

# Sistema para la Simulación de modelos basados en Statecharts

Diego A. Bottallo

3 de junio de 2008

## Resumen

Este documento detalla los pasos realizados durante el desarrollo de un sistema para la simulación de modelos basados en el lenguaje de especificación Statecharts, introducido por David Harel en la década de 1980. Se describen los elementos sintácticos y la gramática contemplada del lenguaje, la tecnología empleada, cómo fue diseñado y los patrones de diseño aplicados en el citado software. Se exhiben las características fundamentales que participan en el proceso de simulación y finalmente se brinda una conclusión junto a recomendaciones para extender las funcionalidades del sistema.

## Índice

<b>1. Introducción</b>	<b>4</b>
1.1. El formalismo visual Statecharts . . . . .	5
1.2. Simulación de modelos . . . . .	5
<b>2. Estado del arte</b>	<b>7</b>
<b>3. Alcance del trabajo</b>	<b>9</b>
<b>4. Elementos de Statecharts considerados</b>	<b>9</b>
4.1. Estado Básico . . . . .	10
4.2. Estado Or . . . . .	11
4.3. Estado And . . . . .	11
4.4. Estado Referencia . . . . .	12
4.5. Estado Temporizado . . . . .	13
4.6. Estado Parametrizado . . . . .	14
4.7. Las Transiciones . . . . .	17
4.8. Los Conectores . . . . .	20
4.9. Transiciones compuestas . . . . .	22
<b>5. Descripción de la solución</b>	<b>23</b>
5.1. Visión general . . . . .	23
5.2. Algoritmo del Sistema de Simulación . . . . .	24
5.3. Tecnología empleada . . . . .	24
5.3.1. Java . . . . .	24

5.3.2.	Desing Patterns . . . . .	25
5.3.3.	OpenDocument . . . . .	25
5.3.4.	Open/Star Office Draw . . . . .	27
5.3.5.	JDOM . . . . .	27
5.3.6.	Scripting . . . . .	28
<b>6.</b>	<b>Documentación</b>	<b>28</b>
6.1.	Manual de Usuario . . . . .	29
6.1.1.	Instalar el Sistema de Simulación . . . . .	29
6.1.2.	Creación del sistema/modelo . . . . .	30
6.1.3.	Gramática permitida para los modelos . . . . .	32
6.1.4.	Almacenar en disco el sistema/modelo . . . . .	37
6.1.5.	Ejecutar el Sistema de Simulación . . . . .	37
6.2.	Instrucciones para el Programador . . . . .	40
6.2.1.	Acerca de XML y Open/Star Office Draw . . . . .	41
6.2.2.	Guía de módulos . . . . .	41
6.2.3.	Detalles del Algoritmo de Simulación . . . . .	44
<b>7.</b>	<b>Caso de estudio</b>	<b>50</b>
7.1.	Ventilador de cuatro velocidades . . . . .	50
7.1.1.	Descripción . . . . .	50
7.1.2.	Modelado . . . . .	51
7.1.3.	Simulación . . . . .	53
<b>8.</b>	<b>Conclusiones y trabajos futuros</b>	<b>56</b>
8.1.	Trabajos futuros . . . . .	57

## **Agradecimientos**

A Verónica por su paciencia y amor inagotable, ella sabe mejor que nadie el tiempo dedicado y los sacrificios realizados durante toda la carrera. A mi familia que siempre estuvo y estará presente y apoyándome.

A mis compañeros de clase, por las anécdotas, trabajos prácticos y algunos nervios de exámenes compartidos.

A Maximiliano Cristiá, que me apoyó durante toda esta etapa, consultándome constantemente sobre los avances y mostrándose siempre dispuesto a reunirnos en cualquier sitio.

A todos MUCHAS GRACIAS sinceramente.

## 1. Introducción

El desarrollo de software se está tornando una actividad cada vez más compleja. Debido a la ubicuidad del mismo es frecuente encontrar software embebido en una amplísima gama de dispositivos, desde los más simples a los más elaborados, utilizados para tareas cotidianas o para tareas críticas donde la calidad, robustez y fiabilidad del código juegan un papel preponderante.

Un programa o sistema correcto parte de un diseño correcto y flexible, que no contenga fallas, permita ser reutilizado y esté adaptado para el cambio.

Los lenguajes de especificación asisten en la función de comprender los requerimientos de los sistemas que el cliente enuncia, ayudan a detectar inconsistencias y/o contradicciones en los mismos y permiten detectar estos problemas de manera temprana que, de otra forma, éstos deben resolverse en etapas posteriores, repercutiendo directamente en los plazos de entrega de proyectos y puesta en producción de los sistemas, o lo que es peor, cuando ya se los está empleando, con consecuencias de una gravedad relativa al uso que se le brinde en cada caso.

Dentro del universo de los sistemas se encuentran los *sistemas reactivos*, que a diferencia de los puramente transformacionales, mantienen una continua interacción con su entorno, respondiendo ante estímulos externos en función de su estado interno. Esta característica hace que el comportamiento de los mismos sea complejo de analizar con el agravante de que a menudo los sistemas reactivos son también sistemas críticos. Otro grupo de sistemas destacados son los que se clasifican como *sistemas de tiempo real*. Las operaciones de éstos son consideradas correctas no sólo porque la lógica e implementación de los programas computacionales sea correcta, sino también porque el tiempo en el que dichas operaciones se efectúan respeta las restricciones temporales preestablecidas.

Todo sistema que sea catalogado como sistema reactivo posee un conjunto de características comunes:

- Respuesta: utilizan entradas/salidas que son continuas o discretas en el tiempo.
- Eventos de interrupción: deben ser capaces de responder a eventos de interrupción de mayor prioridad.
- Operaciones temporizadas: las operaciones y reacciones pueden tener restricciones temporales.
- Comportamiento dependiente del *estado*: existen muchos escenarios posibles de operación dependiendo tanto del modo de operación y el estado actual de los datos como de su comportamiento pasado.
- Concurrencia: los procesos interactuantes pueden operar a menudo en paralelo.
- Generalmente poseen un número finito y relativamente pequeño de estados.

Entre los formalismos utilizados para la especificación de sistemas de tiempo real y sistemas reactivos se destacan aquellos con sostén gráfico, por ser fáciles de asimilar, comprender, utilizar y revisar. Es aquí donde el lenguaje de especificación Statecharts encuentra su campo de acción óptimo.

### 1.1. El formalismo visual Statecharts

David Harel en [HA87] y [HA88] introdujo el lenguaje Statecharts para extender las máquinas de estados finitos, *Finite State Machines* (FSM), para su empleo en escenarios complejos.

Los Statecharts extienden a las FSM, soportando una serie de conceptos fundamentales:

- Jerarquía: Este principio introduce la idea de un tipo de estado complejo denominado *super estado*, el cual incluye en su interior otros estados, permitiendo un nivel de inclusión irrestricto. Esto rompe con la limitación de FSM donde los modelos representados adolecen de estructura jerárquica.
- Ortogonalidad: Permite experimentar varios comportamientos al mismo tiempo. Es la idea de concurrencia obtenida a través de los *estados and* como veremos más adelante.
- Broadcasting: Un evento puede disparar varias transiciones en simultáneo correspondientes a statecharts ortogonales o en paralelo.
- Historia: Luego de transicionar desde un estado a otro, muchas veces es requerido retornar a un estado anterior en el cual se estuvo en lugar del estado por defecto. Para esto Statecharts ofrece elementos especiales.

La siguiente ecuación resume la idea de cómo Statecharts extiende a las FSM.

$$\boxed{\text{Statecharts} = \text{FSM} + \text{Jerarquía} + \text{Ortogonalidad} + \text{Broadcasting} + \text{Historia}}$$

Al utilizar diagramas de estados para la especificación de sistemas se accede a una serie de beneficios, uno de los principales es la *abstracción*, pues los diagramas de estados poseen una semántica simple para la representación de sistemas complejos y ofrecen una vista a nivel de *sistema*. Otros de los beneficios son el de la *auto documentación* y su natural *escalabilidad*.

Para acceder a una completa reseña histórica de este formalismo desarrollada por su mentor, referirse al paper [HA07].

### 1.2. Simulación de modelos

El Sistema desarrollado provee los medios para *simular* el comportamiento de un *sistema* a través de un *modelo* estrechamente relacionado con el mismo. Antes de entrar en detalles se explican brevemente cada uno de estos términos.

- *Sistema*: Conjunto de elementos organizados y relacionados que interactúan entre sí para lograr un objetivo. Los sistemas reciben (entradas) datos, energía o materia del ambiente y proveen (salidas) información, energía o materia. Un sistema existe y opera en tiempo y espacio.
- *Modelo*: Es una representación de un sistema en un punto de tiempo y espacio cuyo objeto es fomentar el entendimiento total o parcial del sistema real.
- *Simulación*: Es la técnica o disciplina de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias -dentro de los límites impuestos por un cierto criterio o un conjunto de ellos- para el funcionamiento del sistema.

El modelado y la simulación conforman una disciplina útil para desarrollar un alto nivel de entendimiento de la interacción de las partes de un sistema y del sistema como un todo. Es una práctica aceptada tanto en el ambiente académico como industrial. El grado de comprensión de los sistemas conseguido a través de este medio rara vez es alcanzado vía otras disciplinas. Obviamente interactuar con un modelo ofrece la posibilidad de repetir los experimentos o pruebas, puesto que un modelo tiende a ser considerablemente más económico y fácil de construir que el sistema original.

La validación y verificación (V&V) son protagonistas principales en el desarrollo de software correcto, definamos sucintamente estos conceptos:

- *Verificación*: Trata de construir el *sistema correctamente*. La verificación es la tarea de comprobar matemáticamente la corrección de un programa con respecto a su especificación.
- *Validación*: Trata de construir el *sistema correcto*. Constata con el cliente las características funcionales y no funcionales del sistema.

Entre las herramientas destacadas que participan durante el proceso de V&V se encuentran los simuladores y los generadores de casos de pruebas, éstas son principalmente aplicadas en la etapa de *testing* del mencionado proceso.

Mediante la simulación de modelos Statecharts es posible verificar las propiedades de *safety* y *liveness* de los sistemas, más frecuentemente en los sistemas reactivos. De manera informal, una propiedad de *safety* estipula que "algo malo" no ocurrirá durante la ejecución de un programa, mientras que una propiedad de *liveness* determina que "algo bueno" sucederá (eventualmente). Según [LL02] las propiedades de *liveness* son consideradas menos importantes que las propiedades de *safety*. Por ejemplo, si se modela cierto proceso químico que tiene lugar en un reactor nuclear, mediante el modelado y simulación es posible estudiar que la ocurrencia de un determinado suceso imprevisto no genere consecuencias graves en dicho ámbito.

Con el lenguaje Statecharts se pueden modelar diversos sistemas, comúnmente sistemas reactivos o de tiempo real. Al realizar simulaciones sobre estos modelos es posible constatar que nunca se alcanzarán estados o

situaciones indeseadas. Enviando eventos (estímulos) al modelo durante la simulación, es posible interpolar el comportamiento del sistema original y realizar análisis sobre los resultados observados a fin de aplicar correcciones (conceptuales o funcionales) al sistema original y hacerlo más robusto.

## 2. Estado del arte

En esta sección se hace un repaso sobre la existencia de software similar al aquí implementado.

Naturalmente como herramienta más reconocida surge el nombre de **STATEMATE**, desarrollada en 1986 por I-Logix Israel Ltd. La decisión de David Harel y sus colegas fue la de desarrollar una aplicación que permita el diseño y simulación de modelos Statecharts dentro de la órbita de una Empresa debido a la alta probabilidad de aplicación del lenguaje Statecharts al ambiente comercial. De haberlo hecho solo con fines académicos corrían el riesgo de ser únicamente los impulsores teóricos perdiendo posiblemente el control de la evolución del lenguaje.

STATEMATE permitía la edición de modelos Statecharts con un editor propio, la simulación de éstos y la generación de código ejecutable en Ada y C, además de documentación de diseño útiles para los ingenieros participantes en los proyectos. Para más detalles sobre todas las características funcionales y el modo de operación de este producto referirse a [STMPB].

Siempre fue una herramienta comercial y hoy es distribuida por Telelogic (en marzo del 2006 I-Logix fue adquirida por Telelogic, una compañía de IBM, por 80 millones de dólares, [ILTL]) bajo el nombre de *Telelogic Statemate*, la cual usa como lenguaje base UML (Unified Modeling Language, [UML]). UML incluye una variante de Statecharts con el nombre de *State Machines*, que no posee toda la potencia del lenguaje original. Se ofrece una descripción finamente detallada sobre STATEMATE en [HP99].

En [HA07] se menciona a **RoseRT** (IBM Rational Rose RealTime), conocida anteriormente como **ObjectTime**, el cual es un software basado en UML desarrollado por Rational. Como comentario particular, esta herramienta no soporta la concurrencia propia de Statecharts y el código generado (soporta Ada, C, C++ y Java) no es lo suficientemente claro. Se ejecuta sobre Windows y UNIX. Se trata de una herramienta comercial que permite la construcción de modelos ejecutables diseñados en UML que se pueden compilar y simular. Rational Rose RealTime cubre el ciclo de vida completo de un proyecto, desde el análisis primario de casos de uso, siguiendo con el diseño, implementación y testing. Para más detalles ingresar a la web de la cita [RRRT].

