DEVS Y PATRONES DE DISEÑO

LAURA POMPONIO

Tesina de Grado

Licenciatura en Ciencias de la Computación

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Febrero 2009

SUPERVISOR
PhD. Maâmar El-Amine Hamri

LSIS - Laboratoire des Sciences de l'Information et des Systèmes
Marsella - Francia

a MIA,
quien es parte fundamental de mi vida y
quien me ha enseñado que la libertad es interna.

RESUMEN

Los sistemas computacionales de hoy día creados por el hombre, pertenecen a la clasificación de lo que se llama *sistemas a eventos discretos*. La característica esencial de estos es que están gobernados por la existencia asíncrona de eventos.

Los avances tecnológicos han hecho que estos sistemas sean cada vez más complejos y mayores, y por tanto, resulte importante contar con recursos que faciliten su análisis, diseño e implementación. Formalismos de especificación, mecanismos de simulación, patrones de diseño y buenas prácticas de la Ingeniería de Software, son algunos de los medios con los que se cuenta para construir estos sistemas.

En particular, DEVS (*Discrete Event Specification System*) es un formalismo general fundado sobre una fuerte base teórica, que permite la especificación de sistemas a eventos discretos. Este formalismo, trae además a la par, la definición de un *simulador abstracto* cuya implementación puede ser relativamente sencilla.

Debido a sus bondades, DEVS es ampliamente utilizado para la modelización y simulación de sistemas. Si bien existen herramientas que acompañan estas tareas, en la mayoría de los casos la implementación de modelos requiere cierta intervención manual por parte del usuario. Más aún, en ocasiones la implementación es realizada en forma completa por quien modela. Esto trae aparejado que modificaciones en los modelos especificados, puedan resultar difíciles cambios en el código.

En este trabajo se sugieren patrones generales para diseñar la implementación de modelos DEVS, de manera tal, que esta cuente con un código fácilmente legible y mantenible. Para esto se utilizarán los popularmente conocidos *patrones de diseño*, los cuales ayudan a crear diseños flexibles y sistemas reutilizables.

AGRADECIMIENTOS

A mis viejos, por contagiarme el amor a los libros y por enseñarme a valorar la educación.

A mis hermanos, Cintia y Santi, por siempre hacerme sentir que tiene sentido el esfuerzo.

A Luciano, por acompañar los tragos dulces, y los no tan dulces, de este transcurrir.

A mi supervisor, M. Hamri por acompañar este trabajo.

A Claudia Frydman, por el apoyo y contención.

A Maxi Cristiá, un especial agradecimiento por su generosidad, por los espacios abiertos, por ser un referente para mí y por encontrar en él además de un gran docente, un amigo.

A Brian, mi buen amigo, por la inmensidad de horas de estudio y por todo lo compartido más allá de lo académico.

A los portugueses, por hacer de Portugal una experiencia maravillosa.

A mis íntimos amigos actores y de la vida, Marián, Sergio y Virginia, que han acompañado este proceso desde un otro lugar, fundamental para mí.

A los docentes que hacen de la educación pública un orgullo.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ACRÓNIMOS

DARPA Defense Advanced Research Projects Agency

DEDS Discrete Event Dynamic Systems (Sistemas Dinámicos a Eventos Discretos)

DES Discrete Event System (Sistemas a Eventos Discretos)

DESS Differential Equation System Specification (Especificación de Sistemas de Ecuaciones Diferenciales)

DEV&DESS Discrete Event & Differential Equation System Specification (Especificación de Sistemas de Ecuaciones Diferenciales y Eventos Discretos)

DEVS Discrete Event System Specification (Especificación de Sistemas a Eventos Discretos)

DoD Department of Defense

DTTS Discrete Time System Specification (Especificación de Sistemas de Tiempo Discreto)

EIC External Input Coupling (Acoplamiento de Entrada Externo)

EOC External Output Coupling (Acoplamiento de Salida Externo)

FSM Finite State Machine

HLA High Level Architecture (Arquitectura de Alto Nivel)

IC Internal Coupling (Acoplamiento Interno)

IS Ingeniería de Software

M&S Modeling & Simulation (Modelización y Simulación)

OO Orientación a Objetos

TGS Teoría General de Sistemas

1 INTRODUCCIÓN

Sistemas de control de tráfico aéreo, redes de computadoras, protocolos de comunicación y sistemas de producción automatizada, son algunos ejemplos de sistemas, cuya dinámica se rige por la ocurrencia de eventos discretos. En consecuencia, estos sistemas son conocidos como *sistemas a eventos discretos* (DES).

Existen diferentes formalismos, como Redes de Petri, Statecharts, Grafcet, etc., que permiten la representación de DES. Cada uno de estos cuenta con particularidades que lo hacen más adecuado para ciertas áreas de aplicación.

Por su parte, DEVS es un formalismo general de especificación de sistemas a eventos discretos, propuesto por B. Zeigler en la década del setenta, con el objetivo de tratar problemas de modelización y simulación (M&S). Este formalismo fue fundado sobre la base de *teoría de sistemas* (TGS)[1]; esto lo hace general y por tanto, todos los sistemas representables en los formalismos antedichos pueden ser representados en este.

Por su universalidad, por su potencial para modelar sistemas complejos y por la posibilidad de implementar simulaciones simples y eficientes, DEVS es empleado en diversas áreas de aplicación y es ampliamente utilizado por la comunidad de M&S.

Entre algunas de sus variantes, DEVS contempla modelos y simulación simple, paralela y/o distribuida. Dada las diferentes alternativas, las implementaciones de modelos DEVS y sus simuladores son diversas. Distintos trabajos existen acerca de algoritmos de simulación y, diseño de simuladores y modelos [19, 14, 21, 20]. No obstante, el uso de buenas prácticas de Ingeniería de Software (IS), aplicadas en el marco de M&S de este formalismo, es un área que aún no ha sido completamente explorada y muchas cuestiones siguen abiertas.

Varias son las herramientas de software que existen para modelar y simular en DEVS. Algunas de ellas requieren gran conocimiento de programación por parte del usuario, mientras que otras, brindan una interfaz más amigable y admiten un usuario menos experto. Sin embargo, en muchas ocasiones los modelos son implementados manualmente por quien modela y luego simulados. A su vez, los modelos implementados, a veces, sufren modificaciones que luego resulta dificultoso plasmar en la implementación. Esto es, un cambio en el modelo puede significar diversas modificaciones en el código y por tanto, resultar esto una tarea pesada.

Para este tipo de problemas, la Ingeniería de Software (IS) cuenta con los llamados *patrones de diseño*. Estos establecen soluciones elegantes y flexibles a problemas recurrentes en el diseño de software. Gamma, Helm, Johnson y Vlissides, llamados GoF (o Gang of Four o banda de cuarto), publicaron en la década del noventa un catálogo de patrones de diseño, a veces referido como el *libro de patrones* [10]. Esta gran contribución facilitó muchas de las tareas en el proceso de desarrollo de software.

1 Llamada también teoría general de sistemas (TGS), propuesta a mediados del siglo XX por el biólogo austríaco Ludwing von Bertalanffy. La TGS es un campo interdisciplinario de la ciencia que estudia la naturaleza de sistemas complejos y los principios comunes que los rigen.

El objetivo de este trabajo es proponer un patrón para el diseño de modelos DEVS, que permita una buena legibilidad del código de implementación y flexibilidad al momento de reflejar los cambios que un modelo pudiera experimentar. Con esta finalidad, aquí se utilizarán los patrones de diseño mencionados antes, como base fundamental para la definición de los utilizados en modelos DEVS.

En la siguiente sección se expondrá una breve reseña sobre el origen de DEVS y sobre el origen de los patrones de diseño. A su vez se describirán extensiones del formalismo, algunas herramientas disponibles, trabajos relativos a la estandarización de este, y trabajos relacionados. En la sección 3 se presentará el formalismo DEVS con las definiciones de modelo atómico y acoplado, junto con el simulador abstracto simple, que el formalismo define. La sección 4 mostrará algunos de los patrones de diseño del *libro de patrones* sobre los cuales se basará este trabajo. En sección 5 se establecerán los patrones de diseño propuestos para modelos DEVS. Por último, la sección 6 contará con las conclusiones de este trabajo.

2

ESTADO DEL ARTE

2.1 EL FORMALISMO DEVS Y SUS EXTENSIONES

Un sistema dinámico es un sistema que presenta cambios de estado en el tiempo. Este se dice discreto si el tiempo es medido en pequeños intervalos, y se dice continuo si el tiempo se mide en forma continua. En el primer caso, los sistemas pueden ser modelados con relaciones recurrentes; en el segundo, con ecuaciones diferenciales.

El concepto *sistema dinámico* implica la formalización matemática, de alguna regla que describa la dependencia de tiempo de un punto en el espacio de estados. La regla de evolución de estos sistemas, está dada por una relación que determina el estado en un tiempo futuro. Estas relaciones son en origen ecuaciones diferenciales o ecuaciones en diferencia. Determinar el estado para todo instante de tiempo futuro requiere iterar sobre la relación, lo que significa integrar o resolver el sistema.

Como es sabido, la representación matemática de sistemas existió, en muchas ocasiones, antes de la versión computarizada de hoy. Tal es el caso de los sistemas de ecuaciones diferenciales, que teniendo estado y tiempo continuos, fueron formulados como la clase de *especificaciones de sistemas de ecuaciones diferenciales* (DESS). También, los sistemas que operan sobre una base de tiempo discreto, como los autómatas celulares, fueron formulados como la clase de *especificaciones de sistemas de tiempo discreto* (DTSS). En ambos casos, la representación matemática ha alcanzado, hoy en día, su versión en un sistema de cómputo.

Por su parte, los sistemas computarizados se rigen por la existencia de eventos asíncronos. Debido a los avances tecnológicos y a la existencia de grandes sistemas complejos de este tipo, una nueva clasificación de sistemas dinámicos ha surgido, esta se llama *sistemas a eventos discretos* (DES). Por su naturaleza, inicialmente estos sistemas no contaron con una representación formal; por el contrario, esta consistió en código de implementación. Con el tiempo, surgieron una variedad de formalismos de representación de DES como Redes de Petri, Statecharts, Grafcet, Grafos de Eventos, etc. Mientras cada uno de estos tiene su área de aplicación, ninguno fue desarrollado intencionalmente como una subclase de formalismo, de *teoría de sistemas* (TGS).

En la década de los setenta Bernard Zeigler [26], basándose en TGS, propuso un marco teórico y una metodología para M&S de sistemas. Es aquí que surge DEVS (*Discrete Event System Specification*), un formalismo de modelización con una sólida semántica, fundado en una base teórica de sistemas.

Este formalismo, a veces caracterizado como universal, permite describir *sistemas dinámicos a eventos discretos* (DEDS). Su universalidad se refiere a que cualquier otro formalismo para DEDS puede ser ajustado[1] a él. En definitiva, esto significa que los formalismos para DES mencionados antes, pueden ser absorbidos por DEVS en cuanto a que todo modelo especificable en los primeros puede ser especificado en este último.

1 El término en inglés que usa Zeigler es *embedded* que significa incrustado o encajado.

DEVS no solo representa una herramienta general de modelado y análisis de sistemas a eventos discretos; sino también, propone un marco de generación de comportamiento de modelos, mediante la definición de un algoritmo de simulación llamado *simulador abstracto*. De esta manera, da la posibilidad de construir, de un modo jerárquico y modular, modelos a eventos discretos simulables.

En la década del noventa, un avance importante fue la combinación de DESS y DEVS en un solo formalismo, llamado *DEV&DESS*, que incluyó a los dos antedichos. Este nuevo concepto permitió el modelado y la simulación de sistemas híbridos cuyo comportamiento dinámico se caracteriza por procesos continuos y discretos [28, 29].

Otras extensiones DEVS surgieron más tarde, dándole al formalismo un mayor plano de acción al momento de tratar sistemas dinámicos. *Parallel DEVS* (o DEVS Paralelos) es una de ellas. En esta extensión, diferentes componentes pueden ser activados y enviar sus salidas a otros componentes al mismo tiempo. Estos conceptos traen aparejado algoritmos de simulación paralela y distribuida [5, 29].

Además aparecieron extensiones como *Dynamic Structure DEVS*, *Symbolic DEVS*, *Real Time DEVS*, *Fuzzy DEVS*, *Cell-DEVS*, *HFSS*, *GDEVS*, etc. *Dynamic Structure DEVS* (o DEVS de Estructura Dinámica) posibilita que un modelo cambie su estructura durante la simulación [29]. *Symbolic DEVS* (o DEVS Simbólicos) representa eventos que ocurren en un tiempo simbólico [29]. Así, se crea a partir de un estado particular, una familia de trayectorias que constituyen todas las trayectorias de estado posibles. Estas familias pueden utilizarse, por ejemplo, para rastrear el origen de un error o evaluar rendimientos de tiempo en la simulación. *Real Time DEVS* (o DEVS de Tiempo Real) permite a un modelo DEVS ser desarrollado en un entorno de simulación y ser ejecutado en tiempo real. *Fuzzy DEVS* (o DEVS Difusos) incorpora la idea de incertidumbre dentro de modelos DEVS [17, 29]. *Cell-DEVS* es utilizado para especificar DES celulares. Esta extensión se basa en combinar Autómatas Celulares y DEVS. *HFSS* (Heterogeneous Flow System Specification) es una formalización alternativa a DEV&DESS para sistemas continuos y discretos. *CFSS* (Continuous Flow Specified System) es su precursor y fue considerado un subformalismo de DEVS. No obstante, aunque HFSS puede estar definido como una extensión de DEVS; en [27] se señala que no ha habido una discusión en cuanto a las condiciones sobre las cuales un modelo está bien definido en el formalismo, ni si la clausura bajo acoplamiento es una propiedad de este. Por último, *GDEVS* (Generalized Discrete Event System) es una generalización de DEVS. En esta, las señales de entrada y salida son una lista de coeficientes polinomiales, que permiten aproximar las trayectorias de entrada y salida de procesos continuos. De este modo, sistemas continuos pueden ser modelados como abstracciones de eventos discretos, de un modo más exacto. En consecuencia, GDEVS permite la especificación de sistemas híbridos.

Como puede observarse, varios son los formalismos basados en DEVS que surgen para cubrir necesidades de análisis, especificación y simulación de distintos tipos de sistemas. Debido a esto, DEVS con sus diversas extensiones es ampliamente utilizado en M&S.

2.2 HERRAMIENTAS DEVS

El formalismo y muchas de sus extensiones cuentan con herramientas para la construcción y simulación de modelos. Generalmente, cada herramienta está orientada a ciertas áreas de aplicación donde el tipo de usuario, modelo y simulación pueden variar.

Un ejemplo son los procesos biológicos. La complejidad de estos sistemas hace que la simulación por computadora sea un buen recurso para estudiarlos. En [9] los autores presentan modelos de simulación sobre biología mitocondrial utilizando el formalismo DEVS y la herramienta CD++. Esta herramienta, desarrollada por G. Wainer y sus estudiantes (Carleton University, Canada; Universidad de Buenos Aires, Argentina), permite la definición de DEVS y Cell-DEVS, admitiendo en ciertas versiones simulación paralela y manejo de tiempo real.

Otro ejemplo es el uso del formalismo y de la herramienta DEVSJAVA, que hacen los autores en [22, 13] para modelar y simular procesos de producción y abastecimiento de productos. Esta herramienta es un entorno de modelización y simulación de DEVS desarrollado en JAVA por H. Sarjoughian y B. Zeigler (Arizona State University, U.S.A.; University of Arizona, U.S.A.).

Por su parte, LSIS_DME [11] también es un entorno de M&S de sistemas DEVS, desarrollado en JAVA por el laboratorio LSIS (Francia). Esta herramienta cuenta con una interfaz gráfica, de construcción de modelos, que se basa en la representación definida en [23]. LSIS_DME es utilizada en [3], trabajo que es parte del proyecto PIOVRA (Polyfunctional Intelligent Operational Virtual Reality Agents). En este proyecto se apunta a desarrollar una nueva generación de CGF (Computer Generated Forces)[2] para ejercicios y planificación de defensa militares.

También existen ADEVS y DEVS/C++, herramientas que fueron desarrolladas en University of Arizona (U.S.A.). La primera es una biblioteca en C++ que soporta la construcción de modelos de sistemas a eventos discretos, usando Parallel DEVS y Dynamic Structure DEVS, y permite su simulación. La segunda es un entorno de simulación de eventos discretos basada en Parallel DEVS e implementada en C++.

PowerDEVS es una herramienta creada en la Universidad Nacional de Rosario (Argentina) que permite la modelización y simulación DEVS de sistemas híbridos.

DEVS/HLA, creado en Arizona State University (U.S.A.), es un entorno de modelización y simulación distribuida basado en el formalismo DEVS y en la arquitectura de alto nivel HLA (definida por el Departamento de Defensa de Estados Unidos) [8].

Otras tantas herramientas como JDEVS, DEVSim++, VLE y demás; existen para modelizar y simular sistemas especificados en DEVS y sus extensiones. La mayoría de las mencionadas aquí, y otras, pueden encontrarse en `http://www.sce.carleton.ca/faculty/wainer/standard/`.

Finalmente, cabe señalar que aunque algunas de las herramientas cuentan con una interfaz de usuario más amigable que otras; generalmente, estas requieren del usuario conocimientos sobre programación para definir los modelos. Más aún, algunas son solo bibliotecas que el usuario puede utilizar; y por tanto, la implementación en código del modelo, requiere gran intervención de este.

2.3 ESTANDARIZACIÓN DE M&S Y DEVS

La comunidad de M&S es muy amplia y existen diversos esfuerzos por facilitar la comunicación entre distintos grupos, compartir recursos y estandarizar los conceptos.

2 Se llama CGF a ciertos componentes de simulación. Esencialmente, es una representación basada en computadora de un participante en un escenario dado. Ejemplo de esto podría ser: un soldado, un avión, un conjunto de naves, etc.

HLA (High Level Architecture) es una arquitectura de propósito general, definida por el Departamento de Defensa (DoD) de Estados Unidos, para sistemas de simulación distribuida. La definición preliminar de esta, fue aceptada como estándar por dicho departamento en 1996.

HLA provee la especificación de una arquitectura técnica estándar, para usar en todas las clases de simulación en el DoD. Los objetivos principales de este estándar son la reutilización y la interoperabilidad en las simulaciones.

En 1997 un proyecto de investigación dispuesto por DARPA (Defense Advanced Research Projects Agency) demostró cómo un entorno DEVS, conforme a HLA, puede significativamente mejorar el rendimiento de prácticas de modelización y simulación distribuida. HLA contribuye con un marco técnico común facilitando la interoperabilidad entre distintos tipos de modelos y simulaciones. DEVS cuenta con fundamentos teóricos que hace que sus implementaciones sean, en principio, independientes de lenguajes de programación y de plataformas de hardware.

Además de las implementaciones DEVS bajo HLA , existen otras bajo CORBA, MPI y demás. Hay una gran variedad de grupos trabajando sobre extensiones del formalismo y desarrollando herramientas de modelización para estas. Aunque cada grupo adhiere a las definiciones de DEVS básico, cuestiones de implementación surgen debido a diferencias en las plataformas subyacentes. Estas diferencias llevan a la dificultad de compartir y reutilizar modelos DEVS.

Durante SCSC '2000 (Summer Computer Simulation Conference 2000) en Vancouver, ciertos investigadores de DEVS decidieron armar un grupo de estudio. El objetivo sería el desarrollo de un estándar para herramientas y extensiones basadas en DEVS, que sería presentado a SISO (Simulation Interoperability Standards Organization). Este grupo internacional, llamado DEVS Standarization Group, intenta desarrollar estándares para una representación procesable del formalismo DEVS. Este comité debate sobre estandarizaciones para contar con comprensión común, intercambio e interoperabilidad de implementaciones DEVS. Más detalles sobre sus actividades pueden encontrarse en `http://www.sce.carleton.ca/faculty/wainer/standard/`.

Por su parte, SISO es una organización internacional dedicada a promover, en beneficio de la comunidad de M&S, la reutilización e interoperabilidad de modelos y simulaciones. SISO, entre otras cosas, desarrolla y mantiene estándares de interoperabilidad de simulación, y organiza conferencias y talleres relativos a este tema. Puede accederse a más información sobre esta organización en `http://www.sisostds.org/`.

Otras organizaciones como SCS (The Society for Modeling and Simulation International) y ACIMS (Arizona Center for Integrative Modeling & Simulation) son importantes respecto a las actividades de M&S.

SCS es una organización cuyo objetivo es facilitar la comunicación entre quienes integran la comunidad de M&S. Para esto organiza encuentros y conferencias, y lleva a cabo distintas publicaciones en forma periódica. El sitio web de SCS es `http://www.scs.org/`.

ACIMS (Arizona Center for Integrative Modeling & Simulation), organización codirigida por B. Zeigler (University of Arizona, U.S.A.) y H. Sarjoughian (Arizona State University), promueve conceptos, herramientas y metodologías de M&S, con el objetivo de utilizar sistemas de cómputo para resolver problemas emergentes, que requieren soluciones multidisciplinarias. Una de las ideas principales es reunir, de un modo integral, una variedad de investigadores alrededor de la modelización y simulación, con la finalidad de integrar ciencias computacionales, M&S y distintas disciplinas. El sitio web de ACIMS se encuentra en `http://www.acims.arizona.edu/`.

Como se mencionó antes, DEVS ha sido aplicado en numerosos contextos de industria e investigación, y ha sido implementado sobre varias plataformas con diferentes lenguajes de programación. Uno de los recientes trabajos [12] de B. Zeigler, consiste en presentar descripciones estandarizadas para modelos DEVS, simuladores DEVS y su interacción. El objetivo de esto es estandarizar las implementaciones y facilitar desarrollos futuros. Además de posibilitar que modelos implementados en diferentes lenguajes sean simulados en forma conjunta, otra meta es permitir que modelos no-DEVS participen de la simulación. La documentación de este trabajo, hasta el momento, está publicada en carácter de borrador.

2.4 PATRONES DE DISEÑO

El arquitecto Christopher Alexander, en la década del sesenta, observó ciertas características y cuestiones recurrentes, en el campo de la arquitectura, y las capturó en descripciones e instrucciones a las cuales denominó patrones. El término *patrón*, refirió a la similitud replicada en un diseño y en particular a aquella que permitiera hacer ambientes variables y posibles de personalizar. Por ejemplo, *ventanas en dos lados de un ambiente* es un patrón, sin embargo no determina ni el tamaño de las ventanas, ni la altura desde el suelo, ni otros detalles.

