Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Realizada en el Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas

Tesis Doctoral

Traducción y Validación Sistemática y Automática de Modelos DEVS Abstractos

Diego Ariel Hollmann
Doctorando

Directora: Dra. Claudia Frydman
Co-Director: Dr. Maximiliano Cristiá

Miembros del Jurado:
Jurado A
Jurado B
Jurado C

Tesis presentada en la Facultad de Ciencias Exactas, Ingeniería y Agrimensura en cumplimiento parcial de los requisitos para optar al título de

Doctor en Informática

Rosario, Argentina, Noviembre de 2014

Certifico que el trabajo incluido en esta Tesis es el resultado de tareas de investigación originales y que no ha sido presentado previamente para optar a un título de posgrado en ninguna otra Universidad o Institución.

Diego Ariel Hollmann

Índice de contenidos

Índice de figuras

Índice de tablas

Resumen

Esta tesis presenta dos aportes originales que se enmarcan dentro del modelado y simulación de sistemas, más precisamente, de modelos basados en el formalismo DEVS.

El primer aporte consiste en un lenguaje que permite la descripción de modelos DEVS abstractos (basados en matemática y lógica) y que luego pueden ser traducidos en forma automática a diferentes lenguajes de simulación. Esto permite describir un modelo DEVS sin que sea necesario conocer el lenguaje de entrada de la herramienta de simulación que se desea utilizar. En general, se escribe el modelo directamente utilizando el lenguaje concreto, o la traducción del modelo abstracto al modelo concreto (en el simulador) se realiza manualmente. Ambas alternativas, son propensas a la inserción de errores durante el modelado y requiere que el especialista tenga nociones o conocimientos de programación para implementar sus modelos o necesita de alguien para esto. Entonces, con el lenguaje de modelado propuesto en esta tesis se evita por un lado, la incorporación de errores en la traducción de los modelos y por otro, permite al especialista modelar el sistema sin la necesidad de conocer un lenguaje de programación en particular. Podemos agregar, además, que permite usar varios simuladores para simular el modelo.

La segunda contribución es una nueva metodología para validar modelos DEVS en forma rigurosa y sistemática permitiendo la semi-automatización del proceso de validación. Generalmente los modelos DEVS se validan simulándolos con alguna herramienta de simulación, comparando el comportamiento y los resultados obtenidos con los requerimientos del sistema que está siendo modelado. El número de escenarios de simulación es usualmente infinito, entonces, seleccionar cuáles de estos escenarios simular se convierte en una tarea crucial. El conjunto de escenarios seleccionados debe incluir aquellos que puedan revelar posibles errores en el modelo sin dejar ningún aspecto del mismo sin analizar y no incluir escenarios redundantes convirtiendo a la tarea de validación es un proceso eficiente (hallazgo de errores en función de la cantidad de simulaciones realizadas). Esta selección, actualmente, se realiza siguiendo la experiencia o intuición de un especialista. La metodología presentada en esta tesis se basa en una familia de criterios de simulación que permite seleccionar los escenarios de simulación de modelos DEVS en forma disciplinada, cubriendo las simulaciones más significativas. Esto se obtiene analizando la representación y estructura matemática y lógica del modelo DEVS.

Para automatizar este proceso es necesario que el modelo se encuentre descripto en forma abstracta, posiblemente, utilizando el lenguaje de modelado presentado en esta tesis.

Abstract

This thesis presents two original contributions on the modeling and simulation field, more precisely, of models based on the DEVS formalism.

The first contribution is a language that allows the description of DEVS abstract models (based on mathematics and logic) and later can be translated automatically into different simulation languages. This allows describing a DEVS model without having to know the input language of the target simulation tool. In general, models are written directly in the concrete language or the translation of the abstract model to a concrete model (in the simulator) is done manually. Both alternatives are error prone during the modeling phase and requires the specialist having programming skills or he or she needs someone to translate its models. Then, with the modeling language proposed, the insertion of errors during the modeling phase is avoided and, on the other hand, allows the specialist to describe its models without knowing any particular programming language. Moreover, it allows using several simulation tools to simulate the model.

The second contribution is a new methodology to rigorously and systematically validate DEVS models allowing the semi-automation of this process. In general, DEVS models are validated through simulation, comparing the behavior and the results against the requirements of the system being modeled. The number of simulation scenarios is usually infinite, therefore, selecting which scenarios must be simulated becomes a crucial task. The selected scenarios must include those that can reveal possible errors in the model, leaving no aspect of the model without analyzing and excluding redundant scenarios, turning the validation task in an efficient process (number of errors found over the number of simulations performed). This selection, is currently done by following the experience or intuition of an specialist. The methodology presented in this thesis is based on a family of simulation criteria to guide the selection of simulation scenarios in a disciplined way covering the most significant simulation. This is achieved by analyzing the mathematical and logical structure of the DEVS model. To automate this process, it is necessary that the model be described in its most abstract form, possibly, using the modeling language presented in this thesis.

Capítulo 1

Introducción y Motivaciones

En esta tesis se presenta un lenguaje que permite la descripción abstracta de modelos DEVS y una nueva metodología para validar dichos modelos. El formalismo DEVS es el formalismo más general para describir sistemas de eventos discretos y, en la actualidad, su uso se ha extendido en los ambientes más diversos. DEVS es utilizado tanto por centros de investigación y universidades, por la industria e ingeniería en general e incluso por departamentos de defensa o instituciones militares. Para poder describir el lenguaje de modelado y esta nueva metodología de validación que se presenta, es necesario primero introducir tanto el modelado y simulación de sistemas a eventos discretos como el formalismo DEVS. Este capítulo es, por tanto, una introducción a estos temas y termina con una descripción de las motivaciones y los aportes que esta tesis presenta y cómo se estructura la misma.

1.1. Modelado y Simulación de Sistemas a Eventos Discretos

El desarrollo y uso de modelos de simulación se ha incrementado considerablemente en los últimos años. Frecuentemente son utilizados como la primera representación de sistemas y luego son utilizados para tomar decisiones sobre situaciones críticas. Otra veces son utilizados para realizar experimentos dado que es poco viable o muy costoso experimentar sobre el sistema real.

Este gran crecimiento, ha llevado al desarrollo de sistemas cada vez más complejos, como sistemas de control de tráfico aéreo, sistemas automatizados de manufacturación, sistemas de navegación o pilotos automáticos, control de velocidad crucero, etc. La complejidad de estos sistemas se debe a la variedad de requerimientos de los mismos, incluyendo propiedades sobre funciones, requerimientos de rendimiento, restricciones de tiempo real.

Todos estos sistemas se caracterizan por la ocurrencia de *eventos discretos* que suce-

den en forma asincrónica. Estos sistemas se denominan *Sistemas de Eventos Discretos* (DES). En esta clase de sistemas, se asume que un número finito de eventos puede ocurrir en un intervalo de tiempo finito.

Existen varios formalismos para representar estos sistemas, i.e. Redes de Petri, Statecharts, Timed Automata, Máquinas de Estados Finitos, Cadenas de Markov. A finales de los años 70, abordando la problemática de la representación y simulación de DES, Bernard Zeigler propuso una teoría para el modelado y simulación de DES creando el formalismo DEVS [1] el cual permite modelar (y la posterior simulación) la dinámica de estos sistemas. DEVS está basado en la teoría general de sistemas y es el formalismo más general para describir sistemas de eventos discretos. Todo modelo representado por alguno de los formalismos mencionados anteriormente, puede ser representado utilizando DEVS [2].

La generalidad de DEVS se deriva del hecho de que permite el modelado de sistemas con un conjunto infinito de posibles estados; donde el nuevo estado, después de un evento de llegada, puede depender del tiempo (continuo) transcurrido en el estado anterior [3]. Por otro lado, DEVS es un formalismo independiente de toda implementación.

1.2. Descripción del Formalismo DEVS

Existen dos clases de modelos, modelos *atómicos* y modelos *acoplados*. Todo el trabajo de esta tesis involucra solamente modelos atómicos. De hecho, un modelo acoplado es equivalente a un (tal vez complicado) modelo atómico [1]. Un modelo DEVS atómico está definido por la estructura:

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

donde:

- $X = \{(p, v) \mid p \in InPorts, v \in X_p\}$ es el conjunto de puertos de entrada y valores, donde *InPorts* representa el conjunto de puertos de entrada y X_p, el conjunto de valores para los puertos de entrada;
- $Y = \{(p, v) \mid p \in OutPorts, v \in Y_p\}$ es el conjunto de puertos de salida y valores, donde *OutPorts* representa el conjunto de puertos de salida e Y_p, el conjunto de valores para los puertos de salida;
- S es el conjunto de valores del estado,
- δ_{int}: $S \to S$ es la *Función de Transición Interna* (*Internal Transition Function*),
- δ_{ext}: $Q \times X \to S$ es la *Función de Transición Externa* (*External Transition Function*), donde:

$Q = \{(s, e), s \in S, 0 \leq e \leq ta(s)\}$ es el conjunto de estados totales, y

e es el *tiempo transcurrido* (*Elapsed Time*) desde la última transición;

- $\lambda : S \to Y$ es la función de salida; y
- $ta : S \to \mathbb{R}^{+}_{0,\infty}$ es la *Función de Avance de Tiempo* (*Time Advance Function*).

Las funciones δ_{int}, δ_{ext} y ta definen la dinámica del sistema. En un momento dado, el sistema se encuentra en un estado s. Si no ocurre ningún evento externo el sistema permanecerá en dicho estado el tiempo determinado por $ta(s)$. Cuando $ta(s)$ es 0, ocurre una transición interna. En ese caso, el sistema produce una salida $y \in Y$ determinado por la función de salida λ ($y = \lambda(s)$). Luego, el sistema asume un nuevo estado, s' determinado por $\delta_{int}(s)$ ($s' = \delta_{int}(s)$). Si en cambio $ta(s) = \infty$, se dice que el sistema está en un estado pasivo y permanecerá en el mismo hasta que un evento externo ocurra.

Cuando llega un evento de entrada, $x \in X$, el sistema cambia de estado inmediatamente y se dice que el sistema ejecuta una transición externa. El nuevo estado s' no depende, en este caso, solamente del estado previo, sino también del tiempo transcurrido desde la última transición, e, y del valor del evento de entrada, $s' = \delta_{ext}(s, e, x)$. En este caso, no se produce ningún evento de salida.

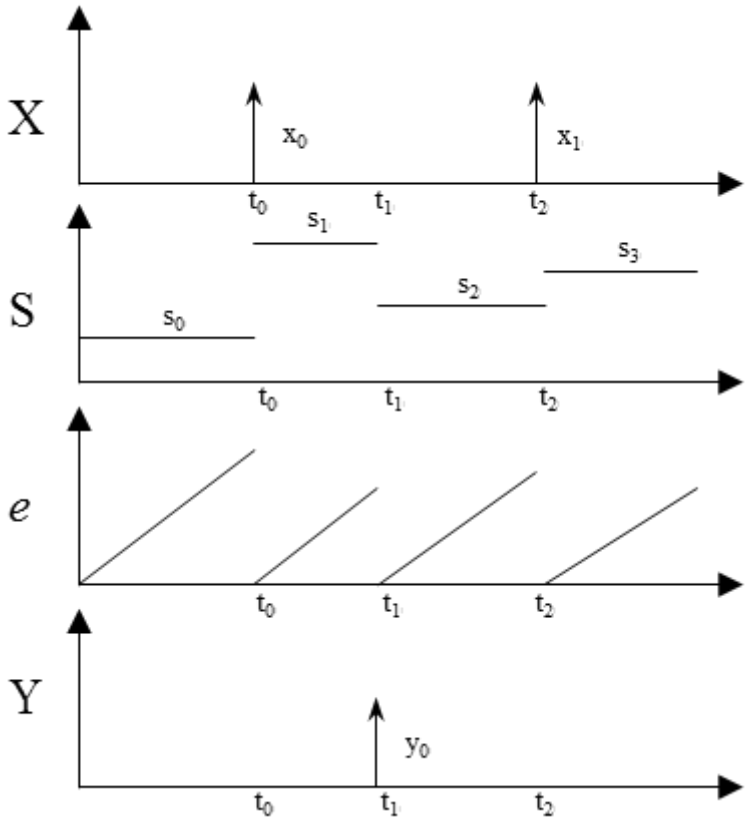

Figura 1.1: Trayectorias de un modelo DEVS

En la figura 1.1 vemos en un pequeño ejemplo la trayectoria de un modelo DEVS. Inicialmente, el sistema se encuentra en el estado s_0. En el instante t_0, y antes de que se cumpla el tiempo de avance asociado a s_0, $ta(s_0)$, i.e. $t_0 < ta(s_0)$, se produce un evento de entrada, x_0. En ese instante, el sistema asume un nuevo estado determinado por la función de transición externa, $s_1 = \delta_{ext}(s_0, e, x_0)$. El tiempo transcurrido, e, en este caso es igual a t_0. Luego, al transcurrir $ta(s_1)$ unidades de tiempo ($t_1 = t_0 + ta(s_1)$), y en la ausencia de eventos de entrada, el sistema emite un evento de salida, $y_0 = \lambda(s_1)$, y realiza una transición interna asumiendo el nuevo estado $s_2 = \delta_{int}(s_1)$. Más tarde, en el instante de tiempo t_2 ($t_2 < ta(s_2) + t_1$) se produce un nuevo evento de entrada, x_1, haciendo que el sistema realice una nueva transición interna y asuma un nuevo estado, $s_3 = \delta_{ext}(s_2, e, x_1)$, con $e = t_2 - t_1$.

1.2.1. Ejemplos de modelos DEVS

Presentamos a continuación algunos simples ejemplos para entender mejor el comportamiento de los modelos DEVS y cómo se describen los mismos. Los ejemplos están basado en los del libro de Zeigler et al [1].

Generador

El primer ejemplo, es el modelo muy simple de un generador, M_{gen}, que no recibe ningún evento de entrada y emite, periódicamente, un 1. Al no recibir ningún evento de entrada, la función de transición externa, δ_{ext} no se describe. El período cada cual se emite el evento de salida es de T_{gen} unidades de tiempo.

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

- $X = \{\}$
- $Y = \{(out, 1)\}$
- $S = \mathbb{R}_0^+$
- $\delta_{int}(\sigma) = T_{gen}$
- $\lambda(s) = (out, 1)$
- $ta(\sigma) = \sigma$

El conjunto de puertos y valores de entrada es vacío, ya que no recibe ningún evento y el de puertos y valores de salida está formado sólo por el par $(out, 1)$, es decir, un solo puerto de salida, out, y un solo posible valor para emitir por dicho puerto, 1. El estado está formado por un solo elemento, que representa el tiempo que falta para emitir el próximo evento. Entonces, en cada transición interna, luego de emitir el evento, se vuelve a configurar el estado en T_{gen}.

Procesador

En este ejemplo, el modelo representa un procesador que recibe trabajos y tarda un tiempo en procesarlos. El valor que recibe en el evento de entrada representa el tiempo que se tarda en procesar el trabajo correspondiente. Si el procesador está ocupado procesando un trabajo y llega uno nuevo, este último es ignorado. Una vez que el trabajo es procesado, el sistema emite un evento de salida con el tiempo que tardó en procesarlo.

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

- $X = \{(in, x) : x \in \mathbb{R}^+\}$
- $Y = \{(out, y) : y \in \mathbb{R}^+\}$
- $S = \{libre, ocupado\} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \mathbb{R}^+$
- $\delta_{int}((fase, \sigma, trabajo)) = (libre, \infty, trabajo)$
- $\delta_{ext}((fase, \sigma, trabajo), e, (in, x)) = \begin{cases} (ocupado, x, x) & \text{si } fase = libre \\ (fase, \sigma - e, trabajo) & \text{caso contrario} \end{cases}$
- $\lambda((fase, \sigma, trabajo)) = (out, trabajo)$
- $ta((fase, \sigma, trabajo)) = \sigma$

El conjunto de puertos y valores para los eventos de entrada, está formado por un solo puerto, *in* y por los reales positivos que representan el tiempo de procesamiento del trabajo que debe ser procesado. Lo mismo ocurre con los eventos de salida, en el puerto *out*. El estado está conformado por tres variables[1]. La primera representa el estado del procesador (*ocupado*, *libre*); la segunda, el tiempo de procesamiento que queda (∞ en caso de que no haya ningún trabajo siendo procesado); y la tercera, el tiempo total de procesamiento del trabajo actual. Cuando llega un trabajo, si está en la fase *libre*, se configura el tiempo restante de procesamiento con el valor de la entrada y se cambia de fase. En cambio, si la fase es *ocupado*, directamente se ignora el trabajo que llega y solo se actualiza el tiempo transcurrido. Cuando se termina el tiempo de procesamiento, se emite el evento con este tiempo como valor por el puerto de salida. En la transición interna correspondiente, se cambia de fase a *libre* y se cambia el tiempo a ∞, en cuyo caso, no habrá ninguna nueva transición a menos que llegue un nuevo trabajo.

[1] O conformado por una variable que tiene tres componentes

Cola de Almacenamiento

La cola consiste en un lista que almacena identificadores de trabajos (números reales positivos), por orden de llegada. Cuando recibe una señal, en lugar de un trabajo, la cola transmite el primer trabajo de la lista (el primero que llegó), si es que hay alguno, y lo saca de la lista.

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

- $X = \{(in, x) : x \in \mathbb{R}^+ \cup \{emitir\}\}$
- $Y = \{(out, y) : y \in \mathbb{R}^+ \cup \{\emptyset\}\}$
- $S = (\mathsf{List}\ \mathbb{R}^+) \times (\mathbb{R}_0^+ \cup \{\infty\})$
- $\delta_{int}((xs, \sigma)) = \begin{cases} (\text{tail } xs, \infty) & \text{si } xs \neq \emptyset \\ (xs, \infty) & \text{si } xs = \emptyset \end{cases}$
- $\delta_{ext}((xs, \sigma), e, (in, x)) = \begin{cases} (xs \frown x, \infty) & \text{si } x \in \mathbb{R}^+ \\ (xs, 0) & \text{si } x = emitir \end{cases}$
- $\lambda((xs, \sigma)) = \begin{cases} (out, \text{head } xs) & \text{si } xs \neq \emptyset \\ (out, \emptyset) & \text{si } xs = \emptyset \end{cases}$
- $ta((xs, \sigma)) = \sigma$

El estado, entonces, es una tupla formada por una secuencia de número reales (el identificador de los trabajos) y un número real positivo o el símbolo ∞. El primer elemento de la tupla representa la lista de los trabajos en la cola y el segundo se utiliza para el comportamiento interno del modelo. El valor 0 indica que se debe emitir un trabajo (si hay) y ∞ indica que hay que esperar a que llegue un evento. El conjunto de valores del puerto de entrada *in* está formado por reales positivos y la señal *emitir*, y el conjunto de valores del puerto de salida *out*, por reales positivos y el símbolo $\emptyset$.

Cuando llega un evento, si es un número real positivo ($x \in \mathbb{R}^+$), i.e. el identificador de un trabajo para ser encolado, se lo agrega a la lista. Si en cambio el valor que arriba es la señal *emitir*, se setea el valor de la variable que controla el funcionamiento interno en 0, lo que generará una transición interna debido a que *ta* usa este valor.

En la transición interna, si la lista de trabajos no está vacía, se emite por el puerto *out* el primer elemento de la lista, y se lo quita de la misma. Si cambio está vacía, se emite el valor $\emptyset$ indicando este hecho.

Extensiones y Variantes del Formalismo DEVS

En las últimas décadas, numerosas variantes o extensiones del formalismo DEVS han sido desarrolladas, generalmente, pensadas para modelar con mayor facilidad o precisión sistemas de un dominio específico, como por ejemplo, sistemas de tiempo real o sistemas paralelos. Algunas de estas extensiones son: P-DEVS (Parallel DEVS) para el modelado y simulación de sistemas concurrentes y/o sistemas distribuidos; RT-DEVS (Real Time DEVS), pensada para modelar y simular sistemas de tiempo real; Cell-DEVS, para modelar sistemas que pueden ser representados como *cell spaces* (espacios de celdas, o espacios celulares); STDEVS (Stochastic DEVS), para modelar y simular modelos DEVS no deterministas; y VECDEVS (Vectorial DEVS), que proporciona una manera simple de representar sistemas de gran escala.

1.3. Simuladores DEVS

Como se menciona en la introducción de este capítulo, el uso del formalismo DEVS se ha extendido notablemente en comunidades muy diversas. Esto ha llevado al desarrollo de varias herramientas de modelado y simulación (M&S) cada una de ellas con sus propias características y lenguaje de modelado. Algunas nacieron basadas en DEVS *puro* y otras como implementaciones de alguna extensión o variante del mismo. Entre los muchos simuladores existentes podemos mencionar: DEVS-C++ [4], DEVSim++ [5], PowerDEVS [6], CD++ [7, 8], DEVS-Suite [9] y JDEVS [10]. En esta sección sólo presentamos dos, PowerDEVS y DEVS-Suite, ya que son los que fueron utilizados en esta tesis.

1.3.1. PowerDEVS

PowerDEVS [6] es una herramienta para modelar y simular sistemas híbridos, i.e. sistemas que combinan dinámicas discretas y continuas, donde estas últimas deben ser discretizadas de algún modo para poder ser simulados. PowerDEVS implementa los métodos de discretización *Quantized State Syetem* (QSS) [11].

Los modelos pueden ser definidos en C++ y luego acoplarse gráficamente en bloques jerárquicos para crear modelos más complejos. PowerDEVS traduce automáticamente los modelos acoplados gráficamente a código C++ que es ejecutado por el simulador.

PowerDEVS permite la posibilidad de ejecutar simulaciones bajo un sistema operativo de tiempo real llamado RTAI. Además, puede interconectarse con herramientas como Scilab, pudiendo PowerDEVS hacer uso de las variables y funciones de Scilab para el posterior procesamiento y análisis de datos.

1.3.2. DEVS-Suite

DEVSJAVA [12], que fue desarrollado por Zeigler, es un entorno para el modelado y simulación de modelos basados en DEVS y Parallel DEVS. Esta basado en Java y soporta la visualizaión jerárquica de modelos y la animación del intercambio de mensajes entre componente atómicos y acoplados.

Más tarde, DEVSJAVA junto con DEVS Tracking Environment (DTE) [13] fueron utilizados para el desarrollo de DEVS-Suite [9], una herramienta más completa que permite la visualización *on-the-flay* the los datos de simulación. Los datos generados por el modelo pueden ser recolectados en forma dinámica y mostrados como trayectorias basadas en el tiempo.

1.4. Motivaciones y Contribuciones de esta Tesis

Las motivaciones de de esta tesis, a pesar de estar conectadas entre ellas, pueden separarse en dos.

Si bien varias de las herramientas de modelado y simulación, como las presentadas anteriormente, están basadas en el mismo lenguaje de programación, C++ o Java la mayoría, no comparten el lenguaje de modelado, teniendo cada una su forma particular de describir un modelo, o componentes del mismo. Esto hace que no sean compatibles unas con otras o no puedan interactuar entre ellas. Un sistema modelado en, por ejemplo, DEVSim++ no puede ser simulado utilizando DEVS-Suite, al menos no sin un arduo y tedioso trabajo de adaptación.

Además, un especialista que desea modelar y/o simular un sistema con alguna herramienta de M&S en particular, necesita conocer las características y particularidades del lenguaje de modelado de dicha herramienta.

Entonces, la primera contribución de esta tesis consiste en un lenguaje que permite la descripción de modelos DEVS abstractos, independiente de cualquier plataforma o implementación, basado en expresiones lógicas y matemáticas. Los modelos descriptos utilizando dicho lenguaje, luego pueden ser traducidos (*compilados*) en forma automática al lenguaje de modelado de diferentes herramientas de M&S para que puedan ser simulados. El objetivo es, por tanto, que el especialista pueda diseñar sus modelos expresándolos en su forma más abstracta, como se mostró en los ejemplos de la sección 1.2.1, sin necesidad de tener conocimientos o habilidades de programación. Luego puede traducirlo, en forma automática, y simularlo con la herramienta que desee. Esto permite, entonces, que ingenieros o especialistas no relacionados con la informática puedan modelar y simular sus sistemas.

Por otra parte, junto con el gran crecimiento en el uso de modelos de simulación, se ha hecho cada vez más necesaria la definición de técnicas rigurosas para asegurar que

estos modelos representan lo mejor posible el sistema real que se está modelando. En otras palabras, la Verificación y Validación (V&V) de los modelos de simulación se ha convertido en una tarea crucial.

Una técnica para validar modelos DEVS, es simularlos varias veces, bajo diferentes condiciones y escenarios y comparar los resultados de dichas simulaciones con los requerimientos del sistema para determinar su exactitud.

Inspirados en técnicas bien conocidas y aceptadas en el ámbito de la Ingeniería de Software, más precisamente en el Testing Basado en Modelos; como segundo aporte, presentamos una familia de criterios para guiar las simulaciones en forma rigurosa y disciplinada cubriendo las simulaciones más significativas o importantes. Esta técnica permite, además, la semi-automatización del proceso de validación. Para lograrlo, es necesario que el modelo esté descripto en forma abstracta y para esto, se podría utilizar el lenguaje de modelado enunciado anteriormente.

El resto de la tesis se estructura de la siguiente manera. En el capítulo que sigue, se resumen los trabajos relacionados al modelado utilizando el formalismo DEVS y se presenta el lenguaje de modelado abstracto. Además, se describen las reglas para traducir esto modelos abstractos a modelos concretos de diferentes herramientas de M&S. En el Capítulo 3 se hace un resumen de la verificación y validación de modelos, se describen los trabajos relacionados y se presenta la técnica de validación propuesta. En el Capítulo 4 se muestran los aportes de la tesis a través de dos casos de estudio. Finalmente, en el Capítulo 5 se analizan las conclusiones generales de la tesis y se detalla el trabajo a futuro que de la misma se desprende.

Capítulo 2

Especificación Abstracta de Modelos DEVS

2.1. Introducción

Cómo se mencionó en el Capítulo 1, el gran crecimiento del uso de modelos de simulación llevó al desarrollo de numerosas herramientas de M&S basadas en el formalismo DEVS o alguna de sus extensiones, como por ejemplo, DEVS-C++ [4], DEVSim++ [5], CD++ [14], PowerDEVS [15], JDEVS [10], DEVS-Suite [9]). Cada una de estas herramientas posee su propio lenguaje de entrada, basado en algún lenguaje de programación, utiliza sentencias en algún lenguaje de programación específico o, simplemente, es un lenguaje de programación de propósito general como C++ o Java. Esto hace que quien desee utilizar dichas herramientas para modelar y simular un sistema requiera conocimientos, no necesariamente triviales, de programación o de alguien con dichos conocimientos para que implemente sus modelos.

Más aún, estos lenguajes son diferentes unos de otros, lo que dificulta la interoperabilidad entre herramientas de simulación, así como también la cooperación entre diferentes comunidades de M&S. Un modelo de simulación descripto utilizando, por ejemplo PowerDEVS, no puede ser simulado en JDEVS. Por otro lado, el modelo descrito en PowerDEVS modela, esencialmente, el mismo sistema que sería modelado en JDEV.

En función de lo antes comentado, sería deseable poder describir el modelo abstracto como es pensado o visto por el especialista y luego ser traducido en forma automática a uno o más lenguajes de modelado de algunas de las herramientas de M&S.

Entonces, esta primera contribución consiste en un lenguaje formal, CML-DEVS (por su siglas en ingles, Conceptual Modeling Language for DEVS), para la descripción abstracta de modelos DEVS en término de expresiones lógicas y matemáticas sin involucrar conceptos o nociones de programación. De este modo los ingenieros o espe-

cialistas pueden usar CML-DEVS para escribir modelos DEVS utilizando matemática *cotidiana*, como fue propuesto originariamente por Zeigler a la hora de definir el formalismo. Los modelos CML-DEVS pueden luego ser analizados y traducidos, en forma automática, a los lenguajes de entrada de diferentes herramientas de M&S para poder ser simulados con éstos. Esta es la misma idea que ocurre en la comunidad de métodos formales (cf. Z [16], B [17], Alloy [18]), dónde especificaciones formales son escritas con matemática y lógica clásica, de modo que estas especificaciones pueda proporcionarse tal cual están a herramientas tales como chequeadores de tipos, probadores de teoremas, generadores de código, etc.

CML-DEVS se describe dando su sintaxis y semántica. La sintaxis, se describe a través de su EBNF (Extendend Backus Naur Form, Forma Norma de Backus Extendida) y por medio de ejemplos, mientras que la semántica es dada en términos del código que debe ser generado en el lenguaje de entrada de dos herramientas, PowerDEVS y DEVS-Suite. En este sentido, se puede construir un compilador multi-objetivo (multi-target compiler) de CML-DEVS para producir modelos de PowerDEVS y DEVS-Suite a partir de modelos DEVS abstractos o matemáticos. Para utilizarlo con otros lenguajes objetivo, sólo bastaría dar las reglas de traducción a esos nuevos lenguajes y programar la parte del compilador correspondiente. La elección de estos dos simuladores es relativamente arbitraria. Teniendo, éstos, diferentes lenguajes de entrada (C++ y Java) la intención es mostrar como puede traducirse en forma automática un modelo CML-DEVS al lenguaje de diferentes simuladores.

En la Figura 2.1 se esboza el proceso de modelado utilizando CML-DEVS. A partir de los requerimientos, el especialista describe el modelo abstracto o conceptual con CML-DEVS. Luego, un compilador multi-target se encarga de traducir, en forma automática, el modelo abstracto en modelos concretos de diferentes lenguajes de herramientas de M&S.

A pesar que definir un estándar para describir modelos DEVS es un tarea más que necesaria a esta altura, y de hecho ya se está intentando llevar a cabo, no es la intención de este trabajo proveer dicho estándar, sino mostrar que es posible escribir un modelo DEVS en su forma más abstracta, y luego traducirlo en forma automática a la herramienta de simulación que se desee. La definición de un estándar involucra gran interacción, desarrollo y esfuerzo entre la industria y las comunidades de investigación que trabajan con DEVS.

El resto de este capítulo está organizado como sigue. En la siguiente sección hacemos una revisión sobre otros enfoques y trabajos que atacan el problema de la descripción de modelos DEVS. En la sección 2.3 presentamos CML-DEVS y mostramos su propósito y utilidad través de algunos ejemplos. En la Sección 2.5 explicamos como un modelo CML-DEVS es traducido a modelos DEVS-Suite y PowerDEVS dando las reglas de traducción. Finalmente, en la Sección 2.6 algunas conclusiones y otras consideraciones

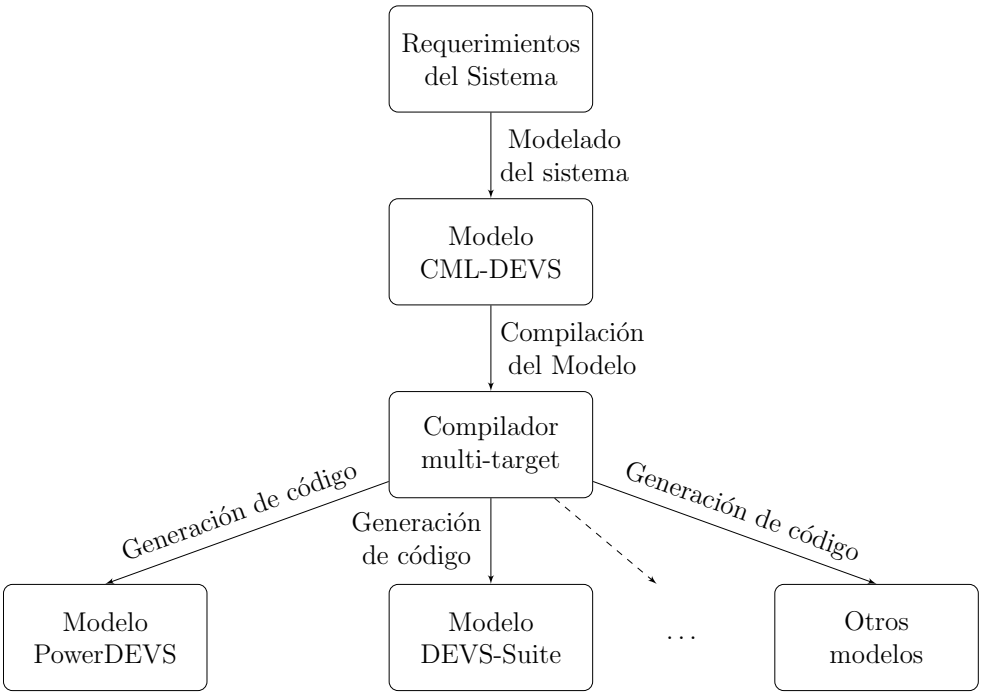

Figura 2.1: Modelado de un sistema con CML-DEVS

son discutidas. En el Capítulo 4 mostramos con dos casos de estudio, un poco más complejos que los ejemplos de este capítulo, como se describen modelos DEVS abstractos utilizando CML-DEVS.

2.2. Trabajos Relacionados y Otros Enfoques

Siendo DEVS un formalismo tan utilizado y ya que existen tantas herramientas para modelar y simular modelos DEVS, se ha empleado mucho esfuerzo en desarrollar un lenguaje general para describir dichos modelos. Incluso, se ha creado un grupo internacional, compuesto por investigadores de varias universidades y centros de investigación al rededor del mundo, abocado a la definición de un estándar para la representación procesable por computadoras del formalismo DEVS [19].

Este grupo de estandarización ha identificado cuatro áreas del entorno DEVS que necesitan estandarización [20, 21]:

- El formalismo DEVS en sí mismo, y sus variantes,
- La representación de modelos, que debe describir la estructura del modelo y la dinámica de forma independiente de la plataforma o herramienta,
- Los requerimientos mínimos para que un simulador soporte DEVS,
- Librerías de modelos, destinados a proporcionar una colección de modelos listos para usar.

Con CML-DEVS intentamos atacar el segundo punto, representando de forma abstracta los modelos DEVS de manera independiente de toda plataforma y lenguaje de programación.

A nuestro criterio, vemos CML-DEVS como una versión extendida o mejorada de DEVSpecL, un lenguaje de especificación para modelos DEVS desarrollado por Hong y Kim [22]. CML-DEVS tiene el objetivo de preservar el concepto abstracto del formalismo DEVS e incorpora nuevas nociones matemáticas abstractas para acercar DEVSpecL al DEVS original; como por ejemplo, una teoría de conjuntos útil para dicha abstracción. Como Hong y Kim mencionan, DEVSpecL soporta solo características básicas y para ser más general, soporta APIs definidas por el usuario. El modelador debe definir estas APIs en el lenguaje de programación del entorno en el que el modelo es ejecutado. Además, en cuanto a los *tipos compuestos* que soporta, solo incluyen secuencias (y en realidad vistas como pilas). Estas últimas consideraciones limitan el poder de abstracción del modelo. Además, sigue exigiendo que el modelador tenga que "programar" para hacer algo más o menos complejo. CML-DEVS está más inspirado en las notaciones formales de ingeniería de software agregando teorías matemáticas útiles para modelar sistemas.

Creemos que los modelos definidos con CML-DEVS son más "DEVS puros", i.e. preservando el formato definido por Zeigler. Por ejemplo, con DEVSpecL el modelador debe definir *tipos de mensajes* e *interfaces* para representar los eventos de entrada/salida, los cuales pueden contener código escrito en algún lenguaje de programación de propósito general. Con CML-DEVS intentamos mantener los eventos de entrada/salida en forma abstracta definiendo solamente el tipo de los valores de los puertos.

Por otro lado, el artículo dónde presentan DEVSpecL, carece de datos suficientes en algunos aspectos. Por ejemplo, las transiciones internas y externas consisten en secuencias de **expr**, como muestran por medio de la BNF, pero no se da ningún detalle sobre dichas expresiones más allá de algunos ejemplos. Además, en la función de transición externa no se hace referencia al tiempo transcurrido desde la última transición, e.

Finalmente, en esta tesis se presentan las reglas de traducción que permiten implementar un compilador multi-objetivo para los lenguajes de entrada de dos simuladores. Esto no parece haber sido presentado por Hong y Kim.

Mitaal and Douglass [23] presentan otro lenguaje de dominio específico para Parallel DEVS como un componente de la versión revisada del entorno DEVSML (DEVSML 2.0), basado en Finite Deterministic DEVS (FDDEVS), DEVS Finitos Deterministas. También tiene la intención de ser utilizado (entre otras cosas) como una representación abstracta de modelos DEVS. Utilizando el entorno Xtext y la EBNF de la gramática de su lenguaje, integraron a Eclipse un editor de DEVSML para definir de manera simple modelos con algunas características elegantes y útiles como asistente de código y validación del modelo (en cuanto a sintaxis y semántica) en tiempo de ejecución.

Sin embargo, la gramática de DEVSML tiene algunas diferencias y limitaciones con respecto a Parallel DEVS. Por ejemplo, los valores de entrada/salida son definidos como *entidades de mensajes*. El tiempo transcurrido, e, en la transición externa es omitido y reemplazado por las sentencias **continue** y **reschedule**. Finalmente, FDDEVS es un subconjunto, restringido y menos expresivo que el formalismo DEVS original.

Ambos trabajos antes mencionados, permiten la generación automática de código, de modo de obtener código DEVS ejecutable, en diferentes implementaciones DEVS como DEVSJAVA, DEVSim++ y DEVSim-Java.

En varios trabajos [21, 24–26] se propone a XML como lenguaje para describir modelos DEVS. Según Touraille et al. [21] parece ser una buena elección, ya que existen numerosas herramientas para trabajar con XML y es independiente de la plataforma, entre otras ventajas. Sin embargo ésta tampoco sería precisamente una representación abstracta de un modelo DEVS. Tal vez, XML sí sea una opción adecuada para una representación intermedia, i.e. entre la descripción abstracta del modelo y su representación en alguna herramienta de simulación.

En un trabajo reciente, enmarcado en su tesis doctoral, L. Touraille [27] desarrolló un entorno que apunta a modelar sistemas con DEVS. El núcleo de este entorno es un meta-modelo DEVS. El mismo está virtualmente separado en dos partes, una para especificar la parte "estática" del modelo, i.e. estados, entradas y salidas; y otra para definir el comportamiento, es decir, transiciones internas y externas, avance del tiempo y función de salida. Esta última, se basa en un lenguaje semi-genérico definido por él, el cual dista de ser una representación abstracta de dichas funciones de DEVS. Basado en el meta-modelo realiza traducciones hacía varias plataformas DEVS, permitiendo la interoperabilidad entre varias herramientas de simulación. Esta transformación, no se hace a partir de la descripción abstracta de un modelo DEVS sino, precisamente, a partir de este meta-modelo.

En general, existen muchos lenguajes que permiten definir modelos DEVS. Sin embargo, hasta donde sabemos, no hay ninguno que tome como punto de partida la descripción abstracta (basado en matemática y lógica) de un modelo DEVS. Todos ellos, describen el modelo utilizando directamente el lenguaje de una herramienta de simulación, construido sobre otro de propósito general, como C++ o Java; o un meta-lenguaje que luego se traduce a diferentes idiomas de simulación.

2.3. Definiendo modelos DEVS abstractos con CML-DEVS

Como mencionamos en la introducción de este capítulo, la contribución que presentamos es un lenguaje que permite describir en forma abstracta modelos DEVS, de

modo muy similar a como se describieron los ejemplos de la Sección 1.2.1, sin involucrar nociones de programación, i.e. estructuras de control, manejo de memoria, estructuras de datos.

Si un especialista desea simular con alguna herramienta de modelado y simulación existente uno de esos modelos, incluso uno tan simple, esto involucraría tareas de programación, situación que no ocurre al modelarlo con CML-DEVS. Por ejemplo, en el modelo de la cola de procesamiento, se debe tomar una decisión sobre la representación de X, Y, y S, en cuanto a cómo representar el símbolo $\emptyset$ y la señal *emitir*. Tal vez, en este caso particular, con una variable de tipo `float` o `double` de algún lenguaje alcanzaría para representar X e Y usando, por ejemplo, un valor negativo para representar *emitir* y $\emptyset$ respectivamente. Sin embargo, en un caso ligeramente diferente, por caso $\mathbb{R} \cup \{\emptyset\}$, esta representación no alcanzaría y una distinta debería definirse. Además, es necesario algún tipo de estructura de datos para manejar los elementos de la cola. Nuevamente, utilizando CML-DEVS, el modelador no necesita lidiar con estos asuntos, puede expresar el modelo directamente en su forma abstracta.

En la Figura 2.2 vemos cómo podría describirse el modelo de la cola de procesamiento en CML-DEVS. En este ejemplo, podemos ver, a grandes rasgos, cómo es la estructura de un modelo CML-DEVS. El mismo consiste en varias subestructuras o componentes, cada una de ellas, representando un componente de un modelo DEVS atómico. Cada componente, está enmarcada por *indicadores* o *palabras claves*, como X is ... end X para el caso de los puertos de entrada. Las variables de estado y puertos de entrada y salida se definen de la misma forma, cada uno dentro de las estructuras correspondientes. Esto se hace dándole un nombre y el tipo pertinente (por ejemplo, $s : \mathbb{R} \cup \{\emptyset\}$). Las funciones de transición, salida y avance de tiempo, también tienen estructuras similares entre sí. El cuerpo de estas funciones se compone de diferentes tipos de sentencias. En general, estas sentencias son asignaciones para modificar los valores del estado o para emitir un evento por un puerto de salida. Como puede verse en este ejemplo, hay sentencias un poco más complejas, como la definición por casos (defcases ... end defcases), en las funciones de transición y de salida. Las definiciones por casos tienen una estructura muy similar al modelo abstracto del ejemplo, donde el resultado de la función (el nuevo estado o el valor de salida respectivamente) están ligados a una condición.

En las secciones que siguen explicaremos con más detalle la sintaxis del lenguaje. Además, la EBNF de CMD-DEVS se muestra en el Apéndice A, siendo esta la forma más común para mostrar la gramática de un lenguaje formal.

```
atomic cola is < X, Y, S, δint, δext, λ, ta > where
X is
      in : ℝ ∪ {emitir};
end X
Y is
      out : ℝ ∪ {∅};
end Y
S is
      s : List ℝ × Time;
end S
δint((xs, σ)) is
      defcases
      case s = (tail xs, ∞); if (xs ≠ {})
      case s = (xs, ∞); if (xs = {})
      end defcases
end δint
δext((xs, σ), e, (in, x)) is
      defcases
      case s = (xs ⌢ x, ∞); if (x ∈ ℝ)
      case s = (xs, 0); if (x = emitir)
      end defcases
end δext
λ((xs, σ)) is
      defcases
      case out = head xs; if (xs ≠ {})
      case out = ∅; if (x = {})
      end defcases
end λ
ta((xs, σ)) is
      σ;
end ta
end atomic
```

Figura 2.2: Modelo de la Cola de Procesamiento en CML-DEVS

2.3.1. Estructura de un Modelo Atómico

Un modelo DEVS atómico en CML-DEVS consiste en varias subestructuras o componentes, cada una de ellas representando un componente de un modelo DEVS atómico. El orden en el que se declaran estas estructuras es indistinto, y tampoco es necesario u obligatorio que se declaren en caso de que no sean usadas. Es decir, si un modelo no recibe eventos, puede omitirse la declaración de X y de δext. Si estas no se definen, tampoco deben declararse en el vector que define el modelo: $< \mathsf{X}, \mathsf{Y}, \mathsf{S}, \delta\mathsf{int}, \delta\mathsf{ext}, \lambda, \mathsf{ta} >$. Es decir, todas las componentes que allí se declaren, deben luego estar definidas.

