en la plataforma Storm. El modelo DEVS fue convertido a un modelo equivalente en AT mediante bisimulación. Posteriormente se realizó la verificación del modelo AT. Si bien algunas propiedades no se cumplen, estas corresponden a estados previstos y se consideran como parte del funcionamiento normal de la aplicación. Por lo que se concluye que el modelo de simulación DEVS posee el mismo comportamiento que el descrito para la aplicación de Storm.

Como trabajo futuro se realizarán benchmarks a la aplicación de Storm verificada, para determinar el ajuste parámetros y efectuar la validación de los resultados obtenidos con el modelo de simulación en DEVS.

## REFERENCIAS


[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal and D. Ryaboy. "Storm@twitter". *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, Snowbird, Utah, USA. 2014.

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

[3] G.A. Wainer. "Discrete-Event Modeling and Simulation: A Practitioner's Approach". 1st ed., Boca Raton, FL: CRC Press, Inc. 2009.

[4] I. Oyelowo, B. Artacho, G.A. Wainer and S. O'Young. "Using Cell-DEVS For Prototyping Unmanned Aircraft System Traffic Simulation". *SpringSim 2018 (TMS'18)*, Baltimore, MD, 2018.

[5] A. Inostrosa-Psijas, G. A. Wainer, V. Gil-Costa and M. Marin. "DEVS modeling of large scale Web Search Engines". *Proceedings of the Winter Simulation Conference 2014*, Savannah, GA, USA. 2014.

[6] B.U. Kazi and G.A. Wainer. "Handover oscillation reduction in ultra-dense heterogeneous cellular networks using enhanced handover approach". *SpringSim 2018 (CNS'18)*, Baltimore, MD. 2018.

[7] B.U. Kazi and G.A. Wainer. "Formal Modeling and Simulation to Analyze the Dynamics of Malware Propagation in Networks Using cell-DEVS". *Proceedings of the 20th Communications & Networking Symposium*, San Diego, CA, USA. 2017.

[8] B.P. Zeigler. "Discrete Event System Specification Framework for Self-Improving

Healthcare Service Systems". *IEEE Systems Journal.* Vol. 12 Nº 1, pp. 196-207. 2018.

[9] A.C.H. Chow and B.P. Zeigler. "Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism". *Proceedings of the 26th Conference on Winter Simulation*, Orlando, Florida, USA. 1994.

[10] G.A. Wainer and N. Giambiasi. "Timed Cell-DEVS: modelling and simulation of cell spaces". *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*, H. S. Sarjoughian y F. Cellier, Edits., New York, NY: Springer New York, 2001, pp. 187-214.

[11] G.A. Wainer. "Improved Cellular Models with Parallel Cell-DEVS". *Transactions of the Society for Computer Simulation International*. 2000.

[12] R. Alur and D.L. Dill. "A Theory of Timed Automata". *Journal Theoretical Computer Science.* Vol. 126 Nº 2, pp. 183-235. 2004.

[13] H. Saadawi and G. Wainer. "Rational Time-advance DEVS (RTA-DEVS)". *Proceedings of the 2010 Spring Simulation Multiconference*, San Diego, CA, USA. 2010.

[14] L. Aceto, A. Ingólfsdóttir, K.G. Larsen and J. Srba. "Reactive Systems: Modelling, Specification and Verification". New York, NY, USA: Cambridge University Press. 2007.

[15] H. Saadawi and G.A. Wainer. "Principles of Discrete Event System Specification Model Verification". *SIMULATION.* Vol. 89, pp. 41-67. 2013.

[16] G. Behrmann, A. David and K.G. Larsen. "A Tutorial on Uppaal". *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT.* Vol. 3185, pp. 200-236. Bertinoro, Italy, Springer. 2004.

[17] A. Esuli and F. Sebastiani. "SENTIWORDNET: A Publicly Available Lexical Resource for Opinion Mining". *In Proceedings of the 5th Conference on Language Resources and Evaluation (LREC'06).* 2006.

[18] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". *SIGOPS Oper. Syst. Rev. V*ol. 44, April 2010.

[19] E.M. Clarke and E.A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". *Logic of Programs, Workshop*, Berlin, Heidelberg. 1981.

[20] J.-P. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". *Proceedings of the 5th Colloquium on International Symposium on Programming*, London, UK. 1982.

[21] C. Baier and J.-P. Katoen. "Principles of Model Checking". Cambridge, Massachusetts: The MIT Press, p. 984. 2008.

[22] P.M. Merlin. "A study of the recoverability of computing systems". Irvine, CA: PhD thesis, University of California. 1974.

[23] C.A. Petri. "Kommunikation mit automaten". Bonn, Germany: Schriflen des Rheinlsch-Westfalischen Institutes für Instrumentelle Mathematik an der Universitat Bonn. 1962.

[24] B. Berthomieu, F. Peres and F. Vernadat. "Time Petri Nets - Analysis Methods and Verification with TINA". *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, John Wiley & Sons, Ltd, pp. 19-49. 2010.

[25] N.D. Jones, L.H. Landweber and Y.E. Lien. "Complexity of some problems in Petri Nets". *Theoretical Computer Science.* Vol. 4 Nº 3, pp. 277-299. 1977.

[26] R. Farkas, T. Tóth, Á. Hajdu y A. Vörös. "Ackward reachability analysis for timed automata with data variables". *18th International Workshop on Automated Verification of Critical Systems*, Oxford, England. 2018.

[27] B. Bérard, F. Cassez, S. Haddad, D. Lime and O.H. Roux. "Comparison of the Expressiveness of Timed Automata and Time Petri Nets". Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 211-225. 2005.

[28] S. Yovine. "Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2)". *International Journal on Software Tools for Technology Transfer.* Vol. 1, pp. 123-133. 1997.

[29] P. Fogh, T. C. Hald and B. Nielsen. "WUppaal: A Web-Service for the Uppaal Model-Checker". *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. 2016.

[30] K.G. Larsen, F. Lorber and B. Nielsen. "20 Years of UPPAAL Enabled Industrial Model-Based Validation and Beyond". *Leveraging Applications of Formal Methods, Verification*

*and Validation. Industrial Practice*, Springer International Publishing, pp. 212-229. 2018.

[31] B.P. Zeigler, J.J. Nutaro and C. Seo. "Combining DEVS and Model-Checking: Concepts and Tools for Integrating Simulation and Analysis". *International Journal of Simulation and Process Modelling (IJSPM).* Vol. 12 Nº 1. 2017.