El trabajo de Alexander sobre patrones fue muy influyente en la comunidad de desarrolladores de software. A partir de las ideas de este, los diseñadores de software han encontrado analogías entre los patrones del arquitecto y patrones de software [18, 6].

Los trabajos de W. Cunningham, K. Bech y E. Gamma fueron los primeros en demarcar un camino hacia los patrones de diseño de software. Particularmente Gamma fue quien observó que las estructuras o patrones de diseño que se repetían, eran importantes. El gran desafío era cómo capturar estas ideas y comunicarlas a otros.

Luego de encuentros entre algunos participantes de la comunidad de software e intercambio de ideas acerca de patrones; en la década del noventa, Gamma E., Helm R., Johnson R. y Vlissides J. lograron documentar, en forma de **patrones de diseño**, las experiencias sobre diseño de software orientado a objetos. De este modo surgión el libro popularmente conocido, Patrones de Diseño [10].

Los patrones definidos en el mencionado libro, resuelven problemas concretos en diseño orientado a objetos, y favorecen la construcción de sistemas reutilizables con diseños flexibles y elegantes. El sentido de los patrones es ayudar a los diseñadores a definir diseños, a partir de la idea central de otros, que la experiencia previa ha demostrado que son buenos.

Luego de los patrones de Gamma, en 1993, K. Beck y G. Booch impulsaron una reunión en Colorado donde cierto grupo se encontró para discutir acerca de fundamentos para patrones de software. Este grupo trató las ideas de Alexander y las experiencias propias, con el objetivo de forjar una unión entre objetos y patrones. Aquí surgió Hillside Group, un grupo educativo sin fines de lucro, que patrocina diversas conferencias y mantiene la página `http://hillside.net/patterns/` de patrones, en la cual puede encontrarse amplia información sobre patrones de software en general.

2.5 TRABAJOS RELACIONADOS

No es nueva, la idea de definir patrones de diseño asociados a un formalismo, de modo tal que, aquellos sistemas especificados en este cuenten con un diseño que mantenga la estructura de dicha especificación. Preservar esta estructura provee una mejor mantenibilidad ante modificaciones futuras que debieran ser hechas manualmente.

Existen varios patrones definidos para máquinas de estado finito (FSM) y Statecharts. Muchos de estos son presentados en en la antología descripta en [1]. Algunos de estos patrones están inspirados, principalmente, en el patrón State del libro de patrones.

En [24], [25] y [2], los autores analizan diversos patrones, como así también, las ventajas y desventajas con las que se cuenta, al utilizar State para diseñar implementaciones de FSM y Statecharts.

Si bien para estos formalismo existen herramientas de generación de código, y en tal caso los patrones perderían sentido; algunos ingenieros de software prefieren, en ciertos casos, llevar a cabo sus propias implementaciones.

En el caso de DEVS, no solo las herramientas disponibles requieren, en general, cierto grado de implementación manual, sino también, existe quienes prefieren implementar directamente sus modelos. Con este espíritu, es que se presentará en este trabajo un posible patrón de diseño para modelos DEVS.

3

FORMALISMO DEVS

La teoría de sistemas (TGS), sobre la cual se basa DEVS, distingue entre *estructura* de un sistema (constitución interna) y *comportamiento* (su manifestación externa). El *comportamiento* externo, visto como una caja negra, es la relación entre la historia temporal de entradas y la historia temporal de salidas. La *estructura* interna incluye el estado, el mecanismo de transición (determinando cómo una entrada transforma el estado actual en el estado sucesor) y las salidas del sistema (dadas ante cierto estado y cierta transición).

Conocer la *estructura* de un sistema permite deducir, analizar o simular su *comportamiento*. Pero inferir su *estructura* de su *comportamiento*, generalmente no es posible dado que puede manifestarse igual *comportamiento* en sistemas que tienen diferentes *estructuras* internas. Descubrir una representación válida de un *comportamiento* observado es uno de los intereses claves de las iniciativas de M&S.

Conceptos importantes en TGS son *descomposición*: cómo un sistema puede ser dividido en sistemas de componentes, y *composición*: cómo sistemas de componentes pueden ser acoplados para construir un sistema mayor.

La TGS es cerrada bajo composición; esto es, la *estructura* y el *comportamiento* de una *composición* de sistemas, puede ser expresada en términos de la teoría de sistemas original. Esto permite la *construcción jerárquica* de sistemas más grandes garantizando que estos tendrán *estructura* y *comportamiento* bien definidos.

A su vez, los sistemas *modulares* cuentan con puertos de entrada y salida, a través de los cuales ocurren las interacciones con el entorno. Estos sistemas pueden ser *acoplados* entre sí conectando puertos de entrada a puertos de salida. De este modo, el acoplamiento permite contar con una estructura jerárquica, en la cual sistemas son conectados, conformando un sistema mayor.

Por otro lado, el concepto más general de *modelo* de simulación, es que este consiste en un conjunto de instrucciones, reglas, ecuaciones o restricciones para generar comportamiento de entrada-salida. En este sentido, un modelo se describe con el conjunto de entradas que acepta y con los mecanismos, de transición de estado y generación de salidas.

Además, un modelo necesita algún agente capaz de generar el comportamiento que especifica. Este agente será llamado *simulador*. Así, un simulador es cualquier sistema de cálculo (tal como un procesador, la mente humana o, de un modo más abstracto, un algoritmo) capaz de ejecutar un modelo para generar su comportamiento.

DEVS es un formalismo fundado en TGS, en el cual los conceptos descriptos más arriba lo constituyen, como así también, la definición de un *simulador abstracto*.

Pudo advertirse en la sección 2, que el formalismo cuenta con varias extensiones. Sin embargo, aquí se presentarán las definiciones [29] y conceptos básicos, de modelo y simulador DEVS, que bastarán en este trabajo para conocer los elementos fundamentales del formalismo.

3.1 MODELOS DEVS ATÓMICOS

3.1.1 *DEVS Clásico*

Un modelo DEVS atómico clásico es una estructura

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

donde

X es el conjunto de valores de entrada

Y es el conjunto de valores de salida

S es el conjunto de estados

$\delta_{int} : S \to S$ es la función de *transición interna*

$\delta_{ext} : Q \times X \to S$ es la función de *transición externa*, siendo

$Q = \{(s, e) \mid s \in S, 0 \leqslant e \leqslant ta(s)\}$ el conjunto de *estado total*

e el *tiempo transcurrido* desde la última transición

$\lambda : S \to Y$ es la función de salida

$ta : S \to \Re^{+}_{0,\infty}$ es la función de *avance de tiempo*

Se presenta la Figura 1 para describir el significado de los elementos de la estructura definida.

Un sistema que está en un estado $s \in S$, permanecerá en el mismo por un tiempo $ta(s)$ salvo que, habiendo transcurrido un tiempo e menor o igual que ta(s) en ese estado, ocurra un evento de entrada con valor $x \in X$. En tal caso, el sistema experimentará una *transición externa* cambiando al estado $s' = \delta_{ext}(s, e, x)$. En la figura esto está ilustrado por medio de la flecha continua que va de s a s'.

En cambio, si el tiempo transcurrido e es igual a $ta(s)$ sin que se hayan producido eventos externos, un evento interno ocurrirá dando lugar a una *transición interna*. Esto producirá un evento de salida con valor $y = \lambda(s)$ y un cambio a un nuevo estado $s'' = \delta_{int}(s)$. Esto es graficado a través de la fecha discontinua que va de s a s''.

En ambos casos, el sistema permanecerá en el nuevo estado, por el tiempo que ta lo determine o hasta que otra vez ocurra un evento externo.

estados transitorios y estados pasivos

Obsérvese en la definición de DEVS presentada, que la función de *avance de tiempo* admite asociar un tiempo 0 o ∞ a un estado $s \in S$. En el primer caso, el sistema sólo podrá permanecer en s, 0 instantes de tiempo. Por tanto, cuando alcance s, inmediatamente sucederá una transición interna ocurriendo un evento de salida y un cambio de estado. Este tipo de estado es llamado *estado transitorio*. De otro modo,

cuando el sistema alcance s cuyo tiempo asociado sea ∞, no existirán transiciones internas y permanecerá por siempre en este estado, salvo que, un evento externo ocurra y entonces, una transición externa produzca el cambio. Este tipo de estado es llamado *estado pasivo*.

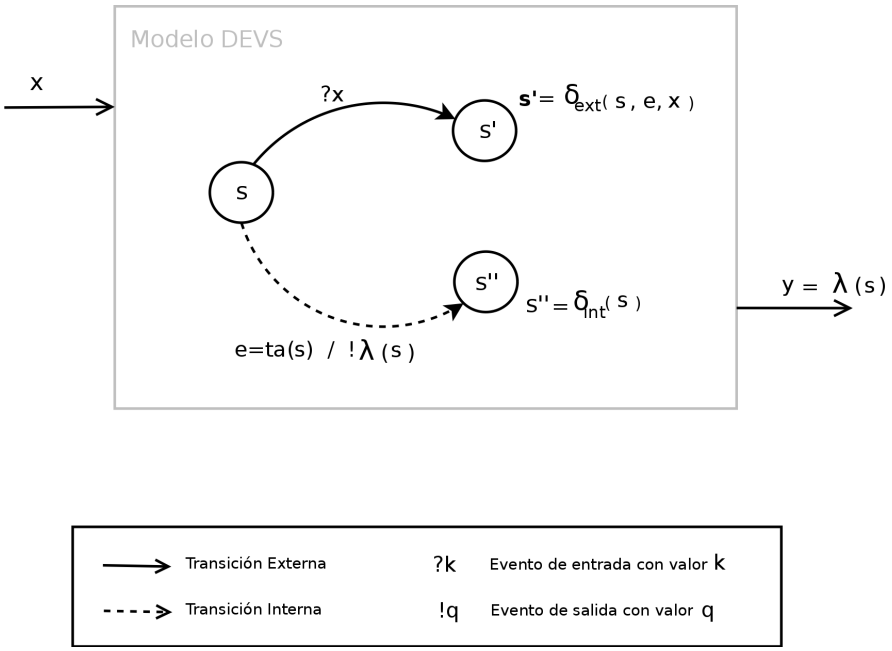


Figura 1: Cambio de Estados DEVS

La Figura 2 contribuye a ilustrar mejor el funcionamiento de un modelo DEVS y a obtener una mayor comprensión semántica de la estructura definida.

Como puede observarse, la *trayectoria de entradas* en Figura 2 es una serie de eventos externos, con valores x_1 y x_3, ocurriendo en los tiempos t_1 y t_3. Entre estos instantes puede haber otros, tal como t_2, que son tiempos de eventos internos.

La *trayectoria de estados* es una serie estados (tales como s_0, s_1, s_2, s_3), en forma escalonada, cuyos cambios se dan cuando ocurren eventos externos o internos. La figura muestra además, los intervalos de tiempo $ta(s_i)$ durante los cuales el sistema permanecería en el estado s_i, si no ocurriesen eventos de entrada.

La *trayectoria de tiempo transcurrido* es un dibujo, con forma de sierra, que ilustra el flujo de tiempo transcurrido en un reloj, el cual volverá a poner su valor en 0 cada vez que un evento tenga lugar. En resumen, grafica el tiempo transcurrido en cada estado. Cuando ocurre un evento (interno o externo) el sistema cambia a un nuevo estado y el contador de tiempo se reinicia.

Por último la *trayectoria de salidas* ilustra los eventos de salida que son producidos por la función λ cuando tiene lugar una transición interna.

En la figura, el sistema está inicialmente en el estado $s_0 \in S$. En un momento t_1, habiendo transcurrido un período e menor que $ta(s_0)$, sucede un evento de entrada con valor x_1. Ocurre entonces, una transición externa y el sistema pasa a estar en el nuevo estado $s_1 = \delta_{ext}(s_0, e, x)$. Por su parte, el tiempo transcurrido (e) vuelve a valer 0.

Como no existen entradas en el sistema durante el intervalo $[t_1, t_2]$, el tiempo $ta(s_1)$ de permanencia en s_1 expira. Esto produce un evento de salida con valor

$y_2 = \lambda(s_1)$ y luego, un cambio de estado a $s_2 = \delta_{int}(s_1)$, como resultado de una transición interna. Nuevamente el valor de e vuelve a 0.

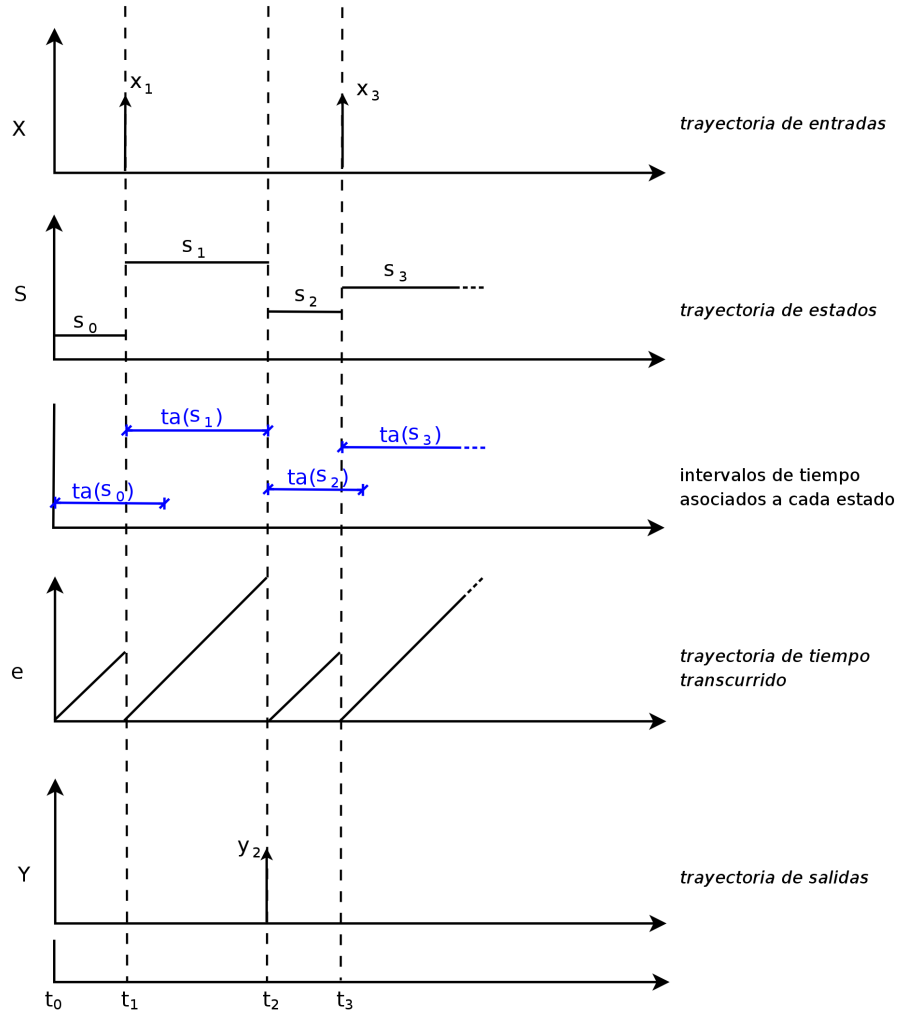


Figura 2: Trayectorias DEVS

Durante la permanencia en el estado s_2, otra entrada x_3 ocurre. El sistema entonces, experimenta una vez más una transición externa y cambia al estado $s_3 = \delta_{ext}(s_2, e, x_3)$.

conflicto de simultaneidad

Si bien la Figura 2 no propone situaciones de colisión, sería natural preguntarse qué sucede si en el mismo instante que debe ocurrir una transición interna, se produce un evento de entrada. Claramente hay dos opciones; se ejecuta la transición interna y luego la transición externa o bien, se ignora la transición interna y se ejecuta solo la transición externa. La estructura de DEVS atómico clásico, definida más arriba, presenta esta indeterminación.

Para resolver este tipo de conflicto, fue definida una extensión del formalismo llamada Parallel DEVS (o DEVS Paralelo), la cual se presenta en la sección 3.1.2. Si bien este trabajo utilizara DEVS clásicos para desarrollar los conceptos, parece importante presentar dicha extensión para dar una visión un poco más completa del formalismo y su poder expresivo.

Por su parte, los DEVS clásicos manejan la simultaneidad de eventos mediante una función Select que forma parte de la especificación de modelos acoplados definida en la sección 3.2.1.

Se presenta a continuación *Ejemplo* 3.1.1, tomado de [16], el cual define un simple modelo que ilustra cómo especificar comportamiento utilizando DEVS clásicos.

Ejemplo 3.1.1 FIFO

Sea una cola a la cual llegan trabajos para ser almacenados y señales *ready*, que indican descargar (o enviar fuera) el primer trabajo encolado. Se asume que la cola demora un tiempo nulo en enviar el trabajo y además, para simplificar el modelo, se considera que la capacidad de almacenamiento con la que cuenta es infinita.

La especificación DEVS del modelo es como sigue.

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

Considerando $J = \{job_1, job2, ..., job_n\}$ el conjunto de trabajos que pueden ingresar a la cola y ϕ como una secuencia vacía, se define

$$X = J \cup \{\text{“ready“}\},$$

$$Y = J,$$

$$S = J^+ \cup \{\phi\} \times \{0, \infty\},$$

$$\delta_{int}((q \,.\, job, \sigma)) = (q, \infty) \quad \text{con } q \in J^+ \cup \{\phi\}$$

$$\delta_{ext}((q, \sigma), e, x) = \begin{cases} (x \,.\, q, \infty) & \text{si } x \in J \\ (q, 0) & \text{en otro caso} \end{cases}$$

$$\lambda((q \,.\, job, \sigma)) = job$$

$$ta((q, \sigma)) = \sigma$$

La definición del modelo establece a X como el conjunto de trabajos y señales que pueden ingresar a FIFO, e Y el conjunto de trabajos de salida.

Por su parte, S es el conjunto de estados posibles, cada uno de los cuales está conformado por una secuencia de trabajos y por su tiempo de vida. Esto último, está claramente especificado por ta.

La función de transición externa δ_{ext} expresa que, si FIFO recibe como evento de entrada un trabajo ($x \in J$), lo encola y pasa a un *estado pasivo* de espera infinita; en otro caso (una señal"ready"), no se modifica la secuencia y pasa a un *estado transitorio* (de tiempo 0).

Obsérvese que cuando el modelo haga la transición a un estado pasivo, simplemente se mantendrá en este hasta que un nuevo trabajo llegue; o bien, hasta recibir una orden “ready“ de descarga de trabajos.

En el caso de pasar a un estado transitorio, el tiempo de permanencia en éste será 0 y por tanto, el sistema llevará a cabo una transición interna a través de la función δ_{int}.

Llegada esta instancia, como lo define λ, un trabajo de salida será descargado y el sistema cambiará al estado $\delta_{int}(q.job, \sigma) = (q, \infty)$, en el que esperará pasivamente por algún evento de entrada.

3.1.2 *DEVS Paralelo*

Un DEVS atómico paralelo es una extensión de un DEVS atómico clásico que permite un *bag*[1] (o multiconjunto) como entrada o salida. Esto es, múltiples ocurrencias de valores de X ingresando al sistema en el mismo instante, o bien la función λ retornando múltiples ocurrencias de valores de Y.

Otro elemento fundamental incorporado en esta extensión del formalismo es la *función de transición confluente* δ_{con}. Esta función provee al modelador la posibilidad de definir, sin ambigüedad, el comportamiento del sistema en el caso en que un evento de entrada, ocurra en el mismo instante en que el sistema deba realizar una transición interna.

Se define entonces un DEVS atómico paralelo como una estructura

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

donde

X es el conjunto de eventos de entrada

Y es el conjunto de eventos de salida

S es el conjunto de estados

$\delta_{int} : S \to S$ es la función de *transición interna*

$\delta_{ext} : Q \times X^b \to S$ es la función de *transición externa*, siendo

X^b un conjunto de *bags* sobre elementos de X

$Q = \{(s, e) | s \in S, 0 \leqslant e \leqslant ta(s)\}$ el conjunto de *estado total*

e el *tiempo transcurrido* desde la última transición

1 o *multiset*, es similar a un conjunto excepto por que múltiples ocurrencias de un mismo elemento pueden pertenecer al *bag*. Por ejemplo; un *bag* sobre $\{a, b\}$ podría ser $\{a, a, b, a\}$.

$\delta_{con} : S \times X^b \to S$ es la *función de confluencia*

$\lambda : S \to Y^b$ es la función de salida

$ta : S \to \Re^{+}_{0,\infty}$ es la función de *avance de tiempo*

La definición por defecto de la *función confluente* es $\delta_{con}(s, x) = \delta_{ext}(\delta_{int}(s), 0, x)$.

Esta definición indica que ante la ocurrencia simultánea de un evento externo y la expiración del tiempo en un cierto estado, el sistema llevará a cabo la transición interna, colocará en 0 el reloj del tiempo transcurrido en el nuevo estado y realizará la transición externa.

Como la función δ_{con} decide el siguiente estado en caso de colisión de eventos externos e internos, otra posibilidad sería invertir el orden de los eventos definiéndola como $\delta_{con}(s, x) = \delta_{int}(\delta_{ext}(s, ta(s), x))$ o bien, determinar que el siguiente estado sea un estado arbitrario s'.

Todas estas son decisiones que quien modela deberá llevar a cabo.

3.1.3 *DEVS con Puertos*

Un modelo DEVS se relaciona con su entorno recibiendo valores de entrada y, emitiendo valores de salida cuando ocurre una transición interna. El lugar por donde ingresan valores, o aquel por donde salen, no es descripto en la sección 3.1.1 debido a que es único. Esto es, un modelo DEVS básico cuenta con un único puerto de entrada y un único puerto de salida que, por su condición de unicidad, no son especificados. Del mismo modo, la sección 3.1.2 nada menciona acerca de los puntos de entrada o salida del sistema.

No obstante, el concepto de puerto es incorporado a la definición de modelo DEVS para permitir que este interactúe con el exterior a través de diferentes puntos de entrada y de salida. Un DEVS puede contar con varios puertos desde donde recibirá valores y otros tantos, por los que enviará al entorno los resultados obtenidos por la función λ.