Además de las estructuras que se pueden observar en la Figura 2.2, del ejemplo anterior, existen dos más que son opcionales. Las mismas no forman parte del formalismo DEVS original pero son utilizadas en casi todas las implementaciones de modelos de simulación. Estas permiten la definición de *parámetros* o *valores constantes* del modelo y *funciones auxiliares*, que simplifican la descripción y mantenimiento de los modelos.

Como en todo lenguaje, existen palabras claves o reservadas, en el ejemplo de la

cola pueden distinguirse de variables o valores porque aparecen en otro formato de texto, como por ejemplo atomic.

2.3.2. Definiciones

La definición de estados y de puertos de entrada y de salida comparten la misma sintaxis, como se ve en la Figura 2.2 del ejemplo de la cola. En la Figura 2.3 se muestran otras posibles definiciones de variables del estado y puertos de entrada y/o salida, dependiendo de la estructura dentro de la cual se declaran. Si se define más de una variable dentro del estado, entonces, el conjunto de posibles valores del estado del modelo es la tupla formada por los conjuntos a los que pertenece cada una de estas variables. Por ejemplo, si las variables de un modelo en particular son *jobs* y σ de la Figura 2.3, el conjunto de posibles valores del estado de dicho modelo sería $S = (\mathsf{List}\ \mathbb{R}) \times (\mathbb{R}_0^+ \cup \{\infty\})$.

$jobs$: List $\mathbb{R}$;
σ : Time;
in : $\mathbb{R} \cup \{\mathsf{signal}\}$;
out : $\mathbb{R} \cup \{\emptyset\}$;
$elevator$: $\{up, down, stopped\}$;
$sensor$: Boolean;
$name$: Text;
$years$: $\mathbb{P}\ \mathbb{N}$;
$pair$: $\mathbb{Z} \times \mathbb{R}$;

Figura 2.3: CML-DEVS - Ejemplos de variables de estado y puertos de entrada y salida

Existe, en CML-DEVS, una variable particular que es σ. Si bien puede ser utilizada como una variable más del estado, su principal uso está en la función de avance de tiempo, *ta*, ya que el valor de σ será el valor de retorno de dicha función. Más adelante se detallará esto.

En el caso de los eventos de entrada y salida, cada definición equivale a un puerto (de entrada o salida, según corresponda) y el tipo o conjunto de los valores asociados a dicho puerto. En forma análoga al estado, los conjuntos puertos de entrada y de salida, y sus valores asociados, quedan conformados por las tuplas de los conjuntos definidos.

CML-DEVS provee varios *tipos básicos* para definir variables y permite la construcción de nuevos *tipos complejos* como tuplas, uniones, conjuntos y listas; como puede verse en la Figura 2.3. Los tipos básicos son $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, Time (equivalente a $\mathbb{R}_0^+ \cup \{\infty\}$), Text, Boolean y tipos enumerados, expresados entre llaves, { }.

$\mathbb{P}$, List, $\cup$ y $\times$ son los constructores para los tipos complejos. $\mathbb{P}$ *Type* representa un conjunto de elementos del tipo *Type* (sin ningún orden y sin elementos repetidos). List

Type, por su parte, es una lista ordenada de elementos del tipo *Type* (las listas pueden contener elementos repetidos). $\cup$ y $\times$ se utilizan para construir, respectivamente, uniones y tuplas de elementos de tipos posiblemente diferentes (de hecho, uniones de elementos del mismo tipo no tendría sentido).

Además, CML-DEVS permite la definición de *sinónimos*. Estos se pueden utilizar para simplificar o facilitar la declaración de variables y puertos, o incluso la declaración de otros sinónimos. Si un tipo complejo es utilizado varias veces, el modelador puede darle un nombre a este tipo complejo con algún sinónimo y luego usar este último en la declaración. Por ejemplo:

$var1 : \mathbb{N} \times \mathbb{R}$;
$var2 : (\mathbb{N} \times \mathbb{R}) \times \mathsf{Boolean}$;
$var3 : \mathsf{List}\ (\mathbb{N} \times \mathbb{R}) \cup \{\emptyset\}$;

Puede ser reemplazado por:

$NRPair == \mathbb{N} \times \mathbb{R}$;
$var1 : NRPair$;
$var2 : NRPair \times \mathsf{Boolean}$;
$var3 : \mathsf{List}\ NRPair \cup \{\emptyset\}$;

2.3.3. Funciones de Transición, Salida y Avance de Tiempo

La descripción de las funciones de transición, así como también las de salida y avance de tiempo, tienen la misma estructura. Un panorama general de estas se mostró en el ejemplo de la Cola de Procesamiento. Estas funciones consisten en un marco, por ejemplo $\delta\mathsf{int}$ `is` ... `end` $\delta\mathsf{int}$, para el caso de la transición interna, con parámetros opcionales. Estos parámetros son, justamente, los parámetros de dichas funciones, i.e. (s, e, x) para el caso de δ_{ext} y (s) para δ_{int}, λ y ta.

Si los parámetros no están presentes, las variables declaradas en el estado son usadas en forma implícita como parámetros, junto con las variables reservadas de CML-DEVS `e`, `value` y `port`, representando el tiempo transcurrido desde la última transición, el valor del evento de entrada y el puerto de dicho evento, respectivamente.

Si los parámetros son declarados en forma explícita, esto debe hacerse en el orden en el que las variables del estado fueron declaradas. Sin embargo, los nombres utilizados como parámetros no necesariamente deben ser los mismos. Esto se debe a que, por ejemplo el especialista podría definir el estado como una tupla, pero, en la definición de la transición interna desea utilizar cada componente del par como una variable individual, dándole un nombre a cada componente dentro de los parámetros de la

función:

S *is*
    $var : \mathbb{R} \times \mathbb{R}$;
end S
⋮
δint$((var1, var2))$ *is*
    $var1 = \ldots$
    $var2 = \ldots$
end δint

donde $var1$ ($var2$) corresponde a la primera (segunda) componente de var

Este es también el caso del ejemplo de la cola de procesamiento, donde el estado está definido por la variable s: List $\mathbb{R} \times$ Time; y luego, tanto en las funciones de transición como en la de salida y la de avance de tiempo, en lugar de usar s como parámetro se utiliza el par (xs, σ).

Los otros dos parámetros, en caso de que se definan en forma explícita, pueden también tener nombre arbitrarios. Obviamente, el modelador puede hacer uso de estas variables directamente si así lo desea.

El cuerpo de cada una de estas funciones se compone de *sentencias*. Hay cuatro tipos de sentencias en CML-DEVS. La más simple de ellas es la *asignación*, utilizada principalmente para actualizar el valor de las variables del estado luego de una transición, como $xs =$ tail xs; o emitir un valor por un puerto de salida, $out =$ head xs. El lado izquierdo de una asignación siempre es el nombre de una variable o un puerto. En cambio, el lado derecho puede ser un simple valor, el nombre de una variable, una operación o una expresión compuesta como tuplas, conjuntos o listas. Algunos ejemplos de asignaciones se pueden observar en la figura 2.4.

$engine = stopped$;
$years = \{1974, 1988, 1990, 1991, 1992, 2004, 2013\}$;
$pair = (-14, 3.1416)$;
$val =$ sin(45);
$jobs = \langle 2.34, 5.98, 6.83 \rangle$;
$elem =$ head xs;
$\sigma = \sigma$ - e;
$out = \emptyset$;
$listA = listB$;

Figura 2.4: CML-DEVS - Ejemplo de asignaciones

En cuanto a los operadores matemáticos, CML-DEVS soporta una amplia variedad, incluyendo operaciones sobre listas y conjuntos, funciones matemáticas y trigonométri-

cas y la aplicación de funciones definidas por el modelador usando CML-DEVS. Los operadores y funciones matemáticas soportadas se listan en la Tabla 2.1.

$+$	$-$	$*$	$\setminus$
$\cup$	$\cap$	$\#$	$\frown$
sin	cos	tan	arcsin
arccos	arctan	log	sign
min	max	sqrt	rev
head	last	tail	front

Tabla 2.1: CML-DEVS - Operadores y funciones matemáticas

El segundo tipo de sentencia de CML-DEVS apunta a modelar expresiones como:

$$\forall\, x \in X \bullet Y = Y \cup \{x * 2\}$$

Estas sentencias se expresan utilizando el comando foreach. La expresión anterior puede describirse en CML-DEVS como se muestra en la Figura 2.5. Se asume que el tipo de x es *Type* y el de X e Y es $\mathbb{P}$ *Type*

```
foreach x in X
    X = (X \ {x}) ∪ {x * 2};
end foreach
```

Figura 2.5: CML-DEVS - Ejemplo de una sentencia *for each*

Estas sentencias consisten en la declaración foreach seguido de un identificador, la palabra reservada in, una expresión o variable, seguido por un conjunto de sentencias y finaliza con end foreach. El tipo de la expresión o variable debe ser List *Type* o $\mathbb{P}$ *Type*.

El tercer tipo de sentencia de CML-DEVS es una estructura que permite la definición de una función por casos. Esta es una forma ampliamente utilizada y una manera muy útil de definir funciones de transición, de salida y de avance de tiempo. En la Figura 2.2 de la cola de procesamiento ya se ha utilizado.

CML-DEVS ofrece dos alternativas para definir funciones de esta forma. Una es más similar a las definiciones matemáticas mostradas en los ejemplos, es decir, dando primero la expresión y luego la condición. La otra es más parecida a la usada en lógica: *if condición then expresión* (*condición* $\Rightarrow$ *expresión*).

Para la primera forma, cada caso se define dentro de una estructura case *sentencias* if *condición* end case. En el caso del segundo estilo, cada caso tiene la estructura: if *condición* $\Rightarrow$ *sentencias*. En los casos de estudio del Capítulo 4 se muestran ambos estilos.

Existe un comando opcional, default, que se puede utilizar para el caso de que ninguna de las condiciones se satisfaga. Este va seguido sólo de las *sentencias*, sin el comando if ni *condiciones*.

Para ambos estilos, el conjunto total de casos se declara dentro del marco:

```
defcases
...
end defcases
```

Las sentencias definen el resultado de la función en caso de se satisfaga la *condición*. Las condiciones van encerradas entre paréntesis, (), y consiste en un *operador relacional* entre dos *operandos*. Los operandos pueden ser nombres de variables, valores u operaciones y los operadores de comparación soportados se listan en la Tabla 2.2. Como se puede notar en el ejemplo de la cola, el operador = está sobrecargado, i.e. es utilizado como un operador de asignación y de comparación. Las condiciones pueden negarse o combinarse utilizando los operadores de conjunción y disyunción lógica, $\neg$, $\wedge$ y $\vee$ respectivamente. La Figura 2.6 muestra algunas condiciones de ejemplo.

$<$	$>$	$\leq$	$\geq$
$=$	$\neq$	$\in$	$\notin$
$\subset$	$\subseteq$		

Tabla 2.2: CML-DEVS - Operadores de comparación

$\neg\ (engine = stopped)$
$(years \neq \{\})$
$(pair.1 \leq 13)$
$((5.63 \in jobs) \wedge (4.13 \notin jobs))$
$((\{1974, 1990\} \subseteq years) \vee (\#years = 1))$
$(\mathsf{value} = emitir)$

Figura 2.6: CML-DEVS - Ejemplos de condiciones

Otra forma usual de definir las funciones de un modelo es utilizando variables o expresiones auxiliares. Por ejemplo:

$\delta_{ext}((xs, \sigma), e, (port, value)) = (ys, \infty)$
where:
$ys = xs \frown value$

El cuarto tipo de sentencia de CML-DEVS intenta, justamente, proveer una forma de definir funciones de este tipo. En la Figura 2.7 se describe el código en CML-DEVS del

ejemplo anterior. La estructura se enmarca por los comandos defwere y end defwhere y consiste en sentencias y definiciones. Las definiciones y las sentencias que van luego del comando where hacen referencia a las variables y operaciones auxiliares.

$\delta\mathsf{ext}((xs, \sigma), \mathsf{e}, (\mathsf{port}, \mathsf{value}))$ is
defwhere
    $xs = ys$;
where
    ys : List$\mathbb{R}$;
    $ys = xs \frown \mathsf{value}$;
end defwhere

Figura 2.7: CML-DEVS - Ejemplo de una sentencia *where*

Notar que las las estructuras *for each*, *where* y las definiciones por casos pueden combinarse una con otra siendo que las tres son consideradas sentencias.

Una observación final sobre las funciones de transición es que si una de las variables de estado no cambia su valor no es necesario que se declare esta situación en forma explícita, por ejemplo $xs = xs$, siendo esta sentencia opcional. Por lo tanto, se asume que las variables que no aparecen del lado izquierdo de ninguna asignación mantienen su valor luego de la transición.

En cuanto a la función de salida, si bien la estructura es igual a la de las funciones de transición, existe una restricción, ésta es, no puede haber ninguna asignación sobre las variables de estado. Es decir, no se puede modificar el valor de dichas variables (no se puede modificar el estado en la función de salida). Entonces, las únicas asignaciones permitidas son aquellas donde la variable del lado izquierdo sea uno de los puertos de salida declarados. En caso de ser necesario, se puede declarar variables auxiliares y modificar su valor (utilizando la estructura defwhere). Entonces, cada asignación sobre un puerto de salida, representa justamente, la producción de un evento de salida por dicho puerto.

Finalmente, en la función de avance de tiempo, *ta*, al igual que en la función de salida no se puede modificar el valor de las variables del estado. Nuevamente, la estructura es similar a las demás funciones, salvo por una particularidad. La función de avance de tiempo retorna siempre el valor de la variable σ. Entonces, existen dos opciones:

(a) la variable σ forma parte del estado del modelo, y, al no poder modificar su valor en esta función, simplemente se retorna su valor:

    ta(. . .) is
        σ;
    end ta

(b) la variable σ no ha sido utilizada por el modelador como variable de estado del modelo, y por consiguiente hay que asignarle un valor dentro de ta:

ta(...) is
...
$\sigma = expresión$;
end ta

El primer caso fue utilizado en el ejemplo de la cola de procesamiento, donde σ forma parte del estado del modelo. El segundo caso, es utilizado en ambos casos de estudio del Capítulo 4.

2.3.4. Parámetros del modelo

Los parámetros, si bien no forman parte del formalismo original DEVS, son utilizados por casi todas las herramientas de M&S y facilitan la descripción y el mantenimiento de los modelos. Si el modelador desea utilizar parámetros para su modelo, debe indicarlo con la sentencia (params) luego del nombre del modelo, y definir, posteriormente, dichos parámetros dentro de la estructura:

params is
$param1 = val1$;
$param2 = val2$;
...
end params

Los parámetros, no se definen como las demás variables del modelo, sino que, al declararse ya con un valor fijo (que no se modifica en todo el modelo) el tipo del parámetro se deriva de dicho valor. Es decir, el tipo queda implícito. Estos parámetros pueden verse como constantes del modelo. Por cuestiones obvias, dichos parámetros o constantes no pueden tener el mismo nombre que ninguna variable del modelo.

2.3.5. Funciones definidas por el modelador

La definición de funciones auxiliares por parte del modelador tampoco forma parte del formalismo definido por Zeigler, pero es muy común y útil definir funciones auxiliares para describir el comportamiento de una parte del modelo o realizar algún tipo de operación. CML-DEVS provee una estructura básica para definir este tipo de funciones

auxiliares. Las mismas se definen todas dentro de la estructura:

functions *LibraryName* is
    ...
end functions

Las funciones que se definen allí, forman una *librería*, y cuando el modelador desea utilizar una función de dicha librería lo hace de la siguiente forma:

LibraryName.funcName(...);

Cabe aclarar que esta librería, puede ser utilizada para definir cualquier modelo que requiera alguna de las funciones allí ya definidas. Es decir, la definición de estas librerías es independiente del modelo.

Las funciones auxiliares se describen usando casi la misma gramática que el resto de las funciones del modelo. La primera línea del cuerpo de una función define su tipo, i.e. el tipo de los parámetros y el tipo del valor de retorno de la función. Esto es seguido, posiblemente, por la definición de otras variables y luego una o más sentencias como las descriptas anteriormente.

Veamos esto con un pequeño ejemplo. Consideremos una función *filtro* que toma dos parámetros, el primero, un conjunto de números enteros y el segundo, un número entero. La función debe devolver un conjunto formado por todos los elementos del primer parámetro menores que el segundo parámetro. La Figura 2.8 muestra una posible forma de describir dicha función en CML-DEVS.

function *filter* is
    $S : \mathbb{P}\ \mathbb{N}, n : \mathbb{N} \rightarrow res : \mathbb{P}\ \mathbb{N}$;
    foreach x in S
        defcases
        if $(x < n) \Rightarrow res = res \cup \{x\}$;
        end defcases
    end foreach
end function

Figura 2.8: CML-DEVS - Ejemplo de una función definida por el modelador

2.3.6. Valores iniciales de las variables de estado para la simulación

La última estructura que nos queda por describir, igual que las dos anteriores no es parte del formalismo DEVS original, y más aún, no forma parte del modelado del sistema sino más bien es un aspecto que concierne a la fase de simulación. Esto es, designar valores iniciales para las variables de estado del modelo. Creemos, sin embargo, que es muy útil proveer esta alternativa en CML-DEVS, de modo que no sea necesario analizar o modificar el código ya traducido al lenguaje del simulador para poder configurar estos valores iniciales. En el próximo capítulo veremos que este tema puntual será de gran ayuda durante el proceso de validación del modelo.

La estructura es muy simple:

simulate *ModelName* from
    asignaciones
end simulate

dónde las *asignaciones* tienen la misma forma que las ya descriptas anteriormente y se utilizan, justamente, para asignarle los valores a las variables del estado del sistema.

2.4. Renderización de un modelo CML-DEVS

La sintaxis de CML-DEVS está pensada, como se puede notar, para que el modelo resultante sea lo más parecido a como el especialista lo describiría "con lápiz y papel", de manera similar a los ejemplos que mostramos en el Capítulo 1. Más aún, un modelo CML-DEVS es fácilmente renderizable a un modelo que quede exactamente igual dichos ejemplos. Es decir, el modelo de la Figura 2.2 puede renderizarse fácilmente para que quede igual al ejemplo de la cola de almacenamiento de la Sección 1.2.1. Esto se podría hacer, por ejemplo, utilizando un lenguaje como LaTeX [28], definiendo en dicho lenguaje las macros y comandos necesarias.

2.5. Traducción de modelos CML-DEVS a modelos concretos

En esta sección describiremos cómo los modelos CML-DEVS se traducen a modelos concretos, i.e. modelos escritos con el lenguaje de herramientas de M&S. Esta tarea es responsabilidad de un compilador multi-objetivo, es decir, un compilador que genera código en diferentes lenguajes. El mismo genera, a través de un analizador sintáctico, lo que se conoce como un árbol de sintaxis abstracta (AST). En este se encuentra

toda la información del modelo en forma estructurada y es utilizado por un analizador semántico, que se encarga de determinar que el modelo "tenga sentido", como por ejemplo, que no se quiera comparar un conjunto con un texto (chequeo de tipos). Luego, una vez que el modelo es semánticamente verificado, y por medio de generadores de código "especializados", cada uno para un lenguaje destino diferente, se genera el código de los modelos en el lenguaje destino. Si se desea generar código para el lenguaje de una herramienta de M&S nueva, solo se necesita agregar un nuevo generador de código, que tome el AST y genere el código correspondiente. Un esquema de este compilador multi-objetivo se esboza en la Figura 2.9

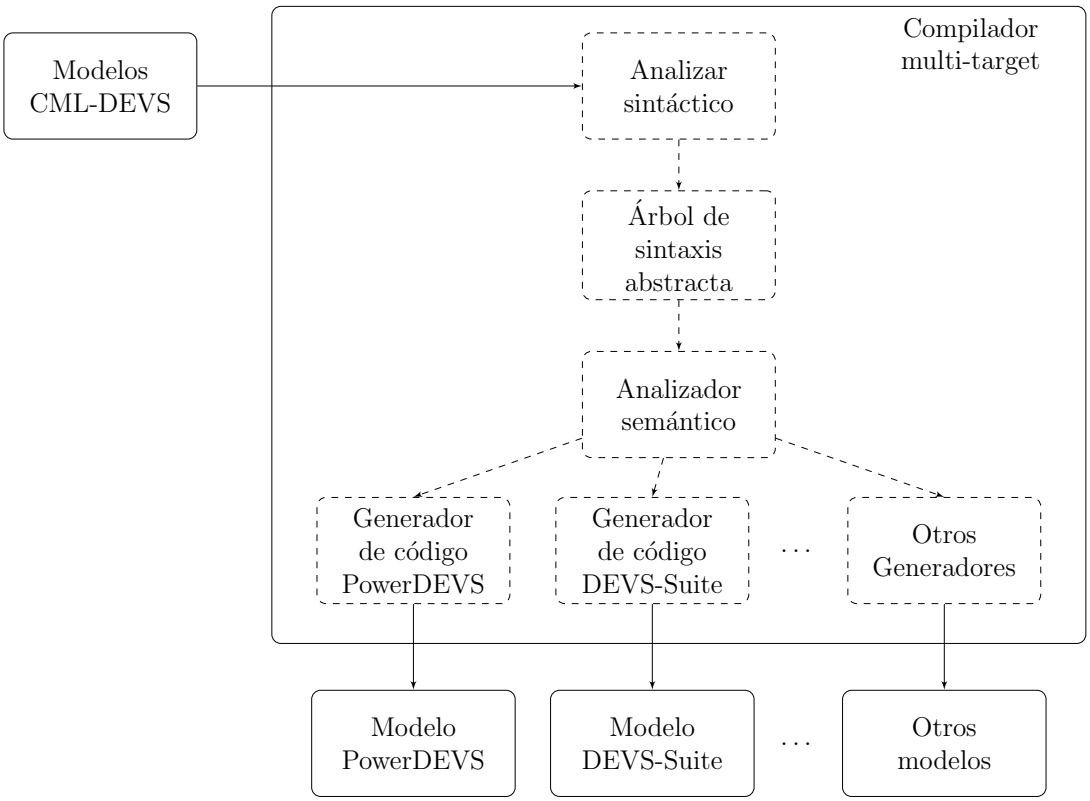

Figura 2.9: Traducción de modelos CML-DEVS a diferentes lenguajes de simulación

2.5.1. Pre-procesamiento

Antes de comenzar con la traducción propiamente dicha del modelo CML-DEVS, debe realizarse un pre-procesamiento para que sea posible dicha traducción. En este pre-procesamiento se "normaliza" la definición del modelo según dos requisitos. El primero hace referencia a la normalización en la definición de los tipos *Unión* y el segundo, a la definición de tipos complejos en general.

Normalización de los tipo *Unión*

Al ser tan flexible la definición de variables utilizando uniones entre conjuntos de diferentes tipos, el tipo resultante puede incluir en forma redundante algunos tipos. Por ejemplo, por algún motivo el modelador podría definir algo como sigue:

$$Type1 == \mathbb{N} \cup \{\emptyset\};$$
$$Type2 == \mathbb{Z} \cup \{\infty\};$$
$$var : Type1 \cup Type2;$$

dónde el tipo de la variable *var*, en definitiva, es $\mathbb{Z} \cup \{\emptyset, \infty\}$ (dado que $\mathbb{N} \subseteq \mathbb{Z}$). Sin embargo, como se verá más adelante en las reglas de traducción, no son analizados esos casos particulares. Las variables de tipo Unión, se representan con estructuras (clases, en los lenguajes orientados a objetos) relativamente complejas utilizando una variable por cada tipo de la unión. Al traducir una definición como la anterior generaría variables innecesarias y dificultaría o produciría resultados tal vez incorrectos en las asignaciones o comparaciones que utilicen variables de este tipo.

Es por esto que, antes de la traducción se deben normalizar todos los tipos definidos mediante uniones. Luego de esta fase de pre-procesamiento, un tipo unión $A_1 \cup \cdots \cup A_N$, es normalizado de la siguiente forma equivalente: $B_1 \cup \cdots \cup B_M$, dónde $M \leq N$ y:

- $A_i = \{x_1^{A_i}, \ldots, x_{N_{A_i}}^{A_i}\} \wedge A_j = \{x_1^{A_j}, \ldots, x_{N_{A_j}}^{A_j}\} \Rightarrow$
  $\exists\, k : B_k = \{x_1^{A_i}, \ldots, x_{N_{A_i}}^{A_i}, x_1^{A_j}, \ldots, x_{N_{A_j}}^{A_j}\}$
- $A_i = \mathbb{N} \wedge A_j = \mathbb{Z} \Rightarrow \exists\, k : B_k = \mathbb{Z} \wedge \nexists l : B_l = \mathbb{N}$
- $A_i = \mathbb{N} \wedge A_j = \mathbb{R} \Rightarrow \exists\, k : B_k = \mathbb{R} \wedge \nexists l : B_l = \mathbb{N}$
- $A_i = \mathbb{Z} \wedge A_j = \mathbb{R} \Rightarrow \exists\, k : B_k = \mathbb{R} \wedge \nexists l : B_l = \mathbb{Z}$
- $k \neq l \Rightarrow B_k \neq B_l$

con $i, j \in 1 \ldots N$ y $k, l \in 1 \ldots M$

Lo anterior significa que, en una unión normalizada, hay (cómo máximo) sólo un tipo enumerado, conteniendo todos los elementos de los tipos enumerados de la unión no normalizada y solo queda el tipo matemático "más grande", $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$.

Por otro lado, luego de esta normalización, no importa el orden de los tipos en una unión. El tipo $B_k \cup B_l$ es el mismo que $B_l \cup B_k$ y por tanto, sólo uno permanece.

Redefinición y normalización de tipos complejos

La otra fase de pre-procesamiento se la conoce como *aplanamiento* de las definiciones de tipo. En la misma, se le asigna un nombre (sinónimo) a cada unión o tupla que

forme parte de un tipo mayor. Luego, la unión o el tipo es reemplazado por el sinónimo asignado. Veamos esta situación con un ejemplo. La expresión:

$$varName : \mathbb{P}((\mathbb{N} \cup \{\emptyset\}) \times (\mathbb{R} \cup \{\emptyset\}));$$

se transforma en:

$$varNameType1 == (\mathbb{N} \cup \{\emptyset\});$$
$$varNameType2 == (\mathbb{R} \cup \{\emptyset\});$$
$$varNameType3 == varNameType1 \times varNameType2;$$
$$varName : \mathbb{P}\ varNameType3;$$

Notar que la redefinición es "de adentro hacia afuera", i.e. las tuplas y uniones internas se redefinen primero y luego las exteriores. Otra observación es que si $Expr_i = Expr_j$ con $i \neq j$, luego $SynonName_i = SynonName_j$. En otras palabras, a expresiones de tipo equivalentes se les asigna el mismo sinónimo.

Entonces, una vez terminado este proceso de redefinición, cada definición de variable tiene la siguiente forma:

$$varName : Type;$$

dónde *Type* es uno de los siguientes:

- $(BT \mid SN)$
- $\mathsf{List}(BT \mid SN)$
- $\mathbb{P}(BT \mid SN)$
- $((BT \mid SN) \mid (\mathsf{List}(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN))) \times \cdots \times ((BT \mid SN) \mid (\mathsf{List}(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN)))$
- $((BT \mid SN) \mid (\mathsf{List}(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN))) \cup \cdots \cup ((BT \mid SN) \mid (\mathsf{List}(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN)))$

con $BT = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \texttt{Text} \mid \texttt{Bool} \mid \texttt{Time}$ y SN es el nombre que se le ha asignado a algún sinónimo.

Por otro lado, cada definición de sinónimo es de la siguiente forma:

$$SynonName == Type;$$

dónde *Type* es también de la forma mencionada anteriormente.

2.5.2. Reglas de traducción

En esta sección mostramos y comentamos, a través de algunos ejemplos, cómo transformar un modelo abstracto escrito en CML-DEVS en modelos implementados en DEVS-Java y en PowerDEVS. En el Anexo B se detalla en forma completa las reglas de traducción. Como puede notarse en dicho anexo, muchas de las reglas se definen en forma recursiva, o utilizan otras reglas, a su vez, para llevar a cabo las traducciones.

Estructura del modelo

Lo primero que hay que hacer para poder simular un modelo, tanto en DEVS-Suite como en PowerDEVS, es generar los archivos necesarios, con la estructura y el código que cada herramienta requiere. Para el caso de DEVS-Suite, se necesita un archivo "ModelName.java" incluyendo en él toda la información del modelo. Este archivo consiste en una clase Java que representa el modelo en sí. Las variables de estado son declaradas como variables de la clase, junto con las funciones de transición, de salida y de avance de tiempo, como métodos de la clase. En el constructor de la clase se instancian las variables que sean necesarias y se declaran los puertos de entrada y de salida. En caso de que corresponda, DEVS-Suite prevé un método, en las clases que representan modelos, para inicializar las variables del mismo.

En el caso de PowerDEVS, son necesarios, al menos, tres archivos. Uno con la estructura del modelo (con la ruta a la implementación del modelo, parámetros y conexiones de puertos), "ModelName.pds". Los otros dos son archivos C++, la cabecera, "ModelName.h", y el código fuente, "ModelName.cpp". En la cabecera se definen las variables de estado y se declaran las funciones. Las definiciones de estas funciones van en el archivo del código fuente. Si además, el especialista quiere simular el modelo utilizando la GUI de PowerDEVS, se necesita un archivo más, "ModelName.pdm", incluyendo la información gráfica (tamaño, colores, posición, etc).

Definiciones

Las definiciones se utilizan, principalmente, para definir variables del estado. Pero también, como mencionamos antes, CML-DEVS permite definiciones de variables en otras partes del modelo, e.g. en las sentencias *where* y en las funciones definidas por el usuario.

Los tipos básicos, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, `Time`, `Text` y `Boolean`, son bastante simple de traducir ya que para cada uno de estos existe un tipo o clase en Java y C++ al cual puede ser traducido. En cuando a los tipos enumerados $\{enum_1, \ldots, enum_n\}$, en lugar de usar el tipo `enum` de Java y C++ decidimos usar `String` y `std::string` respectivamente, para facilitar la inclusión de éstos en tipos complejos. Los conjuntos y listas también son

relativamente simple de traducir, aprovechando las plantillas (templates) ya definidos por ambos lenguajes. La Tabla 2.3 muestra estas traducciones.

CML-DEVS	**DEVS-Suite**	**PowerDEVS**
varName: $\mathbb{N}$;	`Integer varName;`	`unsigned int varName;`
varName: $\mathbb{Z}$;	`Integer varName;`	`int varName;`
varName: $\mathbb{R}$;	`Double varName;`	`double varName;`
varName: Time;	`Double varName;`	`unsigned double varName;`
varName: Boolean;	`Boolean varName;`	`bool varName;`
varName: Text;	`String varName;`	`std::string varName;`
varName: {...};	`String varName;`	`std:string varName;`
varName: Set Type;	`Set<Type>varName;`	`std::set<Type>varName;`
varName: List Type;	`List<Type>varName;`	`std::list<Type>varName;`

Tabla 2.3: Traducción de tipos básicos, Sets y List

En cuanto a las tuplas y uniones, diferentes tipos de clases se definen para representarlas. Además, dependiendo del tipo, cada clase necesita diferentes métodos y constructores que servirán luego para el manejo de variables de estos tipos en asignaciones, comparaciones o inclusiones. Por ejemplo, en las Figuras 2.10 y 2.11 mostramos cómo traducir a Java y C++, respectivamente, la siguiente definición de una tupla:

$$out : \mathbb{R} \cup \{\emptyset\};$$

El método `equals(...)` de la Figura 2.10 y la sobrecarga del operador ==, `operator==(...)`, de la Figura 2.11 son utilizados para comparar instancias de dichas clases a fin de determinar si estas son iguales o no. En cambio, el método `compareTo(...)` y la sobrecarga del operador <, `operator<(...)`, de las mismas Figuras, respectivamente, no son usados propiamente para comparar si una instancia de dicha clase es menor que otra. Por ejemplo, en el caso de Java, las clases usadas para traducir tuplas y uniones implementan `Comparable<Object>` a fin de poder incluirlas en conjuntos y listas. Esta implementación necesita que el método `compareTo` sea implementado y, en nuestro caso, solo basta con determinar si las instancias son diferentes o no. Algo similar ocurre en C++ con la sobrecarga del operador <. Esta operación es utilizada por el template `std::set` en la operación de inserción, `insert`. La sobrecarga del operador = en C++, `operator=(...)`, es usada en las asignaciones. Los diferentes constructores son utilizados para implementar asignaciones y comparaciones.

Para el caso particular de las definiciones de los puertos, hay que mencionar que, en el caso de los puertos de entrada, tanto en Java como en C++ se define una variable extra, cuyo tipo es la unión de los tipos de los diferentes puertos de entrada. Esta variable se utiliza para manejar los valores de entrada en la función de transición externa. En cuanto a los puertos de salida, se define una clase "independiente" por

```
static class T_out extends Object implements Comparable<Object>{
  public Double v_Number;
  public String v_Enum;
  public String type;
  T_out(){}
  T_out(T_out other){
    this.v_Number=other.v_Number;
    this.v_Enum=other.v_Enum;
    this.type=other.type;
  }
  T_out(Double v){
    v_Number=v;
    type="Number";
  }
  T_out(String v){
    v_Enum=v;
    type="Enum";
  }
  @Override
  public boolean equals(Object obj){
    if (obj == this) return true;
    else if (obj==null) return false;
    else if (obj.getClass()!=T_out.class) return false;
    else{
      T_out other = (T_out) obj;
      if (other.type!=this.type) return false;
      else if (this.type=="Number") return (this.v_Number.equals(other.v_Number));
      else return (this.v_Enum.equals(other.v_Enum));
    }
  }
  @Override
  public int compareTo(Object other){
    return this.equals(other)?0:1;
  }
}
T_out out;
```

Figura 2.10: Traducción de una unión de CML-DEVS a DEVS-Suite

```
class T_out{
    public:
    double v_Number;
    std::string v_Enum;
    std::string type;
    bool operator==(const T_out& other) const{
        if (this->type==other.type){
            if (this->type=="Number") return (this->v_Number==other.v_Number);
            else return (this->v_Enum==other.v_Enum);
        }
        else return false;
    }
    bool operator<(const T_out& other) const{
        return !(*this==other);
    }
    T_out& operator=(const T_out& other){
        this->v_Number=other.v_Number;
        this->v_Enum=other.v_Enum;
        this->type=other.type;
    }
    T_out(){}
    T_out(double v){
      v_Number=v;
      type="Number";
    }
    T_out(std::string v){
      v_Enum=v;
      type="Enum";
    }
} out;
```

Figura 2.11: Traducción de una unión de CML-DEVS a PowerDEVS

```
engine = "stopped";
years = buildSet(1974, 1988, 1990, 1991, 1992, 2004, 2013);
pair.v1 = toInteger(-14);
pair.v2 = toDouble(3.1416);
val = sin(45);
jobs = buildList(2.34, 5.98, 6.83);
elem = (xs).get(0);
sigma = sigma - e;
out.type = "Enum";
out.v_Enum = "EMPTY";
listA.clear();
listA.addAll(listB);
```

Figura 2.12: Traducción de asignaciones básicas de CML-DEVS a código DEVS-Suite

```
engine = "stopped";
years = buildSet<int>(7,1974, 1988, 1990, 1991, 1992, 2004, 2013);
pair.v1 = (int)-14;
pair.v2 = (double)3.1416;
val = sin(45);
jobs = buildList<double>(3,2.34, 5.98, 6.83);
elem = (xs).front();
sigma = sigma - e;
out.type = "Enum";
out.v_Enum = "EMPTY";
listA = listB;
```

Figura 2.13: Traducción de asignaciones básicas de CML-DEVS a código PowerDEVS

cada puerto de salida, con el tipo de dicho puerto. Esto último se debe a que los valores deben poder ser accedidos desde fuera del modelo atómico que se está traduciendo.

Asignaciones

Dependiendo del tipo de de la variable del lado izquierdo de la asignación, una asignación puede involucrar una o más sentencias en el lenguaje destino. Más aún, algunas funciones auxiliares como `buildSet`, pueden llegar a ser necesarias. Las Figuras 2.12 y 2.13 muestran las traducciones de las asignaciones de la Figura 2.4 a DEVS-Suite y PowerDEVS respectivamente.

Además de la funciones auxiliares `buildSet`, `builList`, `toInteger` y `toDouble` hay varias más que ayudan con la traducción. Estas pueden verse en el Anexo B.

Sentencias "For Each"

La traducción de las sentencias *foreach* involucra también variables auxiliares. En las Figuras 2.14 y 2.15 se muestra como traducir a código Java y C++, respectivamente la sentencia *foreach* de CML-DEVS de la Figura 2.5. Recordar que estas sentencias funcionan solamente para $\mathbb{P}$ y List.

```
Set<Integer> setTmp = new TreeSet<Integer>(X);
for(Integer x: setTmp){
    X = setDiff(X,buildSet(new Integer(x)));
    X = setUnion(X,buildSet(new Integer(x*2)));
}
```

Figura 2.14: Traducción de una sentencia *foreach* de CML-DEVS a código Java

```
std::set<int> setTmp = X;
for(std::set<int>::iterator it = setTmp.begin(); it!=setTmp.end(); it++){
    int x = *it;
    X = setDiff(X,buildSet<int>(1,x));
    X = setUnion(X,buildSet<int>(1,x*2));
}
```

Figura 2.15: Traducción de una sentencia *foreach* de CML-DEVS a código C++

Sentencias "case"

La clave en la traducción de las sentencias case . . . ; if (. . .) está en la traducción de las condiciones, ya que la traducción de las sentencias antes del if son como las que ya describimos o describiremos a continuación (asignaciones, *foreach*, *case* y *where*). La estructura principal de una sentencia case se traduce usando las clásicas estructuras condicionales de Java y C++, `if`, `else if` y `else`. En las Figuras 2.16 y 2.17 mostramos cómo traducir a Java y C++ las sentencias case de la transición interna del ejemplo de la Cola de Producción y, además, en las Figuras 2.18 y 2.19 las traducciones de las condiciones de CML-DEVS de la Figura 2.6

Sentencias "where"

La traducción de las sentencias where consiste, justamente, en reorganizar apropiadamente el orden de las definiciones y las sentencias en el lenguaje objetivo y realizar las correspondientes traducciones de dichas definiciones y sentencias. Las traducciones a Java y C++ de la sentencia where de DML-DEVS de la Figura 2.7 se describen en las Figuras 2.20 y 2.21, respectivamente.

```
if (xs != buildList()){
    s.v1 = xs.subList(1,xs.size()-1);
    s.v2 = INFINITY;
}
else if (xs == buildList()){
    s.v1 = xs;
    s.v2 = INFINITY;
}
```

Figura 2.16: Traducción de sentencias case de CML-DEVS a DEVS-Suite

```
if (xs != buildList<double>(0)){
    s.v1 = listTail(xs);
    s.v2 = INFINITY;
}
else if (xs == buildList<double>(0)){
    s.v1 = xs;
    s.v2 = INFINITY;
}
```

Figura 2.17: Traducción de sentencias case de CML-DEVS a PowerDEVS

```
!(engine=="stopped")
year != buildSet()
pair.v1 <= 13
jobs.contains(toDouble(5.63))
isSubset(buildSet(toInteger(1974), toInteger(1990)), years)
(in.type=="Enum")&&(in.v_Enum.equals("emitir"))
```

Figura 2.18: Traducción de condiciones de CML-DEVS a código Java

```
!(engine=="stopped")
year != buildSet<int>()
pair.v1 <= 13
jobs.find((double)5.63)!=jobs.end()
isSubset(buildSet<int>((int)1974), years)
(in.type=="Enum")&&(in.v_Enum=="emitir")
```

Figura 2.19: Traducción de condiciones de CML-DEVS a código C++

```
List<Double> ys = new ArrayList<Double>();
ys = listCat(xs,toDouble(value));
xs = ys;
```

Figura 2.20: Traducción de una sentencia “where” de CML-DEVS a código Java

```
std::list<Double> ys;
ys = listCat(xs,(double)value);
xs = ys;
```

Figura 2.21: Traducción de una sentencia “where” de CML-DEVS a código C++

```
public Set<Integer> filter(Set<Integer> S, Integer n){
    Set<Integer> res;
    Set<Integer> setTmp = new TreeSet<Integer>(S);
    for(Integer x: setTmp){
        if (x<n){
            res = setUnion(res, buildSet(toInteger(x)));
        }
    }
    return retVal;
}
```

Figura 2.22: Traducción de una función definida con CML-DEVS a código Java

```
std::set<unsigned int> modelName::filter(std::set<unsigned int> S, unsigned int n){
    std::set<unsigned int> res;
    std::set<unsigned int> setTmp = S;
    for(std::set<unsigned int>::iterator it = setTmp.begin(); it!=setTmp.end(); it++)\{
        unsigned int x = *it;
        if (x<n){
            res = setUnion(res, buildSet<unsigned int>((unsigned int)(x)));
        }
    }
    return retVal;
}
```

Figura 2.23: Traducción de una función definida con CML-DEVS a código C++

2.5.3. Funciones definidas por el usuario

Para traducir una función de CML-DEVS definida por el usuario, antes que todo, debe chequearse si el tipo de retorno de la función ha sido definido ya o no (parte del pre-procesamiento) y definirlo si es necesario. Luego, la función se define (con la correspondiente declaración en la cabecera en el caso de C++) usando las reglas de traducción previamente explicadas. Las figuras 2.22 y 2.23 muestran cómo se traduce la función definida en la Figura 2.8 a Java y a C++.

2.5.4. Post-procesamiento

Una vez que el modelo ha sido traducido, tanto a DEVS-Java como a PowerDEVS, es necesario realizar un post-procesamiento en el que cada variable involucrada en el estado del modelo, es reemplazada, dentro de las definiciones de las funciones (δ_{int}, δ_{ext}, ta y λ), en cada ocurrencia de éstas en el lado derecho de una asignación o en una comparación. Para tal fin, se utiliza una copia de las variables del modelo hecha anteriormente, al inicio de la función (`modelName prev=modelName(this);`).

Por ejemplo, supongamos que la variable `varX` es parte del estado del modelo y la misma aparece en el lado derecho de una asignación:

```
varY=varX+7;
```

entonces, se reemplaza por:

```
varY=prev.varX+7;
```

y, de igual forma, aparece dentro de una comparación (en cualquiera de los lados):

```
(varX >= varY)
```

también se reemplaza:

```
(prev.varX >= varY)
```

Este reemplazo se debe a que, especialmente en las funciones de transición, estas variables pueden modificar su valor pero, cuando se hace referencia a éstas en el modelo abstracto, la referencia es sobre el valor de las mismas en el estado previo a la transición.