Otra herramienta que cabe mencionar es la denominada **EStudio**, herramienta gráfica para desarrollar programas *Esterel*. Esterel es tanto un lenguaje de programación (dedicado a la programación de sistemas reactivos) como un compilador que traduce programas Esterel a FSM. Si bien sólo posee,

en relación a Statecharts, aquello que es también contemplado por FSM, es una herramienta con orígenes similares a la que aquí se desarrolló. Mientras **EStudio** es *sólo* una herramienta de programación, RoseRT tiene como objetivo desarrollar, documentar y presentar proyectos completos.

Existe otra herramienta comercial llamada **IAR visualSTATE**, distribuida por IAR Systems, que posee características gráficas para el diseño, testing e implementación de aplicaciones embebidas basadas en *state machines*. Está basada en el subconjunto *UML State Machine*. Se ejecuta sólo sobre Windows. Cuenta con un diseñador gráfico, herramientas de test, un generador de código C/C++ y utilidades para documentación. Pueden verse detalles en el sitio [VST].

**Rhapsody**, es una herramienta comercial orientada al desarrollo de sistemas embebidos y de tiempo real. El entorno de modelado de Rhapsody está basado en UML, incrementado con perfiles de dominio para sistemas embebidos, tales como SysML. Automatiza por completo el proceso de diseño de sistemas embebidos, lo cual incluye tanto los sistemas como las disciplinas de ingeniería de software. Compila modelos gráficos en aplicaciones de software en varios lenguajes, incluyendo C, C++, Ada y Java. Esta herramienta también fue desarrollada por I-Logix, perteneciendo los derechos actualmente a Telelogic. Más información en [RHPS].

**LabVIEW**, de National Instruments, es un software comercial que posee un módulo dedicado a Statecharts (LabVIEW Statechart Module). Permite la edición de modelos Statecharts y ejecutarlos de manera interactiva. Genera código en C y en dos lenguajes propietarios denominados *Integer LabVIEW code* y *LabVIEW FPGA code*. Puede accederse a más información relacionada con este software en [LBVW].

Por el lado de los editores de Statecharts, todas las herramientas mencionadas poseen uno incorporado, pues poseen editores de UML originalmente. Existe un proyecto del año 1995 donde se desarrolló un editor interactivo de Statecharts, [EDIT].

No se ha encontrado mucho en el ambiente GNU, aunque sí existe un trabajo destacable que permite la construcción de especificaciones Statecharts para modelar y controlar sistemas reactivos. Extiende los lenguajes C++ y Java empleando lex y yacc para facilitar la implementación de sistemas reactivos en estos lenguajes. Posee licencia GPL, aunque también admite licenciamiento comercial. Más detalles en [CHSM].

Como conclusión de este resumen, es posible afirmar que muchas herramientas son comerciales y de código cerrado y que trabajan con una porción de Statecharts, empleando directamente UML con ciertas características adicionales.

### 3. Alcance del trabajo

En el trabajo aquí expuesto se contempló un subconjunto de los elementos que constituyen el lenguaje Statecharts, si bien algunos fueron dejados para etapas de desarrollo posteriores, se incluyeron los más importantes. Estos elementos son suficientes para la confección de una amplia gama de modelos útiles para la incorporación de los conceptos básicos del modelado de sistemas reactivos y de tiempo real.

No se desarrolló un editor para modelos Statecharts integrado al Sistema, esta tarea si bien no es extremadamente compleja no es trivial. En este caso, para la confección de los modelos, se propone el uso de una herramienta Open/Star Office Draw, de simple instalación, disponible para los sistemas operativos más utilizados y cuyos archivos de salida respetan el formato estándar OpenDocument, admitiendo así un alto grado de interoperabilidad con otros sistemas. Más adelante en el documento se brinda información detallada sobre este punto.

La metodología de simulación que propone el Sistema es la de simulación dirigida o asistida por el usuario, lo que significa que el usuario es el que realizará las tareas de carga y configuración de parámetros y eventos previo al proceso de simulación, el cual una vez lanzado actuará como un proceso *batch*, entregando información una vez finalizado.

Existen otros métodos de simulación (aún no compatibles con el Sistema), por ejemplo la simulación aleatoria, donde se permite la generación automática de eventos de entrada al azar o cambios repentinos del entorno en el cual el modelo se desenvuelve para analizar la respuesta a estos estímulos.

El algoritmo de simulación implementado en el Sistema es una variante del que Harel presentó en el texto [HN96].

El espíritu del trabajo es el de brindar a los alumnos y al cuerpo docente de la materia Análisis de Sistemas (DCC - FCEIA - UNR) una herramienta útil y simple de operar, con la finalidad de acompañar el dictado de las clases teóricas y prácticas sobre la especificación de sistemas basada en Statecharts.

### 4. Elementos de Statecharts considerados

El lenguaje Statecharts posee una rica variedad de elementos gráficos, cada uno de ellos con su propia semántica. Este conjunto de elementos y la manera en que se relacionan, conforman el "lenguaje".

Los elementos más importantes pueden clasificarse en:

- *Estados*
- *Transiciones*
- *Conectores*

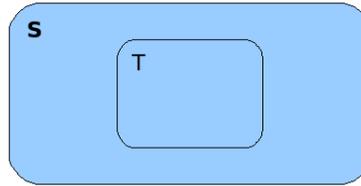


Figura 1: Estado y subestado

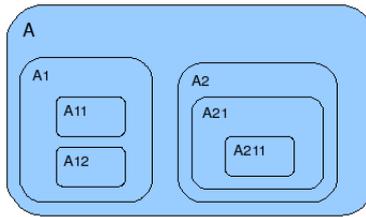


Figura 2: Jerarquía de estados

De esta lista, los *Estados* se destacan por su diversidad y rol protagónico, puesto que conforman los distintos estadios en los cuales un sistema se encuentra en cada momento.

Un sistema, al recibir eventos externos, se encontrará siempre en lo que llamamos un *status*, compuesto por uno o más estados, los cuales a su vez son denominados *estados activos*.

En el desarrollo del Sistema de Simulación se ha considerado un subconjunto de elementos del lenguaje, la mayor parte por cierto, la cual se describe en las próximas secciones.

#### 4.1. Estado Básico

Los estados pueden contener uno o más estados en su interior y dichas inclusiones pueden realizarse con un nivel de anidamiento ilimitado. Estas dos características en conjunto definen lo que se denomina una *jerarquía de estados*.

Si un estado S contiene o incluye a un estado T, se dice que S es el *estado padre* de T, que S es *superestado* de T o, recíprocamente, que T es un *subestado* de S, obsérvese la Figura 1. Como ejemplo de jearquía de estados puede apreciarse la Figura 2, donde el estado A contiene los estados A1 y A2, estos últimos por su parte contienen otros estados.

Un estado que no contiene a ningún otro estado recibe el nombre de *estado básico* o *estado hoja*. Existe además un estado especial, que posee una característica única, es el denominado *estado raíz*, el cual no es *subestado* de ningún otro estado en el sistema o modelo. En cada sistema modelado existe uno y sólo

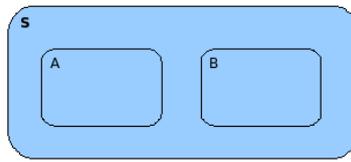


Figura 3: Estado Or

un *estado raíz*. Por ejemplo, en la Figura 2, los estados  $A11$ ,  $A12$  y  $A211$  son estados básicos, mientras que el estado  $A$  es *el* estado raíz.

## 4.2. Estado Or

Además de que un estado pueda contener a uno o más subestados, también puede definir de qué manera dichos subestados se relacionan. Un *estado or* es aquel en que sus subestados se encuentran ligados mediante una *disyunción exclusiva*, provocando que uno y sólo uno de ellos pueda estar *activo* en cada momento.

Estos estados son empleados tan habitualmente que por defecto un conjunto de subestados  $s_1, s_2, \dots, s_n$ , de un estado  $S$ , se encuentran relacionados mediante *o – exclusivo* por el sólo hecho de pertenecer a éste. En la Figura 3 se observa al *estado or*  $S$  que contiene a los subestados  $A$  y  $B$ ; en la Figura 2  $A1$  y  $A2$  son *subestados or* de  $A$ .

**Ejemplo** Un ejemplo típico del uso de estos estados es la representación de sistemas que sólo pueden encontrarse en *una* situación determinada, como encendido/apagado, arriba/abajo, verde/amarillo/rojo, etc.

## 4.3. Estado And

Los subestados de un estado también pueden relacionarse mediante una conjunción, permitiendo así la introducción de concurrencia en los sistemas. Los subestados en una relación de conjunción pueden estar activos simultáneamente y reciben el nombre de *estados and*.

En [HN96] los *estados and* se grafican como se presenta en la Figura 4, aunque en este trabajo, en aras de facilitar la edición de los sistemas, se utilizó una gráfica alternativa presentada en la Figura 5.

Gráficamente, el estado que recibirá el mote de *estado and* es aquel que posee todos sus subestados con borde no continuo, sin importar si estos bordes son idénticos, basta que no sean continuos.

Cabe aclarar que no se permiten estados cuyos subestados indiquen relación *or* y *and* al mismo tiempo, como se muestra en al Figura 6. En este caso  $A$  indica que es un subestado de  $S$  que estará relacionado mediante disyunción

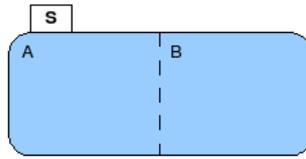


Figura 4: Estado And según [Har96]

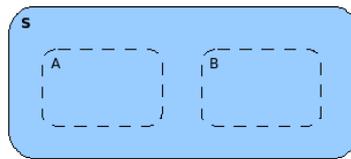


Figura 5: Estado And en el Sistema de Simulación

con los restantes, mientras  $B$  sostiene que dicha relación será una conjunción, impidiendo determinar fehacientemente que *tipo* de estado es en realidad  $S$ .

**Ejemplo** Una situación habitual donde este tipo de estados se emplea es en la descripción de modelos donde la permanencia simultánea en dos o más estados es necesaria; típicamente en todo aquel modelo total o parcialmente concurrente.

#### 4.4. Estado Referencia

Este tipo de estado no se incluye en la definición original de Harel, es un elemento introducido en este trabajo con la finalidad de simplificar la representación gráfica de los sistemas. Sólo es un estado que actúa como señalador o *puntero* a otro estado a fin de evitar un anidamiento excesivo y poder graficar sus subestados y demás componentes en un espacio de mayor amplitud. Esto brinda además el beneficio de poder *modularizar* los sistemas graficándolos con una mayor flexibilidad y así ofrecer una representación mucho más legible.

En la Figura 7 puede apreciarse que  $S$  contiene el estado  $A^*$ , este subestado es en realidad un *placeholder* del estado  $A$ , que al tener a su vez una estructura

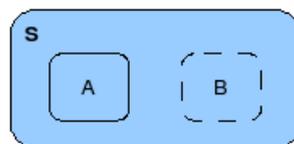


Figura 6: Estado mixto inválido

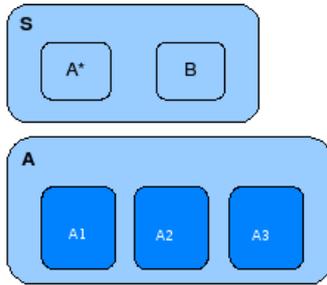


Figura 7: Estado Referencia

propia se pretende graficar en un espacio de mayores dimensiones.

Para el Sistema de Simulación,  $\mathbf{A}$  es un subestado de  $\mathbf{S}$ , internamente  $\mathbf{A}^*$  no es considerado ya que toda interacción de  $\mathbf{A}^*$  con el resto de los elementos en el modelo se interpretará como si el estado  $\mathbf{A}$  es el que interviene.

Un estado con nombre  $\mathbf{A}^*$  es denominado *estado referencia*, mientras que el estado  $\mathbf{A}$  se denomina *estado real* o *estado referenciado*. Gráficamente, las únicas características que deben compartir son el nombre (sin el asterisco por supuesto) y el tipo de borde (continuo o discontinuo).

#### 4.5. Estado Temporizado

Es un estado que posee cotas de tiempo que regulan la permanencia del sistema en el mismo. Pueden especificarse dos cotas temporales distintas y complementarias en estos estados:

- **Cota inferior**, representada por " $t <$ ", indica que sólo es posible abandonar este estado *luego* de transcurridas  $t$  unidades de tiempo. Toda posibilidad de transicionar desde este estado durante el lapso de tiempo  $[0..t)$  se verá impedida.
- **Cota superior**, indicada por " $< T$ ", señala que *inmediatamente luego* de transcurridas  $T$  unidades de tiempo, estando activo el estado, debe abandonarse tomando como transición de salida aquella etiquetada con el texto *timeout*.

Estas cotas de tiempo *deben* ser enteros positivos que cumplan la relación  $t < T$ . Además debe existir *una y sólo una* transición etiquetada con *timeout*.

Observe la Figura 8, la misma posee la estructura típica de uno de estos estados.

Al nominar los estados temporizados debe respetarse la siguiente sintaxis:

$$\mathbf{Nombre\_estado} [cota\_inferior, cota\_superior]$$

donde las cotas pueden expresarse de la siguiente manera ( $a$  y  $b$  son enteros positivos tales que  $a < b$ ):



Figura 8: Estado Temporizado

- [ $< a$ ] o [ $< a, ]$ : sólo cota inferior
- [ $, < b$ ]: sólo cota superior
- [ $a <, < b$ ]: ambas cotas
- Debe definirse al menos una cota
- Los corchetes y la coma son de uso obligatorio

**Ejemplo** Obviamente estos estados se usan para expresar restricciones temporales, las cuales pueden ser de naturaleza interna o ajena al modelo.