Un modelo DEVS atómico clásico o paralelo con puertos, mantiene las estructuras definidas en las secciones 3.1.1 y 3.1.2 respectivamente, junto con el significado de sus elementos; salvo para X e Y, cuya definición es la siguiente:

$X = \{(p, v) \mid p \in \mathrm{InPorts}, v \in X_p\}$ es un conjunto de puertos y valores de entrada, siendo

$\mathrm{InPorts}$ el conjunto de *puertos de entrada*

X_p el conjunto de valores asociados al puerto p

$Y = \{(p, v) \mid p \in \text{OutPorts}, v \in Y_p\}$ es un conjunto de puertos y valores de salida, donde

OutPorts es el conjunto de *puertos de salida*

Y_p es el conjunto de valores asociados al puerto p

Los puertos permiten modelar sistemas en los cuales el comportamiento no solo depende del estado y los valores de entrada, si no también, del lugar por dónde ingresan estos.

En el siguiente *Ejemplo* 3.1.2 se muestra una versión con puertos del modelo FIFO presentado en *Ejemplo* 3.1.1. Ciertas modificaciones se incorporaron sobre el comportamiento antes descripto.

Ejemplo 3.1.2 FIFO con Puertos

Sea FIFO una cola que recibe trabajos por el puerto ip_1 y señales de control por el puerto ip_2. Además, cada vez que se le indique, envía por el puerto op el primer trabajo de la cola.(Ver Figura 3). A los efectos de simplificar el ejemplo, se asume que la capacidad de almacenamiento es infinita.

A medida que FIFO va recibiendo trabajos por el puerto ip_1, los va incorporando en la cola. Cuando una señal de control "ready" ingresa por el puerto ip_2, envía el primer trabajo hacia el exterior por el puerto op. Se asume que el tiempo que demora en enviar el trabajo hacia afuera es nulo. Si una señal de control "clear" ingresa por el puerto ip_2, la cola simplemente se vacía y se pierden los trabajos.

Figura 3: FIFO con Puertos

La especificación del comportamiento de este FIFO es la que sigue.

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

Considerando $J = \{job_1, job2, ..., job_n\}$ el conjunto de trabajos que pueden ingresar a la cola y ϕ como una cola vacía, se define

$$X = \{(p, v) \mid p \in \text{InPorts}, v \in X_p\}$$

$$\text{InPorts} = \{ip_1, ip_2\}, \quad X_{ip_1} = \{job_1, job_2, ..., job_n\} \quad X_{ip_2} = \{''ready'', ''clear''\}$$

$$Y = \{(p, v) \mid p \in \text{OutPorts}, v \in Y_p\}$$

$$\text{OutPorts} = \{op\}, \quad Y_{op} = \{job_1, job_2, ..., job_n\}$$

$$S = J^+ \cup \{\phi\} \times \{0, \infty\},$$

$$\delta_{int}((q \,.\, job, \sigma)) = (q, \infty) \quad \text{con } q \in J^+ \cup \{\phi\}$$

$$\delta_{ext}((q, \sigma), e, (p, x)) = \begin{cases} (x \,.\, q, \infty) & \text{si } p == ip_1 \\ (q, 0) & \text{si } p == ip_2 \wedge x == '' \text{ ready}'' \\ (\phi, \infty) & \text{si } p == ip_2 \wedge x == '' \text{ clear}'' \end{cases}$$

$$\lambda((q \,.\, job, \sigma)) = (op, job)$$

$$ta((q, \sigma)) = \sigma$$

El comportamiento es muy similar al del ejemplo anterior, salvo por que se agregó una señal de control que permite eliminar todos los trabajos en la cola.

En este tipo de especificación, las transiciones externas dependen del estado, del valor del evento de entrada y de por dónde ingresa este. Por otro lado, obsérvese que además de especificar los puntos por donde entran y salen los eventos, se pueden detallar los valores válidos que acepta un puerto.

En este ejemplo particular existe solo un puerto de salida; no obstante, podría contarse con más de uno y el evento resultante de la transición, ser direccionado a diferentes puertos según la necesidad.

Este es un simple ejemplo ilustrativo pero la idea de puertos cobrará real importancia en la sección 3.2 cuando se introduzca el concepto de acoplamiento DEVS.

3.2 MODELOS DEVS ACOPLADOS

La modelización de sistemas complejos puede demandar menos esfuerzo, si se considera a estos como una colección de componentes simples que interactúan entre sí. Por lo general, es más difícil intentar especificar directamente el comportamiento total de un sistema complejo, que identificar componentes simples, modelarlos por separado y especificar la forma en que se relacionan. El formalismo DEVS provee los medios para construir modelos por acoplamiento (o composición) de modelos más pequeños.

El acoplamiento puede ser *no-modular* o *modular*. En el primer caso, las transiciones de estado de un componente pueden directamente cambiar el estado y replanificar los eventos de otro. Un DEVS multicomponente o *multiDEVS* se trata justamente de una especificación *no-modular* de interacción entre componentes.

El resultado de una transición, en un *multiDEVS*, depende del estado en el que estén los componentes que lo influencian. A su vez, cuando dicha transición se lleva a cabo, el estado de cierto conjunto de componentes se ve afectado por esta.

El *acoplamiento modular*, en cambio, propone conectar sistemas autónomos a través de sus interfaces de entrada y salida. De este modo, ningún componente puede acceder ni influir en forma directa sobre el estado o la temporización de otro componente. Todas las interacciones deben realizarse por intercambio de mensajes. Más específicamente, los eventos generados por un componente en sus puertos de salida, son transmitidos a través del acoplamiento a puertos de entrada de los componentes influenciados. Esto provoca, en estos últimos, eventos de entrada y transiciones externas.

acoplamiento modular y no-modular

La especificación no-modular de sistemas es importante porque muchas formas tradicionales de modelización son no-modulares. Sin embargo, la modularización representa una ventaja para la simulación distribuida y para la reutilización de modelos durante el proceso de desarrollo de una nueva simulación. En particular, el estándar HLA [8], determina como objetivos importantes la interoperabilidad y la reutilización de componentes. Contar con componentes modulares bien definidos, que acuerden sobre el significado de los elementos que los constituyen, contribuye a una mayor reutilización de estos.

Dados los objetivos de este trabajo y las ventajas que presenta la modularidad, solo se describirá el acoplamiento modular. Para el lector interesado, en [29] se describe en detalle el acoplamiento no-modular y cómo el formalismo permite el traslado de una especificación no-modular a una modular.

3.2.1 *Acoplamiento Modular de DEVS Clásicos*

Un modelo DEVS acoplado clásico es una estructura

$$N =< X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, Select >$$

donde

- X es el conjunto de eventos de entrada
- Y es el conjunto de eventos de salida
- D es el conjunto de referencias a los componentes del modelo acoplado, tal que

  $\forall d \in D$, M_d es un modelo DEVS Clásico
- I_d es el conjunto de *componentes influyentes* sobre d tal que,

  $\forall d \in D \cup \{N\},\ I_d \subseteq D \cup \{N\} - \{d\}$ y

  $\forall i \in I_d,\ Z_{i,d}$ es la *función de traducción* de salidas desde i hacia d con

$Z_{i,d} : X \to X_d$, si $i = N$

$Z_{i,d} : Y_i \to Y$, si $d = N$

$Z_{i,d} : Y_i \to X_d$, si $d \neq N \wedge i \neq N$

Select es una *función de desempate* que arbitra la ocurrencia de eventos simultáneos

$Select : 2^D \to D$

Cada modelo M_d recibe eventos de entrada que provienen de otros modelos que también constituyen a N. Por esta causa, estos modelos se donominan *influyentes* sobre d y conforman el conjunto I_d asociado a M_d. Obsérvese que dentro de este conjunto puede estar incluido el mismo modelo acoplado N pero, M_d **no puede ser influyente de sí mismo**.

N es influyente de M_d si los eventos de entrada en el primero resultan eventos de entrada en el segundo. Por su parte, el conjunto influyente I_N está constituido por los componentes *internamente* influyentes en N. Estos son, aquellos componentes cuyas salidas resultan en eventos de salida del modelo acoplado.

En muchos casos, los eventos que puede recibir un componente no son del mismo tipo que aquellos que generan sus influyentes. Para salvar estas diferencias el formalismo cuenta con las *funciones de traducción*.

Como se mencionó en la sección 3.1.1, puede haber conflictos de simultaneidad. Las transiciones internas en un modelo, generan eventos de salida que a su vez representan eventos de entrada en otros componentes. El conflicto aparece entonces, cuando un componente y alguno de sus influyentes deben llevar a cabo una transición interna. El funcionamiento del sistema cambiará sustancialmente, dependiendo de qué componente ejecute antes esta transición. Por tal motivo, es necesario especificar cuál es el comportamiento adecuado del sistema. La función Select viene entonces, a cubrir esta necesidad. Esta función especifica cuál será el componente que tendrá prioridad ante la existencia de más de un componente que deba realizar una transición interna.

A continuación, se describe un ejemplo de estructura de modelos acoplados, la cual constituye un nuevo modelo DEVS.

Ejemplo 3.2.1 Acoplamiento DEVS

Sea un DEVS acoplado clásico

$N =< X_N, Y_N, D, \{M_A, M_B\}, \{I_A, I_B, I_N\}, Z_{i,d}, Select >$ donde

$D = \{A, B\}, \quad I_A = \{N\}, \quad I_B = \{A\}, \quad I_N = \{A, B\},$

A es un modelo DEVS clásico cuyos conjuntos de valores de entrada y salida son X_A y Y_A respectivamente,

B es un modelo DEVS clásico cuyos conjuntos de valores de entrada y salida son X_B y Y_B respectivamente,

las funciones de traducción son

$Z_{N,A} : X_N \to X_A$,

$Z_{A,B} : Y_A \to X_B$,

$Z_{A,N} : Y_A \to Y_N$,

$Z_{B,N} : Y_B \to Y_N$;

y la función de desempate es $Select(\{A, B\}) = A$.

La Figura 4 grafica la estructura acoplada del modelo.

Las función *Select* especifica que, si en el mismo instante una transición interna ocurre en A y en B, A tendrá prioridad.

La funciones de traducción indican que los valores que ingresen a N serán traducidos y enviados a A; los valores de salida de A se transformarán en valores de entrada de B y en valores de salida de N; y por último, los valores de salida de B resultarán en valores de salida de N.

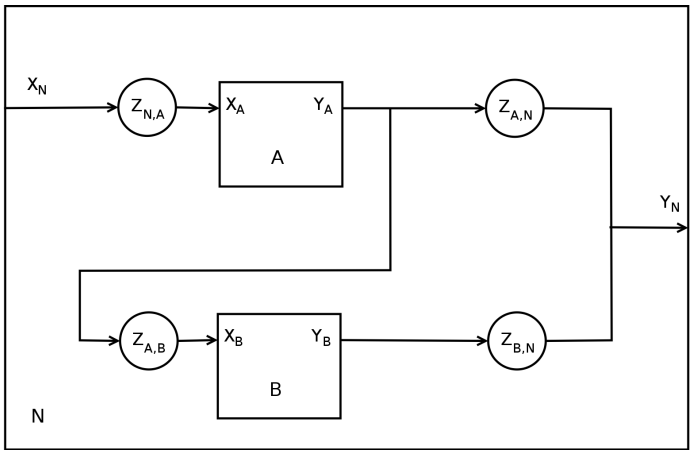


Figura 4: DEVS Acoplado

En la siguiente apartado, se describe muy brevemente cómo sería el acoplamiento para DEVS paralelos.

3.2.2 *Acoplamiento Modular de DEVS Paralelos*

La estructura de un modelo acoplado en DEVS Paralelo difiere de aquella definida en la sección 3.2.1, en que la función *Select* no es parte de la definición. En consecuencia, un DEVS Paralelo acoplado es una estructura

$$N =< X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\} >$$

donde

X es el conjunto de eventos de entrada

Y es el conjunto de eventos de salida

D es el conjunto de referencias a los componentes del modelo acoplado, tal que

$\forall d \in D$, M_d es un modelo DEVS Paralelo

I_d es el conjunto de *componentes influyentes* sobre d tal que,

$\forall d \in D \cup \{N\},\ I_d \subseteq D \cup \{N\} - \{d\}$ y

$\forall i \in I_d,\ Z_{i,d}$ es la *función de traducción* de salidas desde i hacia d con

$Z_{i,d} : X \to X_d$, si $i = N$

$Z_{i,d} : Y_i \to Y$, si $d = N$

$Z_{i,d} : Y_i \to X_d$, si $d \neq N \wedge i \neq N$

Claramente, en este caso la función de desempate no es necesaria. Aunque dos modelos atómicos conectados efectuasen sus transiciones internas al mismo tiempo, aquel que recibiera el evento externo (provocado por el otro) contaría con la función de confluencia que le indicaría cómo actuar ante dicha situación.

3.2.3 *Acoplamiento Modular con Puertos*

Definiendo los conjuntos de entrada y salida de un modelo, como conjuntos de elementos constituidos por múltiples variables; es posible, derivar en la formalización de sistemas acoplados, mediante la conexión directa entre puertos de entrada y salida. Este tipo de acoplamiento es popularmente utilizado en las prácticas de modelización debido a que facilita la tarea de modelado.

En las secciones 3.2.1 y 3.2.2 el acoplamiento se definió por medio de los conjuntos influyentes I_d y por las funciones de traducción $Z_{i,d}$ establecidas entre componentes. No obstante, resulta más natural y claro que los dispositivos que se modelan, envíen sus salidas a través de puertos a otros modelos.

El acoplamiento mediante puertos se define fundamentalmente por medio de tres conceptos:

- el *acoplamiento de entrada externo* EIC (External Input Coupling)

  que une los puertos de entrada externos, del modelo acoplado N, a puertos de entrada de componentes,

- el *acoplamiento de salida externo* EOC (External Output Coupling)

que acopla los puertos de salida de componentes, a puertos de salida externos del modelo acoplado N y, por último,

- el *acoplamiento interno* IC (Internal Coupling)

  que vincula puertos de salida de ciertos componentes a puertos de entrada de otros.

Basándose en estos conceptos, un DEVS acoplado modularmente con puertos es una estructura

$N =< X_N, Y_N, D, \{M_d\}, EIC, EOC, IC, Select >$ si se trata de un **DEVS Clásico** y,

$N =< X_N, Y_N, D, \{M_d\}, EIC, EOC, IC >$ si se trata de un **DEVS Paralelo**, donde

- $X_N = \{(p, v) \mid p \in InPorts_N, v \in X_{p_N}\}$ es un conjunto de puertos y valores de entrada, siendo
  - $InPorts_N$ el conjunto de *puertos de entrada*
  - X_{p_N} el conjunto de valores asociados al puerto p
- $Y_N = \{(p, v) \mid p \in OutPorts_N, v \in Y_{p_N}\}$ es un conjunto de puertos y valores de salida, donde
  - $OutPorts_N$ es el conjunto de *puertos de salida*
  - Y_{p_N} el conjunto de valores asociados al puerto p
- D el conjunto de nombres de componentes
- $\forall d \in D,\ M_d$ es un modelo DEVS, clásico o paralelo según corresponda;

  en particular, si es atómico será de la forma
  - $M_d =< X_d, Y_d, S_d, \delta_{int_d}, \delta_{ext_d}, \lambda_d, ta_d >$ si se usa **DEVS Clásico** o,
  - $M_d =< X_d, Y_d, S_d, \delta_{int_d}, \delta_{ext_d}, \delta_{con}, \lambda_d, ta_d >$ si se usa **DEVS Paralelo**,

    donde
    - $X_d = \{(p, v) \mid p \in InPorts_d, v \in X_{p_d}\}$ y
    - $Y_d = \{(p, v) \mid p \in OutPorts_d, v \in Y_{p_d}\}$

y el acoplamiento entre los modelos está definido por:

- $EIC \subseteq \{((N, ip_N),\ (d, ip_d)) \mid ip_N \in InPorts_N,\ d \in D,\ ip_d \in InPorts_d\}$
- $EOC \subseteq \{((d, op_d),\ (N, op_N)) \mid op_N \in OutPorts_N,\ d \in D,\ op_d \in OutPorts_d\}$
- $IC \subseteq \{((a, op_a),\ (b, op_b)) \mid a \neq b,\ op_a \in OutPorts_a,\ a, b \in D,\ ip_b \in InPorts_b\}$

Si el acoplamiento es entre **DEVS Clásicos** la función Select mantiene su significado como se definió en la sección 3.2.1.

Si el acoplamiento es entre **DEVS Paralelos** los valores asociados a un puerto son multivariables que permiten el uso de *bags*.

En términos generales, se dice que un modelo acoplado N experimenta una transición interna, si alguno de sus componentes cambia de estado internamente.

A partir del *Ejemplo* 3.2.1 se presenta un modelo acoplado con puertos similar, donde los componentes internos son modelos atómicos.

Ejemplo 3.2.2 Acoplamiento Modular con Puertos

Sea $N =< X_N, Y_N, D, \{A, B\}, EIC, EOC, IC, Select >$ donde

$X_N = \{ip_N\} \times X_{ip_N} \qquad Y_N = \{op_N\} \times Y_{op_N}$

$D = \{A, B\},$

$A =< X_A, Y_A, S_A, \delta_{int_A}, \delta_{ext_A}, \lambda_A, ta_A >,$

$X_A = \{ip_A\} \times X_{ip_A}, \qquad Y_A = \{op_A\} \times Y_{op_A},$

$B =< X_B, Y_B, S_B, \delta_{int_B}, \delta_{ext_B}, \lambda_B, ta_B >,$

$X_B = \{ip_B\} \times X_{ip_B}, \qquad Y_B = \{op_B\} \times Y_{op_B},$

$EIC = ((N, ip_N), (A, ip_A)),$

$EOC = ((A, op_A), (N, op_N)), ((B, op_B), (N, op_N)),$

$IC = ((A, op_A), (B, ip_B)),$

y la función de desempate es $Select(\{A, B\}) = A$.

La Figura 5 ilustra la estructura de este ejemplo.

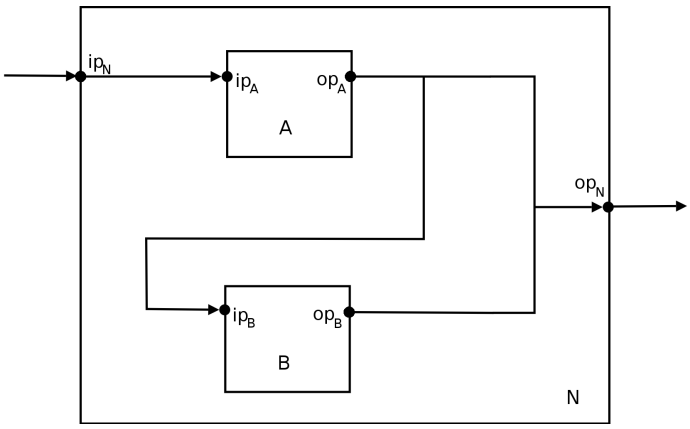


Figura 5: DEVS Acoplado con Puertos

Cada vez que un evento de entrada ingresa a N a través del puerto ip_N, el *acoplamiento de entrada externa* EIC indica que dicho evento será un valor de entrada, en el modelo A, que ingresará por el puerto ip_A.

Cuando el modelo atómico A lleve a cabo una transición interna, λ_A determinará el valor del evento de salida que ocurrirá en el puerto op_A. Debido al *acoplamiento de salida externo* EOC y al *acoplamiento interno* IC, dicho valor saldrá del modelo acoplado N, por el puerto op_N; e ingresará al modelo atómico B, por el puerto ip_B.

3.2.4 *Clausura Bajo Acoplamiento*

La *teoría de sistemas* es cerrada bajo composición en cuanto a que, la estructura y el comportamiento de una composición de sistemas, puede ser expresada en términos de la teoría de sistemas original. La capacidad de seguir componiendo sistemas más y más grandes, a partir de componentes construidos previamente, lleva a lo que se denomina una *construcción jerárquica*. La clausura bajo composición asegura que dicha composición define un sistema *resultante*, con estructura y comportamiento bien definidos. En este sentido, es que el formalismo DEVS es *cerrado bajo acoplamiento* y cuenta con *acoplamiento jerárquico* para la construcción de sistemas complejos.

La ***clausura bajo acoplamiento*** es una de las propiedades fundamentales en DEVS. Esta propiedad ratifica que cualquier acoplamiento de modelos DEVS, tiene asociado un modelo DEVS atómico equivalente. En [29] se demuestra de manera constructiva, esta propiedad de clausura.

La posibilidad de acoplar modelos en forma jerárquica permite la reutilización de los mismos. Esto es; un modelo (acoplado o no) construido como parte de un modelo más general, puede ser reutilizado dentro de otro modelo acoplado, sin la necesidad de realizar modificaciones.

En la bibliografía recién mencionada, se demuestra que las especificaciones de acoplamiento de DEVS Clásicos y Paralelos, presentadas en las secciones 3.2.1 y 3.2.2, cumplen la propiedad de clausura. También se demuestra, que la especificación de acoplamiento con puertos presentada en la sección 3.2.3, es un caso particular de las antedichas, y por tanto, también cumple la propiedad de clausura.

3.3 SIMULADOR ABSTRACTO DEVS

En esta sección, solo se presentará el *simulador abstracto* definido en [29], para simulación de modelos DEVS clásicos. Los autores se refieren a *simuladores abstractos*, en el sentido de que éstos determinan lo que debe ocurrir en la ejecución de modelos atómicos, y acoplados con estructura jerárquica; pero estos simuladores no especifican cómo hacerlo.

Claramente, esta tarea puede realizarse de muchas formas y puede ser optimizada dependiendo de los requerimientos y de los recursos con los que se cuente. Muy amplia es la bibliografía disponible sobre los diferentes criterios y formas de simulación de DEVS [19, 4, 14, 15]. Debido a que los objetivos de este trabajo se centran en los modelos, más que en los simuladores, es que se hará una descripción relativamente breve sobre estos, para presentar la idea conceptual de los mismos.

La simulación de eventos discretos, y en particular la simulación de modelos DEVS, consiste en la ejecución secuencial de una lista de eventos planificados. La

ejecución de cada evento provoca transiciones de estado y nuevos eventos que deberán formar parte de la planificación actualizada.

El *simulador abstracto* consiste en distintas piezas de simulación asociadas a diferentes tipos de especificaciones. Una modelo DEVS atómico tiene asociado lo que se nombrará `devs-simulator` (o *simulador*). Un modelo DEVS modularmente acoplado tiene asociado lo que se denominará `devs-coordinator` (o *coordinador*).