Supongamos que en un modelo el estado tiene la siguiente representación:

$$S = \mathbb{N} \times \mathbb{R} \times \mathsf{Time}$$

y la función de transición interna, $\delta\mathsf{int}$, es como sigue:

$$\delta_{int}((n, r, \sigma)) = (n + 1, r * n, \sigma + n)$$

En este caso, tanto en $r * n$ como en $\sigma + n$ se debe considerar el valor de n antes de realizar dicha transición, i.e. antes de que n asuma el valor $n + 1$. Veamos cómo sería el código Java del modelo luego de la traducción. La definición del estado sería:

```
Integer n;
Double r;
Double t;
```

y la función de transición interna:

```
public void deltint(){
    n = n+1;
    r = r*n;
    sigma = sigma+n;
}
```

Sin embargo, claramente este no es el comportamiento que expresa el modelo (al finalizar la función, σ tendría el valor $\sigma + (n + 1)$ en lugar de $\sigma + n$, y un caso similar ocurre con r). Por lo tanto, las ocurrencias de las variables del estado en la parte derecha de las asignaciones son reemplazadas por la copia hecha previamente:

```
public void deltint(){
    modeName prev=new modelName(this);
```

```
    n = prev.n+1;
    r = prev.r*prev.n;
    sigma = prev.sigma+prev.n;
}
```

Notar que no todos los reemplazos son necesario para preservar el comportamiento del modelo. Sin embargo, para evitar tener que determinar qué variables modifican sus valores antes de ser utilizadas del lado derecho de una asignación o una comparación, todas son reemplazadas sin afectar la complejidad del modelo.

2.6. Conclusiones y otras consideraciones

En este capítulo describimos el primer aporte que presenta esta tesis. Este se centra en CML-DEVS, un lenguaje que permite la descripción de modelos DEVS en forma conceptual, abstracta, como fue presentado originalmente por Zeigler; independiente de toda plataforma o implementación.

Creemos que CML-DEVS es realmente abstracto, en cuanto a la expresividad que tiene y cómo se definen con él los modelos DEVS. Por ejemplo, si el lenguaje proveyese solamente listas (como el caso de DEVSpecL) se pude complicar innecesariamente la descripción del modelo, si el ingeniero desea utilizar conjuntos. En este sentido intentamos acercar el lenguaje a formalismo tales como Z y B dónde se utilizan más la teoría de conjuntos que la teoría de listas para especificar. Muchas entidades del mundo real son esencialmente conjuntos y no listas.

El principal beneficio de este lenguaje es que el especialista, o modelador, puede definir sus modelos sin la necesidad de poseer conocimientos o habilidades de programación y sin tener que conocer el lenguaje de ninguna herramienta de M&S específica.

También presentamos las reglas que permiten traducir, en forma automática, un modelo CML-DEVS en modelos de PowerDEVS y de DEVS-Suite. Estas reglas pueden ser implementadas dentro de un compilador multi-objetivo. De este modo, el especialista puede describir el modelo en forma abstracta y luego simularlo con la herramienta que desee. Se eligieron estas dos herramientas en particular, pero este trabajo puede extenderse al resto de las herramientas de M&S más utilizadas y conocidas. La intención principal es mostrar que es posible describir modelos DEVS en forma abstracta, o conceptual y traducirlos en forma automática a algún lenguaje concreto.

Teniendo los modelos DEVS descritos en su forma abstracta, independiente de toda implementación o plataforma particular permite, por un lado, la interoperabilidad entre especialistas o investigadores y, por el otro, facilita el mantenimiento y modificaciones de dichos modelos

En cuanto a la sintaxis de CML-DEVS y los símbolos como $\mathbb{N}$, $\mathbb{R}$, ∞, $\emptyset$, para des-

cribir el modelo y que luego pueda ser procesado por el compilador, podrían utilizarse varias alternativas. Una, es la de utilizar algún procesador de textos como Write de Libreoffice, o Microsoft Word, por ejemplo y los símbolos que estos procesadores proveen. Luego, estos documentos pueden convertirse fácilmente a documentos XML para ser procesados. También podría escribirse el modelo en texto plano utilizando comandos como los de LaTeX[28] o simplemente definir comandos o palabras claves para reemplazar los símbolos. Otra opción es desarrollar una GUI que ofrezca, entre otras cosas cómo asistente código y verificación de sintaxis, un entorno gráfico para el uso de estos símbolos y también mostrar el modelo y las funciones de una forma elegante.

Actualmente, no hay ningún compilador de CML-DEVS desarrollado. Esto forma parte de nuestro futuro trabajo, por lo tanto, las consideraciones antes mencionadas serán tenidas en cuenta. Creemos que, teniendo definidas ya las reglas de traducción, no es una empresa muy complicada desarrollar una GUI, incluyendo el compilador multi-objetivo, que permita la creación y edición de modelos CML-DEVS de forma sencilla y elegante.

El trabajo futuro incluye, además, extender CML-DEVS para incluir las variantes o extensiones del formalismo DEVS mencionadas en el Capítulo 1, es decir, utilizar CML-DEVS para describir en forma abstracta modelos P-DEVS, STDEVS, Cell-DEVS, etc. También podemos incluir dentro de las futuras líneas de investigación, extender las reglas de traducción de modelos CML-DEVS a otras herramientas de M&S.

Como el lector pudo notar a lo largo del capítulo, el código generado luego de la traducción puede no ser tan elegante como uno espera, podría ser tal vez más simple o quizás el lector podría imaginar una traducción óptima en algunos casos. Siendo que CML-DEVS trata de cubrir un gran espectro de modelos, esto ocasiona que el código objetivo resultante de la traducción de tal rango de modelos sea a veces engorroso. Sin embargo, creemos que el código resultante no debe ser mantenido ni editado. Si el especialista o modelador necesita hacer cambios en el modelo, debe realizarlo sobre el modelo CML-DEVS y luego re-compilarlo. Por ejemplo, por este motivo también se provee de un método para asignar valores iniciales a las variables del estado del sistema, y no tener que editar el código final asignando dichos valores. Por otro lado, una vez que el ingeniero o especialista está seguro de tener el modelo que quiere, es decir el modelo ha sido ya validado (Capítulo 3), puede pedirle a algún programador que haga una implementación óptima en el mejor simulador para dicho modelo.

Capítulo 3

Validación de Modelos DEVS

En este capítulo describimos la segunda contribución que presenta esta tesis, la misma consiste en una novedosa técnica de validación de modelos DEVS y ésta fue publicada durante el desarrollo de esta tesis en un congreso internacional [29] y una revista internacional [30]. En la siguiente sección introducimos la verificación y validación de modelos DEVS y finalizamos describiendo cómo se estructura el resto del capítulo.

3.1. Introducción

El desarrollo y el uso de modelos de simulación ha aumentado considerablemente en los últimos años. Con frecuencia se utilizan como la primera representación de sistemas que más tarde se usarán para la toma de decisiones en situaciones críticas. Se ha convertido, por lo tanto, cada vez más necesario la definición de técnicas rigurosas que aseguren que dichos modelos representan tan bien como sea posible el sistema real que está siendo modelado. Dicho en otras palabras, la utilización de métodos de Verificación y Validación (V&V) de modelos de simulación se ha vuelto crucial.

De acuerdo con el la directiva [31] del Departamento de Defensa de EE.UU., verificación es "el proceso de determinar si la implementación de un modelo representa con exactitud la descripción y especificaciones conceptuales del desarrollador". En el contexto del modelado y simulación, la pregunta que la verificación de modelos trata de responder es: ¿Estamos simulando, o hemos simulado, el modelo correctamente? Por otro lado, validación es "el proceso de determinar el grado en que un modelo es la representación exacta del mundo real desde la perspectiva del uso previsto del modelo". En este caso, la pregunta es: ¿Estamos simulando, o hemos simulado, el modelo correcto?

Realizar la V&V de modelos de simulación ha sido identificado como una actividad primordial ya que esto puede incrementar la confianza del usuario en los resultados

de la simulación y llegar a la acreditación/certificación de los sistemas simulados [32]. Esta acreditación o certificación es de especial importancia cuando los resultados de la simulación se usan en la toma de decisiones sobre temas cruciales.

La Figura 3.1, presentada por Sargent [33] y adaptada luego por Robinson [34], muestra las diferentes fase de V&V sobre el proceso de modelado. A partir del problema o sistema que se intenta modelar se describe el modelo conceptual (o abstracto) que luego se codifica obteniendo el modelo computacional (o concreto). Sobre este modelo concreto se realizan los experimentos (simulaciones). Entre las diferentes etapas existen validaciones y verificaciones. Esto es, se verifican y validan los modelos comparándolos entre sí y con el problema del mundo real o el sistema que se intenta modelar. El trabajo presentado en este capítulo apunta a la fase *Validación del modelo conceptual*, i.e. validar el modelo abstracto o conceptual contra los requerimientos del mismo.

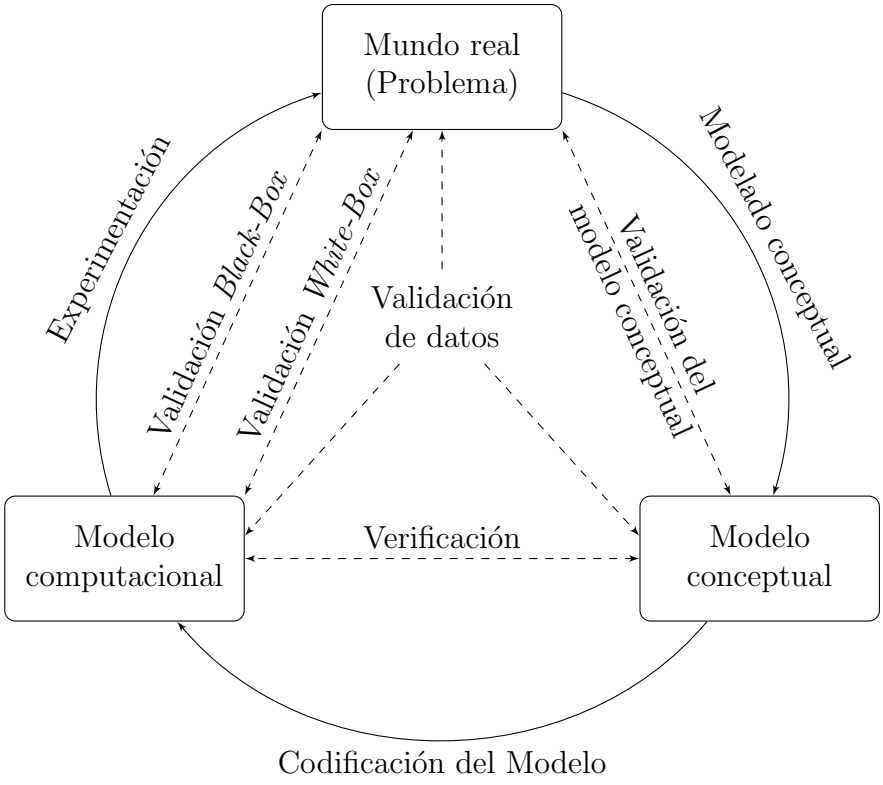

Figura 3.1: V&V de modelos de simulación en el proceso de modelado

3.1.1. Validación de modelos de simulación y el testing basado en modelos

La validación de un modelo contra los requerimientos, usualmente no puede ser realizada matemáticamente porque los requerimientos no son formales. Una forma alternativa es a través de simulaciones. El ingeniero compara los resultados de las simulaciones con los requerimientos con el fin de decidir si el modelo es correcto o no. Esto es

particularmente importante cuando el modelo es demasiado grande, su implementación es crítica y/o las decisiones que se tomen de acuerdo a los resultados provistos por la implementación del modelo y sus simulaciones son críticas. Además, dado que es muy importante encontrar la mayor cantidad de errores posibles y lo más pronto posible, entonces un minucioso proceso de simulación puede ser una actividad que reduzca el costo total del sistema de destino.

Durante la simulación del modelo sería deseable ejecutar todas las posibles simulaciones, i.e. ejecutar simulaciones con todas las configuraciones de simulación (estados iniciales y secuencias de eventos) posibles, y comparar estos comportamientos con los requerimientos. Desafortunadamente, una simulación exhaustiva es impracticable en casi todos los proyectos ya que involucra un número infinito de configuraciones de simulación. Considerando esto, la selección de un conjunto apropiado de configuraciones de simulación es un asunto crucial que debe considerar dos factores opuestos: a) el conjunto de configuraciones de simulación debe ser lo suficientemente grande para dar una seguridad razonable de que el modelo es una representación correcta de los requerimientos; y b) el conjunto debe ser lo suficientemente chico para que la V&V encaje dentro del tiempo y presupuesto con el que se cuenta.

El formalismo de modelado y simulación en el que basamos este trabajo, como en toda la tesis, es DEVS. En la Figura 3.2 presentamos el proceso de validación de modelos DEVS a través de simulaciones. Este proceso es una de las posibles alternativas para llevar a cabo la fase de validación del modelo conceptual de la Figura 3.1. Primero que todo, los requerimientos se extraen del *mundo real*, o de la descripción del problema que está siendo modelado. Basado en éstos, el modelo conceptual o abstracto es definido utilizando un formalismo de modelado.

De acuerdo con este proceso de validación, una vez que el modelo abstracto es definido, un conjunto de configuraciones de simulación es derivado de este. Posteriormente, ambos, el modelo abstracto y el conjunto de configuraciones de simulación son traducidos al lenguaje de alguna herramienta de simulación.

Finalmente, los resultados de la simulación (concretos) necesitan ser traducidos en forma inversa, es decir, desde el lenguaje concreto al abstracto. Entonces, estos resultados abstractos se comparan con los resultados esperados. Esta última tarea, conocida generalmente como el *problema del oráculo*, es un problema complejo en este contexto como también lo es en el testing de software y para el cual no hay una solución general [32, 35].

A pesar del hecho de que el objetivo final es formalizar todo el proceso de validación, el aporte de este capítulo está enfocado en la selección de un conjunto apropiado de configuraciones de simulación. Como mencionamos antes, esta es la parte crucial en este proceso de validación. El mismo consiste en convertir un problema infinito (el conjunto de todas las configuraciones de simulación) en un problema finito. Más

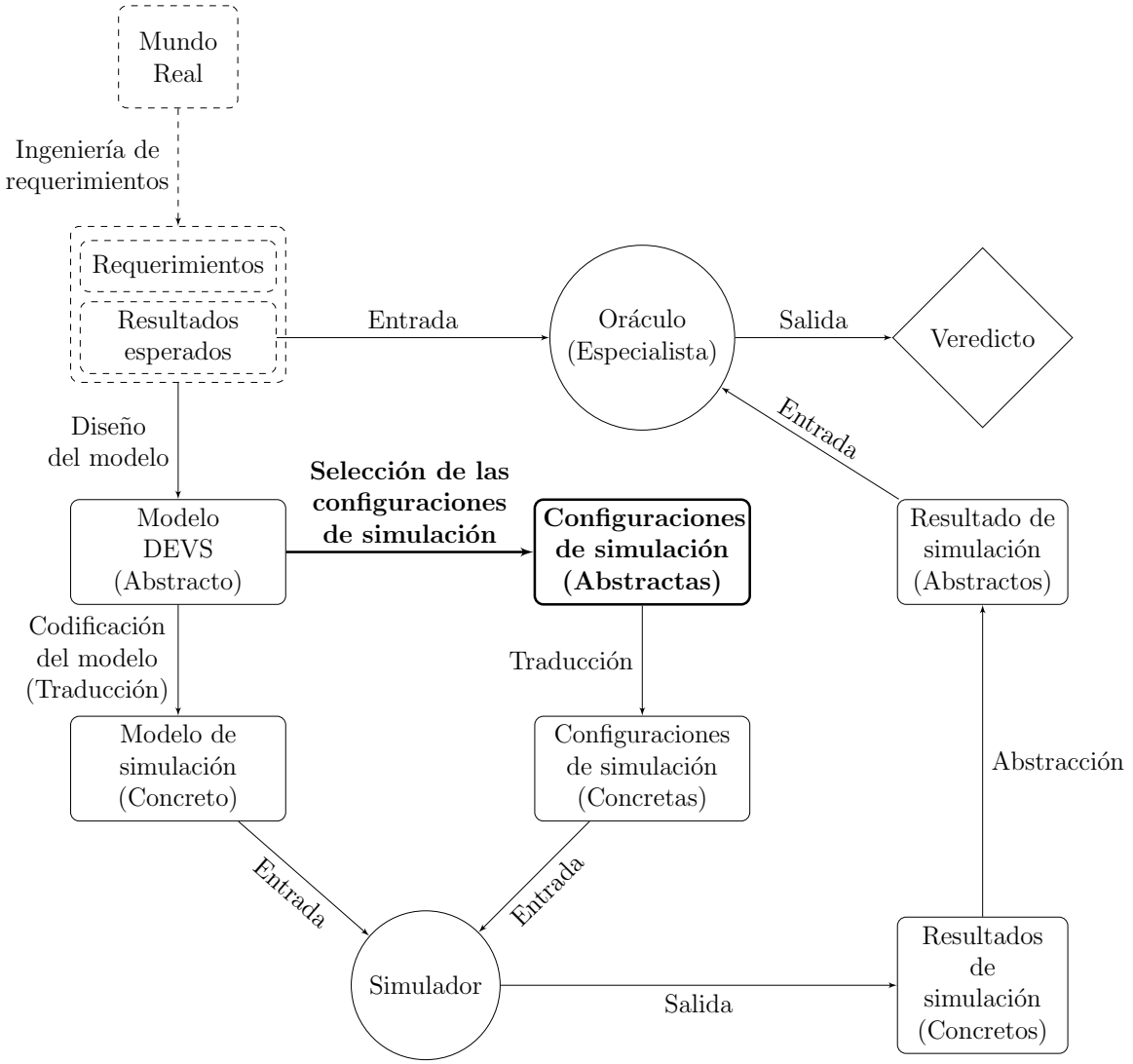

Figura 3.2: Validación de modelos DEVS a través de simulación

aún, esto debe hacerse tratando de mantener en el conjunto final las configuraciones más importantes o relevantes. En otras palabras, el conjunto final de configuraciones de simulación debe ser pequeño y debe cubrir minuciosamente todas las alternativas funcionales descriptas en el modelo. El resto del proceso de validación consiste, principalmente, en la implementación de dicho proceso y se detalla con más profundidad en la Sección 3.7

En general, la simulación de modelos se realiza de acuerdo a la experiencia o intuición de un especialista. Por lo tanto, no se sigue ninguna guía ni criterios rigurosos para definir un conjunto adecuado de configuraciones de simulación, convirtiendo a la simulación de modelos en un proceso informal y propenso a errores. Por otro lado, siendo la selección de configuraciones de simulación una actividad informal, esta no puede ser automatizada. Sin embargo, se puede automatizar hasta cierto punto si el proceso de selección es formalizado en forma tal que, más tarde, una herramienta de software pueda ayudar en esta tarea. Por consiguiente, es deseable definir formalmente

criterios de simulación para considerar la simulación de modelos como un proceso de validación con un aceptable grado de exactitud.

En el campo del *testing de software* existe un escenario análogo. De acuerdo a Utting y Legeard [36] el testing de software se ocupa de la verificación dinámica del comportamiento de un programa en un conjunto finito de casos de prueba, adecuadamente seleccionados de un dominio de ejecución usualmente infinito, contra el comportamiento esperado. Existen varios trabajos tratando de formalizar el proceso de testing de software. Muchos de estos pertenecen al sub-campo del testing conocido como *testing basado en modelos* (MBT, por el inglés Model Based-Testing). Utting y Legeard definen MBT como la generación de casos de test ejecutables, basada en modelos del comportamiento del sistema que se está testeando (SUT, del inglés System Under Test). Entonces, podemos trazar una analogía entre MBT y la validación de modelos vía simulación. El modelo puede ser visto como la especificación del SUT en MBT y la generación de casos de test como la generación de las configuraciones de simulación. Más aún, podemos trazar también una analogía entre CML-DEVS, presentado en el Capítulo 2 y los lenguajes de formales de especificación de MBT.

Siendo tan importante en el desarrollo de software, el proceso de MBT ha sido perfeccionado hasta el punto de convertirse en casi automático, en muchos casos obteniendo muy buenos resultados. [35–37]. En consecuencia, vale la pena explorar si algunas de las técnicas de MBT pueden replicarse en el contexto de la validación de modelos vía simulación.

Una parte importante de esta automatización es posible gracias a que existen criterios de testing precisos. Esto es, criterios de testing qué indican que tests deben generarse a partir del modelo, siendo que usualmente existe un número infinito de posibles tests [36]. Además, es una condición necesaria para lograr la automatización partir de un lenguaje de especificación formal.

A pesar de que muchos métodos de MBT pueden ser utilizados o adaptados para seguir la analogía antes mencionada, basamos el trabajo que se presenta en este capítulo en el *Test Template Framework* (TTF). El TTF es un método de MBT presentado por Stocks y Carrington [38] e implementado luego por Cristiá et al [37]. El TTF fue introducido para definir formalmente conjuntos de datos de test proveyendo de una estructura al proceso de testing. La elección del TTF como método de MBT es motivada por el hecho de que lidia con la definición matemática y lógica del modelo en lugar de analizar, por ejemplo, trazas o ejecuciones del mismo. Por lo tanto, la forma en que se define un modelo DEVS permite adaptar el TTF de forma relativamente sencilla como veremos más adelante.

Teniendo todo esto en cuenta, como segundo aporte de esta tesis se presenta una familia de criterios de simulación proporcionando un novedosa técnica para validar, en forma rigurosa y sistemática, modelos DEVS vía simulaciones. Esta técnica fue publi-

cada en un congreso internacional [29], premiado como mejor artículo del *Symposium on Theory of Modeling & Simulation* (TMS/DEVS 2012) y artículo finalista de la *Spring Simulation Multi-Conference 2012* en reconocimiento a su calidad, originalidad e importancia para el modelado y la simulación. Además, una versión extendida fue publicada en una revista internacional [30].

Creemos que simulando los modelos DEVS siguiendo esta técnica se incrementa la confianza en la corrección del modelo, validando aspectos o características del mismo que pueden ser pasadas por alto por el especialista. Además, la técnica permite automatizar en gran medida este proceso de validación, ahorrando tiempo y costos considerablemente.

El resto de este capítulo se organiza de la siguiente manera. En la Sección que sigue describimos y comentamos otros enfoques sobre la V&V de modelos de sistemas de eventos discretos. En la Sección 3.3 presentamos los criterios de simulación que forman el núcleo de esta contribución. Una posible automatización de este proceso de validación es abordada en la Sección 3.7, junto con otras consideraciones; y en la Sección 3.8 comentamos las conclusiones y trabajo futuro que se desprenden de este trabajo. Los casos de estudio con los que se muestra como se aplica esta técnica de validación se describen en el Capítulo 4.

3.2. Trabajos Relacionados y Otros Enfoques

A continuación discutimos y comentamos diferentes trabajos que involucran verificación y validación de modelos de simulación. Estos trabajos, o bien presentan los enfoques más similares al nuestro, o bien motivan o muestran por qué nuestro trabajo es relevante para la comunidad de M&S.

Balci [39] presenta las directrices para la verificación, validación y acreditación (VV&A) de modelos de simulación. Allí realiza una clasificación de las diferentes técnicas de V&V para modelos de simulación presentando una taxonomía de más de 77 técnicas de V&V para modelos de simulación convencionales y 38 técnicas de V&V para modelos de simulación orientados a objetos. En su trabajo, Balci lista al testing de modelos como uno de los candidatos para la V&V de modelos de simulación. También, Labiche y Wainer [32] hacen una revisión de la V&V de modelos de sistemas de eventos discretos. Ellos proponen aplicar o adaptar técnicas de testing de software existentes a la V&V de modelos DEVS. En particular, ellos afirman que técnicas formales deben ser aplicadas. Por tanto, justificando nuestro aporte. En varios trabajos [40–43] Sargent discute diferentes enfoques, paradigmas y técnicas relacionadas a la validación y verificación de modelos de simulación. Estos son trabajos interesantes para conocer sobre la generalización de diferentes procesos de validación y verificación, sin embargo, no describen ningún proceso particular de validación en detalle. Nuestra contribución

complementa todos estos trabajos dando un método de validación detallado y formal, adaptado de la comunidad de ingeniería de software.

Existen varios trabajos que usan técnicas de verificación, como *model-checking*, para verificar que un modelo sea correcto. Por ejemplo, Napoli y Parente [44] presentan un algoritmo de model-checking para Máquinas de Estados Finitas Jerárquicas como una abstracción de un modelo DEVS. Hacen foco en la generación de configuraciones de simulación para DEVS, pero como contra-ejemplos obtenidos al aplicar su algoritmo de model-checking. Otro trabajo reciente y relevante que involucra técnicas de verificación es [45] donde Saadawi y Wainer introducen una nueva extensión del formalismo DEVS, llamado *Rational Time-Advance DEVS* (RTA-DEVS), DEVS de avance de tiempo racional. Los modelos RTA-DEVS pueden ser formalmente chequeados con algoritmos de model-checking estándares, permitiendo la verificación de modelos DEVS clásicos. A pesar de que las técnicas de model-checking están definidas formalmente, y son útiles para probar propiedades y teoremas sobre el modelo, el principal problema que estas revisten es el así llamado *problema de explosión de estados* [46], i.e. la explosión exponencial del espacio de estados y las variables en cualquier sistema práctico o real. Esto hace casi imposible el uso de estas técnicas en grandes proyectos, a pesar de que model-checking ha sido utilizado en proyectos reales.

K. J. Hong and T. G. Kim [47] introducen un método para la verificación de sistemas de eventos discretos. Proponen un formalismo llamado *Time State Reachability Graph* (TSRG), para especificar módulos de un modelo de eventos discretos; y una metodología para la generación de secuencias de test para testear dichos módulos a un nivel de entrada/salida (E/S). Luego, un análisis teórico de grafos de TSGR genera todas las secuencias temporizadas de E/S posibles con las que se puede construir un conjunto de de secuencias de test temporizadas de E/S con un 100 % de cobertura.

Otro trabajo reciente que aplica técnicas de verificación sobre simulación de eventos discretos es [48] donde da Silva y de Melo presentan un método para realizar simulaciones ordenadamente y verificar propiedades sobre éstas usando sistemas de transiciones. Ambos, los posibles caminos de simulación y las propiedades a ser verificadas se describen usando sistemas de transición. La verificación es alcanzada mediante la construcción de un tipo especial de producto sincrónico entre estos dos sistemas de transición. Ellos enfocan su trabajo en la verificación de propiedades sobre la simulación pero no en la generación de configuraciones de simulación para validar el modelo.

Li et al [49] desarrollaron un entorno para testear herramientas DEVS. En este entorno combinan enfoques de testing *black-box* y *white-box*. En realidad, este trabajo no está relacionado directamente con el nuestro, ya que no validan o verifican modelos DEVS, si no implementaciones DEVS. Sin embargo, es útil para ver cómo introducen técnicas de testing de software en las comunidades que trabajan con el formalismo DEVS.

Luego de revisar muchos trabajos sobre la V&V de modelos de simulación parece que no hay ninguna propuesta similar al método de validación que presentamos en este capítulo. Por ejemplo, no pudimos encontrar ningún trabajo que tome la representación matemática o lógica del modelo como punto de partida para el proceso de validación. Más aún, creemos que, en general, la gente de la comunidad de M&S no tienen este tema en cuenta. Actualmente, desarrollan sus modelos directamente utilizando una herramienta de simulación, con su propio lenguaje, y realizan experimentos directamente sobre ella. Al trabajar con el modelo concreto, se apunta a técnicas de verificación, i.e. verificar que el modelo concreto represente el modelo abstracto. Nosotros, por el otro lado, proponemos una técnica complementaria, para trabajar directamente con el modelo abstracto de modo de comprobar que éste se ajusta a los requerimientos. Más concretamente, nuestro trabajo propone un método para validar formalmente los modelos abstractos e introduce técnicas bien conocidas de MBT en la comunidad de M&S.

3.3. Partición del conjunto de configuraciones de simulación

Dado un modelo DEVS, con el fin de simularlo con alguna herramienta de simulación, es necesario proporcionar un estado inicial (no definido por el formalismo) y una secuencia de eventos de entrada con sus correspondientes tiempos de ocurrencia. Llamamos a estos estados iniciales y secuencias de eventos de entrada una *Configuración de Simulación*. Usualmente, el conjunto de todas las configuraciones de simulación posibles, i.e. todos los estados iniciales posibles y todas las posibles secuencias de eventos de entrada, es infinito, incluso si el modelo tiene conjuntos de estados y de eventos de entrada y de salida finitos. Entonces, la validación vía simulaciones requiere seleccionar algunas de estas configuraciones de simulación, ver Figura 3.3. Actualmente, esta selección se realiza de acuerdo a la experiencia de algún especialista, por tanto, un experto en el dominio es necesario.

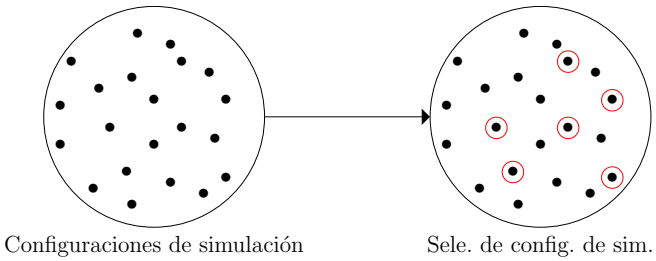

Figura 3.3: Selección tradicional de configuraciones de simulación

La técnica que se presenta en este trabajo propone dividir el conjunto de todas las

configuraciones de simulación en clases de equivalencia, aplicando uno o más *criterios de partición*. Llamamos a cada una de estas clases de equivalencia una *Clase de Configuración de Simulación* (SCC, del inglés Simulation Configuration Class). Una SCC es, entonces, un conjunto de configuraciones de simulación. Posteriormente, una configuración de simulación de cada SCC debe ser elegida, ver Figura 3.4. Estas configuraciones de simulación seleccionadas son las únicas que deben ser ejecutadas.

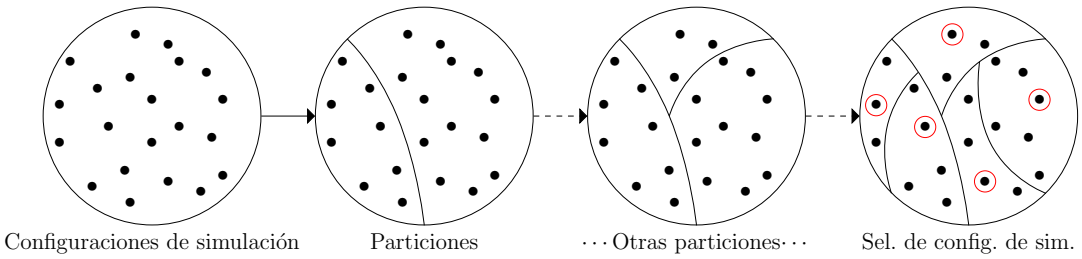

Figura 3.4: Nuestra propuesta: Partición de las configuraciones de simulación

La razón de seleccionar solo una configuración de simulación de cada clase está basada en la *hipótesis de uniformidad* presentada por Bougé et al [50]. Esta afirma, en el testing de software, que "un programa se comporta uniformemente en una clase de equivalencia si se cumple lo siguiente: si el programa trabaja correctamente para algunos datos de entrada de una clase de equivalencia entonces trabaja correctamente para todos los demás datos de la clase". Esta es la suposición clave que hace la comunidad de testing de software porque permite reducir el dominio de entrada potencialmente infinito a uno más chico, finito. Siendo una suposición, no es probada a pesar de que muchos métodos de testing se basan en ésta [35]. De hecho, la idea misma del testing se basa implícitamente en esta hipótesis debido a que un caso de test individual representa toda una clase de casos de test. En otras palabras, cuando un especialista selecciona un caso de test, está asumiendo que representa un conjunto de posibles candidatos.

Nosotros adaptamos este concepto a la validación de modelos. Por lo tanto, decimos que estos subconjuntos, en los que las posibles configuraciones de simulación se dividen son clases de equivalencia porque se asume que el modelo tiene un comportamiento uniforme para estos subconjuntos de configuraciones de simulación. Además, si la hipótesis de uniformidad se cumple y un error en el modelo es encontrado con alguna simulación de alguna clase de configuraciones de simulación en particular, entonces el mismo error debe ser revelado con cualquier otra simulación de esa clase.

En este contexto, la hipótesis de uniformidad puede ser formalizada como sigue. Sea M_{abs} algún modelo abstracto (M_{abs} puede ser visto como una fórmula matemática o lógica) y M_{con} su modelo concreto, i.e. su correspondiente traducción al lenguaje de una herramienta de simulación. Sea, además, a una configuración de simulación derivada de M_{abs} y a' su traducción al lenguaje concreto. Entonces, $M_{con}(a')$ significa la ejecución de a' en M_{con}; y $M_{abs}(a, M_{con}(a'))$ afirma que $M_{con}(a')$ es el resultado esperado con

respecto a a de acuerdo a M_{abs}. Notar que si $M_{con}(a')$ no es el resultado esperado de a de acuerdo a M_{abs}, entonces $M_{abs}(a, M_{con}(a'))$ es falso, i.e. $\neg\ M_{abs}(a, M_{con}(a'))$ es verdadero. Entonces, la hipótesis de uniformidad se cumple si y sólo si para cada SCC_i y para cada $a \in SCC_i$ se cumple lo siguiente:

$$M_{abs}(a, M_{con}(a')) \Rightarrow \forall\ x \in SCC_i : M_{abs}(x, M_{con}(x'))$$

esto es, la hipótesis de uniformidad se cumple si el modelo se comporta de la misma forma para todos los elementos de una SCC dada, SCC_i.

Con esta hipótesis de uniformidad, algunas estrategias o criterios de simulación asumen que cada elemento de una clase es equivalente a todos los otros (de la misma clase) para la simulación de una funcionalidad particular del modelo. Sin embargo, esta es justamente una hipótesis y no siempre se satisface. Por lo tanto, como señalan Stocks y Carrington [38], esta suposición es a veces inválida y proponen aplicar repetidamente las estrategias con el fin de dividir las clases en sub-clases hasta que el ingeniero considere que las clases son razonablemente pequeñas o cada funcionalidad del modelo está cubierta por alguna clase [51]. Por otra parte, si las clases se subdividen al infinito, entonces, cada clase es un conjunto unitario y, por consiguiente, la hipótesis se verifica trivialmente. Por lo tanto, si no se va hasta el infinito pero se subdividen las clases significativamente, se aumenta la probabilidad de que esta hipótesis sea válida.

Como mencionamos al principio de la sección, cada elemento de una SCC consiste en un estado inicial (para inicializar la simulación) y un par conteniendo el evento a simular y su correspondiente tiempo de arribo, i.e. cuándo debe ser simulado dicho evento. Siguiendo esta idea, para describir una SCC necesitamos definir el conjunto de posibles estados iniciales y el conjunto de posibles pares de entrada para la simulación. Un par de entrada es un par ordenado $(evento, tiempo)$. Por lo tanto, una SCC es definida por:

- un conjunto de estados, $IniSt \subseteq S$, y
- un conjunto de pares ordenados de evento y tiempo, $InPairs \subseteq \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$.

donde τ representa simular la situación de *no evento* y una transición interna ocurre en este caso. Esto es necesario para indicar la simulación de una transición interna.

Es importante remarcar, como ya se ha mencionado, que el formalismo DEVS no define estados iniciales. Esto, en realidad, pertenece a la fase de simulación, y siendo que este trabajo apunta a guiar estas simulaciones, un estado inicial debe definirse. Este no es necesariamente el estado inicial del sistema real, y de hecho no lo suele ser, es solamente el punto de partida de la simulación. Más aún, definir este estado inicial hace que la simulación de diferentes funcionalidades del modelo sea más simple, ya

que no es necesario llevar el sistema hasta un estado en particular para comenzar la simulación allí.

La clase total de configuraciones de simulación, i.e. el conjunto de todas las configuraciones de simulación posibles, para un modelo DEVS dado está definido por:

- $IniSt = IniSt_1 \cup \ldots \cup IniSt_n = S$, y
- $InPairs = InPairs_1 \cup \ldots \cup InPairs_n = \{(x,t) : x \in X \cup \{\tau\} \wedge t \in \mathbb{R}_0^+)\}$.

IniSt representa todos los estados iniciales posibles a partir de los cuales puede comenzar una simulación. Notar que, inicialmente, todos los estados del modelo son posibles estados iniciales. Algunos de estos estados podrían ser considerados *inseguros* o *inalcanzables* por parte de algún experto en el área. Sin embargo, hasta que el modelo sea validado, esto no puede ser confirmado porque, precisamente, el ingeniero está tratando de descubrir si se han entendido y formalizado correctamente los requerimientos del modelo, de este modo, necesita validar todos los estados (y descartar aquellos que realmente sean inseguros o inalcanzables). *InPairs* es el conjunto de todos los posibles pares de entrada. Entonces, la idea es partir $IniSt \times InPairs$ analizando el modelo DEVS guiado por los criterios que se definen a continuación.

3.4. Criterios de Partición

Presentamos, ahora, los diferentes criterios para dividir, o partir el conjunto de todas las posibles configuraciones de simulación. Los criterios atacan diferentes aspectos de los modelos DEVS. Algunos criterios se aplican, por ejemplo, a las definiciones de las funciones de transición interna o externa, otros a la definición de los estados o los conjuntos de eventos de entrada y de salida.

Es importante mencionar que en algún momento del proceso de partición, es posible que un criterio no genere nuevas clases. Esto se debe a que el resultado puede ser una clase ya obtenida anteriormente o una clase vacía. Más aún, no importa el orden en el que los criterios se aplican, ya que son independientes unos de otros.

3.4.1. Funciones de transición definidas por casos

Es muy común definir las funciones de transición externa y/o interna por casos. Es decir, definiendo los resultados de las mismas en función de ciertas condiciones. El primer, y tal vez, el criterio más intuitivo es partir el conjunto de posibles simulaciones en varias clases, una por cada caso en la definición de estas funciones.

Sean δ_{ext} y δ_{int} las funciones de transición de un modelo DEVS definidas por casos:

$$\delta_{ext}(s,e,x) = \begin{cases} expr^1_{ext}(s,e,x) \text{ si } P^1_{ext}(s,e,x) \\ \vdots \\ expr^n_{ext}(s,e,x) \text{ si } P^n_{ext}(s,e,x) \end{cases}$$

$$\delta_{int}(s) = \begin{cases} expr^1_{int}(s) \text{ si } P^1_{int}(s) \\ \vdots \\ expr^m_{int}(s) \text{ si } P^m_{int}(s) \end{cases}$$

donde $expr^i_{ext}$ y $expr^i_{int}$ son los resultados de las funciones si la proposiciones P^i_{ext} y P^i_{int}, respectivamente, se cumplen. Este criterio propone generar una clase por cada proposición en la definición de las funciones de transición. Cada clase queda definida por:

- Asociadas a la función de transición externa:

$$IniSt_i = \{s \in S \mid \exists\ e \in \mathbb{R}^+_0, x \in X : P^i_{ext}(s,e,x)\},$$
$$InPairs_i = \{(x,t) \in InPairs \mid \exists\ s \in IniSt_i, e \in \mathbb{R}^+_0 : P^i_{ext}(s,e,x)\}.$$

  con $i \in [1,n]$

- Asociadas a la función de transición interna:

$$IniSt_j = \{s \in S \mid P^j_{int}(s)\},$$
$$InPairs_j = \{(\tau, 0)\}.$$

  con $j \in [1,m]$.

En el caso de la función de transición interna la idea es configurar cierto estado permitiendo que ocurra una transición interna en particular, es por esto que el tiempo relativo al evento τ (el no evento) es 0.

3.4.2. Conjuntos definidos por extensión

En la definición de los modelos DEVS, muchas veces, algunos conjuntos (estados, eventos de entrada o eventos de salida) o parte de estos se definen por extensión. i.e. listando los elementos del conjunto. Necesariamente, estos conjuntos son finitos y relativamente pequeños. Por lo tanto, este criterio propone simular todos los escenarios donde aparezcan, al menos una vez, cada elemento de estos conjuntos.

Supongamos que el conjunto de valores del estado, S, de un modelo DEVS es un conjunto definido por extensión:

$$S = \{s_1, s_2, \ldots, s_n\}$$

entonces, la clases generadas al aplica este criterio deben ser:

$$IniSt_i = \{s_i\},$$
$$InPairs_i = InPairs.$$

con $i \in [1, n]$. Aplicando este criterio, el especialista se garantiza simular el modelo en cada posible estado. Sin embargo, si este criterio se aplica solamente sobre la definición del estado, y ningún otro criterio es aplicado, ciertos eventos de entrada pueden no ser simulados para ciertos estados.

Supongamos ahora que el conjunto de valores de entrada está definido por:

$$X = \{(in, x) : x \in \{x_1, x_2, \ldots, x_n\}\}$$

las clases resultantes serían ahora:

$$IniSt_i = S,$$
$$InPairs_i = \{((in, x_i), t) : t \in \mathbb{R}_0^+\}.$$

con $i \in [1, n]$. Con esto, el especialista se asegura simular todos los posibles eventos de entrada. Combinando estas clases con las anteriores, permite simular todos los posibles estados con todos los posibles eventos. Esto es explicado más extensamente en la Sección 3.5.

3.4.3. Conjuntos definidos por comprensión

Los conjuntos definidos por comprensión son otra forma usual de definir conjuntos de un modelo DEVS, es decir, definiendo las propiedades que cada elemento del conjunto debe satisfacer. Esto se hace a través de un predicado lógico, el cual puede ser simple o una definición muy compleja involucrando varias operaciones. Por tanto, este criterio plantea usar estas definiciones para partir el conjunto de configuraciones de simulación.

Supongamos que el conjunto de estados de un modelo particular es un conjunto definido por comprensión, $S = \{s : TYPE \mid P(s)\}$, donde $TYPE$ es el tipo de los elementos del conjunto y P es un predicado lógico. El criterio indica, primero, escribir P en su forma normal disyuntiva (DNF) [52]:

$$P = (P_1^1 \wedge \ldots \wedge P_{n_1}^1) \vee (P_1^2 \wedge \ldots \wedge P_{n_2}^2) \vee \ldots \vee (P_1^m \wedge \ldots \wedge P_{n_m}^m)$$

y luego, partir el conjunto de configuraciones de simulación de acuerdo a esta DNF:

$$IniSt_i = \{s \in S \mid (P_1^i(s) \wedge \ldots \wedge P_{n_i}^i(s))\},$$
$$InPairs_i = InPairs.$$

con $i \in [1, m]$. Esta misma idea puede aplicarse al conjunto de entradas, salidas o cualquier otro conjunto definido por comprensión del modelo.