#### 4.6. Estado Parametrizado

Sólo consiste en una manera compacta de representar FSM complejas con un número elevado o parametrizado de estados que comparten características comunes, como ser las transiciones entrantes y salientes. No posee una semántica específica, sino que debe analizarse basándose en la semántica de las FSM que pretende representar. Con esto, todo estado parametrizado puede, y debe para ser correcto, ser traducido a un Statecharts no parametrizado mediante reglas de aplicación automática. Si se desean explicitar una o más de las FSM que este tipo de estado representa, las mismas deben graficarse dentro de los límites del estado.

Los nombres de los estados parametrizados poseen una sintaxis particular y obligatoria. El nombre consiste en un nombre común, un parámetro o variable y un rango numérico determinado a partir del cual el parámetro o variable toma valores. Cada uno de estos distintos valores representan un estado (instancia) que comparte iguales características con los estados ligados con el resto de los valores posibles. La estructura de los nombres de los estados parametrizados es la siguiente:

$$\boxed{\text{Nombre}[\text{variable}=\text{valor\_inicial}..\text{valor\_final}]}$$

donde:

- **Nombre** Texto libre válido como nombre de estado.
- **variable** Texto libre.
- *valor\_inicial* y *valor\_final* deben ser constantes enteras no negativas, donde vale la relación  $\text{valor\_inicial} < \text{valor\_final}$ .

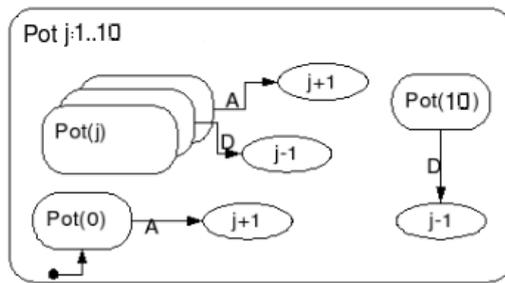


Figura 9: Estado parametrizado

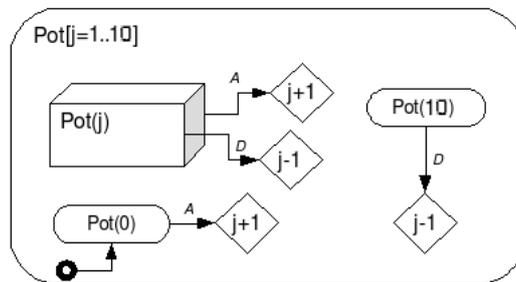


Figura 10: Estado parametrizado contemplado en el desarrollo

- Los corchetes y símbolos "=" y ".." son de uso obligatorio.

En la Figura 9 se muestra un ejemplo de un estado parametrizado según la sintaxis introducida en [CM06].

En la Figura 10 se presentan las modificaciones introducidas de acuerdo a las posibilidades que brinda la herramienta de edición de modelos. Esta representación es un poco más cercana a la que expone Harel en [HA87].

Existen dos estados particulares en Statecharts que sólo pueden estar presentes dentro de estados parametrizados, ningún otro elemento gráfico puede contener dichos estados descriptos a continuación.

El primero de estos estados es el denominado *estado plantilla* o *estado plantilla* que tiene forma de cubo. Debe existir uno y sólo uno de estos por cada estado parametrizado y simboliza a todos los estados que el estado parametrizado pretende representar de manera uniforme. Por ejemplo, un estado parametrizado con nombre  $S[i=1..5]$ , debe contener un único *estado plantilla* con nombre  $S(i)$ , el cual está representando a los estados  $S(1)$ ,  $S(2)$ ,  $S(3)$ ,  $S(4)$  y  $S(5)$ .

El otro tipo de estado es el *estado instancia*, con forma de diamante o rombo. Pueden existir cero o varios de estos estados; aceptan diferentes formatos de nombre y permiten hacer referencia a los estados  $S(i)$ , donde  $i$  es

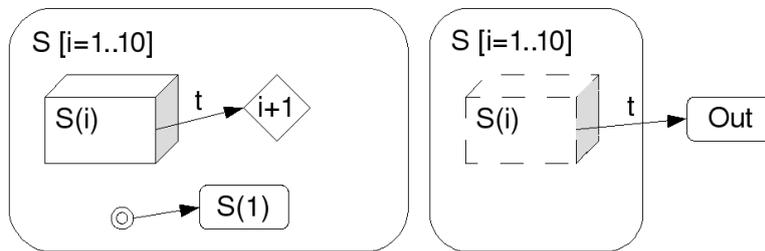


Figura 11: Izq: Estado *parametrizado or*. Der: Estado *parametrizado and*.

la variable o parámetro explicitada por el estado parametrizado. Por ejemplo, siendo  $S[i=1..5]$  un estado parametrizado, el *estado instancia* con nombre 3 representa al estado  $S(3)$ , mientras que otro con nombre  $i$  representa a cada uno de los estados  $S(i)$  asociados a los valores del parámetro  $i$ . Se brindan más detalles en la próxima sección.

Los estados parametrizados también soportan versiones *or* y *and*. Cuando sus estados internos (*template*, *instancia* y estados comunes) se grafican con líneas continuas se trata de un estado *parametrizado or*, mientras que si los estados internos tienen contorno discontinuo recibe la clasificación de estado *parametrizado and*.

La Figura 11 muestra las dos versiones; observar que lo que varía de estilo es el borde del *estado template*. Como aclaración, en el caso de los estados *parametrizados and* no es correcto especificar la *transición por defecto* ya que en definitiva son *estados and* y esta situación va contra la gramática de Statecharts.

### Estados Instancia

Como se mencionó previamente dentro de los estados parametrizados pueden presentarse elementos gráficos con forma de diamante o rombo, éstos son estados que se utilizan para representar a los llamados *estados instancia*. Admiten tres maneras distintas para su nominación:

- Con una variable o parámetro, la misma del nombre del estado parametrizado. Por ejemplo, en el estado parametrizado **Estado** $[i=a..b]$  un *estado instancia* con nombre " $i$ " representa a los estados **Estado**( $a$ ), **Estado**( $a + 1$ ), ..., **Estado**( $b$ ).
- Con una constante numérica comprendida en el rango [*valor inicial*, *valor final*]. Si el estado parametrizado es **Estado** $[i=1..5]$ , un *estado instancia* con nombre "**3**" representa al estado **Estado**(**3**).
- Con una de las siguientes operaciones matemáticas:
  - constante  $\oplus$  variable
  - variable  $\oplus$  constante

donde el operador  $\oplus$  es uno de los operadores  $+$ ,  $-$  o  $\%$  ( $\%$  es el operador módulo). Un *estado instancia* con una operación se traduce en definitiva a

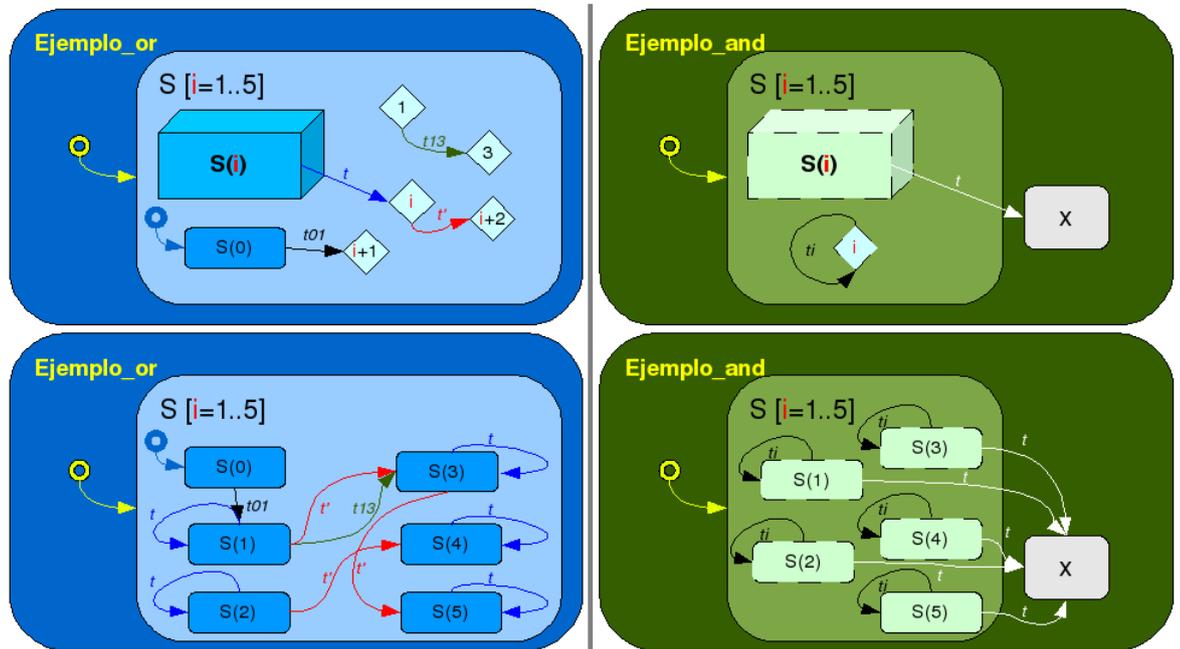


Figura 12: Estados parametrizados y sus equivalentes representados

un estado con nombre **Estado( $x$ )** donde  $x$  es una constante (el resultado de la operación calculada).

La Figura 12 muestra dos estados parametrizados, y debajo el estado no parametrizado equivalente correspondiente a cada ejemplo. Cabe aclarar que en las representaciones inferiores el nombre "S[i=1..5]" es tomado como un literal simple.

**Ejemplo** Puede aplicar a todo ejemplo donde exista un número de estados, generalmente mayor a dos, que respondan a un patrón de forma o comportamiento común.

#### 4.7. Las Transiciones

Las transiciones se emplean para vincular estados con estados o estados con conectores y poseen tres características distintivas:

- **Origen:** el cual puede ser un estado o conector.
- **Destino:** el cual también puede ser un estado o conector. En el caso de estados, el destino puede ser exactamente igual al origen, este tipo de transición recibe el nombre de *bucle* o *lazo*.
- **Etiqueta:** eventualmente nula, compuesta por eventos, condiciones y acciones.



Figura 13: Transición por defecto

Además, cabe mencionar que las transiciones son unidireccionales y siempre deben poseer origen y destino sin excepción. Dos transiciones idénticamente etiquetadas deben diferir en origen y/o destino.

Transición por defecto: En todo *estado or* debe existir una manera de indicar cual será el estado activo entre los subestados que contiene en caso de que éste se active. Para ello se emplea una transición especial, denominada *transición por defecto*, la cual se diferencia del resto por poseer como **origen** un *conector especial*, como **destino** el subestado que debe activarse al acceder al estado que lo contiene y, finalmente, no poseer etiqueta alguna (de tenerla la misma será ignorada).

El mencionado conector especial de Open/Star Office es el *Círculo*, el cual asemeja a un anillo. En la mayoría de las notas bibliográficas se emplea en su lugar el símbolo "●" como origen de la transición por defecto, aunque si bien esto puede emularse con Open/Star Office Draw cambiando el extremo de origen de la transición directamente, se decidió emplear un conector dedicado a fin de evitar confundir una transición por defecto con una transición donde se ha olvidado el origen. La Figura 13 muestra una transición por defecto con el símbolo especial que indica su origen.

Transición común o típica: Este tipo de transición, a diferencia de la transición por defecto, considera el etiquetado (aunque también puede ser nulo). El formato general para etiquetar estas transiciones responde al siguiente patrón:

$\boxed{\text{evento [ condición ] / acción}}$

Cada uno de estos componentes pueden o no estar presentes y se describen a continuación:

- **evento:** son las "señales" recibidas desde el exterior del sistema modelado.
- **condición:** se restringe a las constantes *true* y *false* y a los predicados *in(Estado)*, *notin(Estado)*; en todo caso no se realizan distinciones entre mayúsculas y minúsculas. Se admiten los operadores lógicos **and (&)** y **not (!)** y el empleo de paréntesis para forzar el orden de la evaluación de las expresiones. Por ejemplo  $(in(S) \ \& \ IN(T)) \ \& \ notIn(U)$  es una condición válida. Por convención, los estados empleados en los predicados *in* y *notin* en las transiciones de cierto estado **S** deben corresponder a subestados de un estado ortogonal a **S**.
- **acción:** puede estar conformada por una o más acciones separadas por punto y coma (;). El desarrollo actual restringe las acciones a eventos.

El formato de etiqueta para las transiciones que aquí se expuso es tomado directamente de lo descrito en [HA87].