Cada modelo jerárquico, puede vincularse a un *simulador abstracto* jerárquico constituido por *simuladores*, *coordinadores* y un *coordinador raíz* llamado `devs-root-coordinator`.

El *simulador abstracto* mantiene la jerarquía que establece el acoplamiento del formalismo. La Figura 6 muestra la relación entre modelo y simulador asociado.

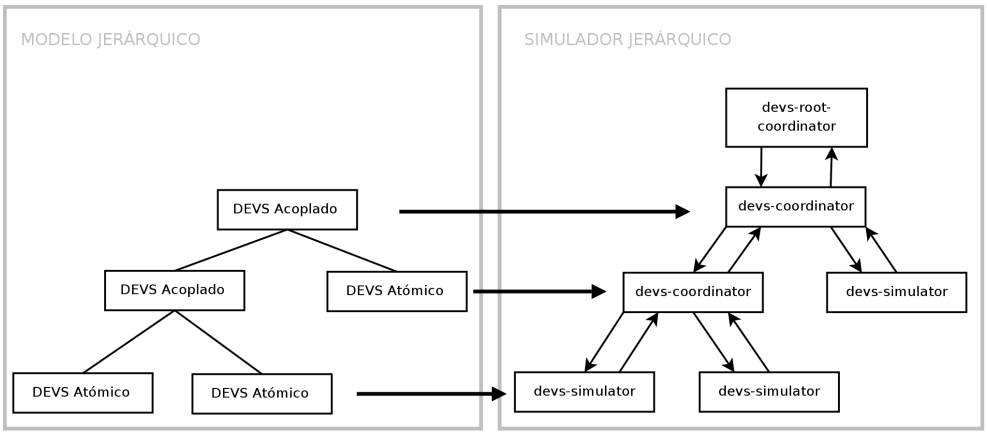


Figura 6: Simulador Jerárquico asociado a Modelo Jerárquico

Para posibilitar la comunicación entre los elementos de la jerarquía, un protocolo general de intercambio de mensajes es definido. Estos mensajes son ilustrados en Figura 7.

Los mensajes que un padre (*coordinador*) puede enviar a sus hijos son de tres tipos:

- ✉ `(i,t)`, llamado `i-message`, es un mensaje de inicialización en el instante `t` que envía un padre a todos sus hijos.
- ✉ `(x, t)`, llamado `x-message`, es un mensaje de entrada que envía cada padre a sus subordinados causando así, eventos externos.
- ✉ `(*, t)`, llamado `*-message`, es un mensaje de transición de estado interno, enviado por el coordinador a los hijos cuyo cambio es inminente. El próximo evento interno que esté planificado, es llevado a cabo a través del envío de estos mensajes.

A su vez, un hijo (*simulador* o *coordinador*) envía a su padre un solo tipo de mensaje:

- ✉ `(y, t)`, llamado `y-message`, es un mensaje indicando un evento de salida, enviado desde cada hijo a su padre.

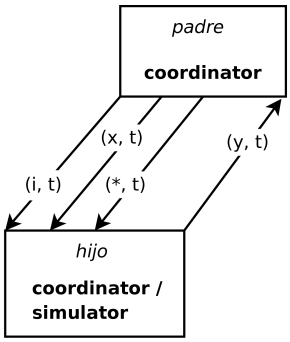


Figura 7: Protocolo de Simulación

3.3.1 *Simulador*

Cada `devs-simulator` cuenta fundamentalmente con dos variables `tl` y `tn`. La primera es el instante de tiempo en que ocurrió el último evento. La segunda es el momento en que deberá ocurrir el próximo evento que debería provocar una transición interna.

En consecuencia, siendo $\mathtt{ta}(\mathtt{s})$ la función de avance de tiempo, se tiene que la próxima transición interna ocurrirá en el tiempo

$$\mathtt{tn} = \mathtt{tl} + \mathtt{ta}(\mathtt{s}).$$

Además, si `t` es el tiempo de simulación, cada *simulador* puede calcular

$$\mathtt{e} = \mathtt{t} - \mathtt{tl} \qquad \text{y}$$

$$\sigma = \mathtt{tn} - \mathtt{t} = \mathtt{tl} + \mathtt{ta}(\mathtt{s}) - \mathtt{t}$$
$$\sigma = \mathtt{ta}(\mathtt{s}) - \mathtt{e}$$

donde `e` es el tiempo transcurrido en el estado actual y σ es el tiempo restante para el próximo evento.

A continuación se presenta el algoritmo para *simuladores* del *simulador abstracto*.

```
devs-simulator
   variables:
       parent         //coordinador padre
       tl             // tiempo del último evento
       tn             //tiempo del próximo evento
       DEVS           // modelo asociado con estado total (s, e)
       y              // valor de salida actual asociado al modelo
   when receive i-message (i, t) at time t
       tl = t - e
       tn = tl + ta(s)
   when receive *-message (*, t) at time t
       if t != tn then
          error: bad synchronization
       y = λ(s)
       send y-message (y, t) to parent coordinator
       s = δint(s)
       tl = t
       tn = tl + ta(s)
   when receive x-message (x, t) at time t with input value x
       if not (tl ⩽ t ⩽ tn) then
          error: bad synchronization
       e = t - tl
```

```
        s = δext(s, e, x)
        tl = t
        tn = tl + ta(s)
end Devs-Simulator
```

Cuando `devs-simulator` recibe un mensaje `i-message` con valor `(i,t)`, calcula `tl` y `tn`. El momento `tn`, en que debe tener lugar la próxima transición interna, es informado al padre u obtenido por este[2].

Un mensaje `*-message` con valor `(*,t)` indica que debe llevarse a cabo una transición interna. Cuando un *simulador* recibe este mensaje, controla que el tiempo de simulación `t` sea el tiempo `tn` planificado de su transición interna. Si esto no se cumple, se detecta un error. En caso contrario, calcula $y = \lambda(s)$, envía al `devs-coordinator` padre un mensaje `y-message` con el valor de salida, calcula el nuevo estado $s = \delta_{int}(s)$, calcula `tl` y computa `tn` para que su padre lo utilice.

Cuando un mensaje `x-message` es recibido por `devs-simulator`, significa que un evento externo ha ocurrido. Se controla entonces, que el tiempo de simulación `t` en que este evento aparece, se encuentre dentro del período determinado por el tiempo del último evento `tl` y el tiempo `tn` próximo en el que debería efectuarse una transición interna. Luego, se calcula el valor $e = t - tl$, que será necesario para calcular el nuevo estado resultante de $\delta_{ext} = (s, e, x)$, se computa `tl` y después `tn`, para que el padre acceda de alguna manera a este último valor.

En esencia, un *simulador* recibe órdenes de su *coordinador* padre de llevar a cabo transiciones internas y externas. A su vez, le envía los valores de salida y calcula el tiempo `tn` de la próxima transición para que su padre lo utilice.

3.3.2 *Coordinador*

Cada componente de un modelo acoplado tiene asociado su propio simulador (o coordinador), que será el responsable por su simulación. Un *coordinador* de un modelo acoplado, es responsable por la sincronización de los componentes que contiene, y por manejar los eventos externos que lleguen.

Para controlar la sincronización de sus subordinados, `devs-coordinator` mantiene una lista de eventos que se actualizará a medida que transcurra el proceso de simulación. Esta lista está constituida por una secuencia de pares, donde los elementos del par son el identificador `d` de un componente hijo y el tiempo tn_d, asociado a este, de la próxima transición interna. El orden de esta lista está dado por `tn` (en forma ascendente) y la función `Select` que establece la prioridad entre los componentes.

El primer componente en la lista ordenada de eventos determina el próximo evento interno del *coordinador*. Recordar que se dice que un modelo acoplado experimenta una transición interna, cuando alguno de sus componentes efectúa una de estas. Con lo cual, el tiempo `tn` de la próxima transición interna de `devs-coordinator` es

$$tn = \min\{tn_d \mid d \in D\},$$

valor que será, a su vez, utilizado por su padre para realizar la sincronización en el nivel siguiente superior de la jerarquía.

De un modo análogo, el tiempo del último evento en `devs-coordinator` es

$$tl = \max\{tl_d \mid d \in D\},$$

2 Esta acción no aparece representada en el algoritmo. De alguna manera el padre conocerá el valor de `tn`.

El algoritmo de simulación para *coordinadores* es el siguiente.

```
devs-coordinator
   variables:
       DEVN = (X, Y, D, Md, Id, Zi,d, Select) //modelo acoplado asociado
       parent                  // coordinador padre
       tl                      // tiempo del último evento
       tn                      // tiempo del próximo evento
       event-list              // lista de elementos (d, tnd) ordenada por tnd and Select
       d*                      // hijo inminente seleccionado
   when receive i-message (i, t) at time t
       foreach d in D do
          send i-message (i, t) to child d
       sort event-list according to tnd and Select
       tl = max tld | d ∈ D
       tn = min tnd | d ∈ D
   when receive *-message (*, t) at time t
       if t != tn then
          error: bad synchronization
       d* = first (event-list)
       send *-message (*, t) to d*
       sort event-list according to tnd and Select
       tl = t
       tn = min tnd | d ∈ D
   when receive x-message (x, t) at time t with external input x
       if not (tl ⩽ t ⩽ tn) then
          error: bad synchronization
       //consulta el acoplamiento externo para obtener el hijo influenciado por la entrada
       receivers = r | r ∈ D, N ∈ Ir , ZN,r(x) ≠ Φ
       foreach r in receivers
          send x-messages (xr, t) with input value xr= ZN,r(x) to r
       sort event-list according to tnd and Select
       tl = t
       tn = min  tnd | d ∈ D
   when receive y-message (yd*, t) with output yd* from d*
       //controla el acoplamiento externo para ver si hay un evento de salida externo
       if d* ∈ IN & Zd*,N(yd*) ≠ Φ then
          send y-message (yN, t) with value yN = Zd*,N(yd*) to parent
       //revisa el acoplamiento interno para obtener el hijo influenciado por la salida yd* de d*
       receivers = r | r ∈ D, d* ∈ Ir , Zd*,r(yd*) ≠ Φ
       foreach r in receivers
          send x-messages (xr, t) with input value xr = Zd*,r(yd*) to r
   end Devs-coordinator
```

Cuando `devs-coordinator` recibe un mensaje de inicialización `i-message`, lo retransmite a todos sus hijos. Sus hijos una vez iniciados, de algún modo, le harán saber cuál es el próximo tiempo $\texttt{tn}_\texttt{d}$ de cada uno. Con esta información y de acuerdo a la función `Select`, el *coordinador* reordena la lista de eventos. Luego calcula sus propios tiempos `tl` y `tn`, como se describe más arriba; y así, su padre puede acceder al tiempo `tn` de la próxima transición interna [3].

Un `*-message` debiera llegar, en el momento de realizar una transición interna. Es decir, este mensaje debería ser recibido si el tiempo de simulación `t`, es el tiempo `tn` planificado para el próximo evento interno. En este caso, `devs-coordinator` controla que el tiempo de simulación coincida con su tiempo `tn`. Entonces, reenvía el mensaje

3 Al igual que para `devs-simulator`, no se especifica en el algoritmo de qué manera el padre accede al valor de `tn`.

de transición interna al hijo inminente d*; es decir, a aquel que estuviese primero en la lista ordenada list-event. El *coordinador* sabrá cuál es el tiempo del próximo evento de su hijo d* y entonces, replanificará su lista de eventos. Después, asignará los nuevos valores a sus variables tl y tn, y como antes, su padre deberá conocer el nuevo valor de tn.

Un mensaje x-message, es un evento externo (x,t) que sucede en el tiempo t, con un valor de entrada x. Cuando un mensaje de este tipo llega, el *coordinador* controla que el tiempo de simulación esté en el período determinado por [tl, tn]. Después, consulta el acoplamiento de entrada externo para saber cuál de sus hijos debe ser afectado por este evento. Pensando en la especificación de modelos, estos son los componentes que se ven influenciados por las entradas de N, donde N es el modelo acoplado asociado a devs-coordinator. Así, el *coordinador* envía a los componentes influenciados, un mensaje (x_d, t) de transición externa. El tiempo de simulación t no cambia pero el valor de x será recalculado por las funciones de traducción, resultando en x_d. Con los nuevos tiempos tn_d de planificación de sus hijos, el *coordinador* reordena la lista de eventos, define el tiempo del último evento como tl=t y calcula tn para que su padre pueda conocer este dato.

Por último, un devs-coordinator recibirá mensajes y-message de sus hijos, cuando estos hayan llevado a cabo una transición interna. Si un mensaje (y_{d^*}, t) de este tipo es recibido, el *coordinador* revisa el acoplamiento de salida externo y el acoplamiento interno entre sus subordinados. En el primer caso, si el componente pertenece a su conjunto influyente, recalcula el valor de y_{d^*} con las funciones de traducción y envía el nuevo valor y_N a su padre. Si además, detecta en el acoplamiento interno que el componente d^* es influyente de otros de sus subordinados, recalcula el valor de y_{d^*} con las funciones de traducción, convirtiéndolo en un valor de entrada x_r para los componentes receptores r. Entonces, envía mensajes (x_r, t) a los receptores correspondientes.

3.3.3 *Coordinador Raíz*

El *simulador raíz* implementa el bucle de simulación global. Además, devs-root-coordinator cuenta con solo un hijo que es el devs-simulator o devs-coordinator asociado al modelo total. Su algoritmo es el siguiente.

```
devs-root-coordinator
   variables:
       t                        // tiempo de simulación actual
       child                    //devs-simulator o devs-coordinator subordinado directo
   t = t0
   send initialization message (i, t) to subordinate
   t = tn of its subordinate
   loop
       send (*, t) message to child
       t = tn of its child
   until end of simulation
end devs-root-coordinator
```

El *simulador raíz* envía un mensaje de inicialización a su subordinado. El tiempo tn del próximo evento de este último es asignado como el tiempo de simulación actual t. Luego, se inician los ciclos de simulación hasta que ocurra una condición de parada. En cada ciclo, devs-root-coordinator envía a su subordinado un evento

de transición interna `(*,t)`; entonces, toma el valor `tn` de su hijo como el tiempo actual de simulación y vuelve a iniciar un ciclo enviando otro mensaje `*-message`.

Notar, que el *coordinador raíz* solo envía mensajes para inicialización y para ejecución de transiciones internas, no recibe ni envía eventos externos. Esto indica que el modelo total asociado a su hijo, no contempla puertos o entradas desde el exterior. Por tanto, a los fines de la simulación, si un sistema total debiese recibir eventos desde fuera, estos deberían ser generados a partir de la simulación de un modelo DEVS que participe de la estructura definida.

Como comentario final y en términos generales, es conveniente destacar que aunque el *simulador abstracto* DEVS ha sido presentado aquí, este trabajo se centra en los modelos y no en los simuladores. Es importante tener presente la diferencia entre modelo y simulador debido a que los patrones de diseño serán sugeridos solo para modelos y no para simuladores.

4

PATRONES DE DISEÑO

Los patrones de diseño describen soluciones simples y elegantes, a problemas específicos del diseño de software orientado a objetos. Estas soluciones han sido desarrolladas, y han ido evolucionando a lo largo del tiempo, con el objetivo de contar con mayor reutilización y flexibilidad en el software.

Los patrones de diseño, descriptos en el libro de patrones [10], son en definitiva la recopilación de dichas soluciones. Así, las experiencias previas en el diseño de software pueden ser capitalizadas a través de un catálogo en el que se han registrado ciertas decisiones de diseño.

En esta sección se describirán los patrones de diseño State y Composite, que pertenecen a la recopilación mencionada. Sobre estos se basarán los patrones propuestos para DEVS.

En el libro de patrones, los autores utilizan una notación propia basada en OMT. En este trabajo se usarán algunas de las convenciones de notación que estos utilizan, y se agregarán otras. Dichas convenciones y notación se describen en el siguiente apartado.

4.1 CONVENCIONES Y NOTACIÓN

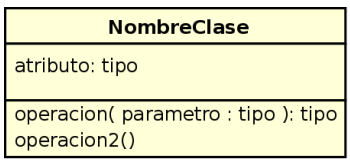
Un **objeto** encapsula tanto datos como procedimientos que operan sobre estos. Estos procedimientos se conocen como **métodos** u **operaciones**.

Cada operación declarada por un objeto, especifica el nombre de la operación, los objetos que toma como parámetro y el valor de retorno de la operación. Esto es lo que se conoce como **signatura** de la operación.

Al conjunto de todas las signaturas definidas por las operaciones de un objeto, se le denomina **interfaz** del objeto. Dicha interfaz caracteriza al conjunto completo de peticiones que se pueden enviar a este último.

Es importante destacar, que aquí el término **interfaz** no se refiere por ejemplo, a aquel utilizado en el contexto UML, en el cual una interfaz es similar a una clase abstracta aunque no puede contar con métodos implementados ni con atributos. Aquí, el término interfaz alude a las operaciones que el objeto deja ver desde el exterior.

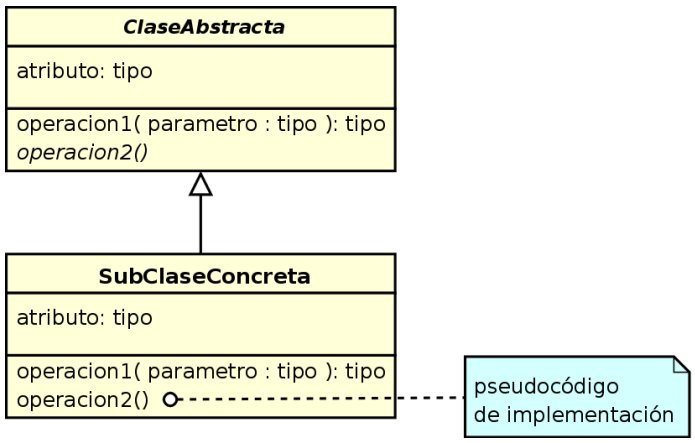
Por otro lado, una clase será mostrada como un rectángulo con el nombre de la clase en **negrita**. Los atributos de la clase se describen debajo del nombre y las operaciones debajo de estos:



Como en el libro de patrones, los tipos de retorno y de las variables son opcionales ya que no se supondrá un lenguaje estáticamente tipado.

Una **clase abstracta**, es aquella cuyo propósito principal es definir una interfaz común para sus **subclases**. Este tipo de clase delegará parte o toda su implementación, en las operaciones definidas, en las clases que hereden de ella. Las operaciones que una clase abstracta declara, pero no implementa, se denominan **operaciones abstractas**.

La **herencia** entre clases se grafica con una línea vertical y un triángulo. El nombre de una clase abstracta, junto con la signatura de las operaciones que no implementa, se mostrarán en letra *itálica*:

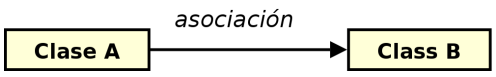


Las clases que no son abstractas, se llaman **clases concretas**; su nombre y sus operaciones serán mostradas en letra normal.

En la figura anterior, aunque la subclase es concreta, podría ser abstracta. De un modo similar, la clase padre y la subclase podrían ser ambas clases concretas y por tanto, ser escritas con letra normal.

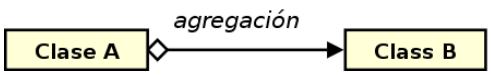
Por otra parte, como el diagrama anterior muestra, podrá presentarse pseudocódigo de implementación, en una caja de texto con una esquina doblada, unida a la operación por una línea discontinua.

Además, en este trabajo se utilizarán las relaciones de ***asociación*** y ***agregación*** en el sentido que se presentan en el libro de patrones. Un flecha continua entre dos clases identificará una relación de ***asociación***:



Esta relación indica que un objeto simplemente *conoce a otro*. En ocasiones, a la asociación también se la conoce como relación de "uso". Los objetos relacionados de esta manera, pueden pedirse operaciones entre sí pero no son responsables unos de otros.

Una flecha continua con un rombo en su base, identifica una relación de ***agregación***:



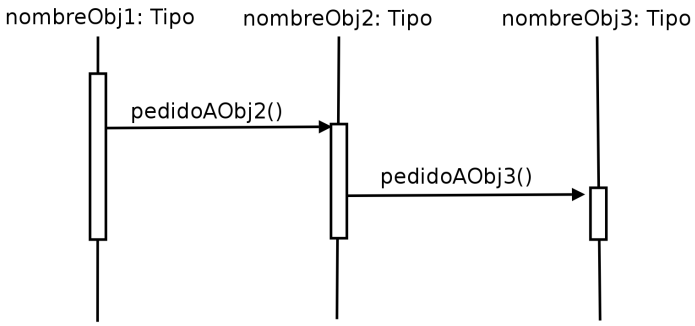
Esta relación indica que un objeto *tiene a otro* o que un objeto es *parte de otro*. Esto implica que un objeto es responsable de otro y que el objeto agregado y su propietario tienen el mismo tiempo de vida.

Estas relaciones pueden contar además, con números o el símbolo '*' (que significa 0 o más) en alguno de los extremos, indicando la multiplicidad de cada clase en la relación. La ausencia de números indica "1".

En ocasiones estas dos relaciones pueden parecer confusas debido a que a veces, en ciertos lenguajes de programación, se implementan de la misma manera. No obstante, la asociación y la agregación están determinadas por su intención más que por mecanismos explícitos del lenguaje.

También, se utilizará notación para indicar a qué puede accederse desde el fuera de un objeto. Los métodos y los atributos podrán ser precedidos por un signo '+', indicando que pueden ser accedidos desde el exterior; o bien, por un signo '-', indicando que son privados del objeto.

Por último, se utilizará un diagrama para mostrar la interacción entre los objetos y el orden en que se ejecutan las peticiones:



Aquí, el tiempo fluye de arriba hacia abajo. La línea vertical indica el tiempo de vida de un determinado objeto. El nombre del objeto, estará en la parte superior de dicha línea, e irá seguido de ":" y del tipo o clase a la que pertenezca el objeto. El rectángulo vertical sobre la línea indica que el objeto está activo; es decir, que está procesando un pedido. La operación de un objeto, puede enviar pedidos a otros objetos; esto se indica con una flecha horizontal que apunte al objeto receptor. El nombre de la solicitud va encima de dicha flecha.

4.2 PATRÓN STATE

El patrón State (Estado), también llamado Objects for States (Objetos como Estados), permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

La motivación radica en que un objeto que recibe peticiones de otro, pueda comportarse de modo diferente según el estado en que se encuentre, siendo esto transparente para el objeto cliente.