Por ejemplo, supongamos que el conjunto de estados, S, de un modelo dado es $S = \{(n, m) : \mathbb{Z} \times \mathbb{Z} \mid n * m > 0 \Rightarrow n > m\}$. Entonces, el predicado $P = n * m > 0 \Rightarrow n > m$ se escribe en su DNF, aplicando algún algoritmo conocido [52], $P = \neg\ (n * m > 0) \vee (n > m)$ y dos SCCs deben definirse, una para el predicado $\neg\ (n * m > 0)$ y la otra para $n > m$.

3.4.4. Particiones estándar

En casi todos los modelos, diferentes operadores matemáticos o lógicos aparecen en las definiciones de los elementos del modelo (funciones de transición, de avance de tiempo, valores del estado) y estos pueden ser simples (suma, unión de conjuntos) o más complejos (operadores definidos en términos de otros más simples, funciones definidas por el modelador). Cada uno de estos operadores tiene un dominio de entrada particular y con este criterio se busca dividir este dominio asociando a este una *partición estándar*. Una partición estándar es una partición del dominio del operador en conjuntos llamados sub-dominios; cada sub-dominio está definido por las condiciones que cada operando de la operación debe satisfacer. En consecuencia, cada sub-dominio es transformado en una condición para generar una SCC.

Por lo tanto, para cada operador del modelo debe definirse una partición estándar. Por ejemplo, para el operador $<$ $(a < b)$, la partición estándar podría ser [51]:

$$\begin{array}{lll} a < 0, b < 0 & a < 0, b = 0 & a < 0, b > 0 \\ a = 0, b < 0 & a = 0, b = 0 & a = 0, b > 0 \\ a > 0, b < 0 & a > 0, b = 0 & a > 0, b > 0 \end{array}$$

Supongamos que $x_1, \ldots, x_n$ son algunas de las variables que definen el estado, S, de algún modelo y $\theta(x_1, \ldots, x_n)$ es un operador de aridad n con la partición estándar asociada $SP_1(x_1, \ldots, x_n)$, $\ldots$, $SP_m(x_1, \ldots, x_n)$. Cuando θ aparece en una expresión del modelo, el conjunto de configuraciones de simulación debe partirse usando la partición estándar asociada a él:

$$\begin{aligned} IniSt_i &= \{s \in S \mid SP_i(x_1, \ldots, x_n)\}, \\ InPairs_i &= InPairs. \end{aligned}$$

3.4.5. Propagación de dominios

Este es un criterio particular, ya que no genera nuevas particiones por sí mismo. El propósito es obtener particiones estándar de operadores complejos combinando las particiones estándar de los sub-operadores más simples que lo componen.

Cada sub-operación tiene su propia partición del dominio de entrada, que pueden ser ignorados por el criterio de particiones estándar si este se aplica al operador complejo. Usando la propagación de dominios las particiones de los dominios de entrada de los sub-operadores son propagados al de más alto nivel [53].

Por ejemplo, sea $\blacksquare$ un operador complejo definido como: $\blacksquare(A, B, C) = (A \blacktriangle B) \blacklozenge C$ donde $\blacktriangle$ y $\blacklozenge$ son operadores simples.

Supongamos que $\blacktriangle$ y $\blacklozenge$ tienen las siguientes particiones estándar:

$$SP^{\blacktriangle}(S, T) = D_1^{\blacktriangle}(S, T) \vee \ldots \vee D_n^{\blacktriangle}(S, T)$$
$$SP^{\blacklozenge}(U, V) = D_1^{\blacklozenge}(U, V) \vee \ldots \vee D_k^{\blacklozenge}(U, V)$$

Entonces, aplicamos primero $SP^{\blacktriangle}$ a la sub-expresión $(A \blacktriangle B)$, reemplazando los parámetros formales que aparecen en $SP^{\blacktriangle}$ por A y B respectivamente:

$$SP^{\blacktriangle}(A, B) = D_1^{\blacktriangle}(A, B) \vee \ldots \vee D_m^{\blacktriangle}(A, B)$$

con $m \leq n$.

Posteriormente, hacemos lo mismo con $SP^{\blacklozenge}$, obteniendo:

$$SP^{\blacklozenge}(A \blacktriangle B, C) = D_1^{\blacklozenge}(A \blacktriangle B, C), \vee \ldots \vee D_j^{\blacklozenge}(A \blacktriangle B, C)$$

con $j \leq k$.

Finalmente, combinamos ambas proposiciones obtenidas y simplificamos:

$$SP^{\blacksquare} = SP^{\blacktriangle}(A, B) \wedge SP^{\blacklozenge}(A \blacktriangle B, C)$$

3.4.6. Particiones del tiempo

En formalismos temporizados, como DEVS, el modelado del tiempo en un aspecto importante. Es muy común, en modelos DEVS, usar variables adicionales para modelar el tiempo. Además, una característica de estos modelos es que el tiempo transcurrido desde la última transición, e, aparece como un parámetro en la función de transición externa y se puede hacer uso e, como una variable más, en la definición de dicha función de transición. Por lo tanto, en la validación de estos modelos, surge la pregunta *"¿cómo sabemos si cada evento ha sido ya simulado en todos los intervalos de tiempo relevantes o significativos?"*. Esto es, en aquellos intervalos de tiempo donde es posible encontrar algún error de modelado. Nuevamente, para responder a esta pregunta habría que simular todos los posibles eventos en todos los posibles intervalos de tiempo haciendo de la validación un proceso infinito.

Entonces, este criterio consiste en, primero, identificar aquellas variables del estado que interactúan con el tiempo transcurrido, e, o son utilizadas para simular el avance del tiempo o temporizadores, por ejemplo. Posteriormente, se definen *instantes de tiempo claves* o *intervalos de tiempo claves* de acuerdo con la interacción de estas variables.

Una vez que estos intervalos de tiempo están definidos, se deben generar clases por cada evento de entrada en cada instante de tiempo. Por ejemplo, supongamos que el intervalo de tiempo $[a, b]$ es relevante para ver una funcionalidad particular del modelo, por consiguiente, sería significativo simular eventos (x, t) con $x \in X \cup \{\tau\}$ y tiempos $t < a$, $t = a$, $a < t < b$, $t = b$ y $t > b$. Formalmente, para cada intervalo de tiempo definido $[a_i, b_i]$, cinco SCCs deben crearse:

- $IniSt_i^1 = \{s \in S\}$,
  $InPairs_i^1 = \{(x, t) \mid x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge t < a_i\}$.

- $IniSt_i^2 = \{s \in S\}$,
  $InPairs_i^2 = \{(x, a_i) \mid x \in X \cup \{\tau\}\}$.

- $IniSt_i^3 = \{s \in S\}$,
  $InPairs_i^3 = \{(x, t) \mid x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge a_i < t < b_i\}$.

- $IniSt_i^4 = \{s \in S\}$,
  $InPairs_i^4 = \{(x, b_i) \mid x \in X \cup \{\tau\}\}$.

- $IniSt_i^5 = \{s \in S\}$,
  $InPairs_i^5 = \{(x, t) \mid x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge t > b_i\}$.

y por cada instante de tiempo establecido t_j, tres clases deben ser definidas:

- $IniSt_j^1 = \{s \in S\}$,
  $InPairs_j^1 = \{(x, t) \mid x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge t < t_j\}$.

- $IniSt_j^2 = \{s \in S\}$,
  $InPairs_j^2 = \{(x, t_j) \mid x \in X \cup \{\tau\}\}$.

- $IniSt_j^3 = \{s \in S\}$,
  $InPairs_j^3 = \{(x, t) \mid x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge t > t_j\}$.

La intensión de este criterio es simular diferentes escenario donde eventos ocurran en diferentes instantes, para validar la interacción de aquellas variables utilizadas para simular el tiempo entre ellas y la interacción entre estas y e (en el caso dela transición externa).

Luego, cuando una de estas clases se combina con aquellas generadas, por ejemplo, por el criterio *Conjuntos definidos por extensión*, será posible simular todos los eventos de entrada en todos los intervalos de tiempo relevantes

3.5. Combinando clases

Como se mencionó en la Sección 3.4, los criterios de partición son independientes unos de otros. Más aún, cada configuración de simulación apunta a validar una característica particular del modelo. Sin embargo, suele ser más eficiente[1] validar simultáneamente más de una característica.

Con el fin de lograr esto, es beneficioso combinar las clases generadas al aplicar cada criterio. Observar que, sin embargo, no todos los criterios son siempre aplicados. Esto dependerá del modelo y del tiempo disponible para validarlo. Además, notar que el mismo criterio puede ser aplicado más de una vez sobre el mismo modelo. Por ejemplo, el criterio *Conjuntos definidos por extensión* se puede aplicar primero sobre el conjunto de estados y luego sobre el conjunto de eventos de entrada. El resultado de esto, son dos conjuntos de configuraciones de simulación independientes. Estos conjuntos pueden combinarse con el fin de simular el arribo de cada evento sobre cada estado. En este sentido, estas SCCs combinadas encontrarían errores, por ejemplo, por la llegada de un evento particular estando el sistema en un estado equivocado.

Veamos cómo las SCCs pueden combinarse en un simple ejemplo. Sean $X = \{(in, x) : x \in \mathbb{N}\}$ y $S = \mathbb{N} \times \{ON, OFF\}$, parte de un modelo de algún sistema. Supongamos que luego de aplicar algunos criterios se obtienen las siguiente SCCs:

- SCC_a:
  $IniSt_a = \{(n, m) \in S \mid n \leq 10\}$,
  $InPairs_a = \{(1, t) \mid t \in \mathbb{R}_0^+\}$

- SCC_b:
  $IniSt_b = \{(n, m) \in S \mid m = ON\}$,
  $InPairs_b = \{(1, t) \mid t \in \mathbb{R}_0^+\}$

- SCC_c:
  $IniSt_c = \{(n, m) \in S \mid m = OFF\}$,
  $InPairs_c = \{(1, t) \mid t \in \mathbb{R}_0^+\}$

Ahora, podemos combinar estas clases haciendo la intersección de los correspondientes conjuntos *IniSt* y *InPairs* como sigue:

[1] Usamos eficiencia como una medida del número de errores encontrados en el modelo. La eficiencia computacional no es abordada aquí.

- $SCC_d = SCC_a \wedge SCC_b$:

$$\begin{aligned} IniSt_d &= IniSt_a \cap IniSt_b = \{(n,m) \in S \mid n \leq 10\} \cap \{(n,m) \in S \mid m = ON\} = \\ &= \{(n,m) \in S \mid n \leq 10 \wedge m = ON\}, \\ InPairs_d &= InPairs_a \cap InPairs_b = \{(1,t) \mid t \in \mathbb{R}_0^+\} \cap \{(1,t) \mid t \in \mathbb{R}_0^+\} = \\ &= \{(1,t) \mid t \in \mathbb{R}_0^+\} \end{aligned}$$

- $SCC_e = SCC_a \wedge SCC_c$:

$$\begin{aligned} IniSt_e &= IniSt_a \cap IniSt_c = \{(n,m) \in S \mid n \leq 10\} \cap \{(n,m) \in S \mid m = OFF\} = \\ &= \{(n,m) \in S \mid n \leq 10 \wedge m = OFF\}, \\ InPairs_e &= InPairs_a \cap InPairs_c = \{(1,t) \mid t \in \mathbb{R}_0^+\} \cap \{(1,t) \mid t \in \mathbb{R}_0^+\} = \\ &= \{(1,t) \mid t \in \mathbb{R}_0^+\} \end{aligned}$$

- $SCC_f = SCC_b \wedge SCC_c$:

$$\begin{aligned} IniSt_f &= IniSt_b \cap IniSt_c = \{(n,m) \in S \mid m = ON\} \cap \{(n,m) \in S \mid m = OFF\} = \\ &= \{\}, \\ InPairs_f &= InPairs_b \cap InPairs_c = \{(1,t) \mid t \in \mathbb{R}_0^+\} \cap \{(1,t) \mid t \in \mathbb{R}_0^+\} = \\ &= \{(1,t) \mid t \in \mathbb{R}_0^+\} \end{aligned}$$

Notar, sin embargo, que solo SCC_d y SCC_e son SCCs válidas ya que SCC_f es vacía por ser $IniSt_f$ el conjunto vacío.

Como se muestra en el ejemplo, combinar SCCs vía intersección puede producir clases vacías (o insatisfacibles). Si este es el caso se debe proceder de la siguiente forma: a) eliminar las vacías resultantes; y b) mantener las SCCs que producen estas SCCs vacías (salvo que las SCCs originales queden incluidas en otras SCCs resultantes de este proceso de combinación).

Observar que, siendo la intersección conmutativa, es lo mismo combinar, por ejemplo, SCC_a y SCC_b como $SCC_a \cap SCC_b$ o como $SCC_b \cap SCC_a$. Más aún, si SCC_d es el resultado de combinar SCC_a y SCC_b, y SCC_d se combina con alguna SCC_α el resultado es $SCC_b \cap SCC_a \cap SCC_\alpha$, que es lo mismo independientemente del orden en el que las clases se combinan. En resumen, las SCCs pueden combinarse en cualquier orden.

Entonces, el objetivo de generar nuevas SCCs, a través de estas combinaciones, es simular situaciones cada vez más complejas, o involucrando más aspectos del modelo, a través de clases cada vez más refinadas. Cabe aclarar que, idealmente, se combinan todas las clases generadas por los criterios. Esto puede depender del tiempo y presupuesto disponible. Otra aclaración es que, las configuraciones de simulación se elijen

de las clases resultantes de las combinaciones y no de las clases combinadas.

3.6. Secuenciación de simulación

Una vez que todas las SCCs quedan definidas es necesario configurar un estado inicial para la simulación y ejecutar una transición. Sin embargo, el nuevo estado obtenido luego de la transición podría ser un estado inicial de otra SCC. Si este es el caso, sería computacionalmente más eficiente, y práctico, continuar la simulación ejecutando alguna transición indicada por esta nueva SCC. Esto evita tener que configurar una nueva simulación para esta última SCC, reusando la configuración dejada por la transición anterior. Luego, este proceso termina produciendo una secuencia de configuraciones de simulación que deben ejecutarse una tras la otra.

Aquí proponemos un algoritmo para generar estas secuencias analizando las clases obtenidas luego de aplicar los criterios. El pseudo código del algoritmo puede verse en el Algoritmo 1, y se explica a continuación.

Algorithm 1 Secuenciación de simulación

Input: $TotSCC$: Conjunto de todas las SCCs generadas por los criterios
Output: $SimSeq$: Conjunto de secuencias de configuraciones de simulación

```
1:  while TotSCC ≠ ∅ do
2:    select scc ∈ TotSCC
3:    select is ∈ scc.IniSt
4:    select ip ∈ scc.InPair
5:    s' ← simulate(is, ip)
6:    seq ← (is, ip)
7:    TotSCC ← TotSCC \ {scc}
8:    while ∃ scc' ∈ TotSCC | s' ∈ scc'.IniSt do
9:      select scc' ∈ TotSCC | s' ∈ scc'.IniSt
10:     select ip' ∈ scc'.InPairs
11:     seq ← seq ∘ (s', ip')
12:     s' ← simulate(s', ip')
13:     TotSCC ← TotSCC \ {scc'}
14:   end while
15:   SimSeq ← SimSeq ∪ {seq}
16: end while
```

El proceso principal del algoritmo está guiado por la siguiente idea: seleccionar una SCC ($scc \in TotSCC$), un estado inicial de esa SCC ($is \in scc.IniSt$) y simular un evento del conjunto de pares de eventos de entrada $(event, time)$ asociados a esa SCC ($ip \in scc.InPair$). La sentencia $simulate(s, ip)$ significa ejecutar un paso de la simulación del modelo desde el estado s con el par ip. Una vez que el evento seleccionado ha sido simulado y se ha alcanzado un nuevo estado, s', la secuencia actual es actualizada (sentencia 6) y el conjunto de SCCs se reduce removiendo la SCC agregada a

la secuencia (sentencia 7). Ahora, hay dos alternativas, a) esperar a que ocurra una transición interna (si $ta(s) \neq \infty$); o b) simular otro evento. Como se muestra en la sentencia 8, si existe una SCC, scc', tal que s' es uno de sus posibles estados iniciales, entonces scc' es elegida como la siguiente SCC (scc' puede reflejar cualquiera de las situaciones anteriores, a o b). En este caso, uno de sus pares de eventos de entrada es simulado (sentencias 10 y 11). Posteriormente, el proceso continua repetidamente eligiendo un estado o aguardando por una transición interna. A este punto del proceso, si el estado alcanzado por el último paso de la simulación no pertenece al conjunto de estados iniciales de ninguna de las SCCs restantes, esta secuencia termina y se la agrega al conjunto de secuencias de configuraciones de simulación (sentencia 15). Una nueva secuencia de simulación comienza, luego, si quedan SCCs sin explorar.

De esta manera, con el Algoritmo 1 todas las clases se utilizan al menos una vez. Además, criterios de cobertura más complejos podrían definirse para la generación de las secuencias, por ejemplo, de una manera similar a lo que proponen Souza et al [54].

3.7. Automatización y otras cuestiones

Como se ha mencionado en la introducción de este capítulo, la técnica presentada en este trabajo permite automatizar una gran parte del proceso de validación de modelos DEVS. La intensión de esta sección es mostrar cómo se podría construir una pieza de software para asistir a los especialistas cuando apliquen este método. Por consiguiente, explicamos qué es lo que se puede automatizar en cada etapa y describimos los principales temas que necesitan ser abordados para lograr dicha automatización. Además, otras consideraciones sobre la metodología son discutidas.

Un panorama general de este proceso semi-automático es como sigue:

1. Parsear la descripción abstracta o matemática del modelo DEVS.
2. Seleccionar y aplicar los criterios de partición para generar las simulaciones.
3. Traducir las configuraciones de simulación al lenguaje de simulación elegido.
4. Seleccionar las simulaciones y generar las secuencias de simulación.
5. Simular las secuencias.
6. Traducir los resultados al lenguaje abstracto del modelo.
7. Comprar los resultados obtenidos con los requerimientos a fin de lograr un veredicto sobre la validación del modelo.

La descripción matemática o abstracta de un modelo DEVS puede ser parseada automáticamente sólo si se escribe utilizando algún estándar o notación formal. Esto

fue ya abordado y discutido en el Capítulo 2 de esta tesis y, una posibilidad, es utilizar el lenguaje allí presentado, CML-DEVS, para describir estos modelos.

En cuanto a la aplicación de los criterios, un análisis preliminar indica que sería apropiado aplicarlos en forma semi-automática. Creemos que una herramienta debería permitir al especialista seleccionar qué criterios deben ser aplicados sobre qué partes del modelo. Mas aún, el especialista podría agregar nuevos criterios y utilizarlos. Otra alternativa podría ser implementar una heurística que seleccione automáticamente los criterios de acuerdo a un análisis sobre la descripción del modelo.

La aplicación de los criterios involucra un problema importante, que es el número total de las SCCs generadas. A pesar de que el número de clases puede ser considerable, este no es exponencial y el el número de criterios involucrados es fijo y pequeño. La cuestión crucial es la combinación entre las clases. El orden del número de clases generadas, entonces, está dado por $O(x^n)$ donde x es el número de clases producidas por un criterio y n el número total de criterios aplicados (recordar que n es fijo y pequeño).

Las configuraciones de simulación generadas, están descriptas esencialmente en matemática. Por lo tanto, para realizar las simulaciones éstas necesitan ser también traducidas al lenguaje de simulación, como la definición del modelo. Esto podría hacerse con el mismo compilador de CML-DEVS, en caso de que se haya utilizado dicho lenguaje para describir el modelo; o adaptando los trabajos hechos para MBT [55, 56]. Este sería un proceso semi-automático, en el caso que el ingeniero tenga que definir las reglas para esta traducción (si no se utilizó CML-DEVS para describir el modelo).

Para la secuenciación de simulación, y la simulación propiamente dicha, al menos una configuración de simulación de cada clase debe ser elegida. Encontrar una configuración de simulación para una SCC particular significa encontrar un elemento que pertenezca a esta. Usualmente, esto involucra resolver una fórmula sobre la teoría de conjuntos, aritmética sobre enteros y números reales y, posiblemente, otras teorías matemáticas. Además, variables libres se mueven en rangos sobre conjuntos infinitos, convirtiendo el problema en *indecidible*. Una posibilidad para resolver este problema es utilizar un *SMT solver* (Satisfiability Modulo Theories Solver) [57, 58] adaptando el trabajo de Cristiá y Frydman [59]. Otra alternativa podría ser utilizar un *constraint solver* como {*log*} (pronunciado 'setlog') [60–62].

Luego de realizar la simulación, debe llevarse a cabo un proceso inverso. Esto es, los resultados de la simulación deben traducirse al lenguaje matemático o formal usado para describir el modelo DEVS. Nuevamente, este debería ser un proceso semi automático si el ingeniero tiene que definir las reglas de traducción. Luego, los resultados de la simulación pueden ser comparados con los resultados esperados (los requerimientos). Esta comparación necesariamente es manual, ya que involucra los requerimientos y éstos suelen estar detallados en un lenguaje coloquial.

Por otro lado, es posible utilizar esta metodología para testear el modelo de simu-

lación concreto. Sin embargo, hay que tener en cuenta que ambas validaciones no se puede hacer al mismo tiempo para el mismo modelo. En efecto, si nuestra metodología se utiliza para validar el modelo abstracto, es necesario asumir que el modelo concreto es una fiel representación del modelo abstracto. Nuevamente, utilizando CML-DEVS para describir el modelo, se ayuda notablemente en esta tarea porque, una vez verificado el compilador de CML-DEVS, las traducciones de los modelos abstractos pueden asumirse como correctas (con respecto al modelo abstracto). En cambio, si la metodología se utiliza para validar el modelo concreto, se debe asumir que el modelo abstracto es correcto con respecto a los requerimientos.

Finalmente, como puede observarse y ya fue mencionado durante todo el capítulo, los criterios son aplicados sobre la definición matemática del modelo y también se basan en operaciones lógicas o matemáticas. Por consiguiente, el proceso de validación puede ser considerado como una metodología rigurosa (dado que se basa en matemática, lógica y lenguajes formales). Más aún, esta metodología es sistemática siendo que el conjunto total de configuraciones de simulación es sistemáticamente subdividido obteniendo clases de simulación cada vez más refinadas y expresivas. En este sentido, ya que cubrir todos los caminos posibles del modelo es imposible (por ser éstos un número infinito) proponemos cubrir todas las estructuras lógicas y matemáticas del modelo, en consecuencia, cubriendo todos los caminos significativos del modelo.

3.8. Conclusiones y trabajo futuro

Como segunda contribución de la tesis, en este capítulo se presenta una familia de criterios para conducir las simulaciones de modelos DEVS en forma disciplinada y cubriendo las simulaciones más significativas para incrementar la confianza sobre la corrección del modelo. La principal ventaja de realizar la simulación de un modelo como aquí se propone es que no se necesita de la experiencia de un especialista o grupos de especialistas, ni un experto en el dominio del modelo, para seleccionar las configuraciones de simulación con el fin de validar el modelo. Esta selección es el resultado de seguir un conjunto de reglas formales sobre el modelo matemático, sin tener que conocer sobre el dominio sobre el cual se está modelando. Esto disminuye la posibilidad de pasar por alto alguna configuración de simulación que pudiera revelar errores de modelado.

Otra ventaja de este trabajo es la posibilidad de automatizar al menos parte del proceso de validación de modelos DEVS. Una cuestión importante y necesaria para permitir esta automatización es que los modelos DEVS se definan utilizando una gramática formal y abstracta. Esto fue detallado en el capitulo anterior donde se provee un lenguaje para tal efecto.

También se debe considerar la posibilidad de reutilizar estas técnicas para testear software derivados de modelos DEVS. Un modelo DEVS puede ser usado como una

adecuada forma de especificación de un sistema a ser implementado en un lenguaje de programación. Las secuencias de simulación generadas al aplicar los criterios de partición pueden usarse como casos de test para testear dicha implementación. Más aún, existen herramientas de simulación que generan código automáticamente. De este modo, si el modelo es validado minuciosamente, la pieza de software resultante sería correcta.

El trabajo futuro que se desprende de esta tesis, es la automatización del proceso de validación de modelos DEVS vía simulaciones. Esto se lo hará integrando con el futuro editor y compilador de CML-DEVS ofreciendo un entorno completo para el diseño de modelos DEVS abstractos y la validación de los mismos. Además de implementar la técnica aquí presentada para la generación de las configuraciones de simulación, se pueden agregar nuevos criterios de cubrimiento para la generación de secuencias de simulación.

Otra línea de investigación es extender esta metodología a modelos DEVS acoplados y a las diferentes extensiones o variantes del formalismo DEVS presentadas en el Capítulo 1. Esto significa la definición de nuevos criterios teniendo en cuenta las particularidades y características de los modelos acoplados y de estas extensiones o variantes de DEVS.

Capítulo 4

Casos de Estudio

En este capítulo mostramos, a través de dos casos de estudio, cómo se describe un modelo DEVS *real* en forma abstracta utilizando CML-DEVS y cómo se aplica la técnica de validación presentada en el Capítulo 3. Los modelos representan el sistema de control de un ascensor y de una máquina expendedora de gaseosas, respectivamente.

Por cada caso de estudio, presentamos primero la descripción del sistema a ser modelado y los requerimientos del mismo. Luego describimos el modelo abstracto, en la versión de DEVS "original" y a también utilizando CML-DEVS. Posteriormente, mostramos la traducción de cada modelo a DEVS-Suite y a PowerDEVS. Finalmente, detallamos cómo se aplican los criterios de simulación para cada modelo mostrando algunas de las clases de configuraciones generadas por estos criterios, el resto de las SCCs se muestran en los apéndices C y D

4.1. Ascensor

El primer caso de estudio representa el modelo del sistema de control de un ascensor. El mismo tiene un panel de control con un botón para cada piso y otros dos botones, uno para abrir la puerta y otro para cerrarla. Además, cuenta con un interruptor para detener o reiniciar su funcionamiento. Cada vez que el ascensor alcanza un piso, este recibe una señal (la misma señal para todos los pisos). El ascensor cuenta con un display que debe mostrar el número del piso actual o el símbolo **ST** en caso de que esté detenido por estar activado el interruptor. Para prevenir accidentes o mal-funcionamiento del ascensor, este posee dos sensores, uno para chequear, durante el cerrado de la puerta, si hay alguien o algo cruzándola. El otro sensor, previene que se exceda el límite del peso soportado por el ascensor antes de que comience a operar. Para completar el funcionamiento del ascensor, el sistema cuenta con tres temporizadores, cuyos propósitos se describe a continuación, junto con los requerimientos restantes del sistema:

- Si el ascensor está detenido en algún piso y alguien lo llama desde un piso diferente, o alguien presiona el botón de algún otro piso desde el tablero interior, luego de T_{D_1} unidades de tiempo la puerta debe comenzar a cerrarse, y demora T_{D_2} unidades de tiempo en cerrarse completamente. Sin embargo, si el botón de apertura de puerta se presiona o se activa alguno de los sensores, la puerta detiene el cerrado y se abre, reseteando el temporizador.
- Si luego de T_A unidades de tiempo ($T_A > T_{D_1}$) desde que al ascensor se le indica ir a un piso diferente al actual la puerta no se cierra, se dispara una alarma indicando esta situación. A diferencia del temporizador anterior, este no se resetea a pesar de que alguno de los sensores se active, o alguno de los botones de la puerta es presionado. Solamente se resetea cuando la puerta se cierra totalmente y el ascensor comienza a moverse; si la alarma se había disparado, ésta deberá apagarse en este momento.
- Luego de T_{GF} unidades de tiempo desde que el ascensor se detiene en un piso, diferente a la planta baja, y si no recibe ningún llamado de ningún piso, éste debe regresar a la planta baja y abrir sus puertas.
- Este ascensor no tiene memoria, por lo tanto, va al primer piso que se le indica desde que está detenido, ignorando las siguiente solicitudes, hasta que llegue a destino.
- La puerta nunca debe cerrarse si:
  - Alguno de los sensores está activo (i.e. alguien o algo está cruzando la puerta, o el límite del peso soportado ha sido excedido).
  - El interruptor de funcionamiento está activado.

4.1.1. Modelo DEVS abstracto

En las Figuras 4.1, 4.2, 4.3 y 4.4 se muestra una posible representación del modelo DEVS del sistema de control del ascensor descripto en la sección anterior.

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

$S = ActualFloor \times FloorCalled \times Engine \times Door \times Sensors \times Switch \times Alarm \times Timers \times NextTimer$

donde:

$ActualFloor = \mathbb{N}$

$FloorCalled = \mathbb{N} \cup \{\emptyset\}$

$Engine = \{\mathsf{up}, \mathsf{down}, \mathsf{stopped}\}$

$Door = \{\mathsf{open}, \mathsf{closed}, \mathsf{closing}\}$

$Sensors = \{0,1\} \times \{0,1\}$

$Alarm = Switch = \{0,1\}$

$Timers = (\mathbb{R}_0^+ \cup \{\infty\}) \times (\mathbb{R}_0^+ \cup \{\infty\}) \times (\mathbb{R}_0^+ \cup \{\infty\}) \times (\mathbb{R}_0^+ \cup \{\infty\}) \times (\mathbb{R}_0^+ \cup \{\infty\})$

$NextTimer = \{\mathsf{A}, \mathsf{D_1}, \mathsf{D_2}, \mathsf{GF}, \mathsf{O}\}$

$X = \{(in, x) : x \in \mathbb{N} \cup \{\mathsf{fsig}, \mathsf{ws_{on}}, \mathsf{ws_{off}}, \mathsf{ds_{on}}, \mathsf{ds_{off}}, \mathsf{od_{press}}, \mathsf{cd_{press}}, \mathsf{s_{on}}, \mathsf{s_{off}}\}\}$

$Y = \{(out, y) : y \in (\mathbb{N} \cup \{\mathsf{ST}\}) \times \{\mathsf{up}, \mathsf{down}, \mathsf{stop}, \emptyset\} \times \{\mathsf{opendoor}, \mathsf{closedoor}, \emptyset\} \times \{\mathsf{firealarm}, \mathsf{stopalarm}, \emptyset\}\}$

Figura 4.1: Modelo DEVS del sistema de control de un ascensor (Parte A)

$$
\begin{aligned}
&\delta_{int}((f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt)) = \\
&\begin{cases}
(f, \emptyset, \mathsf{stopped}, \mathsf{open}, (ws, ds), sw, a, (\infty, \infty, \infty, \mathsf{T_{GF}}, \infty), nt'(\infty, \infty, \infty, \mathsf{T_{GF}}, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge eng \neq \mathsf{stopped} \wedge f = fc \wedge f \neq 0 & (4.1) \\
(f, \emptyset, \mathsf{stopped}, \mathsf{open}, (ws, ds), sw, a, (\infty, \infty, \infty, \infty, \infty), nt'(\infty, \infty, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge eng \neq \mathsf{stopped} \wedge f = fc \wedge f = 0 & (4.2) \\
(f, fc, eng, d, (ws, ds), sw, a, (\infty, \infty, \infty, \infty, \infty), nt'(\infty, \infty, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge eng \neq \mathsf{stopped} \wedge f \neq fc & (4.3) \\
(f, fc, \mathsf{stopped}, d, (ws, ds), sw, a, (\mathsf{T_A}, \infty, \infty, \infty, \infty), nt'(\mathsf{T_A}, \infty, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge sw = 1 \wedge eng \neq \mathsf{stopped} & (4.4) \\
(f, fc, eng, d, (ws, ds), sw, a, (at - ot, dt1 - ot, dt2 - ot, gft - ot, \infty), nt'(at - ot, dt1 - ot, dt2 - ot, gft - ot, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge sw = 1 \wedge eng = \mathsf{stopped} & (4.5) \\
(f, fc, \mathsf{up}, d, (ws, ds), sw, 0, (\infty, \infty, \infty, \infty, \infty), nt'(\infty, \infty, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc > f & (4.6) \\
(f, fc, \mathsf{down}, d, (ws, ds), sw, 0, (\infty, \infty, \infty, \infty, \infty), nt'(\infty, \infty, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc < f & (4.7) \\
(f, fc, eng, \mathsf{closing}, (ws, ds), sw, 0, (at - ot, \infty, \mathsf{T_{D_2}}, \infty, \infty), nt'(at - ot, \infty, \mathsf{T_{D_2}}, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{open} \wedge fc \neq \emptyset & (4.8) \\
(f, fc, eng, d, (ws, ds), sw, a, (\infty, \infty, \infty, gft - ot, \infty), nt'(\infty, \infty, \infty, gft - ot, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc = \emptyset & (4.9) \\
(f, fc, eng, \mathsf{open}, (ws, ds), sw, a, (at - ot, \mathsf{T_{D_1}}, \infty, \infty, \infty), nt'(at - ot, \mathsf{T_{D_1}}, \infty, \infty, \infty)) \\
\quad \text{if } nt = \mathsf{O} \wedge d = \mathsf{closing} & (4.10) \\
(f, fc, eng, \mathsf{closing}, (ws, ds), sw, a, (at - dt1, \infty, \mathsf{T_{D_2}}, \infty, ot - dt1), nt'(at - dt1, \infty, \mathsf{T_{D_2}}, \infty, ot - dt1)) \\
\quad \text{if } nt = \mathsf{D_1} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 & (4.11) \\
(f, fc, eng, d, (ws, ds), sw, a, (at - dt1, \infty, \infty, \infty, ot - dt1), nt'(at - dt1, \infty, \infty, \infty, ot - dt1)) \\
\quad \text{if } nt = \mathsf{D_1} \wedge \neg\,(ds = 0 \wedge ws = 0 \wedge sw = 0) & (4.12) \\
(f, fc, \mathsf{up}, \mathsf{closed}, (ws, ds), sw, 0, (\infty, \infty, \infty, \infty, ot - dt2), nt'(\infty, \infty, \infty, \infty, ot - dt2)) \\
\quad \text{if } nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f & (4.13) \\
(f, fc, \mathsf{down}, \mathsf{closed}, (ws, ds), sw, 0, (\infty, \infty, \infty, \infty, ot - dt2), nt'(\infty, \infty, \infty, \infty, ot - dt2)) \\
\quad \text{if } nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f & (4.14) \\
(f, fc, eng, d, (ws, ds), sw, a, (at - dt2, \infty, \infty, \infty, ot - dt2), nt'(at - dt2, \infty, \infty, \infty, ot - dt2)) \\
\quad \text{if } nt = \mathsf{D_2} \wedge \neg\,(ds = 0 \wedge ws = 0 \wedge sw = 0) & (4.15) \\
(f, fc, eng, d, (ws, ds), sw, 1, (\infty, dt1 - at, dt2 - at, gft - at, ot - at), nt'(\infty, dt1 - at, dt2 - at, gft - at, ot - at)) \\
\quad \text{if } nt = \mathsf{A} & (4.16) \\
(f, 0, eng, \mathsf{closing}, (ws, ds), sw, a, (\infty, \infty, \mathsf{T_{D_2}}, \infty, ot), nt'(\infty, \infty, \mathsf{T_{D_2}}, \infty, ot)) \\
\quad \text{if } nt = \mathsf{GF} \wedge f \neq 0 \wedge fc = \emptyset \wedge d = \mathsf{open} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 & (4.17) \\
(f, 0, eng, d, (ws, ds), sw, a, (at - gft, \infty, \infty, \infty, ot), nt'(at - gft, \infty, \infty, \infty, ot)) \\
\quad \text{if } nt = \mathsf{GF} \wedge f \neq 0 \wedge fc = \emptyset \wedge d = \mathsf{open} \wedge \neg\,(ds = 0 \wedge ws = 0 \wedge sw = 0) & (4.18)
\end{cases}
\end{aligned}
$$

$$
nt'(at, dt1, dt2, gft, ot) = \begin{cases}
\mathsf{A} & \text{if } min(at, dt1, dt2, gft, ot) = at \\
\mathsf{D_1} & \text{if } min(at, dt1, dt2, gft, ot) = dt1 \\
\mathsf{D_2} & \text{if } min(at, dt1, dt2, gft, ot) = dt2 \\
\mathsf{GF} & \text{if } min(at, dt1, dt2, gft, ot) = gft \\
\mathsf{O} & \text{if } min(at, dt1, dt2, gft, ot) = ot
\end{cases}
$$

Figura 4.2: Modelo DEVS del sistema de control de un ascensor (Parte B)

$$\delta_{ext}((f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt), e, (in, x)) =$$

$$\begin{cases}
(f, n, eng, d, (ws, ds), sw, a, (\mathsf{T_A}, \mathsf{T_{D_1}}, dt2', \infty, ot'), nt'(\mathsf{T_A}, \mathsf{T_{D_1}}, dt2', \infty, ot')) \\
\qquad \text{if } x = n, n \in \mathbb{N} \wedge n \neq f \wedge eng = \mathsf{stopped} \wedge fc = \emptyset & (4.1) \\
(f + 1, fc, eng, d, (ws, ds), sw, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{fsig} \wedge eng = \mathsf{up} & (4.2) \\
(f - 1, fc, eng, d, (ws, ds), sw, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{fsig} \wedge eng = \mathsf{down} & (4.3) \\
(f, fc, eng, d, (ws, 1), sw, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{ds_{on}} \wedge eng = \mathsf{stopped})) & (4.4) \\
(f, fc, eng, d, (ws, 1), sw, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{ds_{on}} \wedge eng \neq \mathsf{stopped})) & (4.5) \\
(f, fc, eng, d, (ws, 0), sw, a, (at', \mathsf{T_{D_1}}, dt2', gft', ot'), nt'(at', \mathsf{T_{D_1}}, dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{ds_{off}} \wedge d = \mathsf{open} \wedge fc \neq \emptyset \wedge ws = 0 \wedge sw = 0 & (4.6) \\
(f, fc, eng, d, (ws, 0), sw, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{ds_{off}} \wedge (d \neq \mathsf{open} \vee fc = \emptyset \vee ws = 1 \vee sw = 1) & (4.7) \\
(f, fc, eng, d, (1, ds), sw, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{ws_{on}} \wedge eng = \mathsf{stopped} & (4.8) \\
(f, fc, eng, d, (1, ds), sw, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{ws_{on}} \wedge eng \neq \mathsf{stopped} & (4.9) \\
(f, fc, eng, d, (0, ds), sw, a, (at', \mathsf{T_{D_1}}, gft', ot'), nt'(at', \mathsf{T_{D_1}}, gft', ot')) \\
\qquad \text{if } x = \mathsf{ws_{off}} \wedge fc \neq \emptyset \wedge d = \mathsf{open} & (4.10) \\
(f, fc, eng, d, (0, ds), sw, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{ws_{off}} \wedge (fc = \emptyset \vee d \neq \mathsf{open}) & (4.11) \\
(f, fc, eng, d, (ws, ds), 1, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{s_{on}} & (4.12) \\
(f, fc, eng, d, (ws, ds), 0, a, (at', \mathsf{T_{D_1}}, dt2', gft', ot'), nt'(at', \mathsf{T_{D_1}}, dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{s_{off}} \wedge d = \mathsf{open} \wedge fc \neq \emptyset \wedge fc \neq f & (4.13) \\
(f, fc, eng, d, (ws, ds), 0, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{s_{off}} \wedge fc = \emptyset & (4.14) \\
(f, fc, eng, d, (ws, ds), 0, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{s_{off}} \wedge fc \neq \emptyset \wedge d = \mathsf{closed} & (4.15) \\
(f, fc, eng, d, (ws, ds), sw, a, (at', dt1', dt2', gft', 0), nt'(at', dt1', dt2', gft', 0)) \\
\qquad \text{if } x = \mathsf{od_{press}} \wedge d = \mathsf{closing} & (4.16) \\
(f, fc, eng, d, (ws, ds), sw, a, (at', 0, dt2', gft', ot'), nt'(at', 0, dt2', gft', ot')) \\
\qquad \text{if } x = \mathsf{cd_{press}} \wedge d = \mathsf{open} \wedge fc \neq \emptyset \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 & (4.17) \\
(f, fc, eng, d, (ws, ds), sw, a, (at', dt1', dt2', gft', ot'), nt'(at', dt1', dt2', gft', ot')) \\
\qquad \text{Otherwise} & (4.18)
\end{cases}$$

$$at' = at - e,\ dt1' = dt1 - e,\ dt2' = dt2 - e,\ gft' = gft - e,\ ot' = ot - e,$$

Figura 4.3: Modelo DEVS del sistema de control de un ascensor (Parte C)

$\lambda((f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt)) =$

$$
\begin{cases}
(f, \emptyset, \mathsf{closedoor}, \emptyset) & \text{if } nt = \mathsf{D}_1 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 & (4.1)\\
(f, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_1 \wedge (ws = 1 \vee ds = 1) \wedge sw = 0 & (4.2)\\
(ST, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_1 \wedge sw = 1 & (4.3)\\
(f, \mathsf{up}, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f \wedge a = 0 & (4.4)\\
(f, \mathsf{down}, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f \wedge a = 0 & (4.5)\\
(f, \mathsf{up}, \emptyset, \mathsf{stopalarm}) & \text{if } nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f \wedge a = 1 & (4.6)\\
(f, \mathsf{down}, \emptyset, \mathsf{stopalarm}) & \text{if } nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f \wedge a = 1 & (4.7)\\
(f, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_2 \wedge (ws = 1 \vee ds = 1) \wedge sw = 0 & (4.8)\\
(ST, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{D}_2 \wedge sw = 1 & (4.9)\\
(f, \emptyset, \mathsf{closedoor}, \emptyset) & \text{if } nt = \mathsf{GF} \wedge f \neq 0 \wedge fc = \emptyset \wedge d = \mathsf{open} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 & (4.10)\\
(f, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{GF} \wedge (f = 0 \vee ws = 1 \vee ds = 1) \wedge sw = 0 & (4.11)\\
(ST, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{GF} \wedge sw = 1 & (4.12)\\
(f, \mathsf{stop}, \mathsf{opendoor}, \emptyset) & \text{if } nt = \mathsf{O} \wedge eng \neq \mathsf{stopped} \wedge f = fc & (4.13)\\
(f, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge eng \neq \mathsf{stopped} \wedge f \neq fc & (4.14)\\
(f, \emptyset, \mathsf{opendoor}, \emptyset) & \text{if } nt = \mathsf{O} \wedge d \neq \mathsf{open} \wedge eng = \mathsf{stopped} & (4.15)\\
(ST, \mathsf{stop}, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge sw = 1 \wedge eng \neq \mathsf{stopped} & (4.16)\\
(ST, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge sw = 1 \wedge eng = \mathsf{stopped} & (4.17)\\
(f, \mathsf{up}, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc > f \wedge a = 0 & (4.18)\\
(f, \mathsf{down}, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc < f \wedge a = 0 & (4.19)\\
(f, \mathsf{up}, \emptyset, stopalarm) & \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc > f \wedge a = 1 & (4.20)\\
(f, \mathsf{down}, \emptyset, stopalarm) & \text{if } nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc < f \wedge a = 1 & (4.21)\\
(f, \emptyset, \emptyset, \emptyset) & \text{if } nt = \mathsf{O} \wedge d = \mathsf{open} \wedge sw = 0 & (4.22)\\
(f, \emptyset, closedoor, \emptyset) & \text{if } nt = \mathsf{O} \wedge d = \mathsf{open} \wedge fc \neq \emptyset \wedge a = 1 & (4.23)\\
(f, \emptyset, \emptyset, \mathsf{firealarm}) & \text{if } nt = \mathsf{A} \wedge sw = 0 & (4.24)\\
(ST, \emptyset, \emptyset, \mathsf{firealarm}) & \text{if } nt = \mathsf{A} \wedge sw = 1 & (4.25)
\end{cases}
$$

$ta((f, fc, eng, d, (ws, ds), sw, a, f(at, dt1, dt2, gft, ot))) = min(at, dt1, dt2, gft, ot)$

Figura 4.4: Modelo DEVS del sistema de control de un ascensor (Parte D)

Un estado, $s \in S$, del modelo es una tupla que representa, respectivamente, el piso en el cual se encuentra el ascensor, el piso al que debe ir (piso solicitado), el estado del motor del ascensor, el estado de su puerta, los sensores de la puerta y de peso, la alarma, los diferentes temporizadores (incluyendo un temporizador artificialmente agregado para el manejo de eventos externos) y, finalmente, una variable indicando cuál es el próximo temporizador en finalizar.