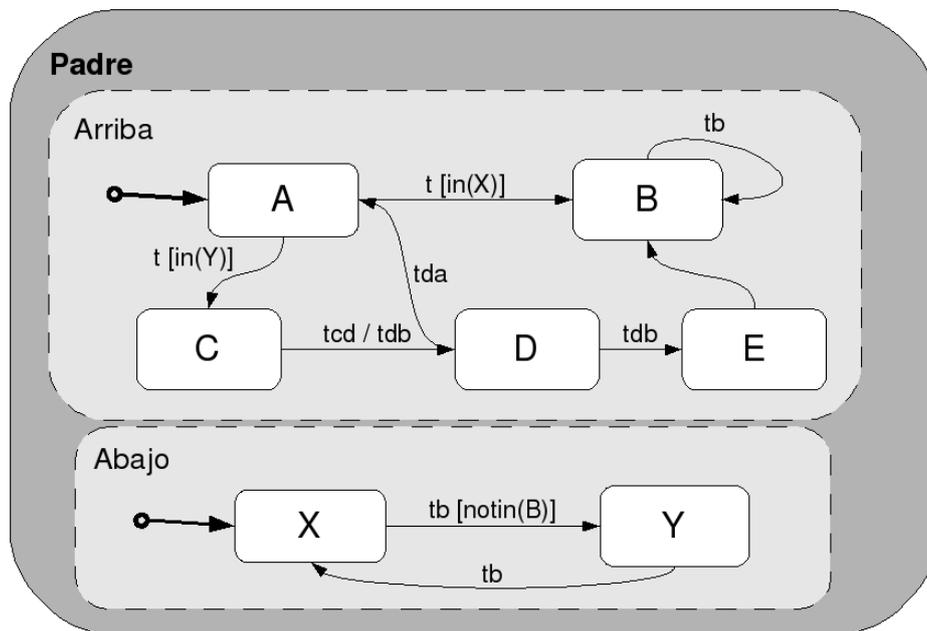


Figura 14: Transiciones

Respecto al carácter opcional de cada uno de los componentes de las etiquetas en las transiciones, se aceptan únicamente las siguientes combinaciones y restricciones:

- **Etiqueta vacía:** No deben incluirse ningún tipo de caracter
- **Etiquetado completo:** "evento [ condición ] / acción"
- **Evento vacío:** "[ condición ] / acción", o bien " / acción"
- **Condición vacía:** "evento", o bien "evento / acción"
- **Acción vacía:** "evento", "[condición]", o bien "evento [condición]"
- Los corchetes son de uso obligatorio al momento de explicitar *condición*
- La barra inclinada debe estar presente cada vez que exista *acción*

En la Figura 14 puede verse que el estado **Padre** posee los subestados **Arriba** (con los subestados **A**, **B**, **C**, **D** y **E**), y **Abajo** (con los subestados **X** e **Y**). Debido al borde discontinuo de los estados **Arriba** y **Abajo** puede determinarse que **Padre** es un *estado and*. Las transiciones por defecto son las graficadas en negro con mayor grosor (observar nuevamente el conector especial en forma de anillo que señala el origen de dicha transición). Existe un *bucle* en el estado **B** y una transición sin etiqueta (vacía) que vincula los estados **E** y **B**. El resto de las transiciones muestran algunas combinaciones de evento, condición y acción. Observar que desde **A**, al ocurrir el evento  $t$ , debido a la condición de cada una de las transiciones salientes, sólo una estará habilitada

en cada momento puesto que es imposible que los estados  $X$  e  $Y$  estén activos al mismo tiempo.

## 4.8. Los Conectores

Los conectores son elementos especiales en el lenguaje Statecharts y permiten la composición de transiciones de varias maneras.

Los conectores (de Statecharts, no confundir con los conectores propios de Open/Star Office Draw los cuales se emplean para graficar transiciones) sirven como método primario para la simplificación de los gráficos reduciendo en número y longitud las transiciones, al empalmar o interconectar a las mismas. No son elementos esenciales de Statecharts, puesto que su semántica puede lograrse haciendo un uso más complejo de las transiciones, bien en cantidad o etiquetado.

Los conectores se clasifican en dos grupos, conectores AND y conectores OR. En el primer grupo todos los segmentos de transiciones empalmados continúan habilitados, es decir, que en presencia de un conector al cual ingresan  $n$  segmentos de transición  $e_1, \dots, e_n$  y salen  $m$  segmentos  $s_1, \dots, s_m$ , existirán  $n \times m$  transiciones de la forma  $e_i + s_j$ , con  $i=1..n$ ,  $j=1..m$ , es decir, cada  $e_i$  se acopla con cada uno de los  $s_j$ .

En cambio, en los conectores OR, dados los segmentos de transición entrantes  $e_1, \dots, e_n$  y los segmentos de transición salientes,  $s_1, \dots, s_m$ , se dará lugar a sólo  $n$  transiciones de la forma  $e_i + s_j$ , con  $i=1..n$ ,  $j=1..m$ , es decir, cada  $e_i$  se une con un único  $s_j$ .

En el desarrollo actual se contemplan cuatro tipos de conectores.

### Conector *Condition*

Conector perteneciente a la clase OR. Está concebido para soportar la ramificación (*branch*) basada en condiciones.

Sintácticamente admite una o más transiciones de entrada y toda salida deberá contener obligatoriamente una condición y, eventualmente, una acción, pero *nunca* un evento.

Semánticamente cada condición deberá ser válida de manera exclusiva. Asegurarse de que esta restricción se cumpla es responsabilidad absoluta de quien construye el modelo.

En la Figura 15 se pueden apreciar las restricciones recientemente mencionadas.  $e$  representa un evento, mientras que  $ci$  y  $ai$  condiciones y acciones respectivamente. Recordar que las condiciones deben ser de tal manera que sólo una evalúe a verdadero en cada momento, de otra forma se obtendrá un error de *duplicación de transiciones* o de *no determinismo*.

La existencia de conectores *condition* en un modelo, como se mencionara, es opcional, la Figura 16 muestra cómo en ausencia de uno de estos conectores se

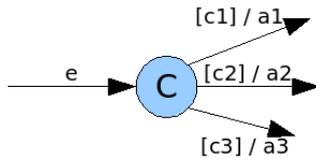


Figura 15: Conector *condición*

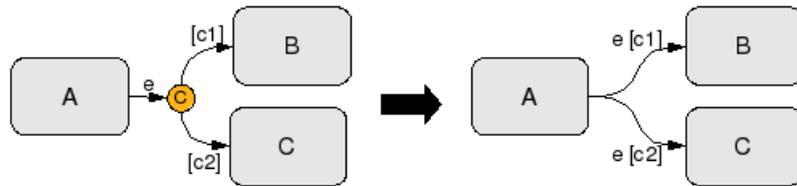


Figura 16: Ejemplo uso de Conector *condición*

puede manifestar lo mismo utilizando transiciones más expresivas.

En Statecharts este tipo de conector se representa mediante un círculo con una letra *C*.

#### Conector *Selection*

Forma parte de la clase OR y brinda la posibilidad de ramificación a nivel eventos. Este conector recibe como entrada una o más transiciones *sin* etiquetar, y cada una de las transiciones salientes pueden ser etiquetadas como cualquier transición típica entre estados, donde cada uno de los componentes, evento, condición y acción, son opcionales. La Figura 17 muestra uno de estos conectores con sus transiciones.

Para introducir uno de estos conectores a un modelo Statecharts debe graficarse un círculo con una letra *S*.

#### Conectores *Fork* y *Joint*

Pertenecen a la clase AND. Se emplean principalmente para reducir la longitud de las transiciones. Devienen de un elemento denominado *Junction* descrito

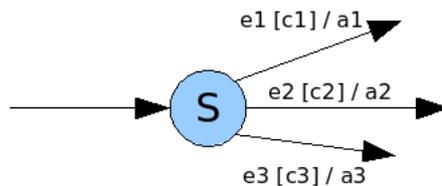


Figura 17: Conector *Selection*

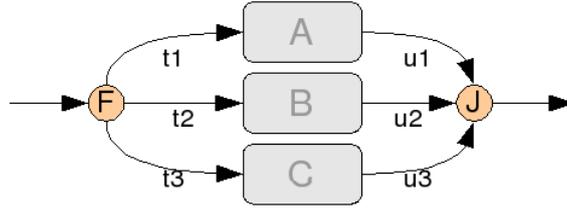


Figura 18: Conectores *Fork* y *Joint*

en [HN96].

En el caso de *Fork* toda transición de entrada se acopla con cada una de las transiciones de salida, generalmente el número de transiciones entrantes es *menor* al de transiciones salientes. En cuanto al conector *Joint* el comportamiento es similar pero el número de transiciones entrantes es *mayor* al número de transiciones salientes. En la Figura 18 se muestran estos conectores.

En Statecharts el conector *Fork* es representado mediante un círculo con una letra *F*, mientras que el conector *Joint* con un círculo con una *J*.

#### 4.9. Transiciones compuestas

Las transiciones compuestas son aquellas transiciones "lógicas", es decir no graficadas, compuestas por dos o más transiciones relacionadas por conectores.

Independientemente del tipo de conector que se emplee, dos transiciones empalmadas por un conector pueden reemplazarse por una única transición cuyo etiquetado proviene de componer las etiquetas de dichas transiciones.

Sean

- $X$  un conector cualquiera
- " $t_{in}: e_1 [c_1] / a_1$ " una transición entrante a  $X$
- " $t_{out}: e_2 [c_2] / a_2$ " una transición saliente de  $X$

donde  $e_i$ ,  $c_i$  y  $a_i$  son eventos, condiciones y acciones respectivamente.

El conector puede ser retirado del modelo y reemplazado por la transición  $t$  cuya etiqueta es " $T[c_1 \& c_2] / a_1; a_2$ ", con  $T=e_1$  o  $T=e_2$ . Los siguientes cuatro casos cubren todas las posibilidades respecto al valor de  $T$ :

- Si  $e_1$  y  $e_2$  son vacíos,  $t: [c_1 \& c_2] / a_1; a_2$ .
- Si  $e_1$  no es vacío y  $e_2$  es vacío,  $t: e_1 [c_1 \& c_2] / a_1; a_2$ .
- Si  $e_1$  es vacío y  $e_2$  no es vacío,  $t: e_2 [c_1 \& c_2] / a_1; a_2$ .
- Si ambos  $e_1$  y  $e_2$  no son vacíos se crean dos transiciones:  
 $t_1: e_1 [c_1 \& c_2] / a_1; a_2$  y  $t_2: e_2 [c_1 \& c_2] / a_1; a_2$ .

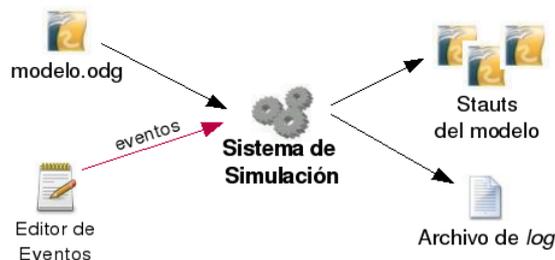


Figura 19: Sistema de Simulación Statecharts

## 5. Descripción de la solución

Esta sección está dedica a brindar una descripción breve y general de lo que realiza el Sistema de Simulación.

### 5.1. Visión general

El Sistema consta de un diseño orientado a objetos, los módulos se describen en una sección posterior y se brinda documentación adjunta para estudiar las interfaces de cada uno de éstos.

Como muestra la Figura 19, el Sistema toma como dato de entrada información contenida en archivos XML en formato OpenDocument generados con las herramientas Open/Star Office Draw. Los modelos se representan mediante diagramas utilizando un subconjunto de los elementos gráficos disponibles (formas básicas) en estas herramientas. Los archivos mencionados poseen extensión **.odg** la cual proviene del nombre *OpenDocument Graph*.

Luego el Sistema traduce esta información a una estructura interna de objetos y analiza si el diagrama descrito corresponde a la gramática Statecharts permitida, determina la validez de elementos individuales y establece las relaciones entre ellos (jerarquía, ligadura mediante transiciones, etc.). Después de establecer cada aspecto del modelo continua una etapa de simplificación del mismo, donde tiene lugar la traducción automática de los eventuales estados parametrizados por estados equivalentes (se eliminan *estados template* y *estados diamante*, resolviendo, de existir, las expresiones matemáticas en estos últimos), se suprimen *estados referencia* y conectores. Todo esto da lugar a un nuevo modelo, más simple que el original en variedad de elementos Statecharts pero manteniendo la semántica de éste.

Posteriormente al nuevo modelo construido se le aplican las actividades pertinentes para llevarlo al denominado *stauts inicial* o *stauts.0*, estadio en el que el modelo se encuentra antes de recibir evento alguno. Para ello deben detectarse las *transiciones por defecto*, analizar los tipos de los estados involucrados y determinarse las transiciones que se encuentran habilitadas a fin de precisar los estados activos.

A partir de este punto el sistema está preparado para recibir los eventos externos indicados por el usuario (a través de un Editor de Eventos desarrollado para tal fin), los que constituyen una simulación.

Existe una instrucción especial para obtener una instantánea del sistema en cualquier momento, es decir para generar su *status* n-ésimo. Cada vez que se envíe una de estas instrucciones se obtendrá, en el directorio temporal del sistema operativo anfitrión, un archivo con el nombre *modelo\_status\_n.odg* útiles para observar el efecto de los eventos externos sobre el sistema.

Junto a los distintos *status* el Sistema entrega un archivo de *log* con los sucesos más destacados que tuvieron lugar para que el usuario pueda, al finalizar la simulación, analizar la ejecución del Sistema.