La idea clave de este patrón es introducir una clase abstracta ***State*** que unifique los estados posibles del objeto, como muestra la Figura 8. Esta clase declara una interfaz común para todas las subclases **ConcreteState** heredadas de la primera, que representan estados concretos operacionales. Estas subclases implementan el comportamiento específico de cada estado.

La clase **Context** refiere al objeto que debe modificar su comportamiento, según el estado en el que se encuentre. Esta clase mantiene un objeto ***State*** que representa su estado actual (`currentState` en la figura).

Context delega todas las peticiones, dependientes del estado, en el estado actual `currentState`. Un cambio de estado en **Context**, implica un cambio del objeto ***State*** que mantiene.

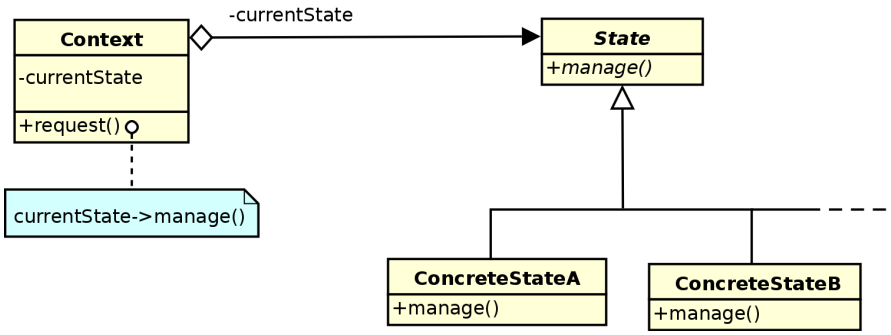


Figura 8: Patrón State

Cuando un cliente hace un pedido `request()` a **Context**, este último delega la petición en `currentState` mediante una llamada a `manage()` de este objeto. Este método tendrá diferentes implementaciones en las distintas subclases **ConcreteState**, lo que permitirá contar con comportamiento diverso, dependiente del estado.

PARTICIPANTES

- **Context**
  - define la interfaz de interés para los clientes.
  - mantiene una instancia de una subclase **ConcreteState** que define el estado actual.
- ***State***
  - define una interfaz para encapsular el comportamiento asociado con un determinado estado de **Context**.
- **ConcreteState**
  - es una subclase de ***State*** que implementa el comportamiento asociado a un estado de **Context**.

Dado que **Context** delega en el estado las peticiones, podría pasarse a sí mismo como parámetro para que el objeto ***State*** pueda acceder al contexto en caso de ser necesario. Otra cosa a tener en cuenta es que cualquiera de las subclases **ConcreteState** o la clase **Context** pueden decidir cuál es el siguiente estado y bajo qué circunstancias.

Si bien el patrón no establece quién define las transiciones entre estados; las notas de implementación mencionan que si los criterios para cambiar de estado son fijos entonces, pueden implementarse en el contexto **Context**; no obstante, generalmente es más flexible y conveniente que las subclases de ***State*** especifiquen su estado sucesor y cuándo llevar a cabo la transición. Si este fuera el caso, debería agregarse una interfaz a **Context** que permita a los estados asignar explícitamente el estado actual.

Descentralizar de este modo los criterios de transición facilita modificarlos o extenderlos agregando nuevos estados. Pero una desventaja de la descentralización es que una subclase de ***State*** conoce al menos a otra y esto introduce dependencia de implementación entre estas.

Este patrón es útil, no solo cuando el comportamiento dependiente de un estado debe cambiar en tiempo de ejecución; sino también, cuando las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado. El patrón propone poner cada rama condicional en una subclase de ***State*** aparte. Esto permite tratar el estado como un objeto que puede variar independientemente de otros.

4.3 PATRÓN COMPOSITE

El propósito de este patrón es componer objetos en estructura de árbol para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme, a los objetos individuales y a los compuestos, describiendo cómo utilizar una composición recurrente.

La Figura 9 grafica la estructura de este patrón.

La idea fundamental es la existencia de una clase abstracta ***Component*** que represente tanto a una hoja o elemento básico, como a su contenedor. De esta manera, los objetos primitivos pueden componerse en objetos más complejos, que a su vez pueden ser compuesto, y así de manera recurrente.

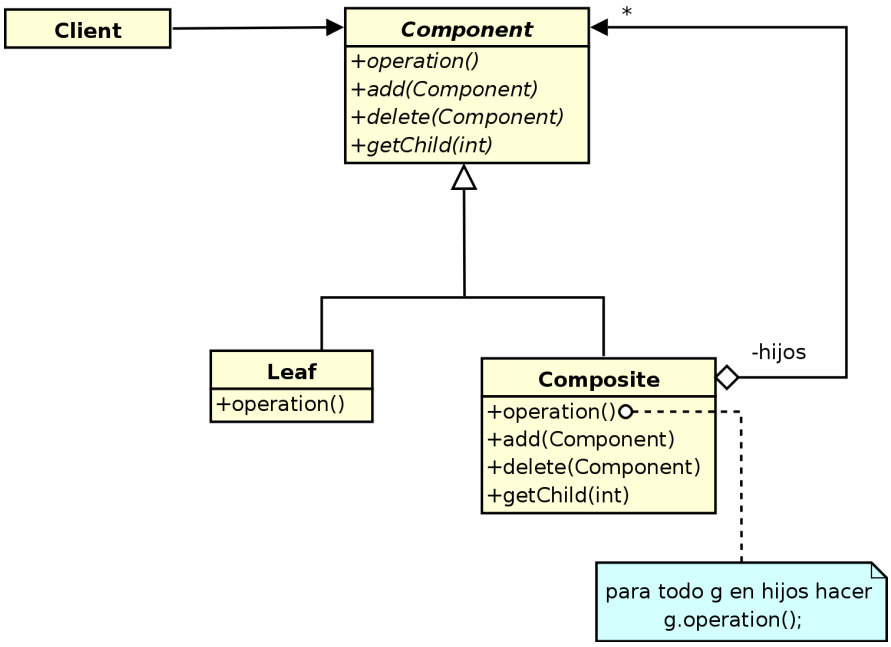


Figura 9: Patrón Composite

Obsérvese que un contenedor **Composite**, se compone de objetos ***Component***, los cuales pueden ser un elemento primario **Leaft** o bien, otro elemento compuesto.

Además, la clase abstracta ***Component*** declara operaciones que permiten el acceso, y la administración, de los hijos de compuestos **Composite**. En este último se encontrará la implementación de dichos métodos.

Los clientes usan la interfaz de la clase ***Component*** para interactuar con los objetos de la estructura compuesta. Si el componente es una hoja, la petición se lleva a cabo directamente. Si es un compuesto, normalmente, este redirige las peticiones a sus hijos, posiblemente realizando operaciones adicionales antes o después.

PARTICIPANTES

- ***Component***
  - declara la interfaz de los elementos de la composición.
  - implementa el comportamiento predeterminado de la interfaz que es común a todas las clases.
  - declara una interfaz para acceder a sus componentes hijos y gestionarlos.
  - (opcional) define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.
- **Leaft**
  - representa objetos hoja en la composición. Una hoja no tiene hijos.
  - define el comportamiento de los objetos primitivos de la composición.
- **Composite**
  - define el comportamiento de los componentes que tienen hijos.
  - almacena componentes hijos.
  - implementa las operaciones de la interfaz ***Component*** relacionadas con los hijos.
- **Client**
  - manipula objetos en la composición a través de la interfaz ***Component***.

Definir la gestión de los hijos en la raíz de la jerarquía de clases, da transparencia puesto que es posible tratar a todos los componentes del mismo modo. Sin embargo sacrifica la seguridad, ya que los clientes pueden hacer cosas sin sentido, como pedir a una hoja que agregue un componente.

De otro modo, definir la interfaz de administración de los hijos, en la clase **Compuesto**, proporciona seguridad pero a la vez, se pierde transparencia debido a que las hojas y los compuestos deben ser tratados de forma diferente.

Aunque podría priorizarse evitar por completo que los clientes hagan cosas inadecuadas, el espíritu de este patrón es dar más importancia a la transparencia renunciando en cierta medida a la seguridad. Por tanto, se sugiere que las operaciones de gestión de los hijos sean definidas en ***Component***.

5

DEVS Y PATRONES DE DISEÑO

Como se mencionó en la introducción, la implementación manual (parcial o total) de modelos DEVS es una tarea que se sigue llevando a cabo, aunque existan diversas herramientas para este formalismo. En principio, la causa de esto es que las herramientas disponibles, generalmente, requieren de cierta programación manual, y además, hay quienes prefieren sus propias implementaciones de modelos.

En ocasiones, estas implementaciones sufren problemas de mantenimiento. Esto se debe, a que cambios en el modelo implican modificaciones que resultan dificultosas la momento de plasmarlas en la implementación.

El paradigma de OO provee gran flexibilidad al momento realizar cambios. En particular, los patrones de diseño constituyen un recurso fundamental para crear sistemas que soporten modificaciones de un modo manejable.

En esta sección se propondrá un patrón para diseñar modelos DEVS, de modo tal que las implementaciones puedan contar con buena legibilidad y las modificaciones futuras puedan realizarse con cierta sencillez.

5.1 DEVS ATÓMICOS

El comportamiento que describe un modelo DEVS está determinado por los eventos de entrada, las transiciones internas y externas, el estado en que se encuentre, el tiempo de vida del estado y los eventos de salida que pueda generar.

En implementaciones con un enfoque procedimental, el cambio de por ejemplo un estado, puede implicar diversas modificaciones en distintos puntos del código, debido al gran acoplamiento en la implementación de los conceptos antedichos. En este tipo de enfoque, sentencias `if`, `switch` y `case` anidas son muy utilizadas. Si el modelo contase con gran cantidad de estados y transiciones, la tarea de modificarlo podría resultar en dificultosos cambios en el código.

El patrón State, descripto en la sección 4.2, es ampliamente utilizado en sistemas cuyo comportamiento interno depende del estado. Además, este patrón se sugiere para cuando se cuenta con operaciones que tienen largas sentencias condicionales, con múltiples ramas que dependen del estado. En consecuencia, a primera vista State es útil para diseñar modelos DEVS.

Recuérdese, que una de las ideas principales de State es definir cada estado como una clase aparte que implemente cierto comportamiento en particular. Ante esto, aparece el inconveniente de la posible existencia de un espacio infinito de estados. Esto es, el formalismo permite especificar sistemas que contemplan infinitos estados y, claramente, no es posible definir infinitas clases que los implementen.

El concepto *fase* (o *phase*) presentado en [7] resuelve en principio esta situación. Se introduce entonces a continuación, una adaptación de la definición de DEVS que contempla la noción de *fase*.

DEVS con Fases

El conjunto de *fases* de un modelo DEVS consiste en una partición finita del espacio de estados del sistema que es modelado. Cada estado está asociado a una fase y a así, en cada una, el sistema puede transitar por un número finito o infinito de estados. Por tanto, un DEVS con fases es una estructura

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

donde

X es el conjunto de valores de entrada

Y es el conjunto de valores de salida

S es el conjunto de estados, siendo

$S = \{(s_1, ..., s_n, fase, \nu_{ta}) \mid s_i \in V_i \wedge \ fase \in P \ \wedge \ \nu_{ta} \in \Re^{+}_{0,\infty}\}$

P es un conjunto finito de fases

V_i es un conjunto de valores

$\nu_{ta} = ta(fase)$

$\delta_{int} : S \to S$ es la función de *transición interna*

$\delta_{ext} : Q \times X \to S$ es la función de *transición externa*, donde

$Q = \{(s, e) | s \in S, 0 \leqslant e \leqslant ta(fase)\}$ es el conjunto de *estado total*

e es el *tiempo transcurrido* desde la última transición

$\lambda : S \to Y$ es la función de salida

$ta : P \to \Re^{+}_{0,\infty}$ es la función de *avance de tiempo*,

$ta(p)$ es el tiempo de vida asociado a la fase p

En este caso, el estado está constituido por una secuencia de variables. Una de estas establece la $fase$ en la cual está el sistema. Otra, el tiempo $ta(fase)$ por el cual el sistema permanecerá en la fase, si no ocurriese un evento de entrada. Notar que la función de avance de tiempo está definida sobre el conjunto de fases y no sobre el conjunto de estados, como en las definiciones anteriores.

La partición del espacio de estados, en un conjunto finito, posibilita que cada uno de estos esté representado por una fase y un grupo de variables. Esto permite utilizar el patrón State y determinar una clase por cada fase definida, donde cada una contenga variables de estado. De este modo, se puede implementar el conjunto total de posibles estados del sistema, independientemente de si es finito o no.

En esta definición, un estado puede ser *activo* o *pasivo* dependiendo del tiempo asociado a su fase. Si el tiempo de vida es infinito, el sistema permanecerá en el estado infinitamente o hasta que ocurra un evento de entrada externo. En este caso, se trata de una fase pasiva y por tanto el estado es llamado estado *pasivo*. Si en cambio, el tiempo asociado a la fase es finito, el sistema permanecerá en el estado a lo sumo por ese período de tiempo y luego, experimentará una transición interna si no existiesen eventos de entrada. En este caso se está ante una fase activa y por tanto se dice que el estado es *activo*.

estados activos y pasivos

Las nociones de fase, estado activo y estado pasivo son importantes al momento de analizar los conceptos del formalismo en pos de abordar la idea de patrones para DEVS.

5.1.1 *Eventos DEVS*

El uso de State para diseñar sistemas orientados a eventos, de alguna manera, indicaría que los estados son definidos como clases concretas separadas, cuyos métodos implementarían acciones y cambios de estado provocados por un evento. En definitiva, cada método representaría un evento que completa una transición de estado (tal vez realizando alguna acción adicional). Desde este punto de vista, cada una de las operaciones de un clase concreta de estado, encerraría los conceptos de evento, transición y acción.

Particularmente, en el formalismo DEVS con acoplamiento modular, los eventos son el medio por el cual diferentes modelos se comunican entre sí, provocando *eventos externos* unos en otros y transportando información a través de estos.

Por otra parte, el concepto de transición interna está dado por la noción de *evento interno*, donde un modelo cambia su estado cuando expira cierto tiempo.

Dadas estas características, la idea de implementar eventos como métodos sugiere no contar con una gran flexibilidad y un fácil manejo de los primeros. Parece más adecuado entonces, pensar en un evento como en un tipo de objeto, que podrá ser transmitido por los modelos para comunicarse con el exterior.

En términos generales; un evento, tanto interno como externo, ocurre en un instante determinado de tiempo. En el caso de tratarse de eventos externos, además de suceder en un momento particular, estos transportan información entre los modelos.

De este modo, los eventos serían objetos de una jerarquía de clases, ilustrada en Figura 10, que establecería las diferencias entre los dos tipos mencionados.

PARTICIPANTES

- ***`Event`***
  - es una clase abstracta que define una interfaz común para los eventos externos e internos.
  - guarda el momento en el cual ocurre el evento (variable `instant`).
  - implementa un método que permite obtener el momento en el que sucede el evento (`getTime()`).

- **`ExtEvent`**
  - es una subclase de ***`Event`*** que implementa un evento externo.
  - mantiene el valor que será transferido (`dat`), por medio del evento, entre los modelos.
  - implementa un método que permite acceder al valor del evento (`getDat()`).
- **`IntEvent`**
  - es una subclase de ***`Event`*** que representa un evento interno.
  - implementa el método de acceso al valor, de modo tal que este retorne un error o levante una excepción (`getDat()`).

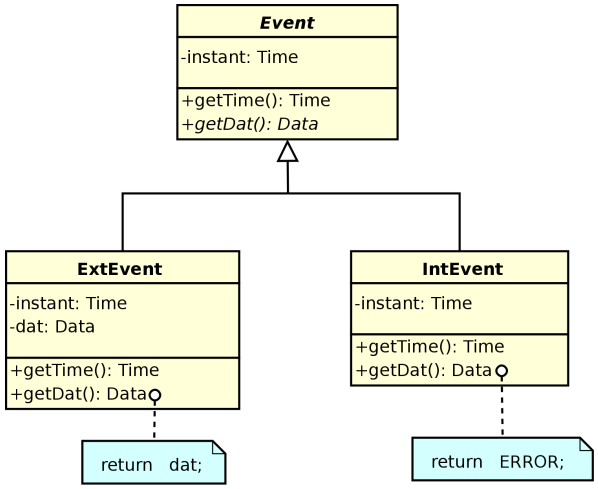


Figura 10: Estructura de Clases de Eventos DEVS

COMENTARIOS

En la figura, `Time` refiere a un tipo de dato para el tiempo y `Data` a un tipo de dato para los valores transmitidos entre los modelos.

En la clase **`IntEvent`**, el método `getDat()` levantará una excepción o retornará error si es invocado. Esto se debe a que las transiciones internas no requieren de un valor para llevarse a cabo. Aunque el método podría solo pertenecer a la clase **`ExtEvent`**, se considera adecuado mantener una interfaz común que permita tratar en forma homogénea los dos tipos de eventos y además, posibilite el manejo de errores.

5.1.2 *Estados DEVS*

Siguiendo el patrón State y la noción de fase presentada antes; se podría considerar como primer idea, definir una clase abstracta ***`State`*** y una subclase concreta **`ConcreteState`** por cada fase en la especificación DEVS.

Vale aclarar, que una subclase concreta del patrón State implementa solo un estado; mientras que en el escenario del formalismo, una subclase concreta podría implementar uno o varios estados, dependiendo del modelo. Esto se debe a que un estado DEVS puede ser simplemente una fase, o bien, una fase y un conjunto de variables.

Como se propuso más arriba, los eventos son objetos que son pasados entre los modelos. La clase **`Context`**, aquí llamada **`AtomicDEVS`**, proveerá la interfaz para los clientes y guardará el estado actual, delegando en este último la función de procesar los objetos ***`Event`*** que lleguen.

El método en ***`State`***, responsable por tratar los eventos, debería en primer lugar diferenciar entre un objeto **`IntEvent`** y uno **`ExtEvent`**. Esto le permitiría establecer si debe realizar una transición interna o externa e identificar el modo de actuar. Además, si se tratase de una transición externa, sería necesario que este método evaluase el objeto ***`ExtEvent`*** para determinar el siguiente estado. En caso de transición interna, no solo debería determinar el próximo estado sino también, debería generar un valor de salida.

Toda esta funcionalidad en solo un método, derivaría en distintas expresiones condicionales anidadas. Ante una gran cantidad de estados y transiciones, el código podría resultar confuso y difícil de mantener. Una de las causas de esto es que el formalismo DEVS cuenta con ciertas particularidades para las cuales el patrón State no puede proveer una representación explícita. Esto pone en evidencia la necesidad de proponer ciertas alternativas, inspiradas en dicho patrón, que se adecúen a los requerimientos que imponen los conceptos del formalismo.

Para empezar, debe observarse que los estados DEVS pueden pertenecer a dos grupos distintos. Como se detalló en la descripción de *fase*; *solo experimentarán transiciones internas los llamados estados activos*, y por tanto fases activas, debido a que el tiempo asociado a estos es finito. En resumen, en los estados activos ocurrirán transiciones internas y externas y, en los estados pasivos solo podrán llevarse a cabo las últimas. Resulta natural entonces, representar esta clasificación mediante la incorporación de dos subclases abstractas ***`ActiveState`*** y ***`PassiveState`***, que hereden de ***`State`*** y de las cuales, deriven a su vez, subclases concretas que constituyan las fases del modelo. Esta jerarquía de clases es ilustrada en Figura 11.

Por otra parte, así como se aislaron los eventos en una clase y los estados en otra, se plantea separar también el concepto de transición. De un modo similar al expuesto en [24] para FMS, las transiciones se proponen como objetos que conocen el próximo estado.

La idea principal es que el objeto que constituye el estado actual de **`AtomicDEVS`**, sólo sepa qué transición debe completarse ante cierto evento; no obstante, ignore totalmente el próximo estado y los detalles de cómo se llevaría a cabo dicha transición.

En el caso de las transiciones DEVS, claramente, estas conforman dos grupos, internas y externas. Esto sugiere que las transiciones sean definidas en dos clases diferentes. Más adelante, en el apartado 5.1.3, esta clasificación se verá con más detalle. Por ahora, basta con tener presente que las transiciones serán objetos de dos tipos, dígase ***`IntTransition`*** y ***`ExtTransition`***, que sabrán cuál es el estado destino.

La clase **`AtomicDEVS`** guarda el estado actual y provee una interfaz para los clientes, por medio de la implementación de métodos como `processEvent` y `doIntTransition` (ver Figura 11). A través del primero, el modelo recibirá eventos de entrada originados por otros modelos, lo que provocará transiciones externas. Por medio del segundo, el modelo recibirá la orden de control de efectuar una transición interna. Además, siguiendo la idea principal del patrón State, **`AtomicDEVS`** delegará en el estado `currentState` procesar los eventos.

La clase ***`State`*** definirá los métodos abstractos *`delta_Int`* y *`delta_Ext`* que en las clases heredadas serán implementados para realizar las transiciones. Por otra parte,

tendrá una variable privada `ta`, que definirá el tiempo de vida $ta(s)$ del estado. Aunque no está presente en la figura, probablemente existirá alguna variable que determine el nombre y, tal vez, otras variables que completen la definición del estado en caso de que el modelo lo requiriese; como así también, deberán existir los métodos adecuados para acceder a atributos privados.