Los valores de entrada pueden ser un número (indicando un piso) o diferentes señales: indicando que el ascensor a alcanzado un piso, el sensor de peso o el sensor del piso se ha activado o desactivado, se ha presionado el botón de apertura o cierre de puerta o el interruptor ha sido encendido o apagado.

La salida, por su parte, consiste en una tupla de cuatro valores donde cada variable representa una indicación, respectivamente, para el *display*, el motor, la puerta y la

alarma. $\emptyset$ significa "no hacer nada" (o más bien, mantener la acción actual).

En cuanto a la función de transición interna, detallamos ahora algunos de los casos con el fin de poder entenderla. Por ejemplo, el caso (16) representa cuando el temporizador de la alarma termina y, por lo tanto, la alarma debe ser disparada, con los correspondientes casos (24) y (25) de la función de salida. El caso (10), por su parte, representa cuando el botón de apertura de puerta es presionado y la puerta debe abrirse, si es que ésta se está cerrando. El caso correspondiente en la función de salida es el (15).

Por otra parte, la función de transición externa es, tal vez, más intuitiva para entender cada caso. Por ejemplo, el caso (1) representa cuando el ascensor es llamado de algún piso, diferente al actual, estando el motor apagado y ningún piso ha sido llamado hasta el momento. Mientras, los casos (10) y (11) simbolizan la situación cuando el sensor de límite de peso se desactiva, habiendo algún piso solicitado o no, y con la puerta abierta o no.

4.1.2. Modelo en CML-DEVS

A continuación presentamos el mismo modelo utilizando, ahora, el lenguaje CML-DEVS.

atomic $Elevator$(params) is $(S, X, Y, \delta int, \delta ext, \lambda, ta)$ where
params is
$TGF = 1000$;
$TD1 = 5$;
$TD2 = 2$;
$TA = 10$;
end params
S is
$f : \mathbb{N}$;
$fc : \mathbb{N} \cup \{\emptyset\}$;
$eng : \{up, down, stopped\}$;
$d : \{open, closed, closing\}$;
$sens$: Boolean $\times$ Boolean;
sw, a : Boolean;
$timers$: Time $\times$ Time $\times$ Time $\times$ Time $\times$ Time;
$nt : NextTimer$;
$NextTimer == \{A, D1, D2, GF, O\}$;
end S
X is
$input : \mathbb{N} \cup \{f\_sig, ws\_on, ws\_off, ds\_on, ds\_off, od\_press, cd\_press, s\_on, s\_off\}$
end X
Y is
$out : DispOut \times EngOut \times DoorOut \times AlarmOut$;
$DispOut == \mathbb{N} \cup \{ST\}$;
$EngOut == \{up, down, stop, \emptyset\}$;
$DoorOut == \{open, close, \emptyset\}$;
$AlarmOut == \{fire, stop, \emptyset\}$;
end Y

δint$(f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt)$ is
defcases
case
$fc = \emptyset$;
$eng = stopped$;
$d = open$;
$timers = (\infty, \infty, \infty, TGF, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, TGF, \infty)$;
if
$((nt = O) \wedge (eng \neq stopped) \wedge (f = fc) \wedge (f \neq 0))$
case
$fc = \emptyset$;
$eng = stopped$;
$d = open$;
$timers = (\infty, \infty, \infty, \infty, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, \infty))$;
if
$((nt = O) \wedge (eng \neq stopped) \wedge (f = fc) \wedge (f = 0))$
case
$timers = (\infty, \infty, \infty, \infty, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, \infty)$;
if
$((nt = O) \wedge (eng \neq stopped) \wedge (f \neq fc)$
case
$eng = stopped$;
$timers = (TA, \infty, \infty, \infty, \infty)$;
$nt = Lib.nt(TA, \infty, \infty, \infty, \infty))$;
if
$((nt = O) \wedge (sw = \mathsf{true}) \wedge (eng \neq stopped))$
case
$timers = (at - ot, dt1 - ot, dt2 - ot, gft - ot, \infty)$;
$nt = Lib.nt(at - ot, dt1 - ot, dt2 - ot, gft - ot, \infty)$;
if
$((nt = O) \wedge (sw = \mathsf{true}) \wedge (eng = stopped))$
case
$eng = up$;
$timers = (\infty, \infty, \infty, \infty, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, \infty))$;
if
$((nt = O) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (d = closed) \wedge (fc \neq \emptyset) \wedge (fc > f))$
case
$eng = down$;
$timers = (\infty, \infty, \infty, \infty, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, \infty)$;
if
$((nt = O) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (d = closed) \wedge (fc \neq \emptyset) \wedge (fc < f))$
case
$d = closing$;
$a = \mathsf{false}$;
$timers = (at - ot, \infty, TD2, \infty, \infty)$;
$nt = Lib.nt(at - ot, \infty, TD2, \infty, \infty)$;
if
$((nt = O) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (d = open) \wedge (fc \neq \emptyset))$
case
$timers = (\infty, \infty, \infty, gft - ot, \infty)$;
$nt = Lib.nt(\infty, \infty, \infty, gft - ot, \infty)$;
if
$((nt = O) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (fc = \emptyset))$

case
$d = open$;
$timers = (at - ot, TD1, \infty, \infty, \infty)$;
$nt = Lib.nt(at - ot, TD1, \infty, \infty, \infty)$;
if
$((nt = O) \wedge (d = closing))$
case
$d = closing$;
$timers = (at - dt1, \infty, TD2, \infty, ot - dt1)$;
$nt = Lib.nt(at - dt1, \infty, TD2, \infty, ot - dt1)$;
if
$((nt = D1) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}))$
case
$timers = (at - dt1, \infty, \infty, \infty, ot - dt1)$;
$nt = Lib.nt(at - dt1, \infty, \infty, \infty, ot - dt1)$;
if
$((nt = D1) \wedge \neg ((ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false})))$
case
$eng = up$;
$d = closed$;
$a = \mathsf{false}$;
$timers = (\infty, \infty, \infty, \infty, ot - dt2)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, ot - dt2)$;
if
$((nt = D2) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (fc > f))$
case
$eng = down$;
$d = closed$;
$a = \mathsf{false}$;
$timers = (\infty, \infty, \infty, \infty, ot - dt2)$;
$nt = Lib.nt(\infty, \infty, \infty, \infty, ot - dt2)$;
if
$((nt = D2) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}) \wedge (fc < f))$
case
$timers = (at - dt2, \infty, \infty, \infty, ot - dt2)$;
$nt = Lib.nt(at - dt2, \infty, \infty, \infty, ot - dt2))$;
if
$((nt = D2) \wedge \neg ((ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false})))$
case
$a = \mathsf{true}$;
$timers = (\infty, dt1 - at, dt2 - at, gft - at, ot - at)$;
$nt = Lib.nt(\infty, dt1 - at, dt2 - at, gft - at, ot - at)$;
if
$(nt = A)$
case
$fc = 0$;
$d = closing$;
$timers = (\infty, \infty, TD2, \infty, ot)$;
$nt = Lib.nt(\infty, \infty, TD2, \infty, ot)$;
if
$((nt = GF) \wedge (f \neq 0) \wedge (fc = \emptyset) \wedge (d = open) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}))$
case
$fc = 0$;
$timers = (at - gft, \infty, \infty, \infty, ot)$;
$nt = Lib.nt(at - gft, \infty, \infty, \infty, ot))$;
if
$((nt = GF) \wedge (f \neq 0) \wedge (fc = \emptyset) \wedge (d = open) \wedge \neg ((ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false})))$
end defcases
end δint

δext$((f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt), \mathsf{e}, (\mathsf{port}, \mathsf{value}))$ is
defcases
case
$timers = (TA, TD1, dt2 - \mathsf{e}, \infty, ot - \mathsf{e})$;
$nt = Lib.nt(TA, TD1, dt2 - \mathsf{e}, \infty, ot - \mathsf{e})$;
if
$((\mathsf{value} \in \mathbb{N}) \wedge (value \neq f) \wedge (eng = stopped) \wedge (fc = \emptyset))$
case
$f = f + 1$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
if
$((\mathsf{value} = f\_sig) \wedge (eng = up))$;
case
$f = f - 1$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0))$;
if
$((\mathsf{value} = f\_sig) \wedge (eng = down))$
case
$sens = (ws, \mathsf{true})$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0))$;
if
$((\mathsf{value} = ds\_on) \wedge (eng = stopped))$
case
$sens = (ws, \mathsf{true})$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = ds\_on) \wedge (eng \neq stopped))$
case
$sens = (ws, \mathsf{false})$;
$timers = (at - \mathsf{e}, TD1, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$timers = Lib.nt(at - \mathsf{e}, TD1, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e}))$;
if
$((\mathsf{value} = ds\_off) \wedge (d = open) \wedge (fc \neq \emptyset) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}))$
case
$sens = (ws, \mathsf{false})$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = ds\_off) \wedge ((d \neq open) \vee (fc = \emptyset) \vee (ws = \mathsf{true}) \vee (sw = \mathsf{true})))$
case
$sens = (\mathsf{true}, ds)$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
if
$((\mathsf{value} = ws\_on) \wedge (eng = stopped))$
case
$sens = (\mathsf{true}, ds)$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = ws\_on) \wedge (eng \neq stopped))$

case
$sens = (\mathsf{false}, ds)$;
$timers = (at - \mathsf{e}, TD1, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, TD1, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = ws\_off) \wedge (fc \neq \emptyset) \wedge (d = open))$
case
$sens = (\mathsf{false}, ds)$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = ws\_off) \wedge ((fc = \emptyset) \vee (d \neq open)))$
case
$sw = \mathsf{true}$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
if
$(\mathsf{value} = s\_on)$
case
$sw = \mathsf{false}$;
$timers = (at - \mathsf{e}, TD1, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, TD1, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = s\_off) \wedge (d = open) \wedge (fc \neq \emptyset) \wedge (fc \neq f))$
case
$sw = \mathsf{false}$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = s\_off) \wedge (fc = \emptyset))$
case
$sw = \mathsf{false}$;
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
if
$((\mathsf{value} = s\_off) \wedge (fc \neq \emptyset) \wedge (d = closed))$
case
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, 0)$;
if
$((\mathsf{value} = od\_press) \wedge (d = closing))$
case
$timers = (at - \mathsf{e}, 0, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, 0, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
if
$((\mathsf{value} = cd\_press) \wedge (d = open) \wedge (fc \neq \emptyset) \wedge (ds = \mathsf{false}) \wedge (ws = \mathsf{false}) \wedge (sw = \mathsf{false}))$
default
$timers = (at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
$nt = Lib.nt(at - \mathsf{e}, dt1 - \mathsf{e}, dt2 - \mathsf{e}, gft - \mathsf{e}, ot - \mathsf{e})$;
end defcases
end δext
ta$(f, fc, eng, d, sens, sw, a, timers, nt)$ is
$\sigma = min(timers)$;
end ta

$\lambda(f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt)$ is
defcases
case $out = (f, \emptyset, closedoor, \emptyset)$;
if $((nt = D1) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}))$
case $out = (f, \emptyset, \emptyset, \emptyset)$;
if $((nt = D1) \wedge ((ws = \text{true}) \vee (ds = \text{true})) \wedge (sw = \text{false}))$
case $out = (ST, \emptyset, \emptyset, \emptyset)$;
if $((nt = D1) \wedge (sw = \text{true}))$
case $out = (f, up, \emptyset, \emptyset)$;
if $((nt = D2) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (fc > f) \wedge (a = \text{false}))$
case $out = (f, down, \emptyset, \emptyset)$;
if $((nt = D2) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (fc < f) \wedge (a = \text{false}))$
case $out = (f, up, \emptyset, stopalarm)$;
if $((nt = D2) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (fc > f) \wedge (a = \text{true}))$
case $out = (f, down, \emptyset, stopalarm)$;
if $((nt = D2) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (fc < f) \wedge (a = \text{true}))$
case $out = (f, \emptyset, \emptyset, \emptyset)$;
if $((nt = D2) \wedge ((ws = \text{true}) \vee (ds = \text{true})) \wedge (sw = \text{false}))$
case $out = (ST, \emptyset, \emptyset, \emptyset)$;
if $((nt = D2) \wedge (sw = \text{true}))$
case $out = (f, \emptyset, closedoor, \emptyset)$;
if $((nt = GF) \wedge (f \neq 0) \wedge (fc = \emptyset) \wedge (d = \text{open}) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}))$
case $out = (f, \emptyset, \emptyset, \emptyset)$;
if $((nt = GF) \wedge ((f = 0) \vee (ws = \text{true}) \vee (ds = \text{true})) \wedge (sw = \text{false}))$
case $out = (ST, \emptyset, \emptyset, \emptyset)$; if $((nt = GF) \wedge (sw = \text{true}))$
case $out = (f, stop, opendoor, \emptyset)$; if $((nt = O) \wedge (eng \neq stopped) \wedge (f = fc))$
case $out = (f, \emptyset, \emptyset, \emptyset)$; if $((nt = O) \wedge (eng \neq stopped) \wedge (f \neq fc))$
case $out = (f, \emptyset, opendoor, \emptyset)$; if $((nt = O) \wedge (d \neq open) \wedge (eng = stopped))$
case $out = (ST, stop, \emptyset, \emptyset)$; if $((nt = O) \wedge (sw = \text{true}) \wedge (eng \neq stopped))$
case $out = (ST, \emptyset, \emptyset, \emptyset)$; if $((nt = O) \wedge (sw = \text{true}) \wedge (eng = stopped))$
case $out = (f, up, \emptyset, \emptyset)$;
if $((nt = O) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (d = closed) \wedge (fc > f) \wedge (a = \text{false}))$
case $out = (f, down, \emptyset, \emptyset)$;
if $((nt = O) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (d = closed) \wedge (fc < f) \wedge (a = \text{false}))$
case $out = (f, up, \emptyset, stopalarm)$;
if $((nt = O) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (d = closed) \wedge (fc > f) \wedge (a = \text{true}))$
case $out = (f, down, \emptyset, stopalarm)$;
if $((nt = O) \wedge (ds = \text{false}) \wedge (ws = \text{false}) \wedge (sw = \text{false}) \wedge (d = closed) \wedge (fc < f) \wedge (a = \text{true}))$
case $out = (f, \emptyset, \emptyset, \emptyset)$; if $((nt = O) \wedge (d = open) \wedge (sw = \text{false}))$
case $out = (f, \emptyset, closedoor, \emptyset)$; if $((nt = O) \wedge (d = open) \wedge (fc \neq \emptyset) \wedge (a = \text{true}))$
case $out = (f, \emptyset, \emptyset, firealarm)$; if $(nt = \text{A} \wedge (sw = \text{false}))$
case $out = (ST, \emptyset, \emptyset, firealarm)$; if $(nt = \text{A} \wedge (sw = \text{true}))$
end defcases
end λ
end atomic
functions Lib is
funtion nt is
$at, dt1, dt2, gft, ot : Time \rightarrow res : NextTimer$;
defcases
case $res = A$; if $(min(at, dt1, dt2, gft, ot) = at)$
case $res = D1$; if $(min(at, dt1, dt2, gft, ot) = dt1)$
case $res = D2$; if $(min(at, dt1, dt2, gft, ot) = dt2)$
case $res = GF$; if $(min(at, dt1, dt2, gft, ot) = gft)$
case $res = O$; if $(min(at, dt1, dt2, gft, ot) = ot)$
end defcases
end function
end functions

Como puede notarse, el modelo DEVS abstracto y el modelo CML-DEVS son muy parecidos. No son idénticos, tal vez, por cuestiones de sintaxis. Sin embargo, el modelo CML-DEVS sigue manteniendo el concepto abstracto, ya que la notación es prácticamente matemática sin involucrar nociones de programación.

4.1.3. Modelos de simulación concretos

Como los modelos, tanto en Java como en C++ (para DEVS-Suite y PowerDEVS, respectivamente) son bastante extensos, no los mostramos en esta tesis, pero están disponibles en forma *online*: `http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS/models`.

El código de estos modelos concretos fue hecho a mano, pero aplicando las reglas de traducción mostradas en el Capítulo 2.

4.1.4. Aplicación de los Criterios

En esta sección, mostramos cómo se aplican los criterios del Capítulo 3 para dividir el conjunto de todas las posibles configuraciones de simulación, generando las diferentes clases de configuraciones de simulación. Por cada criterio, mostramos en este capítulo sólo algunas clases, el resto, pueden observarse en el Apéndice C.

Funciones de transición definidas por casos

El primer criterio que aplicamos para este ejemplo utiliza la definición de las funciones de transición interna y externa, generando una clase para cada caso en la definición de cada función.

Para describir las SCCs de este ejemplo, usaremos varias veces un estado genérico, $s \in S$, definido como $s = (f, fc, eng, d, (ws, ds), sw, a, (at, dt1, dt2, gft, ot), nt)$.

Entonces, algunas de las clases generadas por este criterio son:

- $IniSt_1 = \{s : s \in S \mid eng = \mathsf{stopped} \wedge fc = \emptyset\}$,
  $InPairs_1 = \{((in, n), t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+ \mid n \neq f\}$
- $IniSt_{13} = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge fc \neq f\}$,
  $InPairs_{13} = \{((in, \mathsf{s_{off}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{30} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f\}$
  $InPairs_{30} = \{(\tau, 0)\}$

Con la primera clase, se simula el llamado del ascensor de un piso diferente del actual, estando el motor apagado y no habiendo otro piso llamado. Esto corresponde al primer caso de la función de transición externa.

Conjuntos definidos por extensión

En este ejemplo, hay seis conjuntos definidos por extensión, *Engine*, *Door*, *Sensors*, *Alarm*, *Switch* y *NextTimer*. Además, X es la unión de un conjunto infinito y otro definido por extensión.

Siendo que estos conjuntos tienen un número relativamente pequeño de elementos (considerando solamente el conjunto finito de la unión que forma X), definimos una SCC por cada uno de estos elementos, como el criterio lo indica. Algunas de estas son:

- $IniSt_{36} = \{s : s \in S\}$,
  $InPairs_{36} = \{((in, n), t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+\}$
- $IniSt_{49} = \{s : s \in S \mid d = \mathsf{open}\}$,
  $InPairs_{49} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{57} = \{s : s \in S \mid a = 1\}$,
  $InPairs_{57} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{59} = \{s : s \in S \mid sw = 1\}$,
  $InPairs_{59} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

Con la tercer clase, se simularía el arribo de un evento externo o alguna transición interna estando la alarma disparada.

Particiones estándar

Ahora aplicamos el criterio de particiones estándar sobre los operadores $<$, $>$, $+$ y $-$. Para estos operadores se puede utilizar la misma partición estándar definida en la sección 3.4.4. Sin embargo, algunos casos los debemos ignorar ya que las variables involucradas no pueden ser menores a cero ($f \geq 0$ y $fc \geq 0$).

Las siguientes clases, son algunas de las resultantes al aplicar este criterio. Las primeras dos se refieren a la ocurrencia de los operadores $<$ y $>$ en la definición de δ_{int} y las últimas dos, a la ocurrencia de los operadores $+$ y $-$ en dicha función.

- $IniSt_{67} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc = f \wedge f = 0\}$,
  $InPairs_{67} = \{(\tau, 0)\}$
- $IniSt_{72} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc > f \wedge f = 0\}$,
  $InPairs_{72} = \{(\tau, 0)\}$
- $IniSt_{76} = \{s : s \in S \mid nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f \wedge f > 0\}$,
  $InPairs_{76} = \{(\tau, 0)\}$
- $IniSt_{77} = \{s : s \in S \mid nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f \wedge fc = 0\}$,
  $InPairs_{77} = \{(\tau, 0)\}$

La segunda clase, indica la simulación de una transición interna, en la situación en la que están los sensores y el interruptor desactivado, la puerta cerrada, hay un piso llamado, más alto que el actual, y el ascensor está actualmente en planta baja.

Particiones del tiempo

Finalmente, aplicamos este criterio particular, teniendo en cuenta la relación entre el tiempo transcurrido, e, y las variables usadas como temporizadores, en la tupla: *timers*. Considerando los valores que estas variables asumen, los intervalos de tiempo relevantes son: $[0, \mathsf{T_{D_1}}]$, $[0, \mathsf{T_{D_2}}]$, $[0, \mathsf{T_A}]$, $[0, \mathsf{T_{GF}}]$, $[\mathsf{T_{D_1}}, \mathsf{T_{D_2}}]$, $[\mathsf{T_{D_1}}, \mathsf{T_A}]$, $[\mathsf{T_{D_1}}, \mathsf{T_{GF}}]$, $[\mathsf{T_{D_2}}, \mathsf{T_A}]$, $[\mathsf{T_{D_2}}, \mathsf{T_{GF}}]$, $[\mathsf{T_A}, \mathsf{T_{GF}}]$ (asumiendo $\mathsf{T_{D_1}} < \mathsf{T_{D_2}} < \mathsf{T_A} < \mathsf{T_{GF}}$).

Por lo tanto, los instantes de tiempo, t relevantes para simular cada evento son: $t = 0$, $0 < t < \mathsf{T_{D_1}}$, $t = \mathsf{T_{D_1}}$, $\mathsf{T_{D_1}} < t < \mathsf{T_{D_2}}$, $t = \mathsf{T_{D_2}}$, $\mathsf{T_{D_2}} < t < \mathsf{T_A}$, $t = \mathsf{T_A}$, $\mathsf{T_A} < t < \mathsf{T_{GF}}$, $T = \mathsf{T_{GF}}$ y $T > \mathsf{T_{GF}}$.

Entonces, algunas de las clases generadas por este criterio son:

- $IniSt_{82} = \{s : s \in S\}$
  $InPairs_{82} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid \mathsf{T_{D_1}} < t < \mathsf{T_{D_2}}\}$
- $IniSt_{85} = \{s : s \in S\}$
  $InPairs_{85} = \{(x, \mathsf{T_A}) : x \in X \cup \{\tau\}\}$
- $IniSt_{87} = \{s : s \in S\}$
  $InPairs_{87} = \{(x, \mathsf{T_{GF}}) : x \in X \cup \{\tau\}\}$

Con la primera clase se quiere simular la ocurrencia de algún evento o la ejecución de una transición interna, es decir que suceda algo, después de que la puerta empiece a cerrarse y antes de que se cierre completamente.

Combinando clases

Una vez que todos los criterios se han aplicado, hacemos la intersección de las clases resultantes obteniendo otras nuevas y refinadas. Aquí solamente mostramos algunas de las clases producidas por estas combinaciones. Por ejemplo, si queremos testear la situación cuando alguien llama el ascensor desde un piso, estando el motor detenido y con la puerta abierta, debemos intersecar SCC_1 y SCC_{49} obteniendo:

- $SCC_1 \cap SCC_{49}$:

$$
\begin{aligned}
IniSt_{89} &= IniSt_1 \cap IniSt_{49} = \\
&= \{s : s \in S \mid eng = \mathsf{stopped} \wedge fc = \emptyset\} \cap \{s : s \in S \mid d = \mathsf{open}\} = \\
&= \{s : s \in S \mid eng = \mathsf{stopped} \wedge fc = \emptyset \wedge d = \mathsf{open}\}, \\
InPairs_{89} &= InPairs_1 \cap InPairs_{49} = \\
&= \{((in, n), t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+ \mid n \neq f\} \cap \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\} = \\
&= \{(n, t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+ \mid n \neq f\}
\end{aligned}
$$

Acá se puede observar el efecto de la conmutatividad de la intersección, siendo $SCC_1 \cap SCC_{49}$ igual a $SCC_{49} \cap SCC_1$

Otra combinación interesante resulta de SCC_{13} y SCC_{57}, dónde la clase resultante representa la situación en la que se apaga el interruptor, estando la alarma disparada, algún piso pedido (distinto al actual) y la puerta abierta:

- $SCC_{13} \cap SCC_{57}$:

  $IniSt_{102} = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge fc \neq f \wedge a = 1\}$,
  $InPairs_{102} = \{((in, \mathsf{s_{off}}), t) : t \in \mathbb{R}_0^+\}$

Con las siguientes dos clases, podrían encontrarse errores en las reglas de desempate, cuando este modelo se acople con otros, o en la implementación del simulador. Estas clases se obtienen de combinar SCC_{36} con SCC_{87}, por un lado, y SCC_{13}, SCC_{59} y SCC_{85} por el otro. Por ejemplo, las clases definida por $IniSt_{91}$ y $InPairs_{91}$ simulan el caso cuando el ascenso es llamado en el mismo instante en el que finaliza el temporizador "volver a plata baja".

- $SCC_{36} \cap SCC_{87}$:

  $IniSt_{103} = \{s : s \in S\}$,
  $InPairs_{103} = \{((in, n), \mathsf{T_{GF}}) : n \in \mathbb{N}\}$

- $SCC_{13} \cap SCC_{59} \cap SCC_{85}$:

  $IniSt_{104} = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge fc \neq f \wedge sw = 1\}$,
  $InPairs_{104} = \{((in, \mathsf{s_{off}}), \mathsf{T_A})\}$

4.2. Máquina expendedora de gaseosas

El segundo caso de estudio presentado, consiste en el sistema de control de una máquina expendedora de gaseosas. La máquina acepta monedas de \$0.25, \$0.50 y \$1. As su vez, da vuelto optimizándolo, i.e. dando la menor cantidad de monedas posible. El expendio de las gaseosas en sí, i.e. la situación en la que la máquina entrega la gaseosa al usuario, no se incluye en el modelo.

La máquina tiene dos tipos de gaseosas, normal y dietética, de diferentes precios, y el sistema que controla las operaciones debe cumplir con los siguientes requerimientos:

- Durante una operación, si luego de T_{ret} unidades de tiempo no se inserta ninguna moneda o no se selecciona ninguna gaseosa, la máquina devuelve todas las monedas introducidas hasta el momento durante la operación.

- Los precios de las gaseosas se incrementan a medida que va pasando el tiempo. Cada T_{incr} unidades de tiempo ambos precios aumentan \$0.25.

- Si el dinero devuelto no es retirado por el usuario luego de T_{chg} unidades de tiempo, la máquina lo recupera.

- La máquina tiene una pequeña pantalla donde indica el monto de dinero introducido durante la operación, o el cambio que debe dar la máquina luego de seleccionar una gaseosa.

- En cualquier momento de la operación, pero antes de seleccionar una gaseosa, el usuario puede cancelar dicha operación y la máquina devuelve las monedas insertadas.

Como puede verse, algunos requerimientos temporales (en particular el segundo) fueron incluidos artificialmente a fin de tener más variables relacionadas con el tiempo interactuando en el modelo, permitiendo de esta forma, aumentar las particiones obtenidas al aplicar los criterios del Capítulo 3.

4.2.1. Modelo DEVS abstracto

Las Figura 4.5 y 4.6 muestran un posible modelo DEVS para el sistema de control de la máquina expendedora de gaseosas descripto en la sección anterior.

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

$S = MachState \times Display \times OpTime \times NormalPrice \times DietPrice \times IncrTime \times MoneyStorage \times OperationMoney \times MoneyReturned$

where:

$MachState = \{\text{idle}, \text{operating}, \text{finishOp}, \text{cancelOp}, \text{waitRetChange}\}$

$OpTime = \mathbb{R}_0^+ \cup \{\infty\}$

$Display = IncrTime = NormalPrice = DietPrice = \mathbb{R}_0^+$

$MoneyStorage = OperationMoney = MoneyReturned = Coins1d \times Coins50c \times Coins25c$

where:

$Coins1d = Coins50c = Coins25c = \mathbb{N}_0$

$X = \{(in, x) : x \in \{25, 50, 100, \text{getNormal}, \text{getDiet}, \text{cancel}, \text{moneyRetreated}\}\}$

$Y = \{(out, y) : y \in Display \times MoneyReturned\}$

Figura 4.5: Modelo DEVS del sistema de control de una máquina expendedora de gaseosas (Parte A)

En este modelo, un estado $s \in S$ es una tupla donde cada variable representa, respectivamente, el estado de la máquina, la pantalla, un temporizador interno, los precios actuales (normal y dietética), el temporizador que controla el incremento del precio, el dinero almacenado en la máquina, las monedas insertadas en la operación actual y el vuelto.

Los valores de entrada representan, la denominación de cada moneda, el pedido de una gaseosa normal o una dietética, la cancelación de la operación actual y la señal que indica que el vuelto ha sido retirado de la máquina.

La función de transición externa tiene un caso por cada valor de entrada diferente a una moneda (casos 2, 3, 4 y 5) y un caso para todas las monedas (1). Mientras, la función de transición interna tiene un caso por cada estado de la máquina ($m \in MachState$), para el "temporizador funcional", (casos 1, 2, 3, 4 y 5) y un caso (6) para el temporizador de incremento de precios.

La salida consiste en un par ordenado indicando que se debe mostrar en la pantalla y cuanto dinero debe ser devuelto.

$\delta_{ext}((m, d, ot, np, dp, it, ms, om, mr), e, (in, x)) =$

$$\begin{cases} (\mathsf{operating}, d + x, 0, np, dp, it - e, ms, om \oplus x, (0,0,0)) & \text{if } x \in \{100, 50, 25\} \wedge m \in \{\mathsf{idle}, \mathsf{operating}\} \quad (4.1) \\ (\mathsf{finishOp}, d - np, 0, np, dp, it - e, ms \oplus om, (0,0,0), (d - np) \oslash (ms \oplus om)) & \text{if } x = \mathsf{getNormal} \wedge d \geq np \quad (4.2) \\ (\mathsf{finishOp}, d - dp, 0, np, dp, it - e, ms \oplus om, (0,0,0), (d - dp) \oslash (ms \oplus om)) & \text{if } x = \mathsf{getDiet} \wedge d \geq dp \quad (4.3) \\ (\mathsf{cancelOp}, d, 0, np, dp, it - e, ms, (0,0,0), om) & \text{if } x = \mathsf{cancel} \quad (4.4) \\ (\mathsf{idle}, 0, 0, np, dp, it - e, ms, (0,0,0), (0,0,0)) & \text{if } x = \mathsf{moneyRetreated} \quad (4.5) \end{cases}$$

$\delta_{int}((m, d, ot, np, dp, it, ms, om, mr)) =$

$$\begin{cases} (\mathsf{operating}, d, \mathsf{T_{ret}}, np, dp, it - ot, ms, om, (0,0,0)) & \text{if } m = \mathsf{operating} \wedge ot < it \quad (4.1) \\ (\mathsf{waitRetChange}, d, \mathsf{T_{chg}}, np, dp, it - ot, ms \ominus (d \oslash ms), om, d \oslash ms) & \text{if } m = \mathsf{finishOp} \wedge ot < it \quad (4.2) \\ (\mathsf{waitRetChange}, d, \mathsf{T_{chg}}, np, dp, it - ot, ms, (0,0,0), mr) & \text{if } m = \mathsf{cancelOp} \wedge ot < it \quad (4.3) \\ (\mathsf{idle}, 0, \infty, np, dp, it - ot, ms \oplus mr, (0,0,0), (0,0,0)) & \text{if } m = \mathsf{waitRetChange} \wedge ot < it \quad (4.4) \\ (\mathsf{idle}, 0, \infty, np, dp, it - ot, ms, (0,0,0), (0,0,0)) & \text{if } m = \mathsf{idle} \wedge ot < it \quad (4.5) \\ (m, d, ot - it, np + 0.25, dp + 0.25, \mathsf{T_{incr}}, ms, om, mr) & \text{if } it \leq ot \quad (4.6) \end{cases}$$

$\lambda((m, d, ot, np, dp, it, ms, om, mr)) = (out, (d, ms))$

$ta((m, d, ot, np, dp, it, ms, om, mr)) = min(ot, it)$

$(coins1d, coins50c, coins25c) \oplus x =$

$$\begin{cases} (coins1d + x, coins50c, coins25c) & \text{if } x = 100 \\ (coins1d, coins50c + x, coins25c) & \text{if } x = 50 \\ (coins1d, coins50c, coins25c + x) & \text{if } x = 25 \end{cases}$$

$(coins1d, coins50c, coins25c) \oplus (coins1d', coins50c', coins25c') =$
$(coins1d + coins1d', coins50c + coins50c', coins25c + coins25c')$

$(coins1d, coins50c, coins25c) \ominus (coins1d', coins50c', coins25c') =$
$(coins1d - coins1d', coins50c - coins50c', coins25c - coins25c')$

$d \oslash (coins1d, coins50c, coins25c) = (coins1d', coins50c', coins25c')$,
where:
$coins1d' = min(coins1d, d \text{ div } 1)$
$coins50c' = min(coins50c, (d - coins1d') \text{ div } 0.50)$
$coins25c' = min(coins25c, (d - coins1d' - coins50c') \text{ div } 0.25)$

Figura 4.6: Modelo DEVS del sistema de control de una máquina expendedora de gaseosas (Parte B)

4.2.2. Modelo en CML-DEVS

Al igual que con el modelo del ascensor, presentamos ahora el modelo DEVS abstracto descripto utilizando CML-DEVS.

atomic $Sodas$(params) is $< \mathsf{S}, \mathsf{X}, \mathsf{Y}, \delta_{\mathsf{int}}, \delta_{\mathsf{ext}}, \lambda, \mathsf{ta} >$ where
params is
$Tret = 6$;
$Tchg = 12$;
$Tincr = 129600$;
end params
S is
$m : \{iddle, operating, finishPp, cancelOp, waitRetChange\}$;
$d, it, np, dp : \mathbb{R}$;
ot : Time;
$ms, om, mr : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
end s

X is
$in : \{25, 50, 100, getNormal, getDiet, cancel, moneyRetreated\};$
end X
Y is
$out : \mathbb{R} \times (\mathbb{N} \times \mathbb{N} \times \mathbb{N});$
end Y
δext$((m, d, it, np, dp, ot, ms, om, mr), e,$ (port, value)) is
defcases
if $((\text{value} \in \{100, 50, 25\}) \wedge (m \in \{idle,$ operating$\})) \Rightarrow$
$m = operating;$
$d = d +$ value;
$ot = 0;$
$it = it -$ e;
$om = Library.oplus(om, x);$
$mr = (0, 0, 0);$
if $((\text{value} = getNormal) \wedge (d \geq np)) \Rightarrow$
$m = finishOp;$
$d = d - np;$
$ot = 0;$
$it = it -$ e;
$ms = Library.oplus(ms, om);$
$om = (0, 0, 0);$
$mr = oslash((d - np, Library.oplus(ms, om));$
if $((\text{value} = getDiet) \wedge (d \geq dp)) \Rightarrow$
$m = finishOp;$
$d = d - dp;$
$ot = 0;$
$it = it -$ e
$ms = Library.oplus(ms, om);$
$om = (0, 0, 0);$
$mr = Library.oslash((d - dp, Library.oplus(ms, om));$
if $(\text{value} = cancel) \Rightarrow$
$d = cancelOp;$
$ot = 0;$
$it = it -$ e;
$om = (0, 0, 0);$
$mr = om;$
if $(\text{value} = moneyRetreated) \Rightarrow$
$m = idle;$
$d = 0;$
$ot = 0;$
$it = it -$ e;
$om = (0, 0, 0);$
$mr = (0, 0, 0));$
end defcases
end δext
δint$((m, d, it, np, dp, ot, ms, om, mr))$ is
defcases
if $((m = operating) \wedge (ot < it)) \Rightarrow$
$m = operating;$
$ot = Tret;$
$it = it - ot;$
$mr = (0, 0, 0));$
if $((m = finishOp) \wedge (ot < it)) \Rightarrow$
$m = waitRetChange;$
$ot = Tchg;$
$it = it - ot;$
$ms = Library.ominus(ms, Library.oslash(d, ms));$
$mr = Library.oslash(d, ms);$

if $((m = cancelOp) \wedge (ot < it)) \Rightarrow$
$m = waitRetChange$;
$ot = Tchg$;
$it = it - ot$;
$om = (0, 0, 0)$;
if $((m = waitRetChange) \wedge (ot < it)) \Rightarrow$
$m = idle$;
$d = 0$;
$ot = \infty$;
$it = it - ot$;
$ms = Library.oplus(ms, mr)$;
$om = (0, 0, 0)$;
$rm = (0, 0, 0))$;
if $((m = idle) \wedge (ot < it)) \Rightarrow$
$d = 0$;
$ot = \infty$;
$it = it - ot$;
$om = (0, 0, 0)$;
$mr = (0, 0, 0))$;
if $(it \leq ot) \Rightarrow$
$ot = ot - it$;
$np = np + 0.25$;
$dp = dp + 0.25$;
$it = Tincr$;
end defcases
end δint
$\lambda((m, d, it, np, dp, ot, ms, om, mr))$ is
$out = (d, ms)$;
end λ
ta$((m, d, it, np, dp, ot, ms, om, mr))$ is
$\sigma = min(ot, it)$;
end ta
end atomic
functions $Library$ is
function $oplusx$ is
$m : \mathbb{N} \times \mathbb{N} \times \mathbb{N}, x : \mathbb{N} \rightarrow res : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$;
defcases
if $(x = 100) \Rightarrow res = (m.1 + x, m.2, m.3)$;
if $(x = 50) \Rightarrow res = (m.1, m.2 + x, m.3)$;
if $(x = 25) \Rightarrow res = (m.1, m.2, m.3 + x)$;
end defcases
end oplus
function $oplus$ is
$m1, m2 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow res : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$;
$res = (m1.1 + m2.1, m1.2 + m2.2, m1.3 + m2.3)$;
end oplus
function $ominus$ is
$m1, m2 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow res : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$;
$res = (m1.1 - m2.1, m1.2 - m2.2, m1.3 - m2.3)$;
end ominus
function $oslash$ is
$d : \mathbb{R}, m : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow res : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$;
$res.1 = min(m.1, d/1)$;
$res.2 = min(m.2, (d - min(m.1, d/1))/0.50)$;
$res.3 = min(m.3, (d - min(m.1, d/1) - min(m.2, (d - min(m.1, d/1))))/0.25)$;
end oslash
end functions

4.2.3. Modelos de simulación concretos

Nuevamente, al ser el código completo de los modelos en Java y en C++ muy extenso, se omite en este manuscrito pero puede encontrarse en `http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS/models`.

El código de estos modelos concretos fue hecho a mano aplicando las reglas de traducción mostradas en el Capítulo 2.

4.2.4. Aplicación de los criterios

Como en el ejemplo anterior, vamos aplicando los criterios generando las diferentes SCCs y luego las combinamos obteniendo otras nuevas más refinadas. Nuevamente, mostramos sólo algunas clases en este capítulo. El resto de las clases generadas para este ejemplo pueden observarse en el Apéndice D. También aquí usamos un estado $s \in S$ genérico para describir las SCCs del ejemplo, definido como $s = (m, d, ot, np, dp, it, ms, om, mr)$.

Funciones de transición definidas por caso

Algunas del as clases generadas al aplicar este criterio son:

- $IniSt_3 = \{s : s \in S \mid d \geq dp\}$,
  $InPairs_3 = \{((in, \mathsf{getDiet}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_7 = \{s : s \in S \mid m = \mathsf{finishOp} \wedge ot < it\}$,
  $InPairs_7 = \{(\tau, 0)\}$
- $IniSt_{11} = \{s : s \in S \mid m = \mathsf{idle} \wedge ot \leq it\}$,
  $InPairs_{11} = \{(\tau, 0)\}$

Particiones estándar

Aplicamos ahora el criterio particiones estándar sobre los operadores $\geq$, $+$, $-$, $\oplus$, $\ominus$ y $\oslash$.

- $\geq$ aparece dos veces ($d \geq np$ y $d \geq dp$) y la partición estándar para este operador es igual a la partición estándar de $<$, descripto en la sección 3.4.4. Las siguientes clases son algunas de las resultantes al aplicar este criterio. Las primeras dos corresponden a la aplicación del criterio sobre la comparación $d \geq np$ y la última, sobre $d \geq dp$:

  - $IniSt_{12} = \{s : s \in S \mid d = np = 0\}$,
    $InPairs_{12} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

- $IniSt_{14} = \{s : s \in S \mid d = 0 \wedge np > 0\}$,
  $InPairs_{14} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{19} = \{s : s \in S \mid d > 0 \wedge dp > 0\}$,
  $InPairs_{19} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

- $\oplus$, por definición, está basado en $+$, por lo tanto, puede tener la misma partición estándar. Sin embargo, siendo que involucra sólo elementos mayores a cero, no se pueden agregar nuevas particiones significativas. Excepto si se deseara simular aquellos casos en donde la implementación de esas operaciones en el modelo concreto puedan arrojar algún error, e.g. errores de *overflow*. En este caso, los errores no serían propiamente del modelo sino de sus implementaciones. Esto, obviamente, está relacionado al testing de la implementación y no a la validación del modelo.