## 5.2. Algoritmo del Sistema de Simulación

El algoritmo del Sistema de Simulación está basado en el descrito por Harel y Naamad en [HN96] (página 312), si bien existen diferencias respecto al algoritmo aquí desarrollado, puesto que en el citado documento se contempla semántica propia de STATEMATE (como las denominadas *activities*, las características de historia en los estados, etc.), la estructura general del algoritmo es semejante.

Se brindan pormenores sobre el algoritmo del Sistema de Simulación en la documentación destinada al desarrollador en la sección "Instrucciones para el Programador - Detalles del Algoritmo de Simulación".

## 5.3. Tecnología empleada

En este apartado se lista el conjunto de tecnología utilizada para el desarrollo del Sistema. Se persiguió como meta principal la de utilizar software de código abierto y ampliamente difundido. En el desarrollo intervienen las siguientes tecnologías:

- Java: como lenguaje principal de desarrollo.
- OpenDocument: formato abierto de documento.
- XML: como lenguaje de representación de datos de entrada y de salida.
- JDOM: librería para el tratamiento de archivos XML con Java.
- Open/Star Office Draw: herramienta para la creación de archivos en formato OpenDocument consumidos por el Sistema.
- Bash: para la creación de scripts para Linux/UNIX.

### 5.3.1. Java

El Sistema está desarrollado con el lenguaje de programación Java el cual es orientado a objetos y fue desarrollado por Sun Microsystems a principios de los años 90. Entre noviembre de 2006 y mayo de 2007, Sun Microsystems

liberó la mayor parte de sus tecnologías Java bajo la licencia GNU GPL. La versión aquí empleada fue la **jdk1.6.0\_02**.

Para el desarrollo de las interfaces gráficas se utilizó la biblioteca gráfica **Swing** la cual forma parte de las Java Foundation Classes (JFC). Incluye *widgets* para interfaz gráfica de usuario tales como cajas de texto, botones, desplegados y tablas.

Se eligió Java como lenguaje de desarrollo debido a su fácil integración con XML, expresividad y la posibilidad de generación de documentación directa a través de *JavaDoc*.

### 5.3.2. Desing Patterns

Durante el desarrollo se emplearon patrones de diseño en aras de obtener una codificación más sencilla de comprender y ampliar.

Según Wikipedia, un patrón de diseño es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias).

Los patrones de diseño que se aplicaron fueron:

- *Singleton*: Para asegurar la existencia de una única instancia de la clase que realiza el tratamiento de archivos XML.
- *Strategy*: A fin de permitir la existencia de varios algoritmos para la realización de las transiciones.

Muchos otros patrones los incorpora Java directamente, como por ejemplo el patrón *Iterator*.

### 5.3.3. OpenDocument

El *Formato de Documento Abierto para Aplicaciones Ofimáticas* de OASIS (*OASIS Open Document Format for Office Applications*), también conocido como OpenDocument u ODF, es un formato de archivo estándar para el almacenamiento de documentos de ofimática.

Su desarrollo ha sido encomendado a la organización OASIS (acrónimo de *Organization for the Advancement of Structured Information Standards*) y está basado en un esquema XML inicialmente creado por *OpenOffice.org*. Este estándar fue desarrollado públicamente por un grupo de organizaciones, es de acceso libre, y puede ser implementado por cualquiera sin restricción alguna.

La versión aquí empleada fue la **1.1**.

#### Formato interno de los archivos basados en OpenDocument

Los documentos en formato OpenDocument son almacenados como archivos comprimidos tipo *.zip* que contienen archivos XML. Para acceder a estos

archivos XML individuales, puede descomprimirse un archivo OpenDocument con cualquier programa estilo *unzip*.

Las siguientes acciones despliegan el contenido de un archivo OpenDocument al descomprimirlo:

```
$ mv modelo.odg modelo.zip
$ unzip modelo.zip -d modelo-data/
Archive: modelo.zip
  extracting: modelo-data/mimetype
  creating: modelo-data/Configurations2/statusbar/
  inflating: modelo-data/Configurations2/accelerator/current.xml
  creating: modelo-data/Configurations2/floater/
  creating: modelo-data/Configurations2/popupmenu/
  creating: modelo-data/Configurations2/progressbar/
  creating: modelo-data/Configurations2/menubar/
  creating: modelo-data/Configurations2/toolbar/
  creating: modelo-data/Configurations2/images/Bitmaps/
  inflating: modelo-data/content.xml
  inflating: modelo-data/styles.xml
  inflating: modelo-data/meta.xml
  inflating: modelo-data/Thumbnails/thumbnail.png
  inflating: modelo-data/settings.xml
  inflating: modelo-data/META-INF/manifest.xml
```

El contenido del archivo **content.xml** es el que el Sistema emplea para la captura de los modelos. Este archivo almacena el contenido real del documento (excepto los datos binarios como las imágenes).

OpenDocument hace un uso intensivo de los estilos para el formateo y disposición del contenido. La mayor parte de la información de estilo se almacena en el archivo **styles.xml** (aunque existe también parte de la misma en content.xml). Los estilos también pueden ser vistos en la ventana **Estilo y Formato** de Open/Star Office Draw.

El archivo **meta.xml** contiene meta-información del documento, como el autor y fecha de modificación, la cual puede verse desde **Archivo/Propiedades** en Open/Star Office Draw.

El archivo **settings.xml** contiene la configuración de parámetros del documento, como el factor de zoom o la posición del cursor, que afectan a la apertura inicial del documento, pero no su contenido.

El archivo **manifest.xml** describe la estructura general del archivo XML frente a la cual los documentos se validan.

Al ser el OpenDocument un estándar público existen muchas aplicaciones que emplean este formato (gracias a esto es posible interoperar fácilmente con el Sistema de Simulación desarrollado), puede verse un detalle de estas aplicaciones en el sitio <http://opendocumentfellowship.com/applications>.

Puesto que uno de los objetivos de los formatos abiertos, tales como Open-Document, es garantizar el acceso a largo plazo a los datos producidos eliminando las barreras técnicas o legales, muchas administraciones públicas y gobiernos han comenzado a considerarlo un asunto de política de interés público.

Por esta razón, junto con el poder de expresividad y la facilidad de uso de Open/Star Office Draw, fue escogido el OpenDocument como formato principal para el desarrollo de este Sistema.

#### 5.3.4. Open/Star Office Draw

Draw es un editor para la realización de gráficos vectoriales con posibilidad de exportar al formato estándar SVG (*Scalable Vector Graphic*). Entre los formatos de archivo soportados se encuentra el Open Document Format; los archivos en este formato tienen extensión `.odg`.

Puede ser usado para crear dibujos con diferentes grados de complejidad. Los elementos "connectors" que posee permiten establecer vínculos entre formas, las cuales están disponibles en un amplia gama, desde líneas en diversos estilos hasta figuras geométricas en dos y tres dimensiones. Draw facilita la construcción de gráficos como diagramas de flujo, organigramas, etc.

Las facilidades gráficas que ofrece hacen que la edición de modelos Statecharts sea una actividad simple de realizar.

Esta herramienta está contenida en las suites StarOffice y OpenOffice.org. Ambas fueron desarrolladas por Sun Microsystems y son multiplataforma, siendo posible instalarlas en los sistemas operativos Solaris, Linux, Microsoft Windows, BSD, OpenVMS, OS/2 e IRIX.

En el desarrollo se empleó el OpenOffice.org Draw versión **2.3**.

#### 5.3.5. JDOM

JDOM es una biblioteca liberada bajo la licencia open source Apache para manipulaciones de datos XML optimizada para Java, [JDOM].

Dicho de manera sencilla, JDOM es una representación Java de un documento XML que provee los medios para representar el documento para su fácil y eficiente lectura, manipulación y escritura. Tiene una API liviana y rápida, optimizada para el programador Java. Es una alternativa para DOM y SAX, sin embargo se integra bien con éstos:

- DOM: *Document Object Model*, es un modelo computacional a través del cual los programas pueden acceder y modificar dinámicamente el contenido, estructura y estilo de los documentos HTML y XML. El responsable del DOM es el consorcio W3C (World Wide Web Consortium, [WWW]). Para más detalles visitar [DOM].

- SAX: *Simple API for XML*, es una API para un parser de acceso serial a documentos XML, provee mecanismos para la lectura de documentos XML. Más datos en [SAX].

La principal diferencia entre JDOM y DOM es que mientras DOM fue creado para ser un lenguaje neutral e inicialmente usado para manipulación de páginas HTML y XML con JavaScript, JDOM se creó específicamente para utilizarlo con Java y por lo tanto beneficiarse de las características de Java, incluyendo sobrecarga de métodos, colecciones, etc. Para los programadores de Java, JDOM es una extensión más natural y correcta.

JDOM no es un *parser*, es más bien un *wrapper*, y requiere la presencia de un parser subyacente, tal como Xerces [XRCS] o alguno implementado siguiendo SAX. Para el desarrollo del Sistema se empleó un parser basado en SAX debido a su buen nivel de performance y la documentación disponible para integrar JDOM con SAX. Cabe mencionar como material de referencia a [MB01].

Para más detalles sobre esta biblioteca y su forma de empleo referirse a [BJ] y [JDOM].

### 5.3.6. Scripting

El Sistema cuenta con soporte para los sistemas operativos Solaris, FreeBSD, Linux (testado en Ubuntu) y Windows (testado en XP). Interactúa con estos sistemas a través de simples scripts realizados en *bash* y *command/cmd*. La principal actividad de dichos scripts se restringe al desempaquetado de los archivos *.odg* para acceder a la información interna, la generación de nuevos archivos *.odg* representando los diferentes *status* que el modelo alcanza y la eliminación de archivos temporales necesarios en el transcurso de la simulación.

El Sistema emplea el directorio temporal de Linux/UNIX (`/tmp`) o de Windows (`C:\windows\temp`) para almacenar estos scripts.

## 6. Documentación

La distribución del Sistema de Simulación está conformada por los siguientes documentos

- Archivos *.jar*: Uno para cada grupo de sistemas operativos Linux/UNIX y Windows.
- *Statechart\_Simulator.tar.gz*: Código fuente del Sistema de Simulación.
- Documentación *JavaDoc*: Documentación exhaustiva del código fuente simple de utilizar y consultar debido a su carácter navegable. En la misma se enuncian las clases que componen el Sistema junto con sus interfaces.
- Documento PDF "Tratamiento del OpenDocument": Describe cómo fue contemplada la representación interna de cada uno de los elementos gráficos que componen a los modelos Statecharts, cómo se resolvió la inclusión

de dichos elementos y notas sobre las transformaciones que algunos elementos Statecharts sufren internamente.

- Suite de modelos Statecharts: Consta de más de 90 modelos de diversa complejidad y es acompañada por el documento de texto *Test\_de\_Modelos.txt* que explica de qué trata cada uno de los modelos. Tiene como finalidad la de introducir ágilmente al usuario en la edición de modelos y la práctica de simulación.
- Papers y artículos sobre Statecharts.
- Binarios para Linux y Windows de la *Java Development Kit*.
- Documentación sobre la librería de Java para el tratamiento de los archivos XML (JDOM).
- Manual de Usuario e Instrucciones para el Programador (comprendidos en las siguientes secciones).

Las siguientes secciones contienen un manual de usuario destinado al usuario final, con el objeto de instruirlo para interactuar con el Sistema de Simulación. Además se brindan instrucciones para el desarrollador que pretenda extender las funcionalidades de la herramienta.

## 6.1. Manual de Usuario

En esta sección se describen las actividades típicas que el usuario debe llevar a cabo para interactuar con el Sistema durante el *proceso de simulación*.

### 6.1.1. Instalar el Sistema de Simulación

La aplicación consiste en un archivo `.jar`; existe uno por cada "tipo" de sistema operativo soportado hasta el momento:

- `Statechrts.Simulator-linux.jar` - Soportando las distribuciones de Linux más utilizadas, Solaris y FreeBSD.
- `Statechrts.Simulator-windows.jar` - Soportando Windows en sus versiones 98, XP y Vista.

Los requerimientos bajo entorno **Linux/UNIX** son los siguientes:

- Open/Star Office versión 2.3 o superior para la edición y apertura de los documentos *.odg*.
- JDK versión 1.6.02 o superior.
- Acceso a través de la variable **PATH** a los binarios: *cp*, *mv*, *rm*, *zip* y *unzip*.

Por el lado de **Windows**:

- Open/Star Office versión 2.3 o superior para la edición y apertura de los documentos *.odg*.

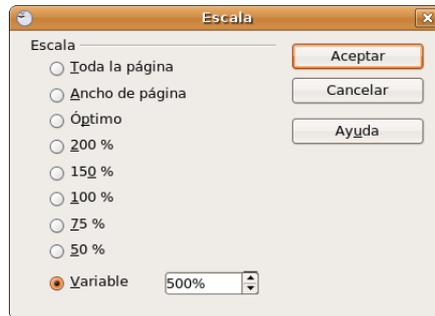


Figura 20: Open/Star Office - Ventana Escala

- JDK versión 1.6.02 o superior.
- Acceso a través de la variable **PATH** a los binarios *copy*, *move*, *del*, *rd* y *7z.exe* (este último se entrega con la distribución del Sistema de Simulación).

### 6.1.2. Creación del sistema/modelo

Los sistemas o modelos se describen utilizando la herramienta *Open/Star Office Draw*.

*Draw* almacena la información de archivo en formato XML (*OpenDocument*) con extensión *.odg*.