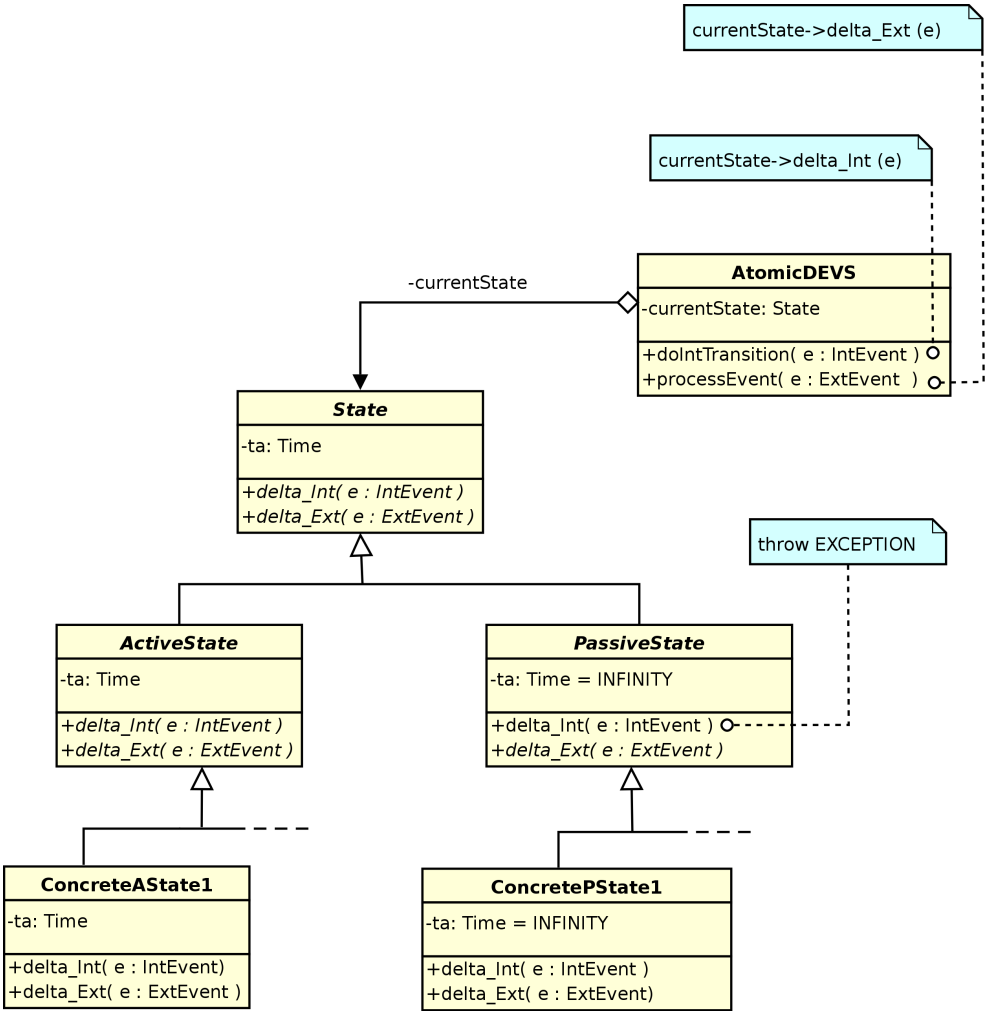


Figura 11: Estructura de Clases de Estados DEVS

Como antes se dijo, los estados de un modelo pueden ser activos o pasivos. Dependiendo de esto, corresponderá que una transición interna pueda o no existir en ellos. Así pues, ***`PassiveState`*** y ***`ActiveState`*** contarán con características distintas y las subclases heredadas de estos implementarán los estados concretos del modelo.

`PassiveState` especifica que el tiempo de cada estado pasivo es infinito, con lo que `ta` será igual a `INFINITY` (un valor arbitrario de tipo `Time` que representa el valor infinito). Además, cada estado ***`PassiveState`*** sólo sabrá qué transiciones externas pueden ocurrir en él. Con lo cual, sólo conocerá los objetos de tipo ***`ExtTransition`*** correspondientes. Como puede esperarse en esta clase, un pedido de transición interna, por medio de una invocación a `delta_Int`, levantará una excepción.

De forma análoga, cada estado ***`ActiveState`*** sabrá qué transiciones internas y externas pueden ocurrir en él. Esto es, tendrá conocimiento de los objetos de tipo ***`IntTransition`*** y ***`ExtTransition`***, a los que está asociado.

Para contar con una mejor legibilidad, por el momento las relaciones entre estados y transiciones no aparecen en la Figura 11; no obstante, serán ilustradas más adelante.

En términos generales, un estado tendrá asociados los objetos del tipo de transición que le correspondan. La idea fundamental es que un estado sólo conoce cuáles serán sus transiciones ante un evento, pero nada sabe acerca de cómo se llevan a cabo estas, ni cuál será el estado destino. Simplemente, debe determinar la transición correcta. En definitiva, cada estado oculta las relaciones *evento-transición* que le competen.

PARTICIPANTES

- **`AtomicDEVS`**
  - implementa una interfaz para los clientes que solicitarán, se efectúe una transición interna o externa (o se procesen eventos internos y externos).
  - mantiene el estado actual (un objeto de tipo ***`State`***) y delegará en este los pedidos de transición.
- ***`State`***
  - es una clase abstracta que define una interfaz que oculta el tratamiento de eventos internos y externos.
  - mantiene el tiempo $\mathrm{ta}(s)$ por el cual el sistema podrá permanecer en el estado.
- ***`ActiveState`***
  - es una subclase abstracta de ***`State`*** que define la interfaz que oculta el tratamiento de eventos internos y externos en un estado activo. En este tipo de estados el tiempo de vida es finito.
- ***`PassiveState`***
  - es una subclase abstracta de ***`State`*** que define una interfaz para ocultar el tratamiento de eventos externos en un estado pasivo.
  - define el tiempo de vida igual a infinito.
- **`ConcreteAState1`**
  - es una subclase concreta de ***`ActiveState`*** que implementa cierto comportamiento de un estado activo de un DEVS atómico.
  - conserva las relaciones *evento-transición* que le permiten determinar, ante un evento, la transición adecuada.
  - conoce objetos de tipo ***`IntTransition`*** y ***`ExtTransition`***.
- **`ConcretePState1`**
  - es una subclase concreta de ***`PassiveState`*** que implementa el comportamiento de un estado pasivo de un DEVS atómico.
  - mantiene las relaciones *evento-transición* por medio de las cuales establece la transición que corresponde a un evento.
  - sólo conoce transiciones de tipo ***`ExtTransition`***.

5.1.3 *Transiciones DEVS*

Las transiciones de un modelo son definidas como objetos de una clase que conocen el estado destino. Como se ha dicho en el apartado anterior, existen transiciones internas y externas que serán representadas por medio de dos clases abstractas diferentes, ***IntTransition*** y ***ExtTransition***. Particularmente, cada transición será implementada en una subclase concreta que herede de alguna de estas. Esta jerarquía de clases se ilustra en Figura 12.

Cada objeto de tipo ***Transition*** conocerá a un objeto ***State***, el cual indicará el estado destino (`targetState`) de la transición. El método `processTransition` permitirá determinar este nuevo estado.

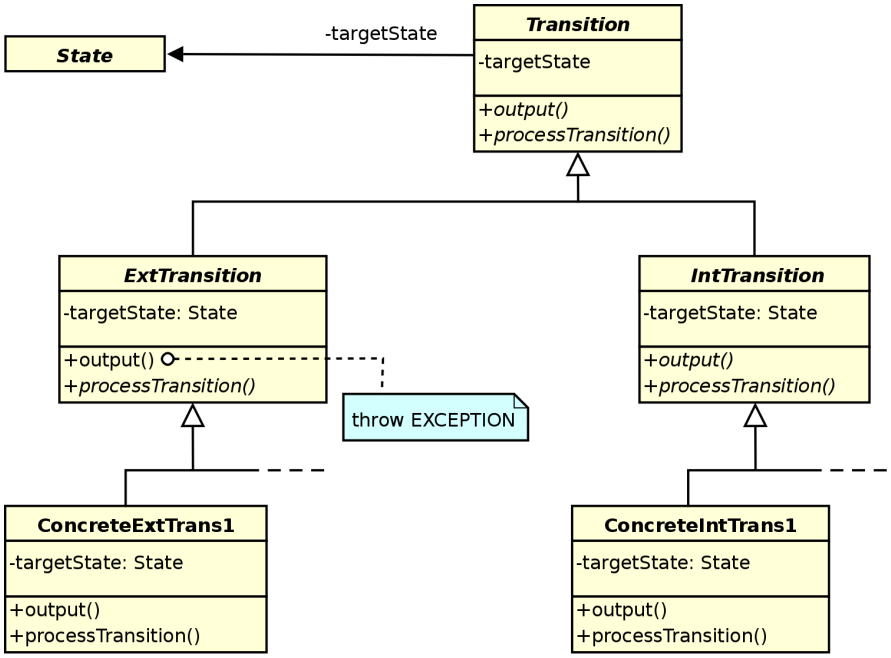


Figura 12: Estructura de Clases de Transiciones DEVS

Por otro lado, un objeto ***IntTransition*** no solo conocerá el siguiente estado de la transición, sino también, será necesario que sepa computar el valor de salida. El método `output()` es el que permitiría calcular este valor. En ***ExtTransition*** este método debería retornar un error o levantar una excepción, en el caso de ser invocado.

PARTICIPANTES

- ***Transition***
  - es una clase abstracta que define una interfaz común para transiciones internas y externas.
  - conoce un objeto de tipo ***State***, que determina el estado destino de la transición.
- ***ExtTransition***
  - es una subclase abstracta de ***Transition*** que define la interfaz que oculta cómo se efectúa una transición externa.

- ***IntTransition***
  - es una subclase abstracta de ***Transition*** que define la interfaz que oculta cómo se realiza una transición interna.
- **ConcreteExtTrans1**
  - es una subclase concreta de ***ExtTransition*** que implementa los métodos que permiten llevar a cabo determinada transición externa.
- **ConcreteIntTrans1**
  - es una subclase concreta de ***IntTransition*** que implementa los métodos que posibilitan completar una transición interna. En particular, implementa el cómputo del valor de salida asociado a la transición.

La idea principal aquí es que cada transición conoce el estado destino que tiene asociado, y en particular, las transiciones internas saben calcular el valor de salida.

Definir las transiciones como elementos de dos tipos diferentes separados de los estados, permite agregar nuevas transiciones o eliminar otras, de un modo relativamente sencillo.

5.1.4 *Estados y Transiciones DEVS*

Hasta aquí, para el diseño de modelos DEVS, se ha recomendado seguir como patrón la desarticulación de eventos, estados y transiciones, en clases de objetos diferentes. Se ha sugerido además, identificar la subcategoría a la que pertenece cada uno y definir esta clasificación, por medio de subclases que están claramente determinadas por la naturaleza de cada concepto. Esto es, por ejemplo, separar estados activos de pasivos, como así también, transiciones internas de externas. Además, se ha propuesto:

- que **AtomicDEVS** defina la interfaz de acceso al modelo atómico, mantenga el estado actual del mismo y delegue en este ciertas tareas;
- que un objeto de estado ***State*** tenga la responsabilidad de establecer cuál es la transición adecuada en cada momento y,
- que un objeto de transición ***Transition*** asuma la tarea de determinar el siguiente estado y, en caso de transición interna, calcule el valor de salida.

La Figura 13 muestra la relación entre estados y transiciones. Obsérvese que el vínculo entre estado y transición se muestra mediante tres relaciones de ***asociación***.

Por un lado, cada estado (***State***) conoce las transiciones externas (***ExtTransition***) que tiene asociadas, y estas pueden ser más de una o ninguna. A su vez, cada transición externa será una transición de uno o más estados. Esto último significa que ante eventos externos, diferentes estados origen, pueden tener el mismo estado destino. En resumen; un estado activo o pasivo puede tener asociado cero o más transiciones externas; y al mismo tiempo, más de un estado puede estar asociado a la misma transición.

Por otro lado, los estados que son activos (***ActiveState***), además conocerán su transición interna (***IntTransition***). Como antes, en este caso también diferentes estados pueden tener asociada la misma transición.

Por último, la figura muestra que cada transición (***Transition***) conoce un estado destino (***State***) que es el que le corresponde.

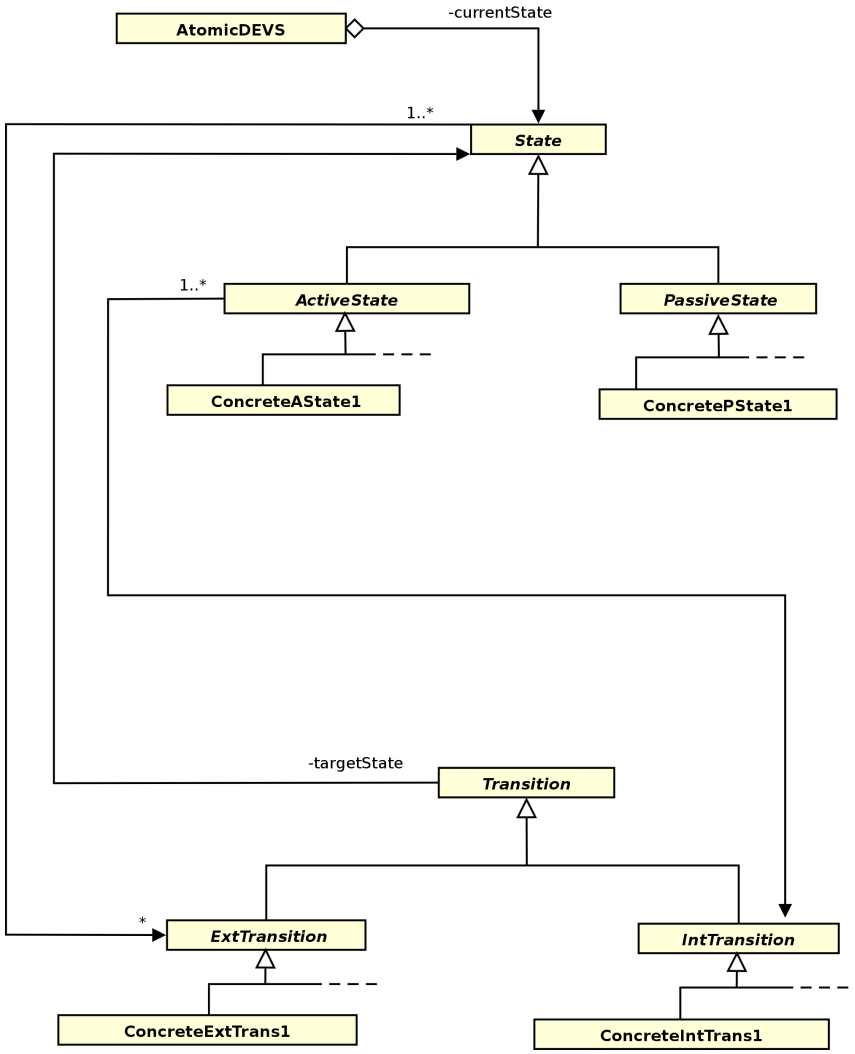


Figura 13: Relaciones entre Estados y Transiciones DEVS

INTERACCIÓN ENTRE LOS OBJETOS

Las clases presentadas en los apartados anteriores definen en sus interfaces la cantidad mínima de métodos que permiten describir el espíritu de lo propuesto. No obstante, otros tantos métodos y parámetros deberán existir en un diseño completo. Por ejemplo, **`AtomicDEVS`** debería permitir acceder al valor resultante de una transición interna, como así también, al estado actual. Por otra parte, aquí quedan sin especificar distintas cuestiones, como por ejemplo quién es responsable de fijar el nuevo estado en **`AtomicDEVS`**.

- Una alternativa a esto es que cuando un cliente señale una transición en **`AtomicDEVS`**, este último solicite a su estado actual que le indique cuál es la transición que corresponde. Una vez obtenida, **`AtomicDEVS`** podría pedir a la transición el nuevo estado y, en caso de tratarse de un transición interna, le solicitaría el valor de salida. Luego, el cliente podría acceder mediante un método provisto por **`AtomicDEVS`**, al nuevo estado; y en caso de tratarse de una transición interna accedería también al valor de salida.

La Figura 14 muestra la estructura de clases de esta alternativa. Aquí se han ocultado, para facilitar la lectura, algunas operaciones y atributos presentes en las figuras anteriores. Además se han agregado métodos en la interfaz de **`AtomicDEVS`** y ***`State`***, y se han especificado valores de retorno para los métodos `processTransition` y `output`, de la clase ***`Transition`***.

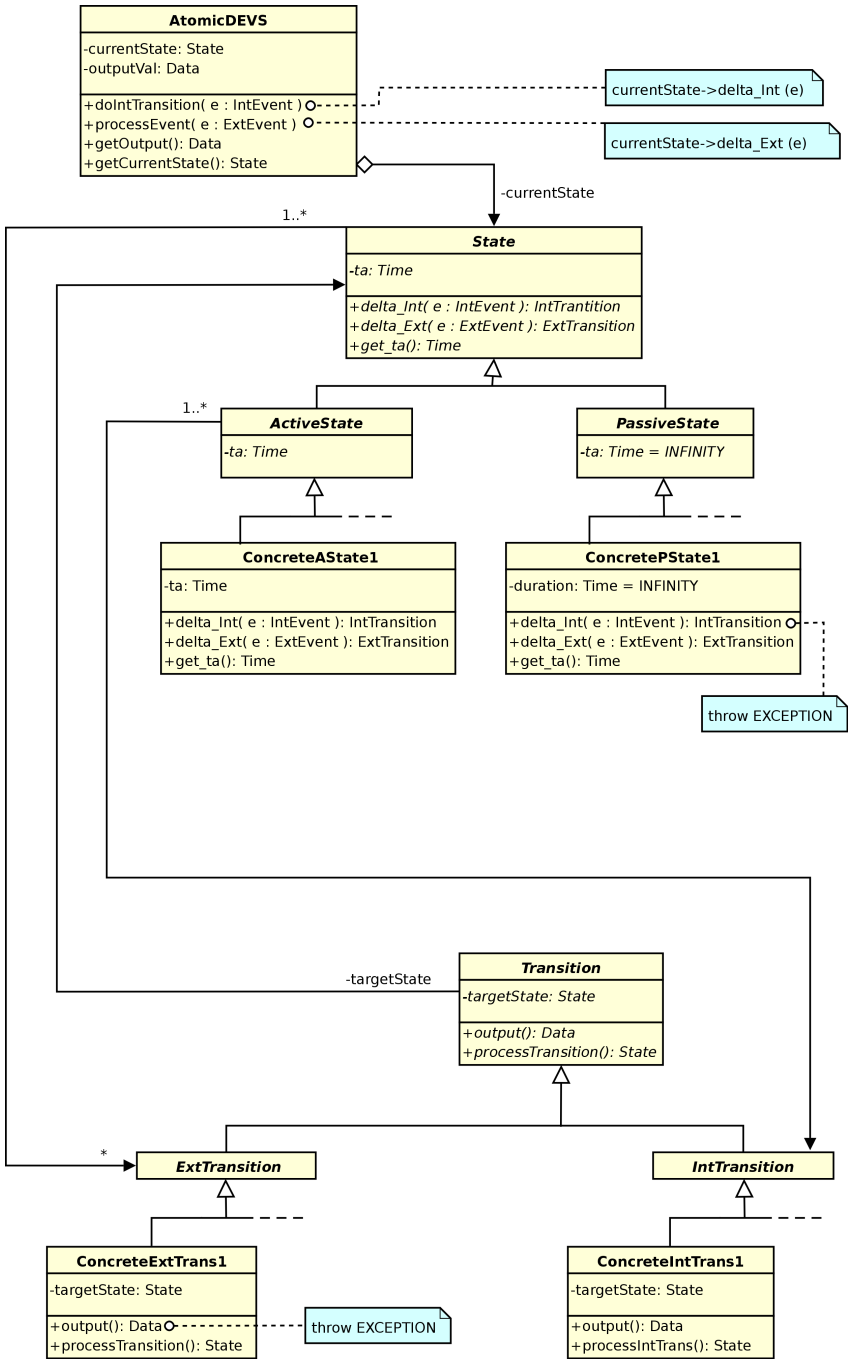


Figura 14: Estructura de Clases de Estados y Transiciones DEVS

La secuencia de interacción entre los objetos, ante un pedido de transición externa, se muestra en Figura 15. Aquí, un cliente `c`, llamando a `processEvent(e)`, solicita a un modelo `devs` que procese un evento externo. El objeto `devs` delega en su estado actual la tarea de determinar la transición externa correspondiente. Para esto, invoca a `delta_Ext(e)` de su estado actual `currentState`.

A su vez, `currentState` cuenta con un mecanismo privado para seleccionar la transición (por ej. un método `selectTransition()`); de esta manera, devuelve a `devs` la transición adecuada `et_trans`. Ante esto, `devs` solicita a dicha transición el siguiente estado y fija entonces, el nuevo valor de `currentState`. Una vez finalizado, el cliente puede acceder al nuevo estado llamando a `getCurrentState()`.

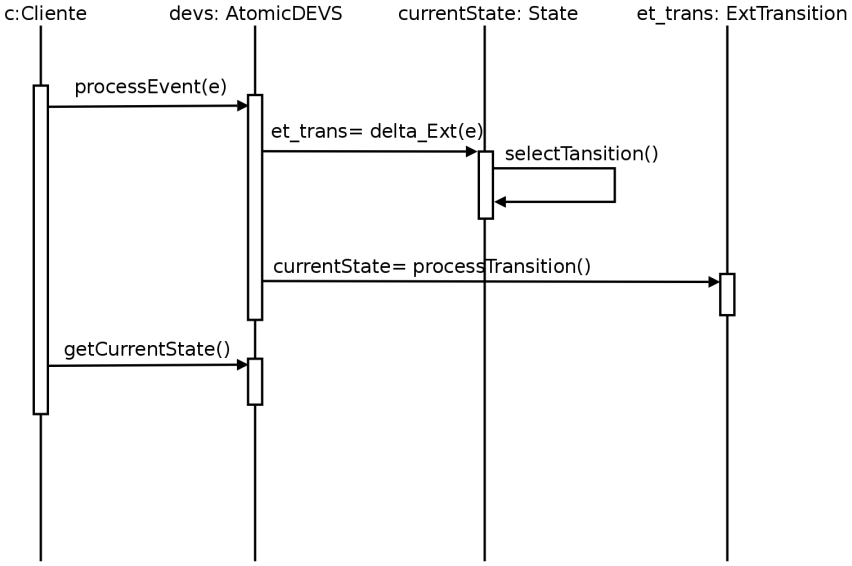


Figura 15: Secuencia de Interacción - Transición Externa

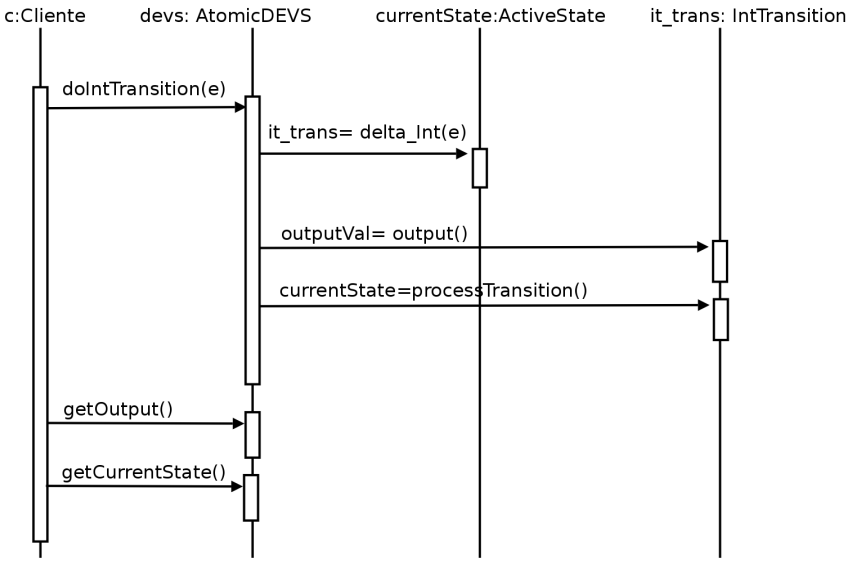


Figura 16: Secuencia de Interacción - Transición Interna

Por otro lado, la secuencia de interacción entre objetos, ante una transición interna, es similar salvo por que además, el valor de salida debe ser calculado. La Figura 16 muestra esta interacción. En este caso, el método de `devs` invocado por el cliente es `doIntTransition(e)`. Ante esto, `devs` solicita a `currentState` la transición correspondiente llamando a `delta_Int(e)`. De este modo, `devs` accede a la transición interna `it_trans`, y le pide el valor de salida y el siguiente estado, por medio de `output()` y `processTransition()` respectivamente.