- En aquellos casos en los que el operador $-$ interactúa con variables usadas para la representación del tiempo o temporizadores, las clases correspondientes serán descriptas más tarde, en el criterio **partición del tiempo**. Algunas de las clases para las restantes ocurrencias del operador $-$ son:

  - $IniSt_{22} = \{s : s \in S \mid 0 < d < np\}$,
    $InPairs_{22} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
  - $IniSt_{26} = \{s : s \in S \mid 0 < d < dp\}$,
    $InPairs_{26} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
  - $IniSt_{27} = \{s : s \in S \mid 0 < dp < d\}$,
    $InPairs_{27} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

- Con el operador $\oslash$ se puede aplicar el criterio **propagación de dominios** ya que está formado por dos operadores más simples. A pesar que $-$ y / tienen la misma partición estándar, las operaciones involucran diferentes variables. Por lo tanto, se pueden obtener nuevas clases al aplicar la propagación de dominios.
  Algunas de las clases generadas son:

  - $IniSt_{28} = \{s : s \in S \mid d = 0\}$,
    $InPairs_{28} = \{(x, t) : x \in X, t \in \mathbb{R}_0^+\}$
  - $IniSt_{29} = \{s : s \in S \mid 0 < d < coins1d \wedge coins25c = d - coins1d' - coins50c' = 0\}$,
    $InPairs_{29} = \{(x, t) : x \in X, t \in \mathbb{R}_0^+\}$
  - $IniSt_{31} = \{s : s \in S \mid 1 < d < coins1d \wedge 0.50 < d - coins1d' < coins50c\}$,
    $InPairs_{31} = \{(x, t) : x \in X, t \in \mathbb{R}_0^+\}$

Conjuntos definidos por extensión

En este ejemplo tenemos dos conjuntos definidos por extensión, X y $MachState$. Al tener estos conjuntos un número pequeño de elementos, definimos una SCC para cada elemento. Algunas de las clases generadas:

- $IniSt_{32} = \{s : s \in S \mid m = \mathsf{operating}\}$,
  $InPairs_{32} = \{(x, t) : x \in X \cup \{\tau\} \wedge t \in \mathbb{R}_0^+\}$
- $IniSt_{34} = \{s : s \in S \mid m = \mathsf{cancelOp}\}$,
  $InPairs_{34} = \{(x, t) : x \in X \cup \{\tau\} \wedge t \in \mathbb{R}_0^+\}$
- $IniSt_{41} = \{s : s \in S\}$,
  $InPairs_{41} = \{((in, \mathsf{getNormal}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{42} = \{s : s \in S\}$,
  $InPairs_{42} = \{((in, \mathsf{getDiet}), t) : t \in \mathbb{R}_0^+\}$

Particiones del tiempo

En este ejemplo, las variables que interactúan con el tiempo son ot e it, además del avance de tiempo e. Nuevamente, considerando los valores que estas variables asumen definimos los intervalos de tiempo claves: $[0, it]$, $[0, ot]$, $[it, ot]$ (cuando $it < ot$) y $[ot, it]$ (cuando $ot < it$). Además, un instante de tiempo clave es: $t = ot = it$, esto es, en el mismo instante en que el se deben aumentar los precios está programada otra transición interna. Algunas clases generadas:

- $IniSt_{46} = \{s : s \in S \mid it > 0\}$,
  $InPairs_{46} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid 0 < t < it\}$
- $IniSt_{53} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{53} = \{(x, it) : x \in X \cup \{\tau\}\}$
- $IniSt_{59} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{59} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid ot < t < it\}$
- $IniSt_{62} = \{s : s \in S \mid ot = it\}$,
  $InPairs_{62} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t < ot\}$
- $IniSt_{64} = \{s : s \in S \mid ot = it\}$,
  $InPairs_{64} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > ot\}$

Combinando clases

Entonces, una vez que los criterios han sido aplicados, hacemos la conjunción lógica entre las clases obtenidas, conservando aquellas distintas de la clase vacía.

Algunas clases obtenidas por este medio:

- $SCC_3 \cap SCC_{19}$:
  $IniSt_{65} = \{s : s \in S \mid d \geq dp \wedge d > 0 \wedge dp > 0\}$,
  $InPairs_{65} = \{((in, \mathsf{getDiet}), t) : t \in \mathbb{R}_0^+\}$
- $SCC_{22} \cap SCC_{41}$:
  $IniSt_{66} = \{s : s \in S \mid 0 < d < np\}$,
  $InPairs_{66} = \{((in, \mathsf{getNormal}), t) : t \in \mathbb{R}_0^+\}$
- $SCC_{27} \cap SCC_{42} \cap SCC_{62}$:
  $IniSt_{67} = \{s : s \in S \mid 0 < dp < d \wedge it = ot\}$,
  $InPairs_{67} = \{((in, \mathsf{getDiet}), t) : t \in \mathbb{R}_0^+ \mid t < ot\}$

Aquí puede observarse cómo se podría encontrar un error en el modelo con la clase definida por $IniSt_{67}$ y $InPairs_{67}$, siendo que esta representa a un caso que no ha sido definido en el modelo, esto es, cuando se inserta menos plata que el precio de la gaseosa solicitada.

Capítulo 5

Conclusiones generales y trabajo futuro

Con este capítulo se concluye la tesis resumiendo brevemente las contribuciones que presenta y las conclusiones que se desprenden de estos.

También discutimos los temas que quedan abiertos para ser abordados y las futuras líneas de investigación referentes a los mismos.

5.1. Conclusiones generales

Esta tesis se enmarca dentro del proceso de modelado y simulación de sistemas a eventos discretos utilizando el formalismo DEVS. La tesis presenta dos contribuciones originales para la comunidad de M&S. La primera, es un lenguaje que permite la descripción de modelos DEVS en forma abstracta, independiente de cualquier implementación y plataforma, basado en nociones lógicas y matemáticas. La segunda, una técnica para validar rigurosa y sistemáticamente estos modelos contra los requerimientos, a través de simulaciones.

El lenguaje que se presenta en el Capítulo 2, CML-DEVS, permite describir modelos DEVS preservando la idea abstracta original del formalismo propuesto por Zeigler. Los modelos se pueden describir utilizando CML-DEVS sin tener que conocer el lenguaje de ninguna herramienta de M&S en particular, y más aún, sin que sean necesarios conocimientos o habilidades de programación, ya que CML-DEVS se abstrae de estos conceptos utilizando nociones basadas en la matemática y la lógica de uso cotidiano.

También mostramos cómo los modelos CML-DEVS se traducen en forma automática al lenguaje de modelado de dos herramientas de M&S diferentes, PowerDEVS y DEVS-Suite. Entonces, el especialista puede modelar el sistema en forma abstracta y luego simularlo con la herramienta que desee. Para realizar esta tarea, se puede construir un compilador multi-objetivo que implemente las reglas de traducción pre-

sentadas.

Hemos descrito la sintaxis de CML-DEVS a través de su EBNF y explicada mediante varios ejemplos. La semántica la detallamos en función del código que debe generarse en el lenguaje se las herramientas de M&S mencionadas.

Las principales ventajas que esta contribución presenta son, como ya mencionamos, la posibilidad de describir modelos DEVS sin la la necesidad conocer algún lenguaje de simulación ni tener habilidades de programación; y poder simularlo luego con la herramienta que se desee. Además, el mantenimiento y modificación de los modelos es muy accesible ya que no hay que examinar un código fuente de ningún lenguaje de programación. Simplemente se modifica el modelo abstracto y se vuelve a generar en forma automática el modelo concreto para ser simulado. Como consecuencia, permite la colaboración entre diferentes investigadores, industrias, o comunidades en general que trabajen con modelos DEVS donde, en la actualidad, cada una usa su propia implementación del formalismo y los modelos de unos no pueden ser simulados con las herramientas de los otros. Finalmente, posibilita la automatización del proceso de validación de modelos DEVS vía simulaciones, referente a la segunda contribución de esta tesis.

El segundo aporte, enunciado y analizado en el Capítulo 3, introduce una nueva metodología para validar modelos DEVS en forma rigurosa y sistemática, siguiendo un conjunto de criterios formalmente definidos. Por tanto, formalizando el proceso de validación de estos modelos contra los requerimientos vía simulaciones. Esta metodología está inspirada en técnicas de ingeniería de software, más precisamente en el área de testing basado en modelos. Estas técnicas fueron probadas en dicha área con muy buenos resultados. Entonces, basados en estas técnicas presentamos una familia de criterios para seleccionar del conjunto infinito de posibles configuraciones de simulación, las configuraciones más significativas y relevantes, sin dejar características o funcionalidades del modelo sin cubrir y dejando afuera aquellas simulaciones redundantes o innecesarias. Es decir, optimizando el conjunto de simulaciones a llevar a cabo para validar el modelo.

Las ventajas de validar un modelo DEVS siguiendo esta metodología son, en primer lugar, que no se necesita un experto del dominio para validar el modelo correspondiente. Esto se debe a que dicha validación se hace siguiendo una rigurosa familia de criterios que analiza la estructura lógica y matemática del modelo, independientemente del área o dominio relativo al modelo. La segunda ventaja importante es la posibilidad de automatizar (en gran medida) este proceso. La consecuencia inmediata es el ahorro de tiempo y recursos empleados durante el proceso general de modelado, validación y desarrollo de sistemas de eventos discretos.

Ambas contribuciones de la tesis fueron aplicadas a dos casos de estudio para mostrar la utilidad y funcionamiento de las mismas. Si bien estos ejemplos son relativamente

pequeños, nos parecen suficientes para mostrar cómo se puede describir en forma abstracta un modelo DEVS, traducirlo a modelos concretos en forma automática para poder simularlos y validarlos con la técnica presentada. Poniendo de esta forma en relieve el aporte que presentamos para la comunidad de M&S.

5.2. Temas abiertos y trabajo futuro

Dentro de los temas que quedan abiertos y creemos merecen ser abordados podemos mencionar, primero, el desarrollo del compilador multi-objetivo para la traducción automática de los modelos CML-DEVS. Básicamente consiste en el desarrollo de un analizador sintáctico y semántico del lenguaje y en la implementación de las reglas de traducción ya presentadas. Existen herramientas como Xtext [63], que generan estos analizadores sintácticos y semánticos a partir de la BNF o EBNF de un lenguaje.

El segundo punto, siguiendo esta línea, sería la implementación de la técnica de validación desarrollando una herramienta que, a partir de un modelo DEVS abstracto, asista al ingeniero en la validación del modelo, generando en forma automática el conjunto de configuraciones de simulación.

Entonces, la idea global es desarrollar un conjunto de herramientas proporcionando un entorno completo para el modelado y validación de modelos DEVS integrando las implementaciones antes mencionadas.

En una segunda etapa, se pretende atacar los demás temas abiertos, extendiendo tanto CML-DEVS como la técnica de validación a las diferentes extensiones y variantes del formalismo DEVS. Además de generar el marco teórico necesario, esto es, la definición del lenguaje CML-DEVS extendido y de los nuevos criterios de simulación; se debe integrar al entorno de modelado y validación, las implementaciones correspondientes. En esta misma etapa, es posible el desarrollo de nuevas reglas de traducción, para convertir modelos CML-DEVS en modelos que puedan ser simulados por el resto de las herramientas de M&S enunciadas en el Capítulo 1. Por otra parte, se puede quizás enriquecer aun más CML-DEVS, sin modificar su concepto abstracto, soportando nuevas estructuras matemáticas como por ejemplo funciones parciales y relaciones binarias.

Apéndice A

EBNF del lenguaje CML-DEVS

En este apéndice, mostramos formalmente la sintaxis completa de CML-DEVS a través de su *Extended Bakus-Naur Normal Form* (EBNF), formal normal de Bakus extendida. Como en casi toda EBNF, los términos entre corchetes angulares, ⟨ ⟩, son no-terminales, y existen reglas de producción para reemplazarlos por terminales u otros no terminales. Las expresiones encerradas entre llaves, { }, significan cero o más repeticiones de dicha expresión. Para identificar a las llaves que forman parte de la sintaxis del lenguaje, las encerramos entre comillas, "{" y "}". Finalmente, las expresiones entre corchetes [ ], son expresiones opcionales.

$\langle CML-DEVS \rangle$ ::= $\langle atomic \rangle$
[$\langle functions \rangle$]
[$\langle simulate \rangle$]

$\langle atomic \rangle$ ::= atomic $\langle id \rangle$[(P)] <[$\langle S \rangle$,][$\langle X \rangle$,][$\langle Y \rangle$,][$\langle \delta int \rangle$,][$\langle \delta ext \rangle$,][$\langle \lambda \rangle$,][$\langle ta \rangle$]> is
[$\langle P \rangle$]
[$\langle S \rangle$]
[$\langle X \rangle$]
[$\langle Y \rangle$]
[$\langle \delta int \rangle$]
[$\langle \delta ext \rangle$]
[$\langle \lambda \rangle$]
[$\langle ta \rangle$]
end atomic

$\langle id \rangle$ ::= $\langle letter \rangle \{ \langle letter \rangle \mid \langle digit \rangle \mid \_ \} \mid \sigma$

$\langle letter \rangle$ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p
| q | r | s | t | uv | w | x | y | z | A | B | C | D | E | F
| G | H | I | J | K | L | M | NO | P | Q | R | S | T | U
| V | W | X | Y | Z

$\langle digit \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\langle val \rangle$ ::= $\langle digit \rangle$ | $\langle letter \rangle \{ \langle digit \rangle$ | $\langle letter \rangle \}$

$\langle S \rangle$ ::= S is
$\langle definitions \rangle$
$[\langle synonyms \rangle]$
end S

$\langle definitions \rangle$ ::= $\langle id \rangle \{, \langle id \rangle \} : \langle type \rangle ;$
$[\langle definitions \rangle]$

$\langle synonyms \rangle$ ::= $\langle id \rangle == \langle Type \rangle ;$
$[\langle synonyms \rangle]$

$\langle type \rangle$ ::= $\mathbb{N}$ | $\mathbb{Z}$ | $\mathbb{R}$ | Time | Text | Boolean | $\langle enum \rangle$
| $\mathbb{P}$ $\langle type \rangle$
| List $\langle type \rangle$
| $\langle type \rangle \cup \langle type \rangle \{ \cup \langle type \rangle \}$
| $\langle type \rangle \times \langle type \rangle \{ \times \langle type \rangle \}$
| $\langle id \rangle$

$\langle enum \rangle$ ::= "{" $\langle id \rangle$ | $\langle symbol \rangle \{, \langle id \rangle$ | $\langle symbol \rangle \}$ "}"

$\langle symbol \rangle$ ::= ∞ | $\emptyset$

$\langle X \rangle$ ::= X is
$\langle definitions \rangle$
$[\langle synonyms \rangle]$
end X

$\langle Y \rangle$ ::= Y is
$\langle definitions \rangle$
$[\langle synonyms \rangle]$
end Y

$\langle deltint \rangle$ ::= δint $[\langle ids \rangle]$ is
$\langle sentence \rangle$
$\{\langle sentence \rangle\}$
end δint

$\langle deltext \rangle$ ::= δext $[\langle ids \rangle]$ is
$\langle sentence \rangle$
$\{\langle sentence \rangle\}$
end δext

$\langle ids \rangle$ ::= (($\langle id \rangle$ | $\langle ids \rangle$){,($\langle id \rangle$ | $\langle ids \rangle$)})

$\langle sentence \rangle$ ::= $\langle assignment \rangle$

| $\langle foreach \rangle$

| $\langle caseblock \rangle$

| $\langle whereblock \rangle$

$\langle assignment \rangle$ ::= $\langle var \rangle$=$\langle expr \rangle$;

$\langle var \rangle$ ::= $\langle id \rangle$ | $\langle idComp \rangle$ | value | port | e

$\langle idComp \rangle$ ::= $\langle id \rangle$.$\langle digit \rangle${.$\langle digit \rangle$}

$\langle expr \rangle$ ::= $\langle var \rangle$ | $\langle val \rangle$ | $\langle vals \rangle$ | $\langle set \rangle$ | $\langle list \rangle$ | $\langle operation \rangle$

$\langle vals \rangle$ ::= ($\langle expr \rangle$,$\langle expr \rangle${,$\langle expr \rangle$})

$\langle set \rangle$ ::= "{"$\langle expr \rangle${,$\langle expr \rangle$}"}"

$\langle list \rangle$ ::= <$\langle expr \rangle${,$\langle expr \rangle$}>

$\langle operation \rangle$ ::= [$\langle unOp \rangle$]$\langle operand \rangle$[$\langle binOp \rangle\langle operand \rangle$]

$\langle operand \rangle$ ::= $\langle var \rangle$ | $\langle val \rangle$ | $\langle mathFun \rangle$ | $\langle defFun \rangle$

$\langle mathFun \rangle$ ::= $\langle funId \rangle$([$\langle unOp \rangle$]($\langle var \rangle$ | $\langle operation \rangle$){,[$\langle unOp \rangle$]($\langle var \rangle$ | $\langle operation \rangle$)})

$\langle funId \rangle$ ::= sin | cos | tan | arcsin | arccos | arctan | log | sign | min

| max | sqrt

$\langle defFun \rangle$ ::= $\langle id \rangle$(($\langle var \rangle$ | $\langle operation \rangle$){,($\langle var \rangle$ | $\langle operation \rangle$)});

$\langle unOp \rangle$ ::= $-$ | rev | head | last | tail | front | #

$\langle binOp \rangle$ ::= $+$ | $-$ | $\setminus$ | $*$ | $\frown$ | $\cap$ | $\cup$

$\langle foreach \rangle$ ::= foreach $\langle id \rangle$ in $\langle expr \rangle$do

$\langle sentence \rangle$

{$\langle sentence \rangle$}

end foreach

$\langle caseblock \rangle$ ::= defcases

(case $\langle sentence \rangle$; {$\langle sentence \rangle$;} if $\langle conditions \rangle$)

| (if $\langle conditions \rangle$ $\Rightarrow$ $\langle sentence \rangle$; {$\langle sentence \rangle$;})

({case $\langle sentence \rangle$; {$\langle sentence \rangle$;} if $\langle conditions \rangle$})

| ({if $\langle conditions \rangle$ $\Rightarrow$ $\langle sentence \rangle$; {$\langle sentence \rangle$;}})

[default $\langle sentence \rangle$; {$\langle sentence \rangle$;}]

end defcases

$\langle conditions \rangle$ ::= $\langle condition \rangle$

| $\neg$ ($\langle conditions \rangle$)

| ($\langle conditions \rangle \wedge \langle conditions \rangle$)

| ($\langle conditions \rangle \vee \langle conditions \rangle$)

$\langle condition \rangle$ ::= ($\langle var \rangle$ | $\langle val \rangle$ | $\langle operation \rangle$)$\langle comparation \rangle$($\langle var \rangle$ | $\langle val \rangle$ | $\langle operation \rangle$)

$\langle comparation \rangle$ ::= $<$ | $>$ | $\leq$ | $\geq$ | $=$ | $\neq$ | $\in$ | $\notin$ | $\subset$ | $\subseteq$

$\langle whereblock \rangle$ ::=defwhere
$\langle sentence \rangle$
$\{\langle sentence \rangle\}$
where
$[\langle definition \rangle]$
$[\langle synonyms \rangle]$
$\{\langle sentence \rangle\}$
end defwhere

$\langle \lambda \rangle$ ::= λ $[\langle ids \rangle]$ is
$\langle sentence \rangle$
$\{\langle sentence \rangle\}$
end λ

$\langle ta \rangle$ ::= ta $[\langle ids \rangle]$ is
$\langle sentence \rangle$
$\{\langle sentence \rangle\}$
end ta

$\langle P \rangle$ ::= P is
$\langle id \rangle = \langle val \rangle$;
$[\langle id \rangle = \langle val \rangle;]$
end P

$\langle functions \rangle$ ::= functions $\langle id \rangle$ is
$\langle function \rangle$
$\{\langle function \rangle\}$
end functions

$\langle function \rangle$ ::= function $\langle id \rangle$ is
$\langle id \rangle : \langle type \rangle \{, \langle id \rangle : \langle type \rangle\} \rightarrow \langle id \rangle : \langle Type \rangle$;
$[\langle definitions \rangle]$
$[\langle synonyms \rangle]$
$\langle sentence \rangle$;
$\{\langle sentence \rangle;\}$
end function

$\langle simulate \rangle$ ::= simulate $\langle id \rangle$ from
$\langle assignment \rangle$
$\{\langle assignment \rangle\}$
end simulate

Apéndice B

ModDEVS - Reglas de Traducción

En este apéndice se muestra en detalle cómo traducir modelos CML-DEVS a modelos que se puedan simular en DEVS-Suite y en PowerDEVS. Se empieza describiendo la estructura general del modelo y los archivos que deben ser generados para la simulación. Luego, de dan las reglas para la traducción de cada estructura o componente de un modelo CML-DEVS. Las reglas se enuncian utilizando una estructura general, que se describe en la Sección B.2. Además, en cada sección se muestra la porción de la EBNF que hace referencia a la(s) regla(s) de traducción correspondiente(s).

B.1. Estructura General

⟨*atomic*⟩ ::= atomic ⟨*id*⟩[(params)] is

[⟨*params*⟩]

[⟨*S*⟩]

[⟨*X*⟩]

[⟨*Y*⟩]

[⟨*δint*⟩]

[⟨*δext*⟩]

[⟨*λ*⟩]

[⟨*ta*⟩]

end ⟨*id*⟩

B.1.1. Estructura general de un modelo atómico DEVS-Suite

```
import view.modeling.ViewableAtomic;
import model.modeling.content;
import model.modeling.message;
import ... ;
```

```
public class ModelName extends ViewableAtomic{
    /* Declaración de variables del estado */
    /* Declaración del tipo de los inputs */
    ...
    public ModelName(){
        super("ModelName");
        /* Declaración de puertos */
        addInport("in");
        addOutport("out");
        /* Configuración de parámetros */
        ...
    }
    public void initialize(){
        /* Instanciar la variables que corresonda */
        var = new ... ();
        /* Inicializar variables de estado (si corresponde) */
        ...
    }
    public void deltint(){
        ...
    }
    public void deltext(double e,message x){
        ...
    }
    public message out(){
        ...
    }
    public double ta(){
        ...
    }
}
```

B.1.2. Estructura general de un modelo atómico PowerDEVS

Estructura archivo modelname.pds

```
Root-Coordinator
 {
  Simulator
   {
    Path = "path/modelname.h"
    Parameters =
   }
  EIC
   {
   }
  EOC
   {
   }
  IC
   {
   }
 }
```

Estructura archivo modelname.h

```
#if !defined modelname_h
#define modelname_h
```

```
#include "simulator.h"
#include "path/modelname.h"
#include "event.h"
#include "stdarg.h"
#include "..."

class modelname: public Simulator {
#define INFINITY 1e10
public:
    modelname(const char *n): Simulator(n) {};
    void init(double, ...);
    double ta(double t);
    void dint(double);
    void dext(Event , double );
    Event lambda(double);
    void exit();
};
#endif
```

Estructura archivo modelname.cpp:

```
#include "modelname.h"
void modelname::init(double t,...) {
    ...
    /* Inicializar variables, si corresponde */
}
void modelname::dint(double t){
    ...
}
void modelname::dext(Event x, double t){
    ...
}
Event modelname::lambda(double t) {
    ...
}
double modelname::ta(double t){
    ...
}
void modelname::exit() {
    ...
}
```

B.2. Definiciones

De ahora en más, para representar una regla de traducción utilizaremos la estructura cuyo esquema se describe en al Figura B.1. Cada regla consiste en un nombre; el código a ser traducido; un contexto, i.e. las condiciones bajo las cuales esta regla se aplica; el código traducido (código Java y C++); y, eventualmente, un comentario sobre la regla. Las expresiones entre doble corchetes, $[\![\ldots]\!]$, son expresiones que todavía tienen que traducirse. El subíndice indica el lenguaje destino a traducirse, Java o C++; y el superíndice, la regla que debe ser aplicada (como en el encabezado del esquema de la regla).

NOMBRE DE LA REGLA `Código a traducir`

CONTEXT:
Condiciones sobre las cuales se aplica esta regla

CODE:

- DEVS-Suite:

  `Código Java resultante`

- PowerDEVS:

  `Código C++ resultante`

COMMENT:
(Los comentarios son opcionales)

Figura B.1: Esquema de una regla de traducción

$$
\begin{aligned}
\langle S\rangle ::= & \ \mathsf{S\ is} \\
& \langle definitions\rangle \\
& [\langle synonyms\rangle] \\
& \mathsf{end\ S} \\
\langle definitions\rangle ::= & \ \langle id\rangle\{,\langle id\rangle\}:\langle type\rangle; \\
& [\langle definitions\rangle] \\
\langle type\rangle ::= & \ \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathsf{Time} \mid \mathsf{Text} \mid \mathsf{Boolean} \mid \langle enum\rangle \\
& \mid \mathbb{P}\ \langle type\rangle \\
& \mid \mathsf{List}\ \langle type\rangle \\
& \mid \langle type\rangle\cup\langle type\rangle\{\cup\langle type\rangle\} \\
& \mid \langle type\rangle\times\langle type\rangle\{\times\langle type\rangle\} \\
& \mid \langle id\rangle \\
\langle synonyms\rangle ::= & \ \langle id\rangle{=}{=}\langle Type\rangle; \\
& [\langle synonyms\rangle]
\end{aligned}
$$

DEFINITION
$[\![\langle id\rangle{:}\langle type\rangle;]\!]^{\mathsf{D}}$

CONTEXT:
$\langle id\rangle$="varName";

CODE:
Case $\langle type\rangle$ *of:*

- $\mathbb{N}$:
  - DEVS-Suite:
    `Integer varName;`
  - PowerDEVS:
    `unsigned int varName;`

- $\mathbb{Z}$:
  - DEVS-Suite:
    `Integer varName;`
  - PowerDEVS:
    `int varName;`
- $\mathbb{R}$:
  - DEVS-Suite:
    `Double varName;`
  - PowerDEVS:
    `double varName;`
- Time:
  - DEVS-Suite:
    `Double varName;`
  - PowerDEVS:
    `unsigned double varName;`
- Boolean:
  - DEVS-Suite:
    `Boolean varName;`
  - PowerDEVS:
    `bool varName;`
- Text:
  - DEVS-Suite:
    `String varName;`
  - PowerDEVS:
    `std::string varName;`
- $\langle enum \rangle$
  - DEVS-Suite:
    `String varName;`
  - PowerDEVS:
    `std::string varName;`
- $\text{Type}_1 \cup \ldots \cup \text{Type}_n$
  - DEVS-Suite:

```
static class T_varName extends Object implements Comparable<Object>{
    public [[Type_1]]_J v_[[Type_1]]^N;
    ⋮
    public [[Type_n]]_J v_[[Type_n]]^N;
    public String type;
    public boolean equals(T_varName other){
        ⋮
    }
    public int compareTo(T_varName other){
        ⋮
    }
    T_varName(){}
    T_varName([[Type_1]]_J v){
        v_[[Type_1]]^N = v;
        type="[[Type_1]]^N";
    }
    ⋮
    T_varName([[Type_n]]_J v){
        v_[[Type_n]]^N = v;
        type="[[Type_n]]^N";
    }
```

```
    T_varName(T_varName other){
        v_[[Type_1]]^N = other.v_[[Type_1]]^N;
        ⋮
        v_[[Type_n]]^N = other.v_[[Type_n]]^N;
        type=other.type;
    }
}
T_varName varName;
```

- PowerDEVS:

```
class T_varName{
    [[Type_1]]_C v_[[Type_1]]^N;
    ⋮
    [[Type_n]]_C v_[[Type_n]]^N;
    public std::string type;
    bool operator<(const T_varName& other) const{
        ⋮
    }
    bool operator==(const T_varName& other) const{
        ⋮
    }
    T_varName& operator=(const T_varName other){
        ⋮
    }
    T_varName(){};
    T_varName([[Type_1]]_C v){
        v_[[Type_1]]^N = v;
        type="[[Type_1]]^N";
    }
    ⋮
    T_varName([[Type_n]]_C v){
        v_[[Type_n]]^N = v;
        type="[[Type_n]]^N";
    }
} varName;
```

- $\text{Type}_1 \times \ldots \times \text{Type}_n$
  - DEVS-Suite:

```
static class T_varName extends Object implements Comparable<Object>{
    public [[Type_1]]_J v1;
    ⋮
    public [[Type_n]]_J vn;
    public boolean equals(T_varName other){
        ⋮
    }
    public int compareTo(T_varName other){
        ⋮
    }
    T_varName(){}
    T_varName([[Type_1]]_J v1, ..., [[Type_n]]_J vn){
        this.v1 = v1;
        ⋮
        this.vn = vn;
    }
```

```
    T_varName(T_varName other){
        v_[[Type_1]]^N = other.v_[[Type_1]]^N;
        :
        v_[[Type_n]]^N = other.v_[[Type_n]]^N;
    }
}
T_varName varName;
```

- PowerDEVS:

```
class T_varName{
    [[Type_1]]_C v1;
    :
    [[Type_n]]_C vn;
    bool operator<(const T_varName& other) const{
        :
    }
    bool operator==(const T_varName& other) const{
        :
    }
    T_varName& operator=(const T_varName other){
        :
    }
    T_varName(){};
    T_varName([[Type_1]]_C v1, ..., [[Type_n]]_C vn){
        this.v1 = v1;
        :
        this.vn = vn;
    }
} varName;
```

- $\mathbb{P}$ Type
  - DEVS-Suite:

    ```
    Set<[[Type]]_J> varName;
    ```

  - PowerDEVS:

    ```
    std::set<[[Type]]_C> varName;
    ```

- List Type
  - DEVS-Suite:

    ```
    List<[[Type]]_J> varName;
    ```

  - PowerDEVS:

    ```
    std::list<[[Type]]_C> varName;
    ```

B.3. Puertos de entrada

⟨*X*⟩ ::= X is

⟨*definitions*⟩

end X

CML-DEVS:

$portName_1 : Type_1;$

...

$portaName_n : Type_n;$

Se crea la siguiente variable en CML-DEVS, y se la traduce a DEVS-Suite y PowerDEVS según las reglas de definición (descriptas anteriormente):

$$T\_in : Type_1 \cup \cdots \cup Type_n;$$

Además:

DEVS-Suite:

Dentro del constructor de la clase ModelName:

```
addInport("portName1");
...
addInport("portNameN");
```

PowerDEVS:

Si se utiliza la IDE de PowerDEVS, el número de puertos se especifica en el archivo modelName.pd.

B.4. Puertos de salida

⟨*Y*⟩ ::= Y is

⟨*definitions*⟩

end Y

CML-DEVS:

$portName_1 : Type_1;$

...

$portaName_n : Type_n;$

DEVS-Java:

Dentro del constructor de la clase ModelName:

```
addOutport("portName1");
...
addInport("portNameN");
```

Además, se crea una clase por cada puerto, siguiendo las reglas de traducción de las definiciones detalladas anteriormente:

```
public class T_Typei extends entity implements Comparable<Object>{
    ...
}
```

PowerDEVS:

Si se utiliza la IDE de PowerDEVS, el número de puertos se especifica en el archivo modelName.pd.

Además, se crea una clase por cada puerto, siguiendo las reglas de traducción de las definiciones detalladas anteriormente:

```
public class T_Typei{
    ...
}
```

B.5. Funciones de transición, salida y avance de tiempo

$\langle \delta int \rangle$ ::= δint[$\langle ids \rangle$] is
{$\langle sentence \rangle$}
end δint

$\langle \delta ext \rangle$ ::= δext[$\langle ids \rangle$] is
{$\langle sentence \rangle$}
end δext

$\langle \lambda \rangle$ ::= λ [$\langle ids \rangle$] is
{$\langle sentence \rangle$}
end λ

$\langle ta \rangle$ ::= ta [$\langle ids \rangle$] is
{$\langle sentence \rangle$}
end ta

$\langle ids \rangle$ ::= (($\langle id \rangle$ | $\langle ids \rangle$){, ($\langle id \rangle$ | $\langle ids \rangle$)})

$\langle sentence \rangle$ ::= $\langle assignment \rangle$
| $\langle foreach \rangle$
| $\langle caseblock \rangle$
| $\langle whereblock \rangle$

DEVS-Java:

```
public void deltint(){
    /*sentences...*/
}
public deltext(double e,message x){
    /*sentences...*/
}
public message out(){
    message m=new message();
    content con;
    /*sentences...*/
}
public double ta(){
    /*sentences...*/
    return sigma;
}
```

PowerDEVS:

```
void modelName::dint(double t){
    /*sentences...*/
}
void mdoelName::dext(Event x, double t){
    /*sentences...*/
}
Event modelName::lambda(double t){
    /*sentences*/
    return Event(..., ...);
}
double modeName::ta(double t){
    /*sentences...*/
    return sigma;
}
```

B.5.1. Asignaciones

$$\langle assignment \rangle ::= \langle var \rangle = \langle expr \rangle ;$$

$$\langle var \rangle ::= \langle id \rangle \mid \langle idComp \rangle \mid \mathsf{value} \mid \mathsf{port} \mid \mathsf{e}$$

$$\langle idComp \rangle ::= \langle id \rangle . \langle digit \rangle \{ . \langle digit \rangle \}$$

$$\langle expr \rangle ::= \langle var \rangle \mid \langle val \rangle \mid \langle vals \rangle \mid \langle operation \rangle$$

$$\langle vals \rangle ::= (\langle expr \rangle , \langle expr \rangle \{ , \langle expr \rangle \})$$

$$\langle set \rangle ::= \{ \langle expr \rangle \{ , \langle expr \rangle \} \}$$

$$\langle list \rangle ::= \langle \langle expr \rangle \{ , \langle expr \rangle \} \rangle$$

ASSIGNMENT Basic Numbers

$[\![\langle var\rangle \texttt{ = } \langle expr\rangle\texttt{)}]\!]^{\texttt{A}}$

CONTEXT:

Type($\langle var\rangle$) = $\mathbb{N}$ | $\mathbb{Z}$ | $\mathbb{R}$ | Time

Type($\langle expr\rangle$) = $\mathbb{N}$ | $\mathbb{Z}$ | $\mathbb{R}$ | Time

CODE:

case Type($\langle var\rangle$) of:

- $\mathbb{N}$ | $\mathbb{Z}$:
  - DEVS-Suite:

    $[\![\langle var\rangle]\!]_{\texttt{J}}$`=toInteger(`$[\![\langle expr\rangle]\!]_{\texttt{J}}$`);`
  - PowerDEVS:

    $[\![\langle var\rangle]\!]_{\texttt{C}}$`=(int)(`$[\![\langle expr\rangle]\!]_{\texttt{C}}$`);`
- $\mathbb{R}$ | Time:
  - DEVS-Suite:

    $[\![\langle var\rangle]\!]_{\texttt{J}}$`=toDouble(`$[\![\langle expr\rangle]\!]_{\texttt{J}}$`);`
  - PowerDEVS:

    $[\![\langle var\rangle]\!]_{\texttt{C}}$`=(double)(`$[\![\langle expr\rangle]\!]_{\texttt{C}}$`);`

ASSIGNMENT Basic Not Numbers

$[\![\langle var\rangle \texttt{ = } \langle expr\rangle\texttt{)}]\!]^{\texttt{A}}$

CONTEXT:

Type(varId) = Type($\langle expr\rangle$) = Text | Bool | $\langle enum\rangle$

CODE:

- DEVS-Suite:

  $[\![\langle var\rangle]\!]_{\texttt{J}}$`=`$[\![\langle expr\rangle]\!]_{\texttt{J}}$`;`
- PowerDEVS:

  $[\![\langle var\rangle]\!]_{\texttt{C}}$`=`$[\![\langle expr\rangle]\!]_{\texttt{C}}$`;`

ASSIGNMENT Tuple

$[\![\langle var\rangle \texttt{ = } \langle expr\rangle\texttt{)}]\!]^{\texttt{A}}$

CONTEXT:

$\langle var\rangle$ = varId

Type(varId) = $\text{Type}_1 \times \cdots \times \text{Type}_n$

CODE:

Case $\langle expr \rangle$ *of:*

- $\langle expr \rangle$ = (expr$_1$, ..., expr$_n$)
  $[\![$varId.1= expr$_1]\!]^{A}$
  ⋮
  $[\![$varId.n= expr$_n]\!]^{A}$
- $\langle id \rangle$ = varName
  $[\![$varId.1= varName.1$]\!]^{A}$
  ⋮
  $[\![$varId.n= varName.n$]\!]^{A}$
- $\langle funAppl \rangle$ = funId(var1, ..., varn)
  $[\![$varTmp: TypeVarId$]\!]^{D}$
  varTmp=funId(var1, ..., varn);
  $[\![$varId.1= varTmp.1$]\!]^{A}$
  ⋮
  $[\![$varId.n= varTmp.n$]\!]^{A}$

ASSIGNMENT Union and Number

$[\![\langle var \rangle$ `=` $\langle expr \rangle)]\!]^{A}$

CONTEXT:

Type($\langle var \rangle$) = Type$_1 \cup \cdots \cup$ Type$_n$

Type($\langle expr \rangle$) = $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time

Type$_i \in$ {Type$_1$, ..., Type$_n$}

Type$_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time

CODE:

case Type$_i$ of:

- $\mathbb{N} \mid \mathbb{Z}$:
  - DEVS-Suite:

    $[\![\langle var \rangle]\!]_{J}$`.v_`$[\![$`Type_`$i]\!]^{N}$ `= toInteger(`$[\![\langle expr \rangle]\!]_{J}$`);`
    $[\![\langle var \rangle]\!]_{J}$`.type = "`$[\![$`Type`$_i]\!]^{N}$`";`
  - PowerDEVS:

    $[\![\langle var \rangle]\!]_{C}$`.v_`$[\![$`Type_`$i]\!]^{N}$ `= (int)(`$[\![\langle expr \rangle]\!]_{C}$`);`
    $[\![\langle var \rangle]\!]_{C}$`.type = "`$[\![$`Type`$_i]\!]^{N}$`";`
- $\mathbb{R} \mid$ Time:
  - DEVS-Suite:

    $[\![\langle var \rangle]\!]_{J}$`.v_`$[\![$`Type_`$i]\!]^{N}$ `= toDouble(`$[\![\langle expr \rangle]\!]_{J}$`);`
    $[\![\langle var \rangle]\!]_{J}$`.type = "`$[\![$`Type`$_i]\!]^{N}$`";`

- PowerDEVS:

  $[\![\langle var\rangle]\!]_{\texttt{C}}$.v_$[\![$Type_$i]\!]^{\texttt{N}}$ = (double)($[\![\langle expr\rangle]\!]_{\texttt{C}}$);
  $[\![\langle var\rangle]\!]_{\texttt{C}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

ASSIGNMENT Union and Not a Number

$[\![\langle var\rangle$ = $\langle expr\rangle)]\!]^{\texttt{A}}$

CONTEXT:

Type($\langle var\rangle$) = Type$_1 \cup \cdots \cup$ Type$_n$
Type($\langle expr\rangle$) = Type$_i \in \{$Type$_1, \ldots,$ Type$_n\}$
Type$_i \neq \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time

CODE:

- DEVS-Suite:

  $[\![\langle var\rangle$.v_$[\![$Type_$i]\!]^{\texttt{N}}$ = $\langle expr\rangle]\!]^{\texttt{A}}$
  $[\![\langle var\rangle]\!]_{\texttt{J}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

- PowerDEVS:

  $[\![\langle var\rangle$.v_$[\![$Type_$i]\!]^{\texttt{N}}$ = $\langle expr\rangle]\!]^{\texttt{A}}$
  $[\![\langle var\rangle]\!]_{\texttt{C}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

ASSIGNMENT Union and Number in Union

$[\![\langle var\rangle$ = $\langle expr\rangle)]\!]^{\texttt{A}}$

CONTEXT:

Type($\langle var\rangle$) = Type$_1 \cup \ldots \cup$ Type$_n$
Type($\langle expr\rangle$) = Type$_{n+1} \cup \ldots \cup$ Type$_m$
Type$_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time
$\langle var\rangle$.type = "$[\![$Type$_i]\!]^{\text{N}}$"
Type$_j = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time
$\langle expr\rangle$.type = "$[\![$Type$_j]\!]^{\text{N}}$"

CODE:

case Type$_i$ of:

- $\mathbb{N} \mid \mathbb{Z}$:

  - DEVS-Suite:

    $[\![\langle var\rangle]\!]_{\texttt{J}}$.v_$[\![$Type$_i]\!]^{\texttt{N}}$ = toInteger($[\![\langle expr\rangle]\!]_{\texttt{J}}$.v_$[\![$Type_$j]\!]^{\texttt{N}}$);
    $[\![\langle var\rangle]\!]_{\texttt{J}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

  - PowerDEVS:

    $[\![\langle var\rangle]\!]_{\texttt{C}}$.v_$[\![$Type$_i]\!]^{\texttt{N}}$ = (int)($[\![\langle expr\rangle]\!]_{\texttt{C}}$.v_$[\![$Type_$j]\!]^{\texttt{N}}$);
    $[\![\langle var\rangle]\!]_{\texttt{C}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

- $\mathbb{R} \mid$ Time:

  - DEVS-Suite:

    $[\![\langle var\rangle]\!]_{\texttt{J}}$.v_$[\![$Type$_i]\!]^{\texttt{N}}$ = toDouble($[\![\langle expr\rangle]\!]_{\texttt{J}}$.v_$[\![$Type_$j]\!]^{\texttt{N}}$);
    $[\![\langle var\rangle]\!]_{\texttt{J}}$.type = "$[\![$Type$_i]\!]^{\texttt{N}}$";

- PowerDEVS:

  $[\![\langle var\rangle]\!]_{\texttt{C}}$.v_$[\![\texttt{Type}_i]\!]^{\mathbb{N}}$ = (double)($[\![\langle expr\rangle]\!]_{\texttt{C}}$.v_$[\![\texttt{Type_}j]\!]^{\mathbb{N}}$);
  $[\![\langle var\rangle]\!]_{\texttt{C}}$.type = "$[\![\texttt{Type}_i]\!]^{\mathbb{N}}$";

ASSIGNMENT Union and Not a Number in Union

$[\![\langle var\rangle$ = $\langle expr\rangle)]\!]^{\texttt{A}}$

CONTEXT:

$\text{Type}(\langle var\rangle) = \text{Type}_1 \cup \ldots \cup \text{Type}_n$

$\text{Type}(\langle expr\rangle) = \text{Type}_{n+1} \cup \ldots \cup \text{Type}_m$

$\text{Type}_i \in \{\text{Type}_1, \ldots, \text{Type}_n\}$

$\text{Type}_j \neq \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$

$\langle expr\rangle$.type = "$[\![\text{Type}_j]\!]^{\text{N}}$"

$\text{Type}_i = \text{Type}_j$

CODE:

- DEVS-Suite:

  $[\![\langle var\rangle$.v_$[\![\texttt{Type_}j]\!]^{\mathbb{N}}$ = $\langle expr\rangle$.v_$[\![\texttt{Type_}j]\!]^{\mathbb{N}}]\!]^{\texttt{A}}$
  $[\![\langle var\rangle]\!]_{\texttt{J}}$.type = "$[\![\texttt{Type}_i]\!]^{\mathbb{N}}$";

- PowerDEVS:

  $[\![\langle var\rangle$.v_$[\![\texttt{Type_}j]\!]^{\mathbb{N}}$ = $\langle expr\rangle$.v_$[\![\texttt{Type_}j]\!]^{\mathbb{N}}]\!]^{\texttt{A}}$
  $[\![\langle var\rangle]\!]_{\texttt{C}}$.type = "$[\![\texttt{Type}_i]\!]^{\mathbb{N}}$";

ASSIGNMENT Set

$[\![\texttt{varId}$ = $\langle expr\rangle)]\!]^{\texttt{A}}$

CONTEXT:

$\text{Type}(\text{varId}) = \text{Type}(\langle expr\rangle) = \mathbb{P}\ \text{Type}$