Los modelos pueden describirse con facilidad, aprovechando una serie de cuestiones:

- El tamaño máximo del área para graficar sistemas es de 300cm x 300cm, utilizando el formato de página *Usuario*, (Formato/Página...). Al emplear Ver/Escala... y definir una escala *Variable* alta, este espacio se incrementa notoriamente. Observar la Figura 20.
- El color de cualquier tipo de elemento no conlleva significado semántico alguno, por lo tanto se alienta a utilizarlos para aportar claridad a los sistemas, remarcar estados, relaciones entre ellos, etc.
- El grosor y estilo de las transiciones puede ser de cualquier tipo dado que estas variaciones no alteran la semántica de los modelos representados.
- El grosor y estilo de los contornos pertenecientes a los conectores y estados pueden ser elegidos a gusto, pero siempre considerando que el estilo de borde de los estados *sí* añade semántica, donde un borde continuo expresa disyunción entre estados, mientras que el resto de los tipos de contorno señala conjunción.
- El tamaño y ubicación de los elementos tampoco altera la semántica de los sistemas representados (siempre que no se modifiquen las inclusiones entre estados ni ligaduras con transiciones).

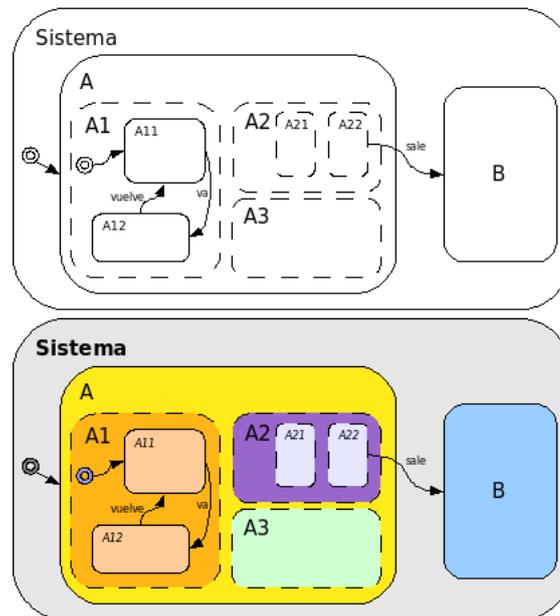


Figura 21: Representaciones equivalentes

- Las etiquetas de las transiciones pueden ser formateadas de cualquier manera (varias líneas, colores, negrita, cursiva, etc.) en aras de la claridad, pero *siempre* deben respetarse las diferentes combinaciones de separadores sintácticos [ , ] / y ; obligatorios para las etiquetas.
- A los nombres de los estados también pueden aplicárseles formato (color, negrita, cursiva, etc.). Para el modelado, los estados pueden poseer el mismo nombre que otro estado, aunque existen algunas excepciones que se detallan más adelante.

Open/Star Office Draw hace de la edición de los modelos Statecharts una tarea muy sencilla. Por ejemplo, para nominar estados y conectores, o etiquetar transiciones, sólo debe elegirse el elemento gráfico a modificar y pulsar la tecla F2 o acceder al menú **Formato/Texto...** También existen menús contextuales accesibles con el botón derecho del mouse.

La Figura 21 muestra dos sistemas equivalentes, en el segundo se hace uso de las facilidades gráficas de Open/Star Office Draw, logrando así representaciones notablemente más claras y fáciles de asimilar.

En La Figura 22 se observan todos los elementos gráficos que deben usarse para la descripción de modelos basados en Statecharts. Dichos elementos se acceden desde la barra de **Dibujo** de Open/Star Office Draw y corresponden a las denominadas **Formas básicas**.

En la parte inferior de la figura se muestra en líneas separadas cuál de las formas básicas le corresponde a cada uno de los elementos Statecharts; el color

del texto empleado es igual al del recuadro que contiene al elemento gráfico relacionado.

Como se mencionara previamente, un estado es un *estado or* o un *estado and* según el tipo de borde de los subestados que contiene. Para cambiar el tipo de borde de un estado, y cualquier otra característica, como el color, texto (que hace las veces de nombre), posición, tamaño, etc., debe seleccionarse tal estado y acceder al menú **Formato** o bien pulsar el botón derecho del mouse. En la Figura 23 se muestran las diferentes opciones para cambiar el tipo o estilo de borde (línea) de un elemento.

Para crear las transiciones entre estados se emplean los *conectores* de Open/Star Office Draw, los mismos también son accedidos desde la barra de **Dibujo**. El conjunto de conectores que se emplean se muestra en la Figura 24 recuadrados en azul. Sólo se emplean estos puesto que indican dirección (en un único sentido). Los otros conectores también son válidos y el sistema los contempla, pero la lectura del modelo se verá afectada y el sentido de las transiciones quedará a la interpretación del usuario, lo cual no es deseable.

No confundir estos conectores, con los *elementos conectores* de Statecharts. Los primeros son elementos de Open/Star Office Draw que representan las transiciones en los modelos Statecharts, mientras que los últimos son elementos propios de Statecharts que permiten reducir la cantidad y longitud de las transiciones.

Todos los elementos gráficos de Open/Star Office Draw poseen un número finito de *puntos de adhesión* por defecto cuya cantidad varía según el elemento introducido. Los rectángulos redondeados, por ejemplo, poseen cuatro, mientras que los círculos ocho y los cubos seis. Cuando el usuario necesita emplear transiciones para vincular elementos debe usar los conectores (transiciones) y hacer que tanto el origen como el destino se acoplen a los elementos a través de estos puntos.

Es posible incrementar la cantidad de los puntos de adhesión, para ofrecer una confección de modelos más claros y evitar cruces o cortes de transiciones. Observar la Figura 25 para ver un ejemplo concreto. El estado de la izquierda posee sólo cuatro puntos de adhesión, mientras que el de la derecha ocho. Los cuatro puntos adicionales se marcan con cruces azules y están siendo usados por las transiciones  $t_2$ ,  $t_5$  y  $t_6$ . Recuadrado en azul, se indica el botón que despliega el menú contextual (también recuadrado) con los controles para la edición de puntos de adhesión.

### 6.1.3. Gramática permitida para los modelos

Al momento de editar los modelos es importante conocer las restricciones gramaticales del lenguaje Statecharts aquí consideradas. Al violar cualquiera de estas restricciones el usuario recibirá un *mensaje de error* relacionado que impedirá al usuario iniciar la simulación.

Existen muchas restricciones; para su fácil enumeración se describirán

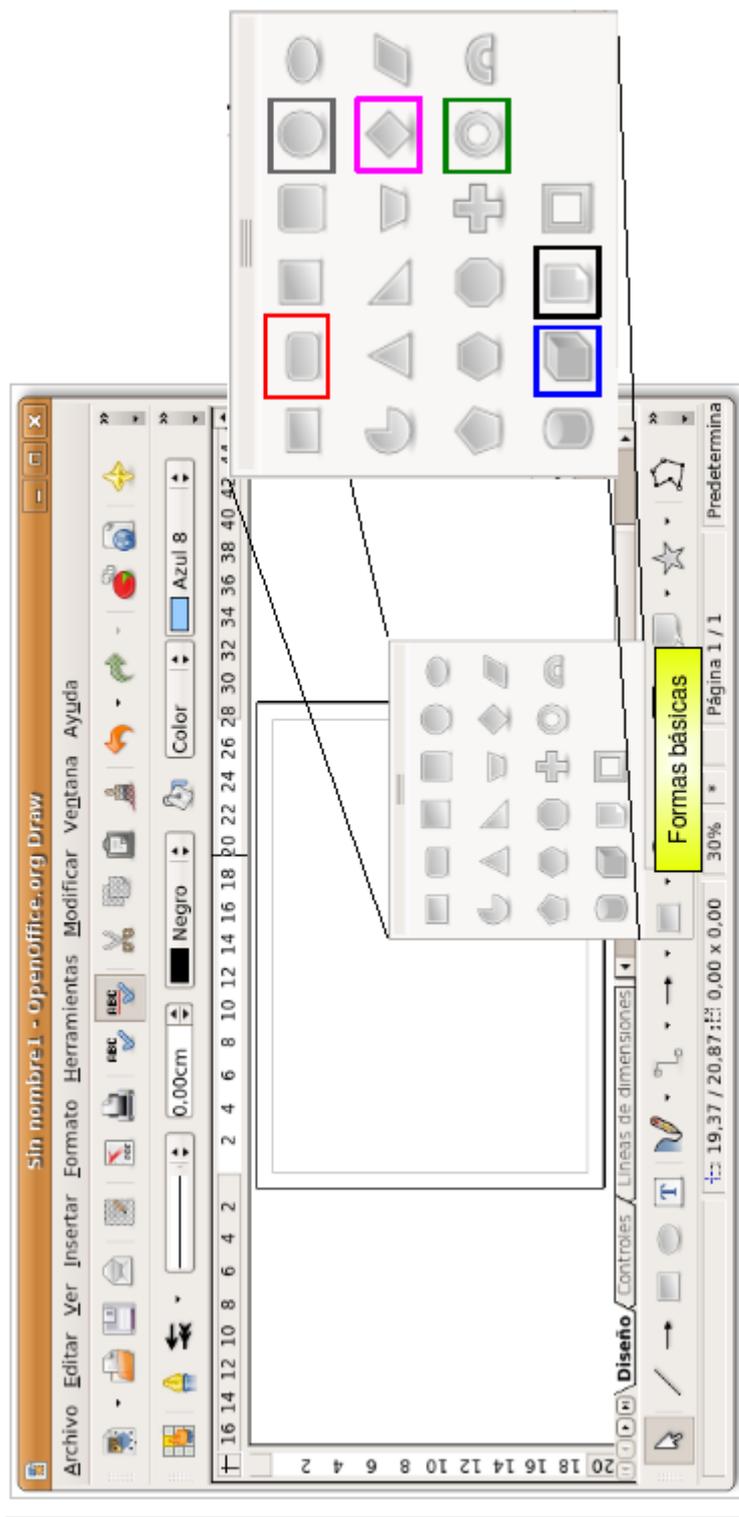


Figura 22: Open/Star Office Draw - Elementos gráficos

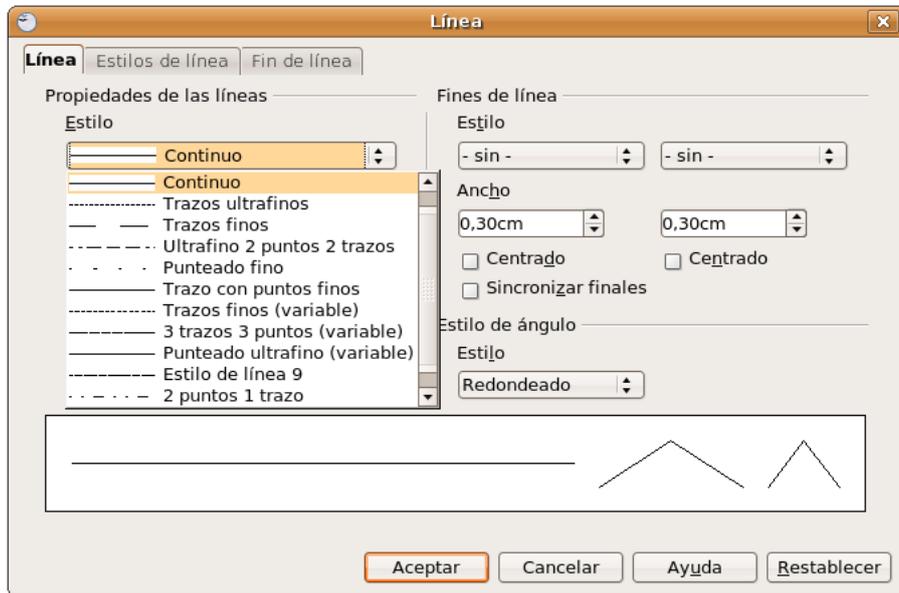


Figura 23: Open/Star Office Draw - Tipos de líneas

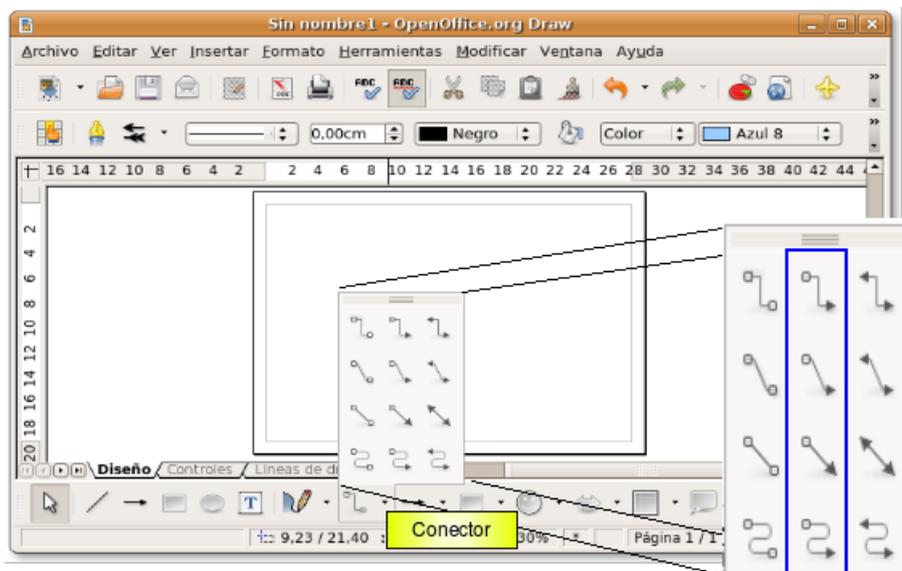


Figura 24: Open/Star Office Draw - Conectores (Transiciones de Statecharts)