Cuando esto finaliza, el cliente podría solicitar el valor de salida, invocando a `getOutput()`, y el nuevo estado, llamando a `getCurrentState()`.

- Otra posibilidad de interacción entre objetos sería la siguiente.

  `AtomicDEVS` delegaría en `currentState`, de forma total, la tarea de llevar a cabo una transición. Entonces, el estado actual seleccionaría la transición correcta y a su vez, delegaría en esta la función de completar el cambio de estado y, en el caso de ser necesario, la tarea de calcular y fijar el valor de salida.

  Para esto, **`AtomicDEVS`** debería definir `setCurrentState(newState:State)` y `setOutput(outval: Data)` en su interfaz, de modo tal que otros pudieran fijar sus valores. Además, pasaría una referencia de sí mismo a `currentState`, el cual la pasaría, a su vez, a la transición responsable por asignar los nuevos valores.

  Debido a que el diagrama de clases sería similar al anterior, no será introducido. En cambio, se describirán las variaciones que sufriría y se mostrarán los diagramas de interacción para describir esta nueva alternativa.

  Como se dijo, el diagrama de clases sería igual al anterior salvo por lo siguiente:

  - **`AtomicDEVS`** definiría además en su interfaz a `setCurrentState(newState:State)` y `setOutput(outval: Data)`,
  - ***`Transition`*** no tendría definida la operación `output()`, al menos de forma pública, debido a que este cálculo se llevaría a cabo dentro de `processTransition()` en las subclases de ***`IntTransition`*** y,
  - los métodos deberían admitir pasar como parámetro la referencia a **`AtomicDEVS`**.

  Obsérvese en Figura 17 cuál sería la secuencia de acción entre los objetos, ante un pedido de transición externa.

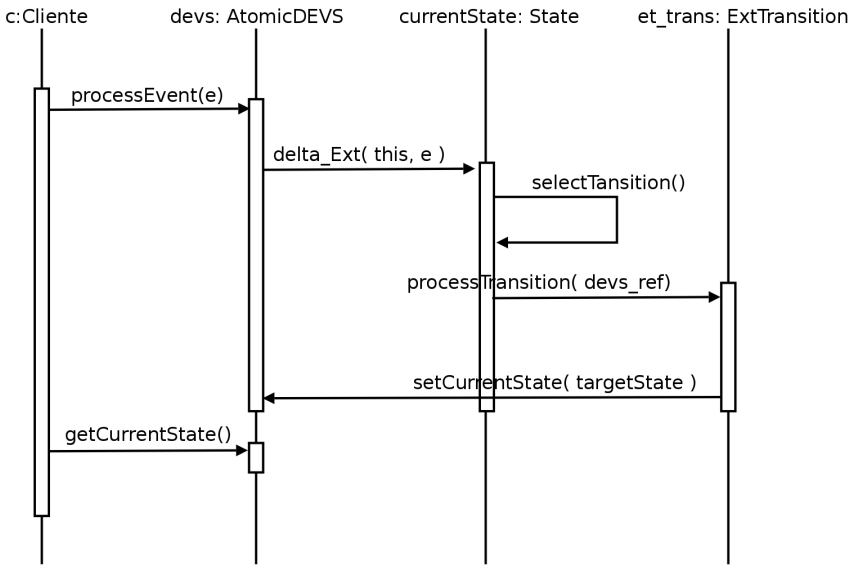


Figura 17: Secuencia de Interacción - Transición Externa

En este caso, cuando un cliente invoca `processEvent(e)`, `devs` delega en `currentState` efectuar la tarea. Para esto, llama a `delta_Ext(this, e)` pasándole al estado actual, una referencia de sí mismo (`this`) como parámetro.

`currentState` selecciona la transición adecuada (`et_trans`) y delega en esta el trabajo restante, por medio de la llamada a `processTransition(devs_ref)`.

El estado le pasa a la transición la referencia a `devs` (`devs_ref`) para que dicha transición fije los nuevos valores. Entonces, `et_trans` determina el nuevo estado `targetState` y, utilizando `setCurrentState(newState:State)` de `devs`, fija el nuevo valor de `currentState`. Luego, el cliente puede acceder al nuevo estado por medio de `getCurrentState()`.

En el caso de procesarse una transición interna, la secuencia sería la que muestra la Figura 18.

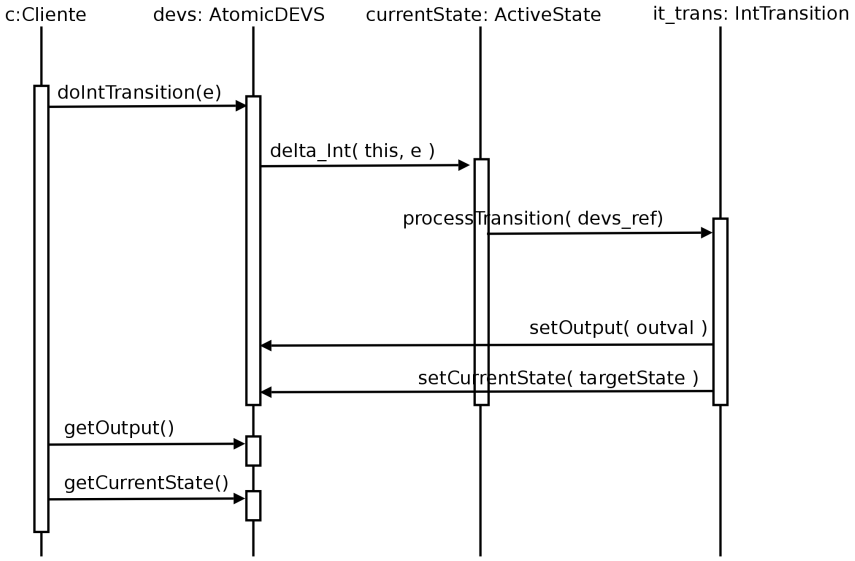


Figura 18: Secuencia de Interacción - Transición Interna

Como puede observarse, la interacción entre los objetos es similar a la del caso de transición externa; salvo que aquí, se requiere el cálculo de un valor de salida. El objeto ***IntTransition*** no solo es el responsable de fijar el nuevo estado, sino también, debe calcular el valor de salida y asignarlo en **AtomicDEVS**.

En la primera de estas alternativa, las tareas se centralizan en **AtomicDEVS** que es quien debe obtener la transición, solicitar el nuevo estado y, en caso de transición interna, pedir el valor de salida. En la segunda opción, **AtomicDEVS** simplemente delega el pedido en `currentState` e ignora todo lo demás; con lo cual sus variables sufrirán cambios transparentes para él y sólo se limitará a responder solicitudes.

Como es de suponer, otras posibilidades podrían definirse. No obstante, las dos alternativas presentadas son admisibles; aunque requeriría de un análisis más profundo determinar, la más adecuada entre éstas y otras que pudieran surgir.

COMENTARIOS

Un método `action()` sería necesario en el caso en que el modelo contase con variables de estado. En principio, este método sería privado de ***Transition***, e invocado en el cuerpo de `processTransition`, para dar valores a las variables de `targetState`, antes de que este último se estableciera como el nuevo estado.

Si el modelo contase además, con condiciones sobre las transiciones, se requeriría en ***Transition*** un método público `guard()` cuya implementación verificaría las condiciones necesarias para realizar la transición. Este método podría ser invocado antes de `processTransition()`.

Por otro lado, en una transición, los valores de las variables del estado destino pueden depender tanto de valores del evento entrante, como de ciertos valores del estado origen. Un ejemplo de esto sería contar con una expresión como

$$\delta_{ext}((faseA,\ r,\ ta_1),\ e,\ x) = (faseB,\ r + x,\ ta_1 + e)$$

Con lo cual, un objeto ***Transition*** podría tener la necesidad de acceder a los valores del evento y del objeto `currentState` (en la expresión anterior, (e, x) y $(faseA,\ r,\ ta_1)$ respectivamente).

Para cubrir esta necesidad, una alternativa sería que cuando se invoque `processTransition`, se pase como parámetro el estado actual junto con el evento entrante. No obstante, podría ocurrir que en un modelo, ciertas transiciones requirieran de esta información y otras no. De esta forma, `processTransition` tendría declarados dos argumentos cuyos valores, en algunos casos, serían pasados al método pero no utilizados por este.

Otra opción sería definir operaciones en ***Transition*** (por ej. `setbeforeState` y `setEvent`) que permitieran registrar el estado origen y/o el evento entrante en la transición. Así, estos métodos solo se utilizarían cuando fuera necesario que la transición accediera al estado actual y al evento, para definir los valores de las variables del nuevo estado.

5.2 DEVS ACOPLADOS

El acoplamiento modular de DEVS posibilita la construcción jerárquica de modelos y la propiedad de clausura, que verifica esta construcción, permite la creación de modelos más y más grandes sin necesidad de modificar los ya existentes. En particular, el acoplamiento modular con puertos es muy utilizado debido a su simplicidad para vincular modelos.

Para diseñar la implementación de este tipo de estructura se propone utilizar las nociones del patrón Composite de la sección 4.3, ya que es de gran utilidad para construir diseños en árbol donde los elementos básicos son tratados del mismo modo que los elementos compuestos.

5.2.1 *Puertos DEVS*

En acoplamiento modular con puertos, los modelos se conectan a través de estos. Así, un valor de entrada o de salida de un modelo, es un par puerto-valor donde el valor pertenece al dominio definido por el puerto.

Por consiguiente, se definirá un puerto como un objeto que cuenta con un dominio determinado. Para esto, se presentará una clase abstracta ***Port*** de la cual derivarán subclases de puertos, con diferentes dominios. La Figura 19 muestra su estructura.

En principio, un puerto tendrá un nombre que podrá ser recuperado por `getName();` método implementado en la clase abstracta ***Port***.

A su vez, cada clase concreta **`ConcretePortType`** implementará un tipo de puerto particular. Esto es, un puerto que admita cierto tipo de valores. El método `checkValue(v:Val)` es el que indicará si un valor pertenece al dominio del puerto o no.

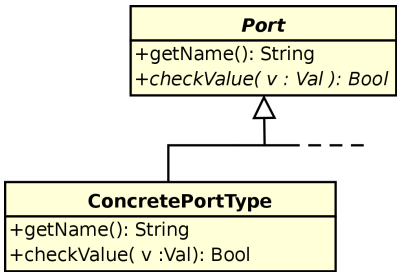


Figura 19: Estructura de Clases de Puertos DEVS

PARTICIPANTES

- ***Port***
  - define la interfaz común para puertos con distinto dominio.
- **ConcretePortType**
  - implementa el comportamiento asociado a un tipo de puerto en particular. Para esto define internamente su dominio y un método que permita saber si un valor está en él.

COMENTARIOS

Notar que en la clase **ExtEvent** definida en 5.1.1, el dato transportado por el evento es de un tipo genérico `Data`. En el contexto de DEVS acoplados con puertos, un objeto de tipo `Data` debería contener un puerto y un valor. Es decir, debería ser un elemento `(`***Port***`, Val)`.

5.2.2 *Conexiones DEVS*

Un modelo acoplado está constituido por un conjunto de modelos conectados entre sí a través de puertos. Las conexiones en un modelo de este tipo pueden corresponder al *acoplamiento de entrada externo* (EIC), al *acoplamiento interno* (IC) o al *acoplamiento de salida externo* (EOC). En el primero, los puertos de entrada del modelo acoplado se conectan a puertos de entrada de modelos internos. En IC, puertos de salida de modelos internos son conectados a puertos de entrada de otros modelos internos. Y en EOC, puertos de salida de modelos internos se conectan a puertos de salida del modelo acoplado.

Por tanto, se propone definir las conexiones como objetos de tres clases diferentes que formarán parte de la definición de un modelo acoplado. En términos generales, una conexión quedará definida por un origen, constituido por un modelo y un puerto, y un destino, conformado por otro modelo y otro puerto.

La Figura 20 muestra la estructura de clases propuesta para las conexiones. Aquí, `DEVSModel` es un tipo de objeto que representa tanto un modelo atómico como uno acoplado. En la siguiente sección, la clase `DEVSModel` será descripta en más detalle. Por ahora, basta saber que un objeto de esta clase tiene puertos de entrada y de salida, y que puede consultársele sobre la existencia de estos.

En la figura, ***Connection*** define la interfaz común para todas la conexiones en un modelo acoplado. Esta interfaz permitiría recuperar el modelo y puerto origen, por

medio de `getmodelSrc()` y `getportSrc()`; como así también, el modelo y puerto destino, a través de `getmodelTgt()` y `getportTgt()`. Dado que la funcionalidad de dichos métodos sería la misma para cualquier tipo de conexión, estos deberían ser implementados en ***`Connection`***.

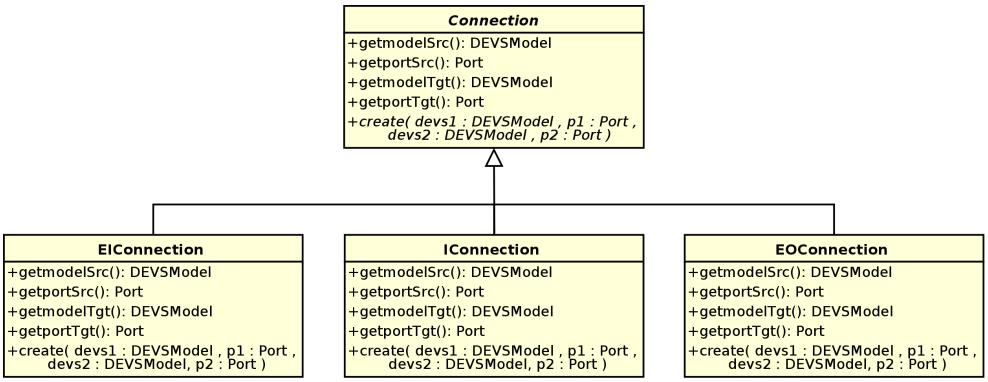


Figura 20: Estructura de Clases de Conexiones DEVS con Puertos

`EIConnection`, **`IConnection`** y **`EOConnection`** definen clases concretas que implementarán las distintas conexiones. En todas, el constructor deberá controlar que el puerto origen pertenezca al modelo origen, y el puerto destino sea parte del modelo destino. Para esto, consultará a cada `DEVSModel` acerca de sus puertos. Además, verificará que los modelos origen y destino, no sean el mismo objeto. Esto constituye uno de los requerimientos en el acoplamiento DEVS; es decir, un modelo no puede estar conectado a sí mismo.

No obstante, la diferencia fundamental entre las tres clases, está en su construcción. El constructor de **`EIConnection`**, deberá constatar que el puerto origen sea un puerto de *entrada* del modelo origen y, el puerto destino sea un puerto de *entrada* del modelo destino. En cambio, el constructor de **`IConnection`** deberá verificar que el puerto origen sea un puerto de *salida* del modelo origen y, el puerto destino sea un puerto de *entrada* del modelo destino. Por último, el constructor de **`EOConnection`** controlará que el puerto origen sea un puerto de *salida* del modelo origen y, el puerto destino sea un puerto de *salida* del modelo destino.

Los modelos origen y destino, de **`EIConnection`** y **`EOConnection`** respectivamente, serán el modelo acoplado que contenga (o sea responsable de) estas conexiones. El control de esto lo hará el mismo modelo como se describirá en la próxima sección.

PARTICIPANTES

- ***`Connection`***
  - tiene un modelo DEVS y un puerto, como origen; y otro par de estos como destino.
  - define la interfaz común para las distintas conexiones.
  - implementa los métodos que permiten obtener los extremos de la conexión.
- **`EIConnection`**
  - implementa una conexión entre un puerto de entrada origen y otro puerto de entrada destino.

- **IConnection**
  - implementa una conexión entre un puerto de salida origen y un puerto de entrada destino.
- **EOConnection**
  - implementa una conexión entre un puerto de salida origen y un puerto de salida destino.

5.2.3 *Modelo DEVS Acoplado*

La implementación de un modelo acoplado define la estructura de este, pero no es responsable por la comunicación que pudiera ocurrir entre sus componentes. Esta es una tarea de las máquinas de simulación, que toman la definición de los modelos para simular comportamiento. En consecuencia, la implementación de un modelo acoplado, solo define una estructura y provee métodos para construirla y obtener información sobre ella.

Un modelo DEVS puede ser atómico o acoplado. Y un modelo acoplado puede estar compuesto de otros modelos DEVS que a la vez, pueden ser atómicos o acoplados. Este carácter recurrente de la definición, lleva al patrón Composite.

La idea fundamental es definir una clase abstracta ***DEVSModel*** que represente a cualquiera de los dos tipos de modelos. De esta clase heredarán la clase **AtomicDEVS**, de los apartados anteriores, y una clase **CoupledDEVS** que implementará la estructura de modelos acoplados y los métodos para su construcción. Los objetos de esta última, deberían poder contener objetos **AtomicDEVS** y objetos de su mismo tipo **CoupledDEVS**. Por tanto, dicha clase estará compuesta por elementos de tipo ***DEVSModel***, como indica la relación de ***agregación*** en Figura 21.

Además, los modelos *tienen* puertos ***Port*** por dónde ingresan o salen valores. Como muestra la figura, se considera que **AtomicDEVS** al menos tendrá un puerto de entrada o uno de salida. En principio, el comportamiento mínimo especificable, sería un modelo pasivo que reciba eventos de entrada o, un modelo que sólo emita valores de salida; y en ambos casos, se requeriría de un puerto. Sin embargo, un modelo acoplado podría no enviar ni recibir valores hacia o desde el exterior; y simplemente, contener otros modelos y sus conexiones.

Por otra parte, **CoupledDEVS** podrá tener conexiones entre los modelos que lo constituyen y, conexiones de él mismo con sus modelos internos. Con lo cual, se define una relación de ***agregación*** hacia la clase ***Connection*** definida antes, indicando que un modelo acoplado *tiene* conexiones.

La clase ***DEVSModel*** define tanto las interfaces de los objetos **AtomicDEVS** como las de los objetos **CoupledDEVS**. De esta manera, tanto un modelo atómico como un modelo acoplado, pueden ser tratados de manera uniforme.

Por otro lado, debería ser igual, en ambos tipos de modelo, determinar si un puerto es un puerto de entra o de salida. Lo mismo ocurre con los métodos para agregar o eliminar puertos. Por tanto, ***DEVSModel*** implementará las operaciones comunes a sus subclases. En la figura, los métodos relacionados a las acciones mencionadas son `isInPort`, `isOutPort`, `add_InPort`, `add_OutPort`, `del_InPort` y `del_OutPort`.

AtomicDEVS implementará los métodos correspondientes a un modelo atómico, como en las secciones anteriores. Los métodos heredados desde ***DEVSModel*** que

tuvieran que ver con la administración de componentes, implementarían aquí algún tipo de manejo de errores. Estos métodos no se muestran en la clase **`AtomicDEVS`** de la figura, a fin de contar con una mejor legibilidad.

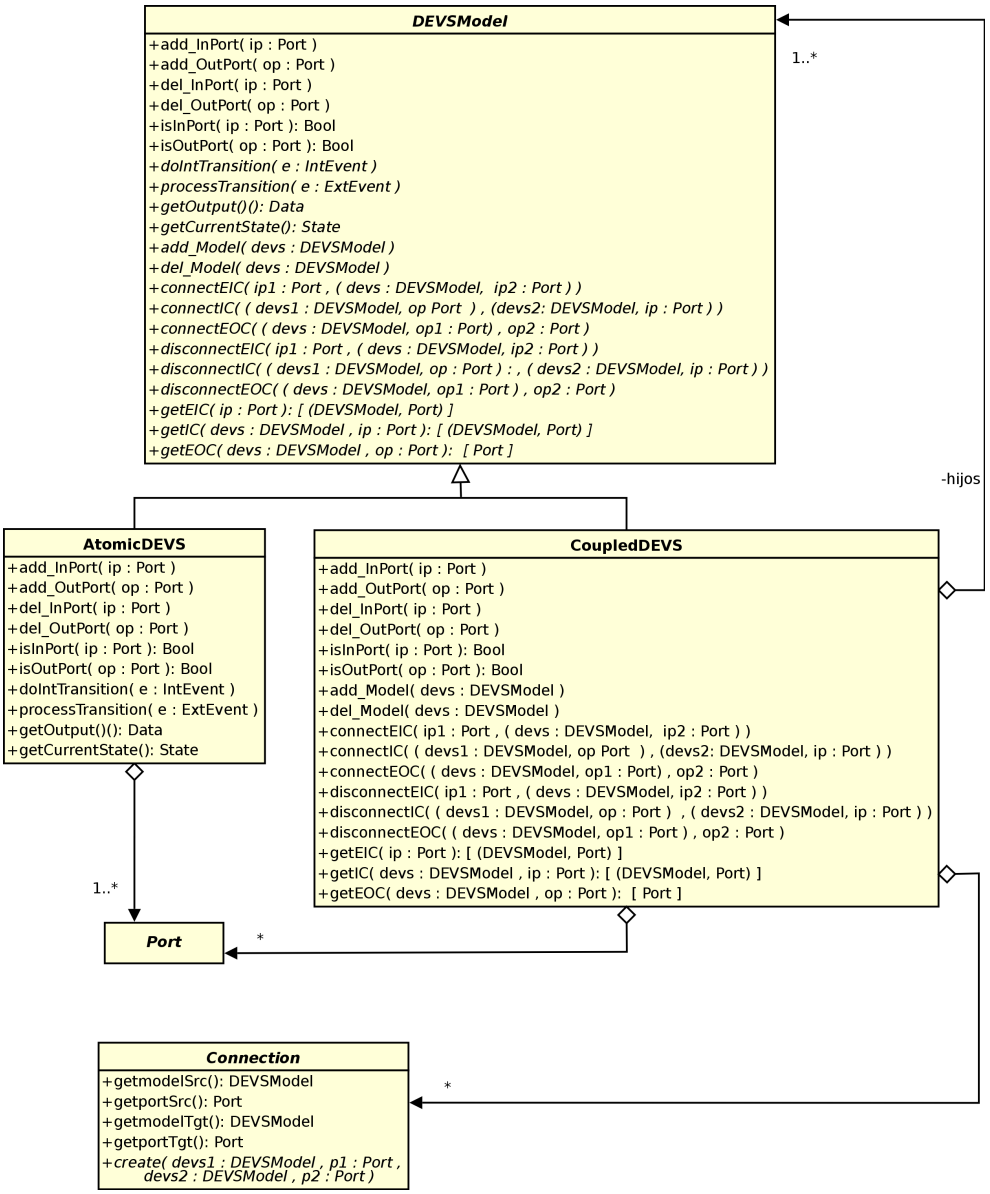


Figura 21: Estructura de Clases de DEVS Acoplado con Puertos

A su vez, **`CoupledDEVS`** definirá la implementación vinculada a la gestión de sus modelos hijos. Por ejemplo, `getModels():[ DEVSModel ]` permitiría recuperar todos los modelos pertenecientes al modelo acoplado. (La notación `[ DEVSModel ]` indica una lista de objetos tipo ***`DEVSModel`***).