CODE:

Case $\langle expr\rangle$ *of:*

- $\langle set\rangle = \{expr_1, \ldots, expr_n\}$
  - DEVS-Suite:

    varId=buildSet(new $[\![\texttt{Type}]\!]_{\texttt{J}}$($[\![expr_1]\!]_{\texttt{J}}$), ..., new $[\![\texttt{Type}]\!]_{\texttt{J}}$($[\![expr_n]\!]_{\texttt{J}}$));

  - PowerDEVS:

    varId=buildSet<$[\![\texttt{Type}]\!]_{\texttt{C}}$>(n, $[\![expr_1]\!]_{\texttt{C}}$, ..., $[\![expr_n]\!]_{\texttt{C}}$);

- $\langle var\rangle \mid \langle operation\rangle$
  - DEVS-Suite:

    varId.clear();
    varId.addAll($[\![\langle expr\rangle]\!]_{\texttt{J}}$);

  - PowerDEVS:

    varId=$[\![\langle expr\rangle]\!]_{\texttt{C}}$;

ASSIGNMENT List

$[\![$`varId =` $\langle expr \rangle$`)`$]\!]^{A}$

CONTEXT:

Type(varId) = Type($\langle expr \rangle$) = List Type

CODE:

Case $\langle expr \rangle$ *of:*

- $\langle list \rangle = \langle expr_1, \ldots, expr_n \rangle$
  - DEVS-Suite:

    `varId=buildList(new` $[\![$`Type`$]\!]_{J}$`(`$[\![expr_1]\!]_{J}$`, ..., new` $[\![$`Type`$]\!]_{J}$`(`$[\![expr_n]\!]_{J}$`));`
  - PowerDEVS:

    `varId=buildList<`$[\![$`Type`$]\!]_{C}$`>(n,` $[\![expr_1]\!]_{C}$`, ...,` $[\![expr_n]\!]_{C}$`);`
- $\langle var \rangle \mid \langle operation \rangle$
  - DEVS-Suite:

    ```
    varId.clear();
    varId.addAll(⟦⟨expr⟩⟧J);
    ```
  - PowerDEVS:

    `varId=`$[\![\langle expr \rangle]\!]_{C}$`;`

B.5.2. Estructura "For Each"

$\langle foreach \rangle$ ::= foreach $\langle id \rangle$ in $\langle expr \rangle${

$\langle sentence \rangle$

{$\langle sentence \rangle$}

}

FOR EACH Set

foreach $\langle id \rangle$ in $\langle expr \rangle${

$\langle sentences \rangle$

}

CONTEXT:

$\langle id \rangle$ = varName

Type($\langle id \rangle$) = TypeId

Type($\langle expr \rangle$) = $\mathbb{P}$ TypeId

CODE:

Case $\langle expr \rangle$ *of:*

- $\langle set \rangle \mid \langle operation \rangle$
  - DEVS-Suite:

    ```
    Set<TypeId> setTmp = ⟦⟨expr⟩⟧J;
    for(TypeId varName: setTmp){
        ⟦⟨sentences⟩⟧J
    }
    ```

- PowerDEVS:

```
std::set<TypeId> setTmp=⟦⟨expr⟩⟧C;
for (std::set<TypeId>::iterator it = setTmp.begin(); it != setTmp.end(); it++)
    TypeId varName = *it;
    ⟦⟨sentences⟩⟧C
}
```

- ⟨*var*⟩ = varName2;
  - DEVS-Suite:

```
Set<TypeId> setTmp = new TreeSet<TypeId>(varName2);
for(TypeId varName: setTmp){
    ⟦⟨sentences⟩⟧J
}
```

  - PowerDEVS:

```
std::set<TypeId> setTmp = varName2;
for(std::set<TypeId>::iterator it = setTmp.begin(); it!=setTmp.end(); it++){
    TypeId varName = *it;
    ⟦⟨sentences⟩⟧C
}
```

B.5.3. Definiciones por caso

⟨*caseblock*⟩ ::= "\begin{defcases}"
\case{
⟨*sentence*⟩;{⟨*sentence*⟩;}
\if
⟨*conditions*⟩
}
{\case{
⟨*sentence*⟩;{⟨*sentence*⟩;}
\if
⟨*conditions*⟩
}}
[\default{
⟨*sentence*⟩;{⟨*sentence*⟩;}
}]
"\end{defcases}"

CASE BLOCK

"\begin{defcases}"

⋮

"\end{defcases}"

CONTEXT:

CODE:

- DEVS-Suite:

```
if (/*conditions*/){
    /*sentences*/
}
else if (/*conditions*/){
    /*sentences*/
}
 ⋮
else{ /*si es que está definido '\default'*/
    /*sentences*/
}
```

- PowerDEVS:

```
if (/*conditions*/){
    /*sentences*/
}
else if (/*conditions*/){
    /*sentences*/
}
 ⋮
else{ /*si es que está definido '\default'*/
    /*sentences*/
}
```

B.5.4. Condiciones

$$\begin{aligned}\langle conditions\rangle ::= &\ \langle condition\rangle \\ &| \neg\ (\langle conditions\rangle) \\ &| \ (\langle conditions\rangle \wedge \langle conditions\rangle) \\ &| \ (\langle conditions\rangle \vee \langle conditions\rangle)\end{aligned}$$

CONDITIONS

$\langle conditions\rangle$

CONTEXT:

CODE:

Case $\langle conditions\rangle$ *of:*

- $\langle condition\rangle$:

  $[\![\langle condition\rangle]\!]^{\texttt{Cond}}$

- $\neg$ ($\langle condition \rangle$):
  - DEVS-Suite:
    `!(`$[\![\langle condition \rangle]\!]^{\texttt{Cond}}$`)`
  - PowerDEVS:
    `!(`$[\![\langle condition \rangle]\!]^{\texttt{Cond}}$`)`
- ($\langle condition1 \rangle \wedge \langle condition2 \rangle$):
  - DEVS-Suite:
    `((`$[\![\langle condition1 \rangle]\!]^{\texttt{Cond}}$`) && (`$[\![\langle condition2 \rangle]\!]^{\texttt{Cond}}$`))`
  - PowerDEVS:
    `((`$[\![\langle condition1 \rangle]\!]^{\texttt{Cond}}$`) && (`$[\![\langle condition2 \rangle]\!]^{\texttt{Cond}}$`))`
- ($\langle condition1 \rangle \vee \langle condition2 \rangle$):
  - DEVS-Suite:
    `((`$[\![\langle condition1 \rangle]\!]^{\texttt{Cond}}$`) || (`$[\![\langle condition2 \rangle]\!]^{\texttt{Cond}}$`))`
  - PowerDEVS:
    `((`$[\![\langle condition1 \rangle]\!]^{\texttt{Cond}}$`) || (`$[\![\langle condition2 \rangle]\!]^{\texttt{Cond}}$`))`

Condición

$\langle condition \rangle ::= (\langle var \rangle \mid \langle val \rangle \mid \langle operation \rangle)\langle comparison \rangle(\langle var \rangle \mid \langle val \rangle \mid \langle operation \rangle)$

$\langle comparison \rangle ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq \mid \in \mid \notin \mid \subset \mid \subseteq$

CONDITION

$[\![\langle condition \rangle]\!]^{\texttt{Cond}}$

CONTEXT:

$\langle condition \rangle = \langle expr1 \rangle \langle comparison \rangle \langle expr2 \rangle$

CODE:

Case $\langle comparison \rangle$ of:

- $=$:
  $[\![\langle expr1 \rangle \ = \ \langle expr2 \rangle]\!]^{\texttt{E}}$
- $\neq$:
  `!(`$[\![\langle expr1 \rangle \ = \ \langle expr2 \rangle]\!]^{\texttt{E}}$`)`
- $<$:
  $[\![\langle expr1 \rangle \ < \ \langle expr2 \rangle]\!]^{\texttt{L}}$
- $>$:
  $[\![\langle expr1 \rangle \ > \ \langle expr2 \rangle]\!]^{\texttt{G}}$
- $\leq$:
  $[\![\langle expr1 \rangle \ \leq \ \langle expr2 \rangle]\!]^{\texttt{LE}}$

- $\geq$:

  $[\![\langle expr1\rangle \ \geq \ \langle expr2\rangle]\!]^{\texttt{GE}}$

- $\in$:

  $[\![\langle expr2\rangle \ \in \ \langle expr1\rangle]\!]^{\texttt{B}}$

- $\notin$:
  - DEVS-Suite:

    `!(`$[\![\langle expr2\rangle \ \in \ \langle expr1\rangle]\!]^{\texttt{B}}$`)`

  - PowerDEVS:

    `!(`$[\![\langle expr2\rangle \ \in \ \langle expr1\rangle]\!]^{\texttt{B}}$`)`

- $\subset$:
  - DEVS-Suite:

    ```
    ((⟦⟨expr2⟩⟧J).containsAll(⟦⟨expr1⟩⟧J)
        && ((⟦⟨expr1⟩⟧J).size()<(⟦⟨expr2⟩⟧J).size()))
    ```

  - PowerDEVS:

    ```
    (std::includes((⟦⟨expr1⟩⟧C).begin(), (⟦⟨expr2⟩⟧C).end(),
        (⟦⟨expr2⟩⟧C).begin(), (⟦⟨expr2⟩⟧C).end())
        && ((⟦⟨expr1⟩⟧C).size()<(⟦⟨expr2⟩⟧C)))
    ```

- $\subseteq$:
  - DEVS-Suite:

    ```
    ((⟦⟨expr2⟩⟧J).containsAll(⟦⟨expr1⟩⟧J))
    ```

  - PowerDEVS:

    ```
    (std::includes((⟦⟨expr1⟩⟧C).begin(), (⟦⟨expr2⟩⟧C).end(),
        (⟦⟨expr2⟩⟧C).begin(), (⟦⟨expr2⟩⟧C).end()))
    ```

Comparación de Igualdad

EQUALS Basic

$[\![\langle expr1\rangle = \langle expr2\rangle]\!]^{\texttt{E}}$

CONTEXT:

$\text{Type}(\langle expr1\rangle) = \text{Type}(\langle expr2\rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time} \mid \text{Text} \mid \text{Bool} \mid \langle enum\rangle$

CODE:

- DEVS-Suite:

  ```
  (⟦⟨expr1⟩⟧J).equals(⟦⟨expr2⟩⟧J)
  ```

- C++

  ```
  (⟦⟨expr1⟩⟧C) == (⟦⟨expr2⟩⟧C)
  ```

EQUALS Union1 $[\![\langle expr1\rangle = \langle expr2\rangle]\!]^{E}$

CONTEXT:

$\text{Type}(\langle expr1\rangle) = \text{Type}_1 \cup \ldots \cup \text{Type}_n$

$\text{Type}(\langle expr2\rangle) = \text{Type}_i \in \{\text{Type}_1, \ldots, \text{Type}_n\}$

CODE:

- DEVS-Suite:

  $([\![\langle expr1\rangle]\!]_{J}$`.type=="`$[\![\texttt{Type}_i]\!]^{N}$`") && (`$[\![(\langle expr1\rangle)$`.v_`$[\![\texttt{Type}_i]\!]^{N}$ `= `$\langle expr2\rangle]\!]^{E}$`)`

- PowerDEVS:

  $([\![\langle expr1\rangle]\!]_{C}$`.type=="`$[\![\texttt{Type}_i]\!]^{N}$`") && (`$[\![(\langle expr1\rangle)$`.v_`$[\![\texttt{Type}_i]\!]^{N}$ `= `$\langle expr2\rangle]\!]^{E}$`)`

COMMENT:

The Union type is already normalized

EQUALS Union2 $[\![\langle expr1\rangle = \langle expr2\rangle]\!]^{E}$

CONTEXT:

$\text{Type}(\langle expr1\rangle) = \text{Type}_1 \cup \ldots \cup \text{Type}_n$

$\text{Type}(\langle expr2\rangle) = \text{Type}_{n+1} \cup \ldots \cup \text{Type}_m$

CODE:

- DEVS-Suite:

  $([\![\langle expr1\rangle]\!]_{J}$`.type==(`$[\![\langle expr2\rangle]\!]_{J}$`).type) && (`$[\![(\langle expr1\rangle)$`.v_`$[\![\texttt{Type}_i]\!]^{N}$ `= (`$\langle expr2\rangle$`).v_`$[\![\texttt{Type}_j]\!]^{N}]\!]^{E}$`)`

- PowerDEVS:

  $([\![\langle expr1\rangle]\!]_{C}$`.type==(`$[\![\langle expr2\rangle]\!]_{C}$`).type) && (`$[\![(\langle expr1\rangle)$`.v_`$[\![\texttt{Type}_i]\!]^{N}$ `= (`$\langle expr2\rangle$`).v_`$[\![\texttt{Type}_j]\!]^{N}]\!]^{E}$`)`

COMMENT:

The Union types are already normalized

EQUALS Tuple

$[\![\langle exprA\rangle = \langle exprB\rangle]\!]^{E}$

CONTEXT:

$\text{Type}(\langle exprA\rangle) = \text{Type}(\langle exprB\rangle) = \text{Type}_1 \times \ldots \times \text{Type}_n$

CODE:

Case $\langle exprA\rangle$, $\langle exprB\rangle$ *of:*

- $\langle exprA\rangle$ = "varName1", $\langle exprB\rangle$ = "varName2"

  - DEVS-Suite:

    `varName1.equals(varName2)`

  - C++

    `varName1 == varName2`

- ($\langle exprA \rangle$ = "varName", $\langle exprB \rangle = (\text{expr}_1, \ldots, \text{expr}_n)$)
  $\vee$ ($\langle exprA \rangle = (\text{expr}_1, \ldots, \text{expr}_n)$, $\langle exprB \rangle$ = "varName")
  - DEVS-Suite:

    $[\![$`varName.1=expr`$_1]\!]^{\text{E}}$

    $\vdots$

    $[\![$`varName.n=expr`$_n]\!]^{\text{E}}$

  - C++

    $[\![$`varName.1=expr`$_1]\!]^{\text{E}}$

    $\vdots$

    $[\![$`varName.n=expr`$_n]\!]^{\text{E}}$

- $\langle exprA \rangle = (\text{expr}_1^A, \ldots, \text{expr}_n^A)$, $\langle exprB \rangle = (\text{expr}_1^B, \ldots, \text{expr}_n^B)$
  - DEVS-Suite:

    $[\![\text{expr}_1^A$`=`$\text{expr}_1^B]\!]^{\text{E}}$

    $\vdots$

    $[\![\text{expr}_n^A$`=`$\text{expr}_n^B]\!]^{\text{E}}$

  - C++

    $[\![\text{expr}_1^A$`=`$\text{expr}_1^B]\!]^{\text{E}}$

    $\vdots$

    $[\![\text{expr}_n^A$`=`$\text{expr}_n^B]\!]^{\text{E}}$

EQUALS Set

$[\![\langle exprA \rangle = \langle exprB \rangle]\!]^{\text{E}}$

CONTEXT:

$\text{Type}(\langle exprA \rangle) = \text{Type}(\langle exprB \rangle) = \mathbb{P}\ \text{Type}$

CODE:

Case $\langle exprA \rangle$, $\langle exprB \rangle$ *of:*

- $\langle var \rangle$ = "varName1", $\langle var \rangle$ = "varName2"
  - DEVS-Suite:

    `varName1.equals(varName2)`

  - C++

    `varName1 == varName2`

- ($\langle var \rangle$ = "varName", $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$)
  $\vee$ ($\langle var \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$, $\langle set \rangle$ = "varName")
  - DEVS-Suite:

    `varName.equals(buildSet(`$[\![\text{expr}_1]\!]_{\text{J}}$`, ..., `$[\![\text{expr}_n]\!]_{\text{J}}$`))`

  - PowerDEVS:

    `varName==(buildSet<`$[\![\text{Type}]\!]_{\text{C}}$`>(n, `$[\![\text{expr}_1]\!]_{\text{C}}$`, ..., `$[\![\text{expr}_n]\!]_{\text{C}}$`))`

- $\langle exprA \rangle = \{\text{expr}_1^A, \ldots, \text{expr}_n^A\}$, $\langle exprB \rangle = \{\text{expr}_1^B, \ldots, \text{expr}_n^B\}$
  - DEVS-Suite:

    ```
    (buildSet([[expr_1^A]]_J, ..., [[expr_n^A]]_J)).equals(
            buildSet([[expr_1^B]]_J, ..., [[expr_n^B]]_J))
    ```

  - PowerDEVS:

    ```
    (buildSet<[[Type]]_C>(n, [[expr_1^A]]_C, ..., [[expr_n^A]]_C))
            ==(buildSet<[[Type]]_C>(n, [[expr_1^B]]_C, ..., [[expr_n^B]]_C))
    ```

EQUALS List

$[\![\langle exprA \rangle = \langle exprB \rangle]\!]^{\text{E}}$

CONTEXT:

Type($\langle exprA \rangle$) = Type($\langle exprB \rangle$) = *List* Type

CODE:

Case $\langle exprA \rangle$, $\langle exprB \rangle$ *of:*

- $\langle exprA \rangle$ = "varName1" $\wedge$ $\langle exprB \rangle$ = "varName2"
  - DEVS-Suite:

    ```
    varName1.equals(varName2)
    ```

  - C++

    ```
    varName1 == varName2
    ```

- ($\langle exprA \rangle$ = "varName', $\langle exprB \rangle = \langle\ \text{expr}_1, \ldots, \text{expr}_n\ \rangle$)
  $\vee$ ($\langle exprA \rangle = \langle\ \text{expr}_1, \ldots, \text{expr}_n\ \rangle$, $\langle exprB \rangle$ = "varName")
  - DEVS-Suite:

    ```
    varName.equals(buildList([[expr_1]]_J, ..., [[expr_n]]_J))
    ```

  - PowerDEVS:

    ```
    varName==(buildList<Type>(n,[[expr_1]]_C, ..., [[expr_n]]_C))
    ```

- $\langle exprA \rangle = \langle \text{expr}_1^A, \ldots, \text{expr}_n^A \rangle$, $\langle exprB \rangle = \langle \text{expr}_1^B, \ldots, \text{expr}_n^B \rangle$
  - DEVS-Suite:

    ```
    (buildList([[expr_1^A]]_J, ..., [[expr_n^A]]_J)).equals(
            buildList([[expr_1^B]]_J, ..., [[expr_n^B]]_J))
    ```

  - PowerDEVS:

    ```
    (buildList<Type>(n,[[expr_1^A]]_C, ..., [[expr_n^A]]_C))==
            (buildList<Type>(n,[[expr_1^B]]_C, ..., [[expr_n^B]]_C))
    ```

Comparación "menor" (<)

LESS Numbers

$[\![\langle expr1\rangle < \langle expr2\rangle]\!]^{\mathrm{L}}$

CONTEXT:

$\mathrm{Type}(\langle expr1\rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

$\mathrm{Type}(\langle expr2\rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

CODE:

- DEVS-Suite:

  `([[`$\langle expr1\rangle$`]]_J) < ([[`$\langle expr2\rangle$`]]_J)`

- C++

  `([[`$\langle expr1\rangle$`]]_C) < ([[`$\langle expr2\rangle$`]]_C)`

LESS Union and number $[\![\langle expr1\rangle < \langle expr2\rangle]\!]^{\mathrm{L}}$

CONTEXT:

$\mathrm{Type}(\langle expr1\rangle) = \mathrm{Type}_1 \cup \ldots \cup \mathrm{Type}_n$

$\mathrm{Type}(\langle expr2\rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

$\mathrm{Type}_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

$\langle expr1\rangle$.type="$[\![\mathrm{Type}_i]\!]^{\mathrm{N}}$"

CODE:

- DEVS-Suite:

  `([[`$\langle expr1\rangle$`]]_J.v_[[`Type_i`]]^N) < [[`$\langle expr2\rangle$`]]_J`

- PowerDEVS:

  `([[`$\langle expr1\rangle$`]]_C.v_[[`Type_i`]]^N) < [[`$\langle expr2\rangle$`]]_C`

COMMENT:

The Union type is already normalized

It must be checked before if $\langle var\rangle$ is currently a number, e.g. $\langle var\rangle \in \mathbb{R}$.

LESS Numbers in Unions $[\![\langle expr1\rangle < \langle expr2\rangle]\!]^{\mathrm{L}}$

CONTEXT:

$\mathrm{Type}(\langle expr1\rangle) = \mathrm{Type}_1 \cup \ldots \cup \mathrm{Type}_n$

$\mathrm{Type}(\langle expr2\rangle) = \mathrm{Type}_{n+1} \cup \ldots \cup \mathrm{Type}_m$

$\mathrm{Type}_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

$\langle expr1\rangle$.type = "$[\![\mathrm{Type}_i]\!]^{\mathrm{N}}$"

$\mathrm{Type}_j = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \mathrm{Time}$

$\langle expr2\rangle$.type = "$[\![\mathrm{Type}_j]\!]^{\mathrm{N}}$"

CODE:

- DEVS-Suite:

  ((⟦⟨*expr*1⟩⟧$_{\texttt{J}}$).type=="⟦Type$_i$⟧$^{\texttt{N}}$")
  && ((⟦⟨*expr*2⟩⟧$_{\texttt{J}}$).type=="⟦Type$_j$⟧$^{\texttt{N}}$")
  && (⟦(⟨*expr*1⟩).v_⟦Type$_i$⟧$^{\texttt{N}}$ < (⟨*expr*2⟩).v_⟦Type$_j$⟧$^{\texttt{N}}$⟧$^{\texttt{E}}$)

- PowerDEVS:

  ((⟦⟨*expr*1⟩⟧$_{\texttt{C}}$).type=="⟦Type$_i$⟧$^{\texttt{N}}$")
  && ((⟦⟨*expr*2⟩⟧$_{\texttt{C}}$).type=="⟦Type$_j$⟧$^{\texttt{N}}$")
  && (⟦(⟨*expr*1⟩).v_⟦Type$_i$⟧$^{\texttt{N}}$ < (⟨*expr*2⟩).v_⟦Type$_j$⟧$^{\texttt{N}}$⟧$^{\texttt{E}}$)

COMMENT:

The Union type is already normalized

It must be checked before if $\langle expr1 \rangle$ and $\langle expr2 \rangle$ are currently numbers, e.g. $\langle expr1 \rangle \in \mathbb{R}$.

Comparación "menor o igual" ($\leq$)

$[\![\langle expr1 \rangle \leq \langle expr2 \rangle]\!]^{\mathrm{L}}$

Estas son las mismas reglas que las de la Sección B.5.4 cambiando $<$ por $<=$ en el código traducido.

Comparación "mayor" ($>$)

$[\![\langle expr1 \rangle > \langle expr2 \rangle]\!]^{\mathrm{L}}$

Estas son las mismas reglas que las de la Sección B.5.4 cambiando $<$ por $>$ en el código traducido.

Comparación "mayor o igual" ($\geq$)

$[\![\langle expr1 \rangle \geq \langle expr2 \rangle]\!]^{\mathrm{L}}$

Estas son las mismas reglas que las de la Sección B.5.4 cambiando $<$ por $>=$ en el código traducido.

Pertenece

BELONGS 1 $[\![\langle exprA\rangle \in \langle exprB\rangle]\!]^{\mathtt{B}}$

CONTEXT:

$\text{Type}(\langle exprB\rangle) = \mathbb{P}\ \text{Type}(\langle exprA\rangle)$

CODE:

Case $\langle exprB\rangle$ *of:*

- $\langle var\rangle \mid \langle defFun\rangle \mid \langle operation\rangle$
  - DEVS-Suite:

    `([[`$\langle exprB\rangle$`]]J).contains([[`$\langle exprA\rangle$`]]J)`
  - C++

    `(([[`$\langle exprB\rangle$`]]C).find([[`$\langle exprA\rangle$`]]C))!=(([[`$\langle exprA\rangle$`]]C).end())`
- $\langle set\rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$
  - DEVS-Suite:

    `(buildSet([[`expr_1`]]J, ... [[`expr_n`]]J)).contains([[`$\langle exprA\rangle$`]]J)`
  - PowerDEVS:

    `((buildSet<[[Type(`$\langle exprA\rangle$`)]]C>(n, [[`expr_1`]]C, ... [[`expr_n`]]J)).find([[`$\langle exprA\rangle$`]]C))!=`
    `((buildSet<[[Type(`$\langle exprA\rangle$`)]]C>(n, [[`expr_1`]]C, ... [[`expr_n`]]C)).end())`
- $\mathbb{N}$
  - DEVS-Suite:

    `isNat([[`$\langle exprA\rangle$`]]J)`
  - PowerDEVS:

    `isNat([[`$\langle exprA\rangle$`]]C)`
- $\mathbb{Z}$
  - DEVS-Suite:

    `isInt([[`$\langle exprA\rangle$`]]J)`
  - PowerDEVS:

    `isInt([[`$\langle exprA\rangle$`]]C)`
- $\mathbb{R}$
  - DEVS-Suite:

    `isReal([[`$\langle exprA\rangle$`]]J)`
  - PowerDEVS:

    `isReal([[`$\langle exprA\rangle$`]]C)`

COMMENT:

No sirve para, por ejemplo, $x \in \mathbb{P}\ \mathbb{N}$

BELONGS 2

$[\![\langle exprA \rangle \in \langle exprB \rangle]\!]^{B}$

CONTEXT:

$\text{Type}(\langle exprA \rangle) = \text{Type}_1 \cup \ldots \cup \text{Type}_n$

$\text{Type}(\langle exprB \rangle) = \mathbb{P}\ \text{Type}_i,\ \text{Type}_i \in \{\text{Type}_1, \ldots, \text{Type}_n\}$

$\text{TypeName}(\langle exprA \rangle) = \text{T\_exprA}$

CODE:

- DEVS-Suite:

```
((⟦⟨exprA⟩⟧J).type=="⟦Typei⟧N")
    && ⟦(⟦⟨exprA⟩⟧J).v_⟦Typei⟧N ∈ ⟨exprB⟩⟧B
```

- PowerDEVS:

```
((⟦⟨exprA⟩⟧C).type=="⟦Typei⟧N")
    && ⟦(⟦⟨exprA⟩⟧C).v_⟦Typei⟧N ∈ ⟨exprB⟩⟧B
```

Subconjunto Propio

PROPER SUBSET

$[\![\langle exprA \rangle \subset \langle exprB \rangle]\!]^{PS}$

CONTEXT:

$\text{Type}(\langle exprB \rangle) = \text{Type}(\langle exprA \rangle) = \mathbb{P}\ \text{Type}$

CODE:

Case $\langle exprA \rangle$, $\langle exprB \rangle$ *of:*

- $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$, $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$
  - DEVS-Suite:

    ```
    isProperSubset(⟦⟨exprA⟩⟧J, ⟦⟨exprB⟩⟧J)
    ```

  - C++

    ```
    isProperSubset(⟦⟨exprA⟩⟧C, ⟦⟨exprB⟩⟧C)
    ```

- $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$, $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$
  - DEVS-Suite:

    ```
    isProperSubset(⟦⟨exprA⟩⟧J, (buildSet(⟦expr1⟧J, ... ⟦exprn⟧J))
    ```

  - PowerDEVS

    ```
    isProperSubset(⟦⟨exprA⟩⟧C, (buildSet<⟦Type⟧C>(n, ⟦expr1⟧C, ... ⟦exprn⟧C))
    ```

- $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$, $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$
  - DEVS-Suite:

    ```
    isProperSubset(buildSet(⟦expr1⟧J, ... ⟦exprn⟧J), ⟦⟨exprA⟩⟧J)
    ```

  - PowerDEVS:

    ```
    isProperSubset(buildSet<⟦Type⟧C>(n, ⟦expr1⟧C, ... ⟦exprn⟧C), ⟦⟨exprA⟩⟧C)
    ```

- $\langle set \rangle = \{\text{expr}_1^A, \ldots, \text{expr}_n^A\}$, $\langle set \rangle = \{\text{expr}_1^B, \ldots, \text{expr}_m^B\}$
  - DEVS-Suite:

```
isProperSubset(buildSet(⟦expr_1^A⟧_J, ... ⟦expr_n^A⟧_J),
               buildSet(⟦expr_1^B⟧_J, ... ⟦expr_m^B⟧_J))
```

- PowerDEVS:

```
isProperSubset(buildSet<⟦Type⟧_C>(n, ⟦expr_1^A⟧_C, ... ⟦expr_n^A⟧_C),
    buildSet<⟦Type⟧_C>(n, ⟦expr_1^B⟧_C, ... ⟦expr_m^B⟧_C))
```

Subconjunto

SUBSET

$[\![\langle exprA \rangle \subseteq \langle exprB \rangle]\!]^{\mathtt{PS}}$

CONTEXT:

$\text{Type}(\langle exprB \rangle) = \text{Type}(\langle exprA \rangle) = \mathbb{P}\ \text{Type}$

CODE:

Case $\langle exprA \rangle$, $\langle exprB \rangle$ *of:*

- $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$, $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$
  - DEVS-Suite:

    ```
    isSubset(⟦⟨exprA⟩⟧_J, ⟦⟨exprB⟩⟧_J)
    ```

  - C++

    ```
    isSubset(⟦⟨exprA⟩⟧_C, ⟦⟨exprB⟩⟧_C)
    ```

- $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$, $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$
  - DEVS-Suite:

    ```
    isSubset(⟦⟨exprA⟩⟧_J, (buildSet(⟦expr_1⟧_J, ... ⟦expr_n⟧_J))
    ```

  - C++

    ```
    isSubset(⟦⟨exprA⟩⟧_C, (buildSet<⟦Type⟧_C>(n, ⟦expr_1⟧_C, ... ⟦expr_n⟧_C))
    ```

- $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\}$, $(\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle)$
  - DEVS-Suite:

    ```
    isSubset(buildSet(⟦expr_1⟧_J, ... ⟦expr_n⟧_J), ⟦⟨exprA⟩⟧_J)
    ```

  - C++

    ```
    isSubset(buildSet<⟦Type⟧_C>(n, ⟦expr_1⟧_C, ... ⟦expr_n⟧_C), ⟦⟨exprA⟩⟧_C)
    ```

- $\langle set \rangle = \{\text{expr}_1^A, \ldots, \text{expr}_n^A\}$, $\langle set \rangle = \{\text{expr}_1^B, \ldots, \text{expr}_m^B\}$
  - DEVS-Suite:

    ```
    isSubset(buildSet(⟦expr_1^A⟧_J, ... ⟦expr_n^A⟧_J),
             buildSet(⟦expr_1^B⟧_J, ... ⟦expr_m^B⟧_J))
    ```

  - C++

    ```
    isSubset(buildSet<⟦Type⟧_C>(n, ⟦expr_1^A⟧_C, ... ⟦expr_n^A⟧_C),
             buildSet<⟦Type⟧_C>(n, ⟦expr_1^B⟧_C, ... ⟦expr_m^B⟧_C))
    ```

B.6. Funciones definidas por el usuario

⟨*function*⟩ ::= function ⟨*id*⟩ is

⟨*id*⟩:⟨*type*⟩{,⟨*id*⟩:⟨*type*⟩}→⟨*id*⟩:⟨*Type*⟩;

[⟨*definitions*⟩]

[⟨*synonyms*⟩]

⟨*sentence*⟩;

{⟨*sentence*⟩;}

end function

Function $[\![\langle function \rangle]\!]^{F}$

CONTEXT:

[⟨*id*⟩] = [funId]

⟨*id*⟩: ⟨*type*⟩{, ⟨*id*⟩: ⟨*type*⟩} → ⟨*id*⟩: ⟨*Type*⟩ = var1: Type1, ..., varn: Typen → retVal: funType

CODE:

- DEVS-Suite:

```
public funType funId(Type1 var1, ..., Typen varn){
    [[retVal: funType]]^D
    [[⟨definitions⟩]]^D /*If any*/
    [[⟨sentence⟩]]^S
    .
    .
    .
    [[⟨sentence⟩]]^S
    return retVal;
}
```

- PowerDEVS:

  - En la cabecera (archivo .h):

```
funType modeName::funId(Type1, ..., Typen);
```

  - En el código fuente (archivo .cpp):

```
funType modeName::funId(Type1 var1, ..., Typen varn){
    [[retVal: funType]]^D
    [[⟨definitions⟩]]^D /*If any*/
    [[⟨sentence⟩]]^S
    .
    .
    .
    [[⟨sentence⟩]]^S
    return retVal;
}
```

COMMENT:

Si el tipo de la función no ha sido ya definido, definirlo.

B.7. Código Java y Código C++

B.7.1. Código Java (Expr)

JAVA CODE Expr

$[\![\langle expr \rangle]\!]_{\mathtt{J}}$

CONTEXT:

TypeName($\langle expr \rangle$)=TypeName

CODE:

Case $\langle expr \rangle$ *of:*

- $\langle id \rangle$ = "Name":

  `Name`

- $\langle idComp \rangle = \langle id \rangle$"."$\langle digit \rangle\{$"."$\langle digit \rangle\}$:

  $[\![\langle id \rangle]\!]_{\mathtt{J}}$`.v`$[\![\langle digit \rangle]\!]_{\mathtt{J}}$`....v`$[\![\langle digit \rangle]\!]_{\mathtt{J}}$

- $\langle digit \rangle$ = "7":

  `7`

- $\langle val \rangle$ = "value":
  Case Type($\langle val \rangle$) of:
  - $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Bool:

    `value`

  - Text:
    ``` ``value'' ```

  - Time:
    - value = ∞
      `INFINITY`
    - otherwise:
      `value`

  - $\langle enum \rangle$ = value:

    `"value"`

  - $\langle Type \rangle \cup \cdots \cup \langle Type \rangle$:

    `new TypeName(`$[\![$`value`$]\!]_{\mathtt{C}}$`)`

- \sigma:

  `sigma`

- \port:

  `port`

- \value:

  `value`

- $\langle vals \rangle = (\mathrm{expr}_1, \ldots, \mathrm{expr}_n)$:

  `TypeName(`$[\![\mathtt{expr}_1]\!]_{\mathtt{J}}$`, ... ,`$[\![\mathtt{expr}_n]\!]_{\mathtt{J}}$`) %Contruye la variable`
    ```

- $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\} \wedge \text{Type}(\langle set \rangle) = \mathbb{P}\ \text{Type}$:

  `buildSet(`$[\![\texttt{expr}_1]\!]_J$`, ..., `$[\![\texttt{expr}_n]\!]_J$`)`

- $\langle list \rangle = \langle \text{expr}_1, \ldots, \text{expr}_n \rangle \wedge \text{Type}(\langle list \rangle) = \text{List Type}$:

  `buildList(`$[\![\texttt{expr}_1]\!]_J$`, ..., `$[\![\texttt{expr}_n]\!]_J$`)`

- $\langle operation \rangle = \langle unOp \rangle \langle operand \rangle$
  Case $\langle unOp \rangle$ *of:*
  - "−":

    `-(`$[\![\langle operand \rangle]\!]_J$`)`
  - "rev":

    `listRev(`$[\![\langle operand \rangle]\!]_J$`)`
  - "head":

    `(`$[\![\langle operand \rangle]\!]_J$`).get(0)`
  - "last":

    `(`$[\![\langle operand \rangle]\!]_J$`).get((`$[\![\langle operand \rangle]\!]_J$`).size()-1)`
  - "front":

    `(`$[\![\langle operand \rangle]\!]_J$`).subList(0,(`$[\![\langle operand \rangle]\!]_J$`).size()-2)`
  - "tail":

    `(`$[\![\langle operand \rangle]\!]_J$`).subList(1,(`$[\![\langle operand \rangle]\!]_J$`).size()-1)`
  - "#":

    `(`$[\![\langle operand \rangle]\!]_J$`).size()`
  - "max":

    `Collections.max(`$[\![\langle operand \rangle]\!]_J$`)`
  - "min":

    `Collections.min(`$[\![\langle operand \rangle]\!]_J$`)`

- $\langle operation \rangle = \langle operand1 \rangle \langle binOp \rangle \langle operand2 \rangle$
  Case $\langle binOp \rangle$ *of:*
  - $+$:

    `(`$[\![\langle operand1 \rangle]\!]_J$`) + (`$[\![\langle operand2 \rangle]\!]_J$`)`
  - $-$:

    `(`$[\![\langle operand1 \rangle]\!]_J$`) - (`$[\![\langle operand2 \rangle]\!]_J$`)`
  - $*$:

    `(`$[\![\langle operand1 \rangle]\!]_J$`) * (`$[\![\langle operand2 \rangle]\!]_J$`)`
  - $\setminus$:
    Case Type($\langle operand1 \rangle$*) of:*
    - $\langle \mathbb{R} \rangle \mid \langle \mathbb{Z} \rangle \mid \langle \mathbb{N} \rangle$:
      `(`$[\![\langle operand1 \rangle]\!]_J$`)\(`$[\![\langle operand2 \rangle]\!]_J$`)`
    - $\mathbb{P}$ Type:
      `setDiff(`$[\![\langle operand1 \rangle]\!]_J$`,`$[\![\langle operand2 \rangle]\!]_J$`)`

- $\frown$:
  `listCat(`$[\![\langle operand1\rangle]\!]_{\texttt{J}}$`,`$[\![\langle operand2\rangle]\!]_{\texttt{J}}$`)`
- $\cap$:
  `setInter(`$[\![\langle operand1\rangle]\!]_{\texttt{J}}$`,`$[\![\langle operand2\rangle]\!]_{\texttt{J}}$`)`
- $\cup$:
  `setUnion(`$[\![\langle operand1\rangle]\!]_{\texttt{J}}$`,`$[\![\langle operand2\rangle]\!]_{\texttt{J}}$`)`

- $\langle mathFun\rangle = \langle funId\rangle(\langle parameter\rangle)$:
  Case $\langle funId\rangle$ *of:*
  - sin:
    `sin(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - cos:
    `cos(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - tan:
    `tan(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - arcsin:
    `asin(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - arccos:
    `acos(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - arctan:
    `atan(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - log:
    `log(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
  - sing:
    `singnum(`$[\![\langle parameter\rangle]\!]_{\texttt{J}}$`)`
- $\langle defFun\rangle$ = funName(param1, ..., paramn):
  `funName(`$[\![$`param1`$]\!]_{\texttt{J}}$`, ..., `$[\![$`paramn`$]\!]_{\texttt{J}}$`)`
- $\langle Type\rangle$:
  Case $\langle Type\rangle$ *of*:
  - $\mathbb{N}$:
    `Integer`
  - $\mathbb{Z}$:
    `Integer`
  - $\mathbb{R}$ | Time:
    `Double`
  - Text | Enum:
    `String`
  - Boolean:
    `Boolean`
  - $\langle synonym\rangle$ = synName:
    `T_synName`

B.7.2. Código C++ (Expr)

C++ CODE Expr

$[\![\langle expr \rangle]\!]_{\texttt{C}}$

CONTEXT:

TypeName($\langle expr \rangle$)=TypeName

CODE:

Case $\langle expr \rangle$ *of:*

- $\langle id \rangle$ = "Name":

  `Name`

- $\langle idComp \rangle = \langle id \rangle$"."$\langle digit \rangle\{$"."$\langle digit \rangle\}$:

  $[\![\langle id \rangle]\!]_{\texttt{C}}$`.v`$[\![\langle digit \rangle]\!]_{\texttt{C}}$`....v`$[\![\langle digit \rangle]\!]_{\texttt{C}}$

- $\langle digit \rangle$ = "7":

  `7`

- $\langle val \rangle$ = "value":
  Case Type($\langle val \rangle$) of:
  - $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Bool:

    `value`

  - Text:
    ``` ``value'' ```

  - Time:
    - value = ∞
      `INFINITY`
    - otherwise:
      `value`

  - $\langle enum \rangle$:

    `"value"`

  - $\langle Type \rangle \cup \cdots \cup \langle Type \rangle$:

    `TypeName(`$[\![$`value`$]\!]_{\texttt{C}}$`)`

- \sigma:

  `sigma`

- \port:

  `port`

- \value:

  `value`

- $\langle vals \rangle = (\text{expr}_1, \ldots, \text{expr}_n)$:

  `TypeName(`$[\![\text{expr}_1]\!]_{\texttt{C}}$`, ... ,`$[\![\text{expr}_n]\!]_{\texttt{C}}$`)`

- $\langle set \rangle = \{\text{expr}_1, \ldots, \text{expr}_n\} \wedge \text{Type}(\langle set \rangle) = \mathbb{P}$ Type:
    ```

`buildSet<Type>(n,`$[\![\texttt{expr}_1]\!]_{\texttt{C}}$`, ..., `$[\![\texttt{expr}_n]\!]_{\texttt{C}}$`)`

- $\langle list \rangle = \langle \text{expr}_1, \ldots, \text{expr}_n \rangle \wedge \text{Type}(\langle list \rangle) = \text{List Type}$:

  `buildList<Type>(n,`$[\![\texttt{expr}_1]\!]_{\texttt{J}}$`, ..., `$[\![\texttt{expr}_n]\!]_{\texttt{J}}$`)`

- $\langle operation \rangle = \langle unOp \rangle \langle operand \rangle$
  Case $\langle unOp \rangle$ *of:*
  - "−":
    `-(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`)`
  - "rev":
    `listRev(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`)`
  - "head":
    `(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).front()`
  - "last":
    `(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).back()`
  - "front":
    `listFront(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`)`
  - "tail":
    `listTail(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`)`
  - "#":
    `(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).size()`
  - "max":
    `*max_element((`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).begin(),(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).end())`
  - "min":
    `*min_element((`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).begin(),(`$[\![\langle operand \rangle]\!]_{\texttt{C}}$`).end())`

- $\langle operation \rangle = \langle operand1 \rangle \langle binOp \rangle \langle operand2 \rangle$
  Case $\langle binOp \rangle$ *of:*
  - +:
    `(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`) + (`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`
  - −:
    `(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`) - (`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`
  - ∗:
    `(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`) * (`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`
  - \:
    Case Type($\langle operand1 \rangle$*) of:*
    - $\langle \mathbb{R} \rangle \mid \langle \mathbb{Z} \rangle \mid \langle \mathbb{N} \rangle$:
      `(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`)\(`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`
    - $\mathbb{P}$ Type:
      `setDiff(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`,(`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`
  - $\frown$:
    `listCat(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`,`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`

- ∩:

  `setInter(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`,`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`

- ∪:

  `setUnion(`$[\![\langle operand1 \rangle]\!]_{\texttt{C}}$`,`$[\![\langle operand2 \rangle]\!]_{\texttt{C}}$`)`

- $\langle mathfun \rangle = \langle funId \rangle(\langle parameter \rangle)$:
  Case $\langle funId \rangle$ *of:*

  - sin:

    `sin(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - cos:

    `cos(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - tan:

    `tan(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - arcsin:

    `asin(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - arccos:

    `acos(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - arctan:

    `atan(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - log:

    `log(`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)`

  - sing:

    `(((`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)==0.0)?0.0:(((`$[\![\langle parameter \rangle]\!]_{\texttt{C}}$`)<0.0)?-1.0:1.0))`

- $\langle defFun \rangle$ = funName(param1, ..., paramn):