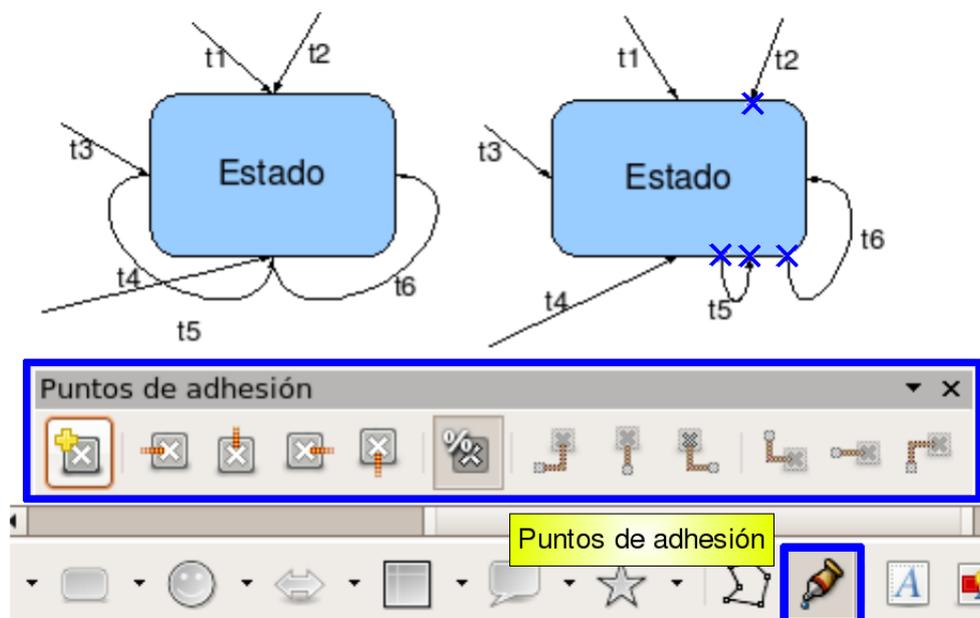


Figura 25: Open/Star Office Draw - Puntos de adhesión

agrupándolas por tipo de objeto Statecharts:

#### Estados

- Los estados no pueden solaparse o superponerse.
- No pueden salir transiciones idénticamente etiquetadas desde un mismo estado.
- Todo *estado or* debe tener una y sólo una transición por defecto.

#### Nominación de los estados

Los nombres de los estados deben respetar ciertos símbolos obligatorios dependiendo del tipo de estado, como se comentó en la sección "Elementos de Statecharts considerados". Un estado puede tener como nombre aquel que ya posea otro estado en el modelo, aunque existen excepciones a esto. Una excepción está en relación a los *estados referencia*, si uno de estos estados se nombra con **A\*** y existen dos o más estados con nombre **A**, se generará un error.

La otra excepción proviene de los *estados parametrizados*. Si un estado parametrizado tiene como nombre **S**[*i* = 1.,10], debe considerarse con especial cuidado la inclusión de dos o más subestados con **S**(1), pues esto conducirá a un error.

#### Estados Temporizados

- Las cotas de tiempo deben ser constantes enteras positivas.

- Al menos debe existir una cota de tiempo.  
Casos permitidos: [ $a <, < b$ ]; [ $< a$ ]; [ $< a, ]$ ]; [ $, < b$ ].
- Deben tener exactamente una transición saliente con etiqueta *timeout*.

#### Estados Parametrizados

- Los nombres pertenecientes a un *estado template* (cubo)  $S(i)$  dentro de un estado parametrizado  $S[i = \dots]$  deben coincidir, en este caso lo que debe ser igual es el nombre "S".
- Dado un estado parametrizado  $S[var = a..b]$  debe ser  $0 \leq a < b$  y con  $a$  y  $b$  constantes enteras positivas incluido el cero.
- Las variables  $S[var1 = \dots]$  y la del estado cubo  $S(var2)$  deben coincidir en nombre ( $var1 == var2$ ).
- Todo *estado instancia* debe pertenecer a un estado parametrizado.
- Los *estados instancia* no pueden contener elementos en su interior.
- Si un *estado instancia* en su nombre presenta una variable, la misma debe coincidir con la del estado padre  $S[var = a..b]$ , en este caso debe ser *var*.
- Deben respetarse las expresiones matemáticas permitidas en los *estados instancia*; los operadores matemáticos admitidos al momento son: +, - y %.
- Dos *estados instancia* pueden estar vinculados por una transición sólo si al menos uno de ellos tiene como nombre o bien una variable o bien una constante.

#### Referencias

- En presencia de un estado referencia cuyo nombre es  $S*$ , *debe existir un único* estado con nombre  $S$  en el modelo.
- Todo estado referencia debe tener un estado padre.
- Los estados referencia no deben contener ningún tipo de elemento interno.

#### Transiciones

- Siempre deben poseer elementos de origen y destino, no pudiendo estar "descolgadas" de ninguno de sus extremos.
- Debe respetarse el formato de etiquetado *evento* [ *condición* ] / *acción*.
- Los lazos o bucles sólo son válidos para los estados.

#### Símbolo de Transición por defecto

- No pueden tener transiciones entrantes.
- No pueden tener más de una transición saliente ni lazos o bucles.

#### Conectores

- Por el momento no se permiten conectores diferentes a los tipos: *C*, *J*, *F* y *S*.
- Deben tener al menos una transición entrante y una saliente.
- Siempre deben pertenecer a un estado.

#### 6.1.4. Almacenar en disco el sistema/modelo

En Open/Star Office Draw, luego de finalizar la confección del modelo o sistema, se prosigue a su almacenamiento en memoria persistente.

Los modelos pueden almacenarse en cualquier lugar del disco, para ello deberá dirigirse al menú **Archivo - Guardar como...** de Open/Star Office Draw.

El formato de almacenamiento *debe* ser exclusivamente el OpenDocument Format cuya extensión de archivo es *.odg*.

#### 6.1.5. Ejecutar el Sistema de Simulación

La aplicación puede iniciarse, dependiendo del entorno del usuario, tanto mediante línea de comandos con la instrucción:

```
java -jar Statechrts.Simulator-sistema_operativo.jar
```

o directamente haciendo doble click sobre el archivo *.jar*.

El *proceso de simulación* consta de cuatro pasos fundamentales

- Selección del sistema/modelo a simular
- Definición de la secuencia de eventos para la simulación
- Ejecución de la simulación
- Análisis de resultados

Cada una de estas etapas son desarrolladas a continuación.

##### Selección del sistema/modelo

Para la selección del sistema a simular pulsar el botón **Examinar...** de la ventana principal, la cual se muestra en la Figura 26. Dicho botón desplegará una ventana desde la cual sólo pueden seleccionarse archivos con extensión *.odg*.

##### Definición de la secuencia de eventos para la Simulación

Para la carga de eventos se ha desarrollado un editor integrado. A este editor puede accederse desde la ventana principal pulsando el botón **Editor** o bien desde el ítem de menú **Eventos/Editor...** La Figura 27 muestra la ventana correspondiente al editor.

Ya en el editor, en el panel izquierdo, titulado *Lista de Eventos Existentes*, se registran todos los eventos presentes en el modelo. Mediante los controles

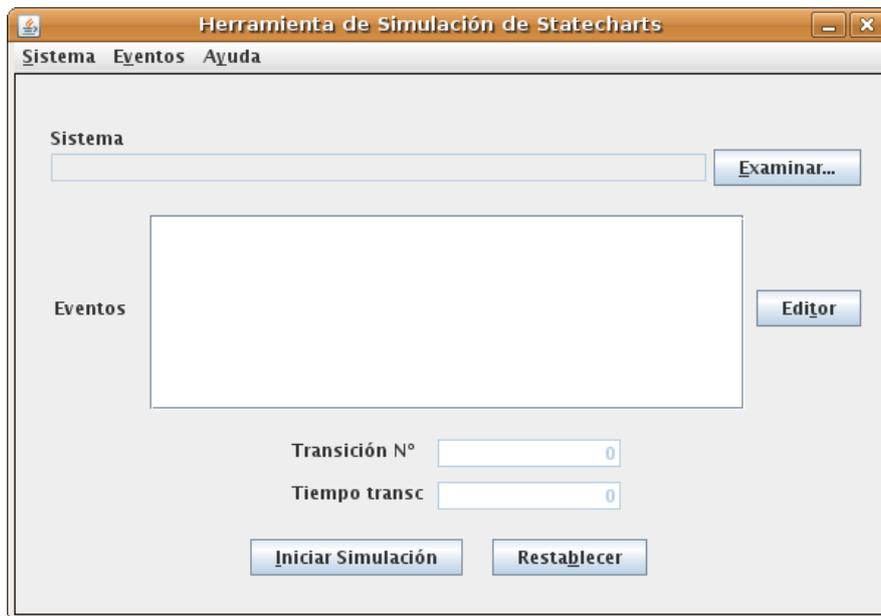


Figura 26: Ventana Principal del Sistema de Simulación

centrales se pueden agregar estos eventos a la *Lista de Eventos Seleccionados* los cuales posteriormente serán enviados al modelo en ese orden para su ejecución. Pueden seleccionarse varios eventos simultáneamente de la primer lista manteniendo la tecla *Control* presionada.

Entre los controles más importantes se encuentran el de agregado de lapsos de tiempo, de la forma  $\{t\}$  (donde  $t$  es una cantidad no negativa de unidades de tiempo), y el de mandato para graficar el *status* del modelo representado por [GRAFICAR\_STATUS].

Respecto al uso de los lapsos de tiempo  $\{t\}$ , estos tendrán sentido sólo en el caso en que el modelo contenga *estados temporizados*, los cuales poseen cotas de tiempo impuestas. Estos lapsos de tiempo sirven para definir espacios de tiempo entre eventos y en ausencia de los mismos el Sistema considera que no media tiempo entre eventos sucesivos, provocando que la ocurrencia de éstos sea de manera instantánea. Las cotas de tiempo presentes en los estados temporizados pertenecen a la misma categoría que los lapsos de tiempo aquí mencionados, sólo el número es lo que vale. Por ejemplo, dado un estado temporizado con una cota de tiempo "3", dicha cota será transgredida luego de transcurridos uno o más lapsos de tiempo cuya suma supere 3 unidades.

Un lapso de tiempo  $\{n\}$ , indica que entre dos eventos transcurren  $n$  unidades de tiempo, estas unidades *no* son segundos o milisegundos, sólo el número  $n$  en sí tiene significado para el Simulador. Cuando el Simulador recibe uno de estos lapsos de tiempo lo divide en  $n$  lapsos  $\{1\}$  y los trata en secuencia; al pasar el primer lapso verifica si alguna de las cotas superiores de tiempo de los estados

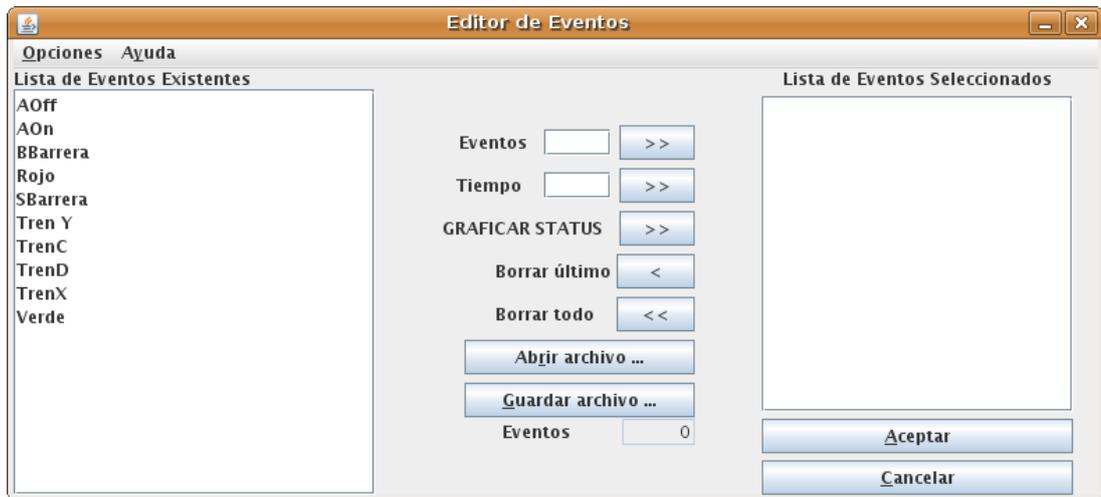


Figura 27: Menú de Edición de Eventos



Figura 28: Cuadros de cantidad de eventos y tiempo

temporizados se transgrede, en cuyo caso realiza las acciones pertinentes y toma el próximo lapso  $\{1\}$  hasta consumir las  $n$  instancias.

Debe quedar claro que el Sistema aun no maneja el tiempo de manera autónoma, el usuario es el que debe explicitar los lapsos de tiempo que considere oportunos para que las cotas de tiempo de los estados temporizados sean transgredidas o el tiempo de permanencia en estos estados avance.

Volviendo al Editor de Eventos, los cuadros de texto vacíos que acompañan a los botones para inserción de eventos existentes y de lapsos de tiempo se emplean para introducir cantidad. Por ejemplo, en la Figura 28, en la fila **1** (Eventos), el número 3 indica introducir tres instancias del evento seleccionado en la lista de *Lista de Eventos Existentes* a la *Lista de Eventos Seleccionados*, por otra parte en la fila **2** (Tiempo), se introduce el lapso de tiempo  $\{5\}$ .

También existen botones para realizar correcciones y para almacenar/recuperar listas de eventos en archivos de extensión **.evs**.

Finalmente, con el botón **Aceptar**, los eventos seleccionados se copian en la ventana principal y se aprestan para ser enviados al modelo en el orden definido.

## Ejecución de la simulación

Una vez seleccionado el sistema y definida la secuencia de eventos se pulsa el botón **Iniciar Simulación** o se selecciona la opción *Iniciar Simulación* del menú **Sistema** en la ventana principal (ver Figura 26), para dar comienzo a la simulación.

En caso de error gramatical o semántico, el usuario recibirá un mensaje de error orientativo.

Al finalizar la simulación el usuario recibirá un cartel informativo anunciando la finalización del proceso.

Por último, es responsabilidad del usuario proponer sólo simulaciones consistentes con el entorno. Por ejemplo aquellas que no vayan contra las características o restricciones naturales del modelo, como encender una luz ya encendida, etc.

### Análisis de resultados

Al finalizar la simulación se encontrarán en el directorio temporal del sistema operativo anfitrión tantos archivos con los distintos *status* del modelo como instrucciones [GRAFICAR.STATUS] se hayan introducido con el Editor de Eventos. Para los sistemas Linux/UNIX este directorio es el (**/tmp**), y en Windows es el (**C:\windows\temp**).

El nombre de los archivos con los distintos *status* sigue el patrón común "*nombre\_modelo.status.n.odg*", donde *n* es un entero positivo. Son archivos .odg que pueden abrirse directamente con Open/Star Office Draw aunque *no* pueden emplearse como archivos de entrada para el Sistema. En los archivos se encuentran destacados los estados activos del modelo en dicho *status*. La forma elegida para discriminar a los estados activos del resto de los estados es que sus nombres titilan o resplandecen. Sólo existe una excepción a esto y proviene de los estados parametrizados, en este caso la información de los estados activos se presenta con una línea de texto estilo marquesina.

Además de estos archivos de salida, al finalizar cada simulación se genera un archivo de *log* con el nombre "*nombre\_modelo.log*", el mismo contiene información de todos los sucesos importantes que acontecieron.

## **6.2. Instrucciones para el Programador**

Esta sección se compone fundamentalmente de una guía descriptiva de las clases (módulos de software) del Sistema y de los detalles del algoritmo de simulación, indicando sus diferencias y semejanzas frente al algoritmo presentado por Harel y Naamad en [HN96]. Con esta información se pretende brindar al programador herramientas suficientes para que éste pueda sumar funcionalidades al Sistema y que tenga un entendimiento acabado sobre el mismo.

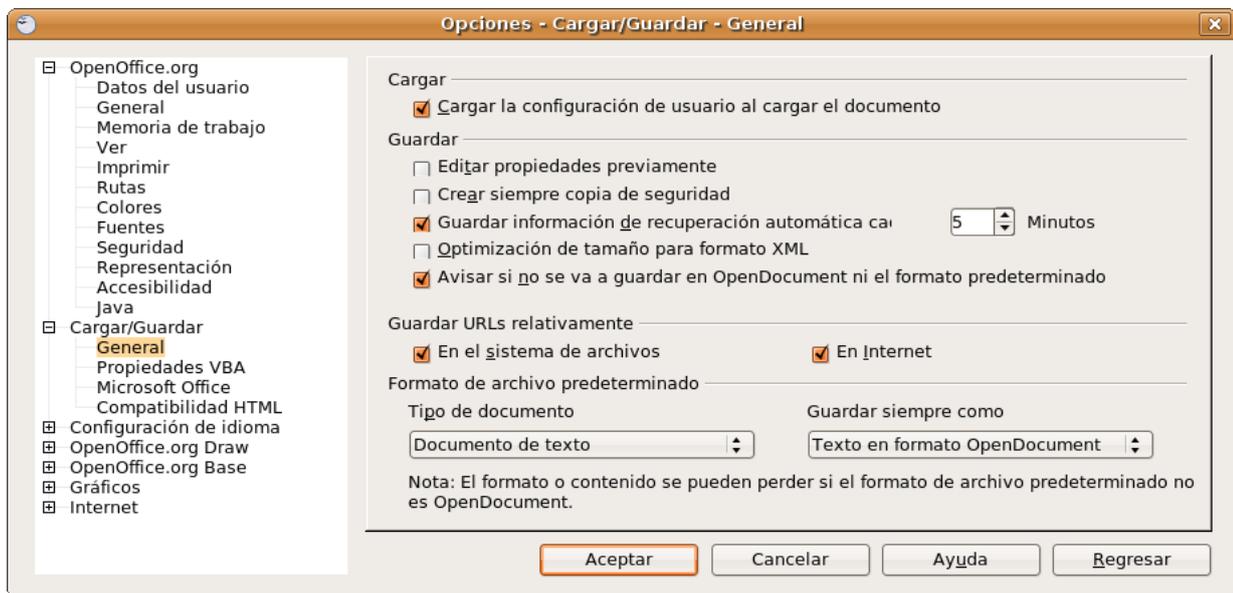


Figura 29: Pretty printing

Antes de adentrarnos en el algoritmo en sí, un comentario acerca de los archivos XML generados por Open/Star Office Draw.

### 6.2.1. Acerca de XML y Open/Star Office Draw

Al momento de almacenar los modelos Statecharts confeccionados con Open/Star Office Draw, es conveniente desmarcar el *checkbox* "Optimización de tamaño para formato XML", del menú **Herramientas - Opciones - Cargar/Guardar - General**. La Figura 29 muestra esta ventana y la opción comentada desmarcada. De esta manera se obtendrán documentos XML con la característica de *pretty printing*, es decir, más legibles.

Aunque esto no representa cambios en las funciones del Sistema, es una opción útil para comprender la estructura general que respetan los archivos XML y realizar tareas de *debugging*.

### 6.2.2. Guía de módulos

A continuación se describen las clases que componen el Sistema, las mismas están listadas respetando el formato propuesto en [MC07].

<b>Clase:</b>	<b>AcercaDe</b>
<b>Función:</b>	Presenta información sobre la versión del Sistema y datos del creador en una ventana gráfica.
<b>Archivos:</b>	AcercaDe.form & AcercaDe.java

<b>Clase:</b>	<b>Algorithms</b>
<b>Función:</b>	Implementa el algoritmo para la ejecución de las transiciones en los modelos.
<b>Secreto:</b>	El algoritmo de transición entre estados. Esta clase emplea el patrón de diseño <i>strategy</i> para permitir que a futuro se puedan implementar otros algoritmos y el usuario pueda seleccionar cual aplicar entre ellos.
<b>Archivo:</b>	<code>Algorithms.java</code>
<b>Clase:</b>	<b>Connector</b>
<b>Función:</b>	Representa a los conectores de Statecharts, su tipo e información del estado que lo contiene y las transiciones entrantes y salientes.
<b>Archivo:</b>	<code>Connector.java</code>
<b>Clase:</b>	<b>Core</b>
<b>Función:</b>	Hace las veces de nexo del Sistema, es la única clase que interactúa con la clase <b>JDOMWorker</b> (la que analiza los archivos XML), realiza la primer etapa de categorización de los elementos del modelo Statecharts, ejecuta los scripts de sistema operativo, aplica funciones de otros módulos al modelo y actúa como canal para el envío de eventos al modelo.
<b>Secreto:</b>	La manera en que interactúa con el sistema operativo anfitrión a través de los scripts.
<b>Archivo:</b>	<code>Core.java</code>
<b>Clase:</b>	<b>Diamond_info</b>
<b>Función:</b>	Representa los estados instancia (diamante) los cuales tienen un tratamiento especial, pues deben pertenecer a un estado parametrizado y tener un nombre validado.
<b>Archivo:</b>	<code>Diamond_info.java</code>
<b>Clase:</b>	<b>Dimension</b>
<b>Función:</b>	Asociar dimensiones espaciales a cada elemento gráfico del sistema y para definir una métrica que permita determinar de manera precisa la inclusión entre elementos.
<b>Secreto:</b>	El algoritmo empleado para el cómputo de las métricas de inclusión entre estados, conectores y símbolos de inicio de las transiciones por defecto.
<b>Archivo:</b>	<code>Dimension.java</code>
<b>Clase:</b>	<b>EditorEventos</b>
<b>Función:</b>	Formulario y clase correspondiente al Editor de Eventos con el que el usuario interactúa para construir la secuencia de eventos a enviar al modelo.
<b>Secreto:</b>	Cómo confecciona la lista de eventos existentes y cómo realiza la presentación de los mismos. Interacción con el sistema de archivos.
<b>Archivos:</b>	<code>EditorEventos.form</code> & <code>EditorEventos.java</code> .
<b>Clase:</b>	<b>Error</b>
<b>Función:</b>	Clase que contiene la definición de todos los mensajes de error que pueden generarse durante la examinación del modelo y el proceso de simulación.
<b>Secreto:</b>	La manera en que estos mensajes se presentan y cómo se organizan.
<b>Archivo:</b>	<code>Error.java</code>

<b>Clase:</b>	<b>JDOMWorker</b>
<b>Función:</b>	Clase que utiliza la librería JDOM para recorrer el archivo <i>content.xml</i> e identificar cada uno de los elementos gráficos relacionados con los modelos Statecharts.
<b>Secreto:</b>	Cómo se implementa la activación y desactivación de estados manipulando los archivos XML. Algoritmo de clasificación de los elementos gráficos.
<b>Archivo:</b>	JDOMWorker.java
<b>Clase:</b>	<b>Param_info</b>
<b>Función:</b>	Clase que complementa a los estados clasificados como parametrizados. Posee información útil para la activación de los mismos.
<b>Secreto:</b>	La manera en que categoriza cada una de las componentes del nombre de un estado parametrizado y cómo colabora con la activación de estos estados.
<b>Archivo:</b>	Param_info.java
<b>Clase:</b>	<b>Point</b>
<b>Función:</b>	Clase simple que representa un punto en el plano, útil para detectar errores en inclusiones entre elementos gráficos (solapamientos) y colabora en la definición de la métrica de inclusión de elementos.
<b>Archivo:</b>	Point.java
<b>Clase:</b>	<b>Ring</b>
<b>Función:</b>	Clase que representa los elementos iniciales de las transiciones por defecto.
<b>Archivo:</b>	Ring.java
<b>Clase:</b>	<b>Sistema</b>
<b>Función:</b>	Es la clase que compone los modelos, estableciendo la relación entre los elementos y categorizándolos; transforma el <i>modelo inicial o gráfico</i> en el denominado <i>modelo real</i> reescribiendo estados parametrizados y eliminando conectores y estados referencia entre otras actividades. Detecta una cantidad de errores antes de la etapa de simulación.
<b>Secreto:</b>	Posee algoritmos para transformar el modelo y todas las estructuras de datos necesarias para representar un sistema Statecharts.
<b>Archivo:</b>	Sistema.java
<b>Clase:</b>	<b>Statecharts</b>
<b>Función:</b>	Clase y formulario dedicados a la interfaz principal del Sistema, permite la elección del modelo a simular y el acceso al Editor de Eventos.
<b>Secreto:</b>	Oculto la manera en que los eventos son enviados al modelo.
<b>Archivo:</b>	Statecharts.form & Statecharts.java
<b>Clase:</b>	<b>State</b>
<b>Función:</b>	Clase relacionada con los estados de los modelos Statecharts. Contiene información como ser el nombre, identificadores internos, tipo de estado, transiciones entrantes y salientes, subestados, etc.
<b>Secreto:</b>	Posee estructuras de datos internas para la representación de todo lo que incumbe a un estado.
<b>Archivo:</b>	State.java
<b>Clase:</b>	<b>Temp_info</b>
<b>Función:</b>	Clase que complementa a los estados, contiene información esencial para los estados categorizados como temporizados.
<b>Archivo:</b>	Temp_info.java

<b>Clase:</b>	<b>Transition</b>
<b>Función:</b>	Clase que representa las transiciones, contiene información de los elementos de origen y destino de las mismas y de cada uno de los componentes de la eventual etiqueta. Categoriza los extremos de la transición (estado, conector, símbolo de inicio de transición por defecto).
<b>Secreto:</b>	Contiene las estructuras de datos necesarias para la representación de las transiciones.
<b>Archivo:</b>	<code>Transition.java</code>
<b>Clase:</b>	<b>Utils</b>
<b>Función:</b>	Clase que contiene funciones de utilidad general y algunas variables globales importantes. Se encarga de la generación de las salidas por pantalla y del archivo de <i>log</i> .
<b>Secreto:</b>	La manera en que se genera el archivo de <i>log</i> .
<b>Archivo:</b>	<code>Utils.java</code>
<b>Clase:</b>	<b>VentanaErrores</b>
<b>Función:</b>	Formulario y clase de la ventana dedicada a la publicación de errores.
<b>Archivos:</b>	<code>VentanaErrores.form</code> & <code>VentanaErrores