Las operaciones `getEIC(ip:Port): [(DEVSModel, Port)]`, `getIC(devs:DEVSModel, ip:Port): [(DEVSModel, Port)]` y `getEOC(devs:DEVSModel, op:Port): [Port]` darían información sobre a qué puertos y modelos destino, está conectado un puerto y modelo determinado.

En el caso de `getEIC(ip:Port)`, el parámetro `ip` es un puerto de entrada del modelo acoplado, que es el origen de la conexión. Este método retornaría todos los modelos internos y los puertos a los que está conectado `ip`.

`getIC(devs:DEVSModel, ip:Port): [(DEVSModel, Port)]` claramente recibe un modelo interno origen y su puerto asociado, y retorna los modelos internos y sus puertos, que definen los extremos de las distintas conexiones.

La función `getEOC(devs:DEVSModel, op:Port): [Port]` toma como entrada un modelo interno y su puerto de salida, y devuelve los puertos de salida del modelo acoplado, a los que se conecta el puerto original.

Por otro lado, la clase **`CoupledDEVS`** además implementaría métodos para agregar nuevos modelos y eliminarlos. En la Figura 21 estos métodos son `add_Model( devs : DEVSModel)` y `del_Model( devs : DEVSModel)`.

A su vez, la definición de conexiones entre modelos, también estaría implementada en dicha clase a través de operaciones como `connectEIC( ip1:Port, (devs:DEVSModel, ip2:Port))`, `connectIC((devs1:DEVSModel, op:Port),(devs2:DEVSModel, ip:Port))` y `connectEOC(op1:Port, (devs:DEVSModel, op2:Port))` definidas en el diagrama de clases de la figura.

`connectEIC` controlaría, en primer lugar, que `ip1` fuera un puerto de entrada del modelo acoplado propio, y luego, que `devs` fuera un componente dentro de este. Entonces, construiría un objeto **`EIConnection`**, el cual incorporaría al conjunto de conexiones de este tipo, en el modelo. Claramente, al constructor de **`EIConnection`**, debería pasarle una referencia `this` (del propio modelo acoplado) y el puerto `ip1` como origen. Recordar, que los controles sobre las características de los extremos de una conexión y aquellos sobre los requerimientos de acoplamiento, son realizados por los constructores vinculados a ***`Connection`***. (Ver sección anterior 5.2.2.)

En el caso de `connectIC`, el método controlaría que los modelos `devs1` y `devs2` fueran parte del modelo acoplado y luego, construiría un objeto **`IConnection`** que agregaría a la colección de conexiones IC.

Por su parte, `connectEOC` antes de crear el objeto **`EOConnection`**, debería verificar que `op1` fuera uno de los puertos de salida del modelo acoplado y que `devs` fuera uno de los hijos de este. Una vez comprobado esto, el objeto de conexión sería creado e incorporado a el conjunto de conexiones EOC.

Los métodos para eliminar conexiones también serían implementados en **`CoupledDEVS`**. En el diagrama de clases presentado más arriba, estos métodos son `disconnectEIC( ip1:Port, (devs:DEVSModel, ip2:Port))`, `disconnectIC((devs1:DEVSModel, op:Port),(devs2:DEVSModel, ip:Port))` y `disconnectEOC(op1:Port, (devs:DEVSModel, op2:Port))`. Estas operaciones eliminarían las conexiones definidas en el modelo acoplado.

Estos métodos, en lugar de recibir tan explícitamente todos los parámetros de una conexión, podrían recibir directamente el objeto a eliminar ***`Connection`*** o bien, un identificador de éste. En tal caso, las operaciones de conexión presentadas antes, tendrían que retornar algún tipo de identificador de la conexión creada.

Por otra parte, obsérvese que diseñar modelos DEVS acoplados basándose en el patrón Composite, permite componer infinitamente modelos cada vez más grandes. Además, este tipo de estructura admite la recurrencia de operaciones, lo cual podría ser de gran utilidad.

Por ejemplo, `getCurrentState()` fue definida como una operación que devuelve un objeto ***`State`***; y en principio, esto parecería no tener sentido en un elemento que es una estructura, como lo es un modelo acoplado. No obstante, el estado de un

modelo de este tipo, en definitiva, es el estado de todos sus componentes. A su vez, el estado de sus componentes atómicos sería un elemento de la clase ***State***; pero el estado de sus componentes acoplados, sería nuevamente el estado de los modelos internos de este, y así sucesivamente hasta llegar a los extremos de la estructura.

Ante esto, podría ser necesario obtener los estados ***State*** de todos los modelos atómicos existentes, en el árbol definido por diversos modelos acoplados jerárquicamente.

Así, podría cambiarse la signatura de la operación, de modo tal que devolviese un conjunto de pares modelo-estado. Es decir, el método podría ser declarado en ***DEVSModel***, como *`getCurrentState(): [( DEVSModel, State)]`*.

La Figura 22 muestra el diagrama de clases anterior en forma resumida, con la nueva signatura de `getCurrentState()` y un pseudocódigo ilustrativo de la implementación de esta operación.

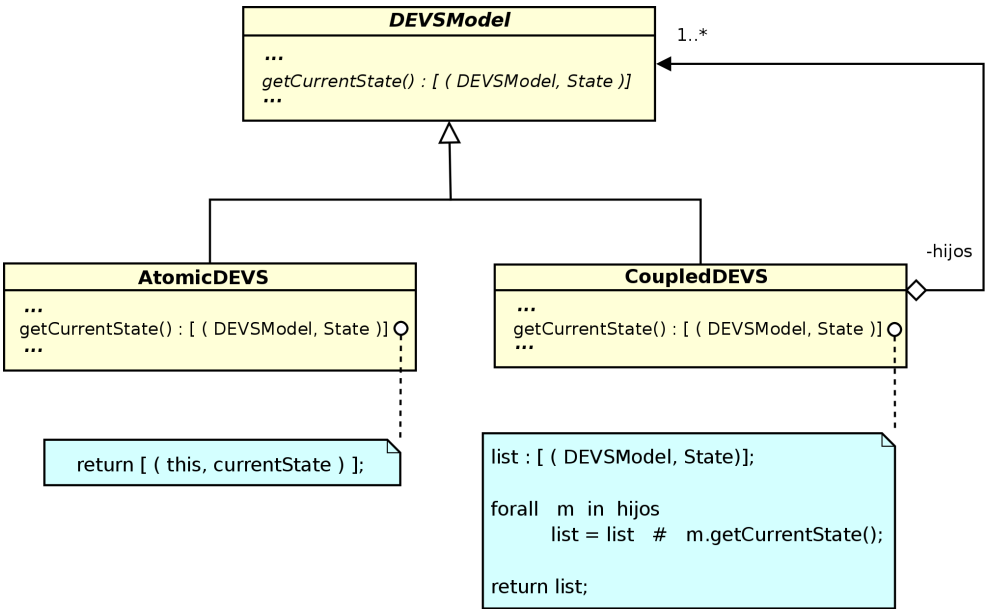


Figura 22: Recurrencia de un método

La implementación de `getCurrentState()` en **`CoupledDEVS`**, recorrería la lista de hijos invocando en cada uno `getCurrentState()`. Como cada uno de los retornos obtenidos sería un elemento de tipo **`[(DEVSModel, State)]`**; **`CoupledDEVS`** debería concatenarlos en una única lista de pares y entonces, retornarla.

`AtomicDEVS`, en la implementación de `getCurrentState()`, simplemente devolvería una lista con un único par, constituido por una referencia a sí mismo y por su estado actual. Es decir un elemento `[(this, currentState)]`.

De esta manera, la definición avanzaría en forma recurrente en la estructura jerárquica, tan profundo como fuera necesario, para alcanzar todos los modelos atómicos.

Esto permitiría, obtener el conjunto total de modelos atómicos con sus respectivos estados, independientemente de los niveles de jerarquía existentes por debajo del modelo acoplado más general.

Este tipo de recurrencia, podría considerarse adecuada en los modelos, para que estos pudieran brindar toda la información necesaria sobre su definición. La

posibilidad de implementar funciones de este tipo, está dada por lo que resulta de utilizar el patrón Composite para la definición de modelos DEVS.

PARTICIPANTES

- ***DEVSModel***
  - tiene un conjunto de puertos.
  - declara las interfaces de modelos atómicos y acoplados.
  - implementa el comportamiento común a los dos tipos de modelos. En particular, aquel vinculado a los puertos.
  - define la interfaz para acceder y gestionar los modelos hijos.
  - define la interfaz para acceder a la información de acoplamiento.
- **AtomicDEVS**
  - implementa las operaciones referentes a un modelo atómico con puertos. (Ver sección 5.1 para más detalles sobre esto.)
  - un modelo atómico no puede tener otros modelos, con lo cual, operaciones sin sentido sobre este llevarán a mecanismos de manejo de errores.
- **CoupledDEVS**
  - tiene un conjunto de modelos hijos.
  - tiene conjuntos de conexiones que representan EIC, IC y EOC.
  - implementa las operaciones de administración de modelos y conexiones.

La idea fundamental aquí es utilizar el patrón Composite para poder implementar un modelo DEVS como un objeto que puede estar compuesto por otros modelos, y que a su vez, puede ser parte de otro modelo mayor. Esto permite construir modelos DEVS agregando infinitamente niveles en la jerarquía de la construcción.

Además, se propone que los puertos y conexiones sean implementados como objetos que posee un modelo acoplado. Contar con estos elementos como objetos da mayor flexibilidad ante posibles cambios en una especificación.

COMENTARIOS

Es importante aclarar, que en el caso de tratarse de DEVS clásicos, debería estar definida la operación `select()`, que implemente la función de desempate en el acoplamiento. Como puede imaginarse, `select()` debería ser declarada en ***DEVSModel*** e implementada en **CoupledDEVS**.

Notar, que en el caso de Paralell DEVS, esta función no existe en los modelos acoplados y en cambio, existe una función confluente en los modelos atómicos. Aquí, esta función probablemente sería implementada como un objeto, de forma similar a la que fue planteada para las transiciones internas y externas, en las secciones anteriores.

6

CONCLUSIONES Y TRABAJOS FUTUROS

La implementación de modelos DEVS puede requerir modificaciones en el tiempo, producto de cambios en las especificaciones del modelo.

Dado que la dinámica de un modelo atómico depende de diferentes elementos como eventos, estados, transiciones y valores de salida; modificaciones en dicho modelo pueden implicar código extenso y confuso, y por tanto, dificultad de cambio.

Además, las características propias del formalismo llevan a la necesidad de reutilización de modelos implementados. El acoplamiento modular por su parte, hace que sea posible construir modelos a partir de modelos más pequeños. Cada uno de estos, en particular, debería poder ser parte de diferentes modelos mayores, sin tener que sufrir modificaciones. Por otro lado, un modelo acoplado podría en algún momento ser parte de la definición de otro modelo mayor. Todas estas características requieren un diseño que se adelante a los eventuales cambios, y así, contar con código fácilmente mantenible.

En este trabajo se han propuesto posibles soluciones al problema de implementar modelos DEVS que pudieran sufrir modificaciones o que se requiriese que fueran reutilizados.

En primer lugar, se ha planteado usar el patrón de diseño State como base del diseño propuesto para DEVS atómicos. Se ha sugerido desglosar los conceptos de evento, transición y estado, de modo tal, de contar con objetos bien definidos que permitan extensiones o cambios de la funcionalidad implementada. De esta manera, agregar estados y/o transiciones, o cambiar los existentes debería convertirse en una tarea relativamente sencilla.

Por otra parte, se ha propuesto basar el diseño de modelos acoplados en el patrón Composite. De este modo, los modelos pueden ser articulados para construir modelos mayores, cuantas veces sea necesario y en los niveles jerárquicos que se quiera.

Además, los puertos de modelos, tanto atómicos como acoplados, y las conexiones de los últimos fueron propuestos como objetos independientes, de tal forma, que puedan ser incorporados o eliminados de manera simple.

Definir los modelos DEVS de este modo permitiría construir modelos acoplados que soporten fácilmente cambios en su estructura interna. Esto significa que modificar conexiones entre componentes y, agregar o eliminar modelos y puertos, resultaría relativamente simple.

En términos generales, la separación de los conceptos inherentes al formalismo, en objetos de diferentes clases, implica mayor población de objetos creados y puede resultar en una implementación menos eficiente que otras. No obstante, se contaría con un código más claro que permitiese mayor mantenibilidad, lo cual fue uno de los objetivos principales de este trabajo.

Como tarea pendiente queda la definición de un caso de estudio que cuente con cierta complejidad y en el cual las especificaciones DEVS sufran distintas

modificaciones. También sería interesante considerar y estudiar las características del patrón Flyweight (o Peso Ligero) para el caso en que los modelos contasen con elementos comunes, usados reiteradas veces. Tal podría ser el caso de un mismo modelo que fuese utilizado por diversos modelos acoplados al mismo tiempo.

Otro trabajo interesante, aunque de mayor de envergadura, sería construir una herramienta la cual no requiriese conocimiento de programación alguno, y los modelos fueran construidos de forma totalmente automática a partir de una especificación, como la que podría hacerse en papel y lápiz. En este caso se prescindiría de la utilización de patrones para definir modelos dado que estos serían completamente generados de un modo automático y no en forma, total o parcialmente, manual. Como se mencionó en otras secciones, existen herramientas que acompañan el proceso de implementar modelos simulables; no obstante, en términos generales el usuario no solo debe saber modelar sino también debe conocer sobre programación. De aquí, que construir una herramienta como la mencionada sería un interesante trabajo a realizar.

BIBLIOGRAFÍA

[1] P. Adamczyk. The Anthology of the Finite State Machine Design Patterns. In *The 10th Conference on Pattern Languages of Programs*, 2003. (Citado en pág. 8.)

[2] P. Adamczyk. Selected Patterns for Implementing Finite State Machines, 2004. URL http://pinky.cs.uiuc.edu/~padamczy/xml/tableofcontents.html. (Citado en pág. 8.)

[3] L. Baati, C. Frydman, and N. Giambiasi. LSIS_DME M&S environment extended by dynamic hierarchical structure DEVS modeling approach. In *Spring Simulation Multiconference. Proceedings of the 2007 spring simulation multiconference*, volume 2, pages 227–234. SCS. ACM/SIGSIM, SCSI, 2007. (Citado en pág. 5.)

[4] S. Cheon, C. Seo, S. Park, and B. P. Zeigler. Design and Implementation of Distributed DEVS Simulation in Peer to Peer Network System. In *Proceedings of the 2004 Military, Government and Aerospace Simulation Symposium*, April 2004. (Citado en pág. 24.)

[5] A. C. H. Chow and B. P. Zeigler. Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism. In *WSC '94: Proceedings of the 26th Conference on Winter Simulation*, pages 716–722. Society for Computer Simulation International, 1994. (Citado en pág. 4.)

[6] J. O. Coplien. Software design patterns: Common questions and answers. In *Rising L., (Ed.), The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320. University Press, 1998. (Citado en pág. 7.)

[7] H. P. Dacharry and N. Giambiasi. Discrete event modeling through a multiformalism approach, from a user-oriented perspective. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, volume 2, pages 207–213. SCS, ACM/SIGSIM, Society for Computer Simulation International, March 2007. (Citado en pág. 37.)

[8] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The Department Of Defense High Level. In *WSC'97. Proceedings of the 1997 Winter*, volume 0, pages 142–149. IEEE Computer Society, Dec 1997. (Citado en pág. 5 y 18.)

[9] R. Djafarzadeh, G. Wainer, and T. Mussivand. DEVS Modeling and Simulation of the Cellular Metabolism by Mitochondria. In *SpringSim '05: Proceedings of the 2005 DEVS Integrative M&S Simposium, Spring Simulation Conference*, April 2005. URL http://www.sce.carleton.ca/faculty/wainer/papers/DEVS-Mito.pdf. (Citado en pág. 5.)

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Patrones de Diseño*. Addison-Wesley, 1th edition, 2003. (Citado en pág. 1, 7, y 31.)

[11] M. Hamri and G. Zacharewicz. LSIS_DME: An Environment for Modeling and Simulation of DEVS Specification. In *the Iinternational Modeling and Simulation Multiconference 2007 (IMSM07).*, Feb 2007. (Citado en pág. 5.)

[12] X. Hu and B. P. Zeigler. A Proposed DEVS Standard: Model and Simulator Interfaces, Simulator Protocol. draft 1.0, 2008. URL http://acims.eas.asu.edu/PUBLICATIONS/publications.shtml. (Citado en pág. 7.)

[13] D. Huang, H. Sarjoughian, W. Wang, G. Godding, D. Rivera, K. Kempf, and H. Mittelmann. SUBMITTED TO THE SPECIAL ISSUE OF IEEE TRANS. ON SEMICONDUCTOR MANUFACTURING. Simulation of Semiconductor Manufacturing Supply-Chain Systems with DEVS, MPC and KIB, 2008. (Citado en pág. 5.)

[14] K. Kim and W. Kang. CORBA-Based, Multithreaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One. In *Computational Science and Its Applications - ICCSA 2004*, volume 3046/2004, pages 167–176. Springer Berlin / Heidelberg, April 2004. (Citado en pág. 1 y 24.)

[15] K. Kim, W. Kang, B. Sagong, and H. Seo. Efficient distributed simulation of hierarchical DEVS models: transforming model structure into a nonhierarchical one. In *Simulation Symposium, 2000. (SS 2000)Proceedings 33rd Annual*, pages 227–233. IEEE, April 2000. (Citado en pág. 24.)

[16] E. Kofman. Introducción a la Modelización y Simulación de Sistemas de Eventos Discretos con el Formalismo DEVS. URL `https://www.fceia.unr.edu.ar/dsf/files/Apunte_DEVS.pdf`. 2003. (Citado en pág. 13.)

[17] Y. W. Kwon, H. C. Park, S. H. Jung, and T. G. Kim. Fuzzy-DEVS Formalism : Concepts, Realization and Applications. In *AIS'96*, pages 227–234, Mar. 1996. (Citado en pág. 4.)

[18] D. Lea. Christopher Alexander: An Introduction for Object-Oriented Designers. *SIGSOFT Softw. Eng. Notes*, 19(1):39–46, 1994. (Citado en pág. 7.)

[19] A. Muzy and J. J. Nutaro. Algorithms for efficient implementation of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling & Simulation.*, June 2005. (Citado en pág. 1 y 24.)

[20] J.Ñutaro and P. Hammonds. Combining the Model/View/Control Design Pattern with the DEVS Formalism to Achieve Rigor and Reusability in Distributed Simulation. In *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology.*, volume 1, pages 19–28, June 2004. (Citado en pág. 1.)

[21] H. S. Sarjoughian and R. K. Singh. Building Simulation Modeling Environments Using Systmes Theory and Software Architecture Principles. In *Advanced Simulation Technology Symposium. (ASTC)*, pages 99–105. SCS, April 2004. (Citado en pág. 1.)

[22] R. K. Singh, H. S. Sarjoughian, and G. W. Godding. Design of Scalable Simulation Models for Semiconductor Manufacturing Processes. In *Proceedings of the Summer Computer Simulation Conference*, volume 36, pages 235–240. SCS, 2004. (Citado en pág. 5.)

[23] H. S. Song and T. G. Kim. The DEVS Framework for Discrete Event Systems Control. In *Proceedings of the fifth Annual Conference on IA, Simulation, and Planning in High Autonomy Systems*, pages 228–294, December 1994. (Citado en pág. 5.)

[24] J. van Gurp and J. Bosch. On the Implementation of Finite State Machines. In *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta*, pages 172–178. Press, 1999. (Citado en pág. 8 y 41.)

[25] S. M. Yacoub and H. H. Ammar. A Pattern Language of Statecharts. In *Proc. Third Conference on Pattern Languages of Programming and Computing, PloP, Monticello/IL*, number WUCS-98-25, 1998. (Citado en pág. 8.)

[26] B. P. Zeigler. *Theory of Modeling and Simulation*. Wiley Interscience, 1th edition, 1976. (Citado en pág. 3.)

[27] B. P. Zeigler. Embedding DEV&DESS in DEVS. In *DEVS Integrative M&S Symposium. Part of the 2006 Spring Simulation Multiconference (SpringSim'06).* SCS, April 2006. (Citado en pág. 4.)

[28] B. P. Zeigler, H. S. Song, T. G. Kim, and H. Praehofer. DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems. In *In Proceedings of HSAC*, pages 529–551. Springer-Verlag, 1996. (Citado en pág. 4.)

[29] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuos Complex Dynamic Systems*. Academic Press, 2th edition, 2000. (Citado en pág. 4, 9, 18, y 24.)