  `funName(`$[\![\texttt{param1}]\!]_{\texttt{C}}$`, ..., `$[\![\texttt{paramn}]\!]_{\texttt{C}}$`)`

- $\langle Type \rangle$:
  Case $\langle Type \rangle$ *of*:

  - $\mathbb{N}$:
    `unsigned int`
  - $\mathbb{Z}$:
    `int`
  - $\mathbb{R}$ | Time:
    `double`
  - Text | Enum:
    `string`
  - Boolean:
    `bool`
  - $\langle synonym \rangle$ = synName:
    `T_synName`

B.7.3. Nombres

Name

$[\![\langle Type \rangle]\!]^{\mathbb{N}}$

CODE:

Case $\langle Type \rangle$ of:

- $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid$ Time:
  `Number`
- Text:
  `Text`
- Boolean:
  `Boolean`
- Enum:
  `Enum`
- $\langle synonym \rangle$ = synName:
  `synName`

B.8. Funciones Auxiliar

AUXILIAR FUNCTIONS

CODE:

- **setUnion:**
  - DEVS-Suite:

```
public static <T> Set<T> setUnion(Set<T> setA, Set<T> setB){
    Set<T> union = new HashSet<T>(setA);
    union.addAll(setB);
    return union;
}
```

  - C++
    Header file:

```
template <class T>
std::set<T> setUnion(std::set<T> setA, std::set<T> setB);
```

    Source Code File:

```
template <class T>
std::set<T> modelName::setUnion(std::set<T> setA,
                                std::set<T> setB){
    std::set<T> res;
    std::set_union(setA.begin(),setA.end(),setB.begin(),
                   setB.end(),inserter(res,res.begin()));
    return res;
}
```

- **setInter:**
  - DEVS-Suite:

```
public static <T> Set<T> setInter(Set<T> setA, Set<T> setB){
    Set<T> inter = new HashSet<T>(setA);
    inter.retainAll(setB);
    return inter;
}
```

  - C++
    Header file:

```
template <class T>
std::set<T> setInter(std::set<T> setA,
                     std::set<T> setB);
```

    Source Code File:

```
template <class T>
std::set<T> modelName::setInter(std::set<T> setA,
                                std::set<T> setB){
    std::set<T> res;
    std::set_intersection(setA.begin(),
                          setA.end(),
                          setB.begin(),
                          setB.end(),
        inserter(res,res.begin()));
    return res;
}
```

- **setDiff:**
  - DEVS-Suite:

```
public static <T> Set<T> setDiff(Set<T> setA, Set<T> setB){
    Set<T> diff = new HashSet<T>(setA);
    diff.removeAll(setB);
    return diff;
}
```

  - C++
    Header file:

```
template <class T>
std::set<T> setDiff(std::set<T> setA, std::set<T> setB);
```

    Source Code File:

```
template <class T>
std::set<T> modelName::setDiff(std::set<T> setA,
                               std::set<T> setB){
    std::set<T> res;
    std::set_difference(setA.begin(),setA.end(),
                        setB.begin(),setB.end(),
        inserter(res,res.begin()));
    return res;
}
```

- **listCat:**
  - DEVS-Suite:

```
public static <T> List<T> listCat(List<T> listA, List<T> listB){
    List<T> cat= new ArrayList<T>(listA);
    cat.addAll(listB);
    return cat;
}
```

  - PowerDEVS:
    Header file:

```
template <class T>
std::list<T> listCat(std::list<T> listA, std::list<T> listB);
```

    Source Code File:

```
template <class T>
std::list<T> modelName::listCat(std::list<T> listA,
                                std::list<T> listB){
    std::list<T> res=listA;
    res.insert(res.end(),listB.begin(),listB.end());
    return res;
}
```

- **listFront:**
  - PowerDEVS:
    Header file:

```
template <class T>
std::list<T> listFront(std::list<T> listA,std::list<T> listB);
```

    Source Code File:

```
template <class T>
std::list<T> modelName::listFront(std::list<T> list){
    std::list<T> res=list;
    res.erase(--res.end());
    return res;
}
```

- **listTail:**

- PowerDEVS:
  Header file:

```
template <class T> std::list<T> listTail(std::list<T> list);
```

  Source Code File:

```
template <class T>
std::list<T> modelName::listTail(std::list<T> list){
    std::list<T> res=list;
    res.erase(res.begin());
    return res;
}
```

- **listRev:**
  - DEVS-Suite:

```
public static <T> List<T> listRev(List<T> list){
    List<T> rev= new ArrayList<T>(list);
    Collections.reverse(rev);
    return rev;
}
```

  - PowerDEVS:
    Header file:

```
template <class T>
std::list<T> listRev(std::list<T> listA, std::list<T> listB);
```

    Source Code File:

```
template <class T>
std::list<T> modelName::listRev(std::list<T> list){
    std::list<T> res=list;
    res.reverse();
    return res;
}
```

- **buildSet:**
  - DEVS-Suite:

```
public static <T> Set<T> buildSet(T ...elements){
    Set<T> set = new HashSet<T>(Arrays.asList(elements));
    return set;
}
```

  - PowerDEVS:
    Header file:

```
template <class T> std::set<T> buildSet(int, ...);
```

    Source Code File:

```
template <class T>
std::set<T> modelName::buildSet(int n, ...){
    std::set<T> res;
    va_list vl;
    va_start(vl,n);
    for (int i=0; i<n; i++){
        T val=(T)(va_arg(vl,T));
        res.insert(val);
    }
    va_end(vl);
    return res;
}
```

- **buildList:**
  - DEVS-Suite:

```
public static <T> List<T> buildList(T ...elements){
    List<T> list=new ArrayList<T>(Arrays.asList(elements));
    return list;
}
```

  - PowerDEVS:
    Header file:

```
template <class T> std::list<T> buildList(int, ...);
```

    Source Code File:

```
template <class T>
std::list<T> modelName::buildList(int n, ...){
    std::list<T> res;
    T val;
    va_list vl;
    va_start(vl,n);
    for (int i=0; i<n; i++){
        val=va_arg(vl,T);
        res.push_back(val);
    }
    va_end(vl);
    return res;
}
```

- **isProperSubset:**
  - DEVS-Suite:

```
public static <T> Boolean isProperSubset(Set<T> setA,
                                         Set<T> setB){
    return setB.containsAll(setA) &&
           (setA).size()<(setB).size();
}
```

  - PowerDEVS:
    Header file:

```
template <class T>
bool isProperSubset(std::set<T>, std::set<T>);
```

    Source Code File:

```
template <class T>
bool modelName::isProperSubset(std::set<T> setA,
                               std::set<T> setB){
    return std::includes(setB.begin(), setB.end(),
                         setA.begin(), setA.end())
        && setA.size()<setB.size();
}
```

- **isSubset:**
  - DEVS-Suite:

```
public static <T> Boolean isSubset(Set<T> setA, Set<T> setB){
    return setB.containsAll(setA);
}
```

  - PowerDEVS:
    Header file:

```
template <class T> bool isSubset(std::set<T>, std::set<T>);
```

    Source Code File:

```
template <class T>
bool modelName::isSubset(std::set<T> setA,
                         std::set<T> setB){
    return std::includes(setB.begin(), setB.end(),
                         setA.begin(), setA.end());
}
```

- **isNat:**
  - DEVS-Suite:

```
public static Boolean isNat(Object var){
    if (var instanceof Integer)
        if ((Integer)var >=0) return true;
        else return false;
    else return false;
}
```

  - PowerDEVS:
    Header file:

```
template <typename T> bool isNat(T var);
bool isNat(int var);
```

    Source Code File:

```
bool test2::isNat(int var){
    return (var>0);
}
template <typename T> bool test2::isNat(T var){
    return false;
}
```

- **isInt:**
  - DEVS-Suite:

```
public static Boolean isInt(Object var){
    if (var instanceof Integer) return true;
    else return false;
}
```

  - PowerDEVS:
    Header file:

```
template <typename T> bool isInt(T var);
```

    Source Code File:

```
template <typename T> bool test2::isInt(T var){
  return (typeid(var)==typeid(int));
}
```

- **isReal:**
  - DEVS-Suite:

```
public static Boolean isReal(Object var){
    if ((var instanceof Double)
        || (var instanceof Integer)) return true;
    else return false;
}
```

  - PowerDEVS:
    Header file:

```
template <typename T> bool isReal(T var);
```

    Source Code File:

```
template <typename T> bool test2::isReal(T var){
  return (typeid(var)==typeid(double)
         || typeid(var)==typeid(int));
}
```

- **toInteger:**
  - DEVS-Suite:

```
public static Integer toInteger(Object var){
  Integer res=null;
  if (var instanceof Integer){
    res=((Integer) var).intValue();
  }
  else if (var instanceof Double){
    res=((Double) var).intValue();
  }
  return res;
}
```

- **toDouble:**
  - DEVS-Suite:

```
public static Double toDouble(Object var){
  Double res=null;
  if (var instanceof Integer){
    res=((Integer) var).doubleValue();
  }
  else if (var instanceof Double){
    res=((Double) var).doubleValue();
  }
  return res;
}
```

Apéndice C

SCCs generadas para el Ascensor

C.1. Funciones de transición definidas por casos

- $IniSt_1 = \{s : s \in S \mid eng = \mathsf{stopped} \wedge fc = \emptyset\}$,
  $InPairs_1 = \{((in, n), t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+ \mid n \neq f\}$

- $IniSt_2 = \{s : s \in S \mid eng = \mathsf{up}\}$,
  $InPairs_2 = \{((in, \mathsf{fsig}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_3 = \{s : s \in S \mid eng = \mathsf{down}\}$,
  $InPairs_3 = \{((in, \mathsf{fsig}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_4 = \{s : s \in S \mid eng = \mathsf{stopped}\}$,
  $InPairs_4 = \{((in, \mathsf{ds_{on}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_5 = \{s : s \in S \mid eng \neq \mathsf{stopped}\}$,
  $InPairs_5 = \{((in, \mathsf{ds_{on}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_6 = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge ws = 0 \wedge sw = 0\}$,
  $InPairs_6 = \{((in, \mathsf{ds_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_7 = \{s : s \in S \mid d \neq \mathsf{open} \vee fc = \emptyset \vee ws = 1 \vee sw = 1\}$,
  $InPairs_7 = \{((in, \mathsf{ds_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_8 = \{s : s \in S \mid eng = \mathsf{stopped}\}$,
  $InPairs_8 = \{((in, \mathsf{ws_{on}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_9 = \{s : s \in S \mid eng \neq \mathsf{stopped}\}$,
  $InPairs_9 = \{((in, \mathsf{ws_{on}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{10} = \{s : s \in S \mid fc \neq \emptyset \wedge d = \mathsf{open}\}$,
  $InPairs_{10} = \{((in, \mathsf{ws_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{11} = \{s : s \in S \mid fc = \emptyset \vee d \neq \mathsf{open}\}$,
  $InPairs_{11} = \{((in, \mathsf{ws_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{12} = \{s : s \in S\}$,
  $InPairs_{12} = \{(\mathsf{s_{on}}, t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{13} = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge fc \neq f\}$,
  $InPairs_{13} = \{((in, \mathsf{s_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{14} = \{s : s \in S \mid fc = \emptyset\}$,
  $InPairs_{14} = \{(\mathsf{s_{off}}, t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{15} = \{s : s \in S \mid fc \neq \emptyset \wedge d = \mathsf{closed}\}$,
  $InPairs_{15} = \{((in, \mathsf{s_{off}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{16} = \{s : s \in S \mid d = \mathsf{closing}\}$,
  $InPairs_{16} = \{((in, \mathsf{od_{press}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{17} = \{s : s \in S \mid d = \mathsf{open} \wedge fc \neq \emptyset \wedge ds = 0 \wedge ws = 0 \wedge sw = 0\}$,
  $InPairs_{17} = \{((in, \mathsf{cd_{press}}), t) : t \in \mathbb{R}_0^+\}$

- $IniSt_{18} = \{s : s \in S \mid nt = O \wedge eng \neq \mathsf{stopped} \wedge f = fc \wedge f \neq 0\}$
  $InPairs_{18} = \{(\tau, 0)\}$

- $IniSt_{19} = \{s : s \in S \mid nt = O \wedge eng \neq \mathsf{stopped} \wedge f = fc \wedge f = 0\}$
  $InPairs_{19} = \{(\tau, 0)\}$

- $IniSt_{20} = \{s : s \in S \mid nt = O \wedge eng \neq \mathsf{stopped} \wedge f \neq fc\}$
  $InPairs_{20} = \{(\tau, 0)\}$

- $IniSt_{21} = \{s : s \in S \mid nt = O \wedge sw = 1 \wedge eng \neq \mathsf{stopped}\}$
  $InPairs_{21} = \{(\tau, 0)\}$

- $IniSt_{22} = \{s : s \in S \mid nt = O \wedge sw = 1 \wedge eng = \mathsf{stopped}\}$
  $InPairs_{22} = \{(\tau, 0)\}$

- $IniSt_{23} = \{s : s \in S \mid nt = O \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc > f\}$
  $InPairs_{23} = \{(\tau, 0)\}$

- $IniSt_{24} = \{s : s \in S \mid nt = O \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc < f\}$
  $InPairs_{24} = \{(\tau, 0)\}$

- $IniSt_{25} = \{s : s \in S \mid nt = O \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{open} \wedge fc \neq \emptyset\}$
  $InPairs_{25} = \{(\tau, 0)\}$
- $IniSt_{26} = \{s : s \in S \mid nt = O \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc = \emptyset\}$
  $InPairs_{26} = \{(\tau, 0)\}$
- $IniSt_{27} = \{s : s \in S \mid nt = O \wedge d = \mathsf{closing}\}$
  $InPairs_{27} = \{(\tau, 0)\}$
- $IniSt_{28} = \{s : s \in S \mid nt = \mathsf{D}_1 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0\}$
  $InPairs_{28} = \{(\tau, 0)\}$
- $IniSt_{29} = \{s : s \in S \mid nt = \mathsf{D}_1 \wedge \neg(ds = 0 \wedge ws = 0 \wedge sw = 0)\}$
  $InPairs_{29} = \{(\tau, 0)\}$
- $IniSt_{30} = \{s : s \in S \mid nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f\}$
  $InPairs_{30} = \{(\tau, 0)\}$
- $IniSt_{31} = \{s : s \in S \mid nt = \mathsf{D}_2 \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f\}$
  $InPairs_{31} = \{(\tau, 0)\}$
- $IniSt_{32} = \{s : s \in S \mid nt = \mathsf{D}_2 \wedge \neg(ds = 0 \wedge ws = 0 \wedge sw = 0)\}$
  $InPairs_{32} = \{(\tau, 0)\}$
- $IniSt_{33} = \{s : s \in S \mid nt = \mathsf{A}\}$
  $InPairs_{33} = \{(\tau, 0)\}$
- $IniSt_{34} = \{s : s \in S \mid nt = \mathsf{GF} \wedge f \neq 0 \wedge f \neq \bot \wedge d = \mathsf{open} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0\}$
  $InPairs_{34} = \{(\tau, 0)\}$
- $IniSt_{35} = \{s : s \in S \mid nt = \mathsf{GF} \wedge f \neq 0 \wedge f \neq \bot \wedge d = \mathsf{open} \wedge \neg(ds = 0 \wedge ws = 0 \wedge sw = 0)\}$
  $InPairs_{35} = \{(\tau, 0)\}$

C.2. Conjuntos definidos por extensión

- $IniSt_{36} = \{s : s \in S\}$,
  $InPairs_{36} = \{((in, n), t) : n \in \mathbb{N}, t \in \mathbb{R}_0^+\}$
- $IniSt_{37} = \{s : s \in S\}$,
  $InPairs_{37} = \{((in, \mathsf{fsig}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{38} = \{s : s \in S\}$,
  $InPairs_{38} = \{((in, \mathsf{ws_{on}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{39} = \{s : s \in S\}$,
  $InPairs_{39} = \{((in, \mathsf{ws_{off}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{40} = \{s : s \in S\}$,
  $InPairs_{40} = \{((in, \mathsf{ds_{on}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{41} = \{s : s \in S\}$,
  $InPairs_{41} = \{((in, \mathsf{ds_{off}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{42} = \{s : s \in S\}$,
  $InPairs_{42} = \{((in, \mathsf{od_{press}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{43} = \{s : s \in S\}$,
  $InPairs_{43} = \{((in, \mathsf{cd_{press}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{44} = \{s : s \in S\}$,
  $InPairs_{44} = \{((in, \mathsf{s_{on}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{45} = \{s : s \in S\}$,
  $InPairs_{45} = \{((in, \mathsf{s_{off}}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{46} = \{s : s \in S \mid eng = \mathsf{up}\}$,
  $InPairs_{46} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{47} = \{s : s \in S \mid eng = \mathsf{down}\}$,
  $InPairs_{47} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{48} = \{s : s \in S \mid eng = \mathsf{stopped}\}$,
  $InPairs_{48} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{49} = \{s : s \in S \mid d = \mathsf{open}\}$,
  $InPairs_{49} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{50} = \{s : s \in S \mid d = \mathsf{closed}\}$,
  $InPairs_{50} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{51} = \{s : s \in S \mid d = \mathsf{closing}\}$,
  $InPairs_{51} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{52} = \{s : s \in S \mid ws = 0\}$,
  $InPairs_{52} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{53} = \{s : s \in S \mid ws = 1\}$,
  $InPairs_{53} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{54} = \{s : s \in S \mid ds = 0\}$,
  $InPairs_{54} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{55} = \{s : s \in S \mid ds = 1\}$,
  $InPairs_{55} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{56} = \{s : s \in S \mid a = 0\}$,
  $InPairs_{56} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{57} = \{s : s \in S \mid a = 1\}$,
  $InPairs_{57} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{58} = \{s : s \in S \mid sw = 0\}$,
  $InPairs_{58} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{59} = \{s : s \in S \mid sw = 1\}$,
  $InPairs_{59} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

- $IniSt_{60} = \{s : s \in S \mid fc = n \in \mathbb{N}\}$,
  $InPairs_{60} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{61} = \{s : s \in S \mid fc = \emptyset\}$,
  $InPairs_{61} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{62} = \{s : s \in S \mid nt = \mathsf{A}\}$,
  $InPairs_{62} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{63} = \{s : s \in S \mid nt = \mathsf{D_1}\}$,
  $InPairs_{63} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{64} = \{s : s \in S \mid nt = \mathsf{D_2}\}$,
  $InPairs_{64} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{65} = \{s : s \in S \mid nt = \mathsf{GF}\}$,
  $InPairs_{65} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{66} = \{s : s \in S \mid nt = \mathsf{O}\}$,
  $InPairs_{66} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

C.3. Particiones estándar

- $IniSt_{67} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc = f \wedge f = 0\}$,
  $InPairs_{67} = \{(\tau, 0)\}$
- $IniSt_{68} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc = f \wedge f > 0\}$,
  $InPairs_{68} = \{(\tau, 0)\}$
- $IniSt_{69} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc < f \wedge fc > 0\}$,
  $InPairs_{69} = \{(\tau, 0)\}$
- $IniSt_{70} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc > f \wedge f > 0\}$,
  $InPairs_{70} = \{(\tau, 0)\}$
- $IniSt_{71} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc < f \wedge fc = 0\}$,
  $InPairs_{71} = \{(\tau, 0)\}$
- $IniSt_{72} = \{s : s \in S \mid nt = \mathsf{O} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge d = \mathsf{closed} \wedge fc \neq \emptyset \wedge fc > f \wedge f = 0\}$,
  $InPairs_{72} = \{(\tau, 0)\}$
- $IniSt_{73} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc = f \wedge f = 0\}$,
  $InPairs_{73} = \{(\tau, 0)\}$
- $IniSt_{74} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc = f \wedge f > 0\}$,
  $InPairs_{74} = \{(\tau, 0)\}$
- $IniSt_{75} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f \wedge fc > 0\}$,
  $InPairs_{75} = \{(\tau, 0)\}$
- $IniSt_{76} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f \wedge f > 0\}$,
  $InPairs_{76} = \{(\tau, 0)\}$
- $IniSt_{77} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc < f \wedge fc = 0\}$,
  $InPairs_{77} = \{(\tau, 0)\}$
- $IniSt_{78} = \{s : s \in S \mid nt = \mathsf{D_2} \wedge ds = 0 \wedge ws = 0 \wedge sw = 0 \wedge fc > f \wedge f = 0\}$,
  $InPairs_{78} = \{(\tau, 0)\}$

C.4. Particiones del tiempo

- $IniSt_{79} = \{s : s \in S\}$
  $InPairs_{79} = \{(x, 0) : x \in X \cup \{\tau\}\}$
- $IniSt_{80} = \{s : s \in S\}$
  $InPairs_{80} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge 0 < t < \mathsf{T_{D_1}}\}$
- $IniSt_{81} = \{s : s \in S\}$
  $InPairs_{81} = \{(x, \mathsf{T_{D_1}}) : x \in X \cup \{\tau\}\}$
- $IniSt_{82} = \{s : s \in S\}$
  $InPairs_{82} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge \mathsf{T_{D_1}} < t < \mathsf{T_{D_2}}\}$
- $IniSt_{83} = \{s : s \in S\}$
  $InPairs_{83} = \{(x, \mathsf{T_{D_2}}) : x \in X \cup \{\tau\}\}$
- $IniSt_{84} = \{s : s \in S\}$
  $InPairs_{84} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge \mathsf{T_{D_2}} < t < \mathsf{T_A}\}$
- $IniSt_{85} = \{s : s \in S\}$
  $InPairs_{85} = \{(x, \mathsf{T_A}) : x \in X \cup \{\tau\}\}$
- $IniSt_{86} = \{s : s \in S\}$
  $InPairs_{86} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge \mathsf{T_A} < t < \mathsf{T_{GF}}\}$
- $IniSt_{87} = \{s : s \in S\}$
  $InPairs_{87} = \{(x, \mathsf{T_{GF}}) : x \in X \cup \{\tau\}\}$
- $IniSt_{88} = \{s : s \in S\}$
  $InPairs_{88} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge t > \mathsf{T_{GF}}\}$

Apéndice D

SCCs generadas para la máquina expendedora de gaseosas

D.1. Funciones de transición definidas por casos

- $IniSt_1 = \{s : s \in S \mid m \in \{\mathsf{idle}, \mathsf{operating}\}\}$,
  $InPairs_1 = \{((in, c), t) : c \in \{100, 50, 25\}, t \in \mathbb{R}_0^+\}$
- $IniSt_2 = \{s : s \in S \mid d \geq np\}$,
  $InPairs_2 = \{((in, \mathsf{getNormal}), t) \mid t \in \mathbb{R}_0^+\}$
- $IniSt_3 = \{s : s \in S \mid d \geq dp\}$,
  $InPairs_3 = \{((in, \mathsf{getDiet}), t) \mid t \in \mathbb{R}_0^+\}$
- $IniSt_4 = \{s : s \in S\}$,
  $InPairs_4 = \{((in, \mathsf{cancel}), t) \mid t \in \mathbb{R}_0^+\}$
- $IniSt_5 = \{s : s \in S\}$,
  $InPairs_5 = \{((in, \mathsf{moneyRetreated}), t) \mid t \in \mathbb{R}_0^+\}$
- $IniSt_6 = \{s : s \in S \mid m = \mathsf{operating} \wedge ot < it\}$,
  $InPairs_6 = \{(\tau, 0)\}$
- $IniSt_7 = \{s : s \in S \mid m = \mathsf{finishOp} \wedge ot < it\}$,
  $InPairs_7 = \{(\tau, 0)\}$
- $IniSt_8 = \{s : s \in S \mid m = \mathsf{cancelOp} \wedge ot < it\}$,
  $InPairs_8 = \{(\tau, 0)\}$
- $IniSt_9 = \{s : s \in S \mid m = \mathsf{waitRetChange} \wedge ot < it\}$,
  $InPairs_9 = \{(\tau, 0)\}$
- $IniSt_{10} = \{s : s \in S \mid m = \mathsf{idle} \wedge ot < it\}$,
  $InPairs_{10} = \{(\tau, 0)\}$
- $IniSt_{11} = \{s : s \in S \mid m = \mathsf{idle} \wedge it \leq ot\}$,
  $InPairs_{11} = \{(\tau, 0)\}$

D.2. Particiones estándar

- $IniSt_{12} = \{s : s \in S \mid d = np = 0\}$,
  $InPairs_{12} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{13} = \{s : s \in S \mid d > 0 \wedge np = 0\}$,
  $InPairs_{13} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{14} = \{s : s \in S \mid d = 0 \wedge np > 0\}$,
  $InPairs_{14} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{15} = \{s : s \in S \mid d > 0 \wedge np > 0\}$,
  $InPairs_{15} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{16} = \{s : s \in S \mid d = dp = 0\}$,
  $InPairs_{16} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}+_0\}$
- $IniSt_{17} = \{s : s \in S \mid d > 0 \wedge dp = 0\}$,
  $InPairs_{17} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}+_0\}$
- $IniSt_{18} = \{s : s \in S \mid d = 0 \wedge dp > 0\}$,
  $InPairs_{18} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}+_0\}$
- $IniSt_{19} = \{s : s \in S \mid d > 0 \wedge dp > 0\}$,
  $InPairs_{19} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}+_0\}$
- $IniSt_{20} = \{s : s \in S \mid d = np = 0\}$,
  $InPairs_{20} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{21} = \{s : s \in S \mid d = np \wedge np > 0\}$,
  $InPairs_{21} = \{(x, t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

- $IniSt_{22} = \{s : s \in S \mid 0 < d < np\}$,
  $InPairs_{22} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{23} = \{s : s \in S \mid 0 < np < d\}$,
  $InPairs_{23} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{24} = \{s : s \in S \mid d = dp = 0\}$,
  $InPairs_{24} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{25} = \{s : s \in S \mid d = dp \wedge dp > 0\}$,
  $InPairs_{25} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{26} = \{s : s \in S \mid 0 < d < dp\}$,
  $InPairs_{26} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{27} = \{s : s \in S \mid 0 < dp < d\}$,
  $InPairs_{27} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{28} = \{s : s \in S \mid d = 0\}$,
  $InPairs_{28} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

Para la operación $d \oslash (coins1d, coins50c, coins25c)$ existen 351 SCCs. aquí solo mostramos algunas de ellas:

- $IniSt_{29} = \{s : s \in S \mid 0 < d < coins1d \wedge coins25c = d - coins1d' - coins50c' = 0\}$,
  $InPairs_{29} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{30} = \{s : s \in S \mid coins1d < d = 1 \wedge 0 < coins25c < d - coins1d' - coins50c'\}$,
  $InPairs_{30} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{31} = \{s : s \in S \mid 1 < d < coins1d \wedge 0.50 < d - coins1d' < coins50c\}$,
  $InPairs_{31} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{32} = \{s : s \in S \mid d > 0 \wedge coins1d = coins50c = coins25c = 0\}$,
  $InPairs_{32} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$

D.3. Conjuntos definidos por extensión

- $IniSt_{33} = \{s : s \in S \mid m = \mathsf{idle}\}$,
  $InPairs_{33} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{34} = \{s : s \in S \mid m = \mathsf{operating}\}$,
  $InPairs_{34} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{35} = \{s : s \in S \mid m = \mathsf{finishOp}\}$,
  $InPairs_{35} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{36} = \{s : s \in S \mid m = \mathsf{cancelOp}\}$,
  $InPairs_{36} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{37} = \{s : s \in S \mid m = \mathsf{waitRetChange}\}$,
  $InPairs_{37} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+\}$
- $IniSt_{38} = \{s : s \in S\}$,
  $InPairs_{38} = \{((in, 25), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{39} = \{s : s \in S\}$,
  $InPairs_{39} = \{((in, 50), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{40} = \{s : s \in S\}$,
  $InPairs_{40} = \{((in, 100), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{41} = \{s : s \in S\}$,
  $InPairs_{41} = \{((in, \mathsf{getNormal}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{42} = \{s : s \in S\}$,
  $InPairs_{42} = \{((in, \mathsf{getDiet}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{43} = \{s : s \in S\}$,
  $InPairs_{43} = \{((in, \mathsf{cancel}), t) : t \in \mathbb{R}_0^+\}$
- $IniSt_{44} = \{s : s \in S\}$,
  $InPairs_{44} = \{((in, \mathsf{moneyRetreated}), t) : t \in \mathbb{R}_0^+\}$

D.4. Particiones del tiempo

- $IniSt_{45} = \{s : s \in S\}$,
  $InPairs_{45} = \{(\tau, 0)\}$
- $IniSt_{46} = \{s : s \in S \mid it > 0\}$,
  $InPairs_{46} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge 0 < t < it\}$
- $IniSt_{47} = \{s : s \in S \mid it > 0\}$,
  $InPairs_{47} = \{(x, it) : x \in X \cup \{\tau\}\}$
- $IniSt_{48} = \{s : s \in S\}$,
  $InPairs_{48} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > it\}$
- $IniSt_{49} = \{s : s \in S \mid ot > 0\}$,
  $InPairs_{49} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge 0 < t < ot\}$
- $IniSt_{50} = \{s : s \in S \mid ot > 0\}$,
  $InPairs_{50} = \{(x, ot) : x \in X \cup \{\tau\}\}$
- $IniSt_{51} = \{s : s \in S\}$,
  $InPairs_{51} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > ot\}$
- $IniSt_{52} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{52} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge 0 < t < it\}$
- $IniSt_{53} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{53} = \{(x, it) : x \in X \cup \{\tau\}\}$

- $IniSt_{54} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{54} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge it < t < ot\}$

- $IniSt_{55} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{55} = \{(x, ot) : x \in X \cup \{\tau\}\}$

- $IniSt_{56} = \{s : s \in S \mid 0 < it < ot\}$,
  $InPairs_{56} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > ot\}$

- $IniSt_{57} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{57} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge 0 < t < ot\}$

- $IniSt_{58} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{58} = \{(x, ot) : x \in X \cup \{\tau\}\}$

- $IniSt_{59} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{59} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \wedge ot < t < it\}$

- $IniSt_{60} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{60} = \{(x, it) : x \in X \cup \{\tau\}\}$

- $IniSt_{61} = \{s : s \in S \mid 0 < ot < it\}$,
  $InPairs_{61} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > it\}$

- $IniSt_{62} = \{s : s \in S \mid ot = it\}$,
  $InPairs_{62} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t < ot\}$

- $IniSt_{63} = \{s : s \in S \mid ot = it\}$,
  $InPairs_{63} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t = ot\}$

- $IniSt_{64} = \{s : s \in S \mid ot = it\}$,
  $InPairs_{64} = \{(x,t) : x \in X \cup \{\tau\}, t \in \mathbb{R}_0^+ \mid t > ot\}$

Bibliografía

[1] Zeigler, B. P., Praehofer, H., Kim, T. G. Theory of Modeling and Simulation, Second Edition. London: Academic Press, 2000.

[2] Zeigler, B. P., Vahie, S. Devs Formalism And Methodology: Unity Of Conception/Diversity Of Application. En: In Proceedings of the 25th Winter Simulation Conference, págs. 573–579. ACM Press, 1993.

[3] Wainer, G. A. Discrete-Event Modeling and Simulation: a Practitioner's approach. CRC Press. Taylor and Francis, 2009.

[4] Cho, H. J., Cho, Y. K. DEVS-C++ Reference Guide. The University of Arizona, 1997.

[5] Kim, T. G. DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models. Korea Advance Institute of Science and Technology, 1994.

[6] Bergero, F., Kofman, E. Powerdevs: a tool for hybrid system modeling and real-time simulation. *SIMULATION*, **87** (1-2), 113–132, 2011.

[7] Wainer, G., Christen, G., Dobniewski, A. Defining models with the CD++ toolkit. En: In Proceedings of the European Simulation Symposium 2001. Marseille, France: SCS Publisher, 2001.

[8] Rodríguez, D. A., Wainer, G. A. New extensions to the CD++ tool. En: In Proceedings of the 32 nd SCS Summer Computer Simulation Conference. Chicago, IL: SCS Publisher, 1999.

[9] Kim, S., Sarjoughian, H. S., Elamvazhuthi, V. DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring. En: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim '09, págs. 161:1–161:7. San Diego, CA, USA: Society for Computer Simulation International, 2009.

[10] Filippi, J. B., Delhom, M., Bernardi, F. The JDEVS Environmental Modeling and Simulation Environment. En: In Proceedings of IEMSS 2002, págs. 283–288. 2002.

[11] Cellier, F. E., Kofman, E. Continuous System Simulation. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[12] Sarjoughian, H. S., Zeigler, B. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. *Simulation Series*, **30**, 29–36, 1998.

[13] Sarjoughian, H. S., Singh, R. Building Simulation Modeling Environments Using Systems Theory and Software Architecture Principles. En: Proceedings of the Advanced Simulation Technology Conference, págs. 99–104. 2004.

[14] Wainer, G. CD++: a toolkit to develop DEVS models. *Software - Practice and Experience*, **32**, 1261–1306, 2002.

[15] Bergero, F., Kofman, E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *SIMULATION*, 2010.

[16] Spivey, J. M. The Z notation: a reference manual. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1992.

[17] Abrial, J.-R. The B-book: Assigning Programs to Meanings. New York, NY, USA: Cambridge University Press, 1996.

[18] Jackson, D. Alloy: A logical modelling language. En: ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings, pág. 1. 2003.

[19] DEVS Standardization Group. Website, http://cell-devs.sce.carleton.ca/devsgroup/, Último Acceso: 13-11-2014.

[20] Vangheluwe, H., Bolduc, L., Posse, E. DEVS Standardization: some thoughts. En: Winter Simulation Conference. 2001.

[21] Touraille, L., Traoré, M. K., Hill, D. R. C. A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models. En: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim '09, págs. 163:1–163:6. San Diego, CA, USA: Society for Computer Simulation International, 2009.

[22] Hong, K. J., Kim, T. G. DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems. *Inf. Softw. Technol.*, **48** (4), 221–234, abr. 2006.

[23] Mittal, S., Douglass, S. A. DEVSML 2.0: The Language and the Stack. En: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, TMS/DEVS '12, págs. 17:1–17:12. San

Diego, CA, USA: Society for Computer Simulation International, 2012. URL http://dl.acm.org/citation.cfm?id=2346616.2346633.

[24] Fishwick, P. Using XML for simulation modeling. En: Proceedings of the Winter Simulation Conference, 2002, tomo 1, págs. 616–622 vol.1. 2002.

[25] Rohl, M., Uhrmacher, A. Flexible integration of XML into modeling and simulation systems. En: Proceedings of the Winter Simulation Conference, 2005, págs. 8 pp.–. 2005.

[26] Sarjoughian, H. S., Chen, Y. Standardizing DEVS Models: An Endogenous Standpoint. En: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, TMS-DEVS '11, págs. 266–273. San Diego, CA, USA: Society for Computer Simulation International, 2011.

[27] Touraille, L. Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation. Tesis Doctoral, Doctoral dissertation, Université d'Auvergne, 2012.

[28] Lamport, L. LaTeX: A Document Preparation System (2nd Edition). Addison-Wesley Professional, 1994.

[29] Hollmann, D. A., Cristiá, M., Frydman, C. Adapting Model-Masked Testing Techniques to DEVS Models Validation. En: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, TMS/-DEVS '12, págs. 6:1–6:8. San Diego, CA, USA: Society for Computer Simulation International, 2012.

[30] Hollmann, D. A., Cristiá, M., Frydman, C. A family of simulation criteria to guide DEVS models validation rigorously, systematically and semi-automatically. *Simulation Modelling Practice and Theory*, **49** (0), 1 – 26, 2014.

[31] DoDD 5000.59. DoD Modeling and Simulation (M&S) Management, Jan 4, 1994.

[32] Labiche, Y., Wainer, G. Towards the verification and validation of DEVS models. En: in Proceedings of 1st Open International Conference on Modeling & Simulation, 2005, págs. 295–305. 2005.

[33] Sargent, R. G. Validation and Verification of Simulation Models. En: Winter Simulation Conference, págs. 104–114. 1992.

[34] Robinson, S. Simulation model verification and validation: increasing the users' confidence. En: Proceedings of the 29th conference on Winter simulation, págs. 53–59. IEEE Computer Society, 1997.

[35] Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., *et al.* Using formal specifications to support testing. *ACM Computing Surveys*, **41** (2), 1–76, 2009.

[36] Utting, M., Legeard, B. Practical Model-Based Testing: A Tools Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[37] Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Monetti, P. R. Tool Support for the Test Template Framework. *Softw. Test., Verif. Reliab.*, 2013. Online Version of Record published before inclusion in an issue – http://onlinelibrary.wiley.com/doi/10.1002/stvr.1477/abstract.

[38] Stocks, P., Carrington, D. A Framework for Specification-Based Testing. *IEEE Trans. Softw. Eng.*, **22**, 777–793, November 1996.

[39] Balci, O. Verification, validation and accreditation of simulation models. En: Proceedings of the 29th conference on Winter simulation, WSC '97, págs. 135–141. Washington, DC, USA: IEEE Computer Society, 1997.

[40] Sargent, R. G. Verification and validation: verification and validation of simulation models. En: Proceedings of the 35th conference on Winter simulation: driving innovation, WSC '03, págs. 37–48. Winter Simulation Conference, 2003.

[41] Sargent, R. G. Verification and validation of simulation models. En: Proceedings of the 37th conference on Winter simulation, WSC '05, págs. 130–143. Winter Simulation Conference, 2005.

[42] Sargent, R. G. Verification and validation of simulation models. En: Proceedings of the 39th conference on Winter simulation, WSC '07, págs. 124–137. Piscataway, NJ, USA: IEEE Press, 2007.

[43] Sargent, R. G. Verification and validation of simulation models. En: Proceedings of the 2010 Winter Simulation conference, WSC '07, págs. 166 –183. 2010.

[44] Napoli, M., Parente, M. Graded CTL Model Checking for Test Generation. En: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, TMS-DEVS '11, págs. 59–66. San Diego, CA, USA: Society for Computer Simulation International, 2011.

[45] Saadawi, H., Wainer, G. Principles of Discrete Event System Specification model verification. *SIMULATION*, **89** (1), 41–67, 2013.

[46] Baier, C., Katoen, J.-P. Principles of model checking. MIT Press, 2008.

[47] Hong, K. J., Kim, T. G. Timed I/O Test Sequences for Discrete Event Model Verification. En: T. Kim (ed.) Artificial Intelligence and Simulation, tomo 3397 de *Lecture Notes in Computer Science*, págs. 275–284. Springer Berlin / Heidelberg, 2005.

[48] da Silva, P. S., de Melo, A. C. V. On-the-fly verification of discrete event simulations by means of simulation purposes. En: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, TMS-DEVS '11, págs. 238–247. San Diego, CA, USA: Society for Computer Simulation International, 2011.

[49] Li, X., Vangheluwe, H., Lei, Y., Song, H., Wang, W. A testing framework for DEVS formalism implementations. En: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, TMS-DEVS '11, págs. 183–188. San Diego, CA, USA: Society for Computer Simulation International, 2011.

[50] Bougé, L., Choquet, N., Fribourg, L., Gaudel, M. C. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, **6**, 343–360, November 1986.

[51] Cristiá, M., Monetti, P. Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing. En: K. Breitman, A. Cavalcanti (eds.) Formal Methods and Software Engineering, tomo 5885 de *Lecture Notes in Computer Science*, págs. 167–185. Springer Berlin Heidelberg, 2009.

[52] Fitting, M. First-order logic and automated theorem proving (2nd ed.). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.

[53] Stocks, P. A. Applying Formal Methods to Software Testing, 1993.

[54] Souza, S. R. S., Maldonado, J. C., Fabbri, S. C. P., Masiero, P. C. Statecharts Specifications: A Family of Coverage Testing Criteria. En: XXVI Conferência Latinoamericana de Informática – CLEI'2000. Tecnologico de Monterrey – México: Springer Berlin / Heidelberg, 2000.

[55] Cristiá, M., Hollmann, D. A., Albertengo, P., Frydman, C. S., Monetti, P. R. A Language for Test Case Refinement in the Test Template Framework. En: ICFEM, págs. 601–616. 2011.

[56] Cristiá, M., Hollmann, D. A., Pomponio, L. Automatic Test Adaptation for Model-Based Testing Methods using Set-Based Specification Languages. *En referato en ACM Transactions on Software Engineering and Methodology*, 2014.

[57] Gomes, C. P., Kautz, H., Sabharwal, A., Selman, B. Satisfiability Solvers. En: Handbook of Knowledge Representation, tomo 3 de *Foundations of Artificial Intelligence*, págs. 89–134. Elsevier, 2008.

[58] Nieuwenhuis, R., Oliveras, A., Tinelli, C. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, **53** (6), 937–977, nov. 2006.

[59] Cristiá, M., Frydman, C. S. Applying SMT Solvers to the Test Template Framework. En: A. K. Petrenko, H. Schlingloff (eds.) MBT, tomo 80 de *EPTCS*, págs. 28–42. 2012.

[60] Dovier, A., Omodeo, E. G., Pontelli, E., Rossi, G. {log}: A Language For Programming In Logic With Finite Sets. *Journal of Logic Programming*, **28**, 28–1, 1996.

[61] Dovier, A., Piazza, C., Pontelli, E., Rossi, G. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, **22** (5), 861–931, sep. 2000.

[62] Cristiá, M., Rossi, G., Frydman, C. {*log*} as a test case generator for the Test Template Framework. En: R. M. Hierons, M. Brevetti, M. Merayo, M. Brevetti (eds.) SEFM, Lecture Notes in Computer Science, págs. 229–243. Springer, 2013. To appear.

[63] Xtext. Website, http://www.eclipse.org/Xtext/, Último Acceso: 13-11-2014.

Publicaciones durante el doctorado

1. Cristiá, M., Hollmann, D.A., Albertengo, P., Frydman, C., Rodríguez Monetti, P. A Language for Test Case Refinement in the Test Template Framework. En *13th International Conference on Formal Engineering Methods*, ICFEM 2011, Durham, UK, 2011.

2. Hollmann, D.A., Cristiá, M., Frydman, C. Adapting Model-Based Testing Techniques to DEVS Models Validation. En *Sympoium on Theory of Modeling and Simulation (TMS/DEVS 2012)*, 2012.

   Premiado como mejor artículo del Symposium on Theory of Modeling & Simulation (TMS/DEVS 2012) y artículo finalista de la Spring Simulation Multi-Conference 2012 en reconocimiento a su calidad, originalidad e importancia para el modelado y la simulación.

3. Hollmann, D.A., Cristiá, M., Frydman, C. A Family of Simulation Criteria to Guide DEVS Models Validation Rigorously, Systematically and Semi-Automatically. En *Simulation Modelling Practice and Theory*, Elsevier, vol. 49, pág. 1-26. 2014.

4. Cristiá, M., Hollmann, D.A., Pomponio, L. Automatic Test Adaptation for Model-Based Testing Methods Using Set-Based Specification Languages. En referato en *Transactions on Software Engineering and Methodology*, ACM. 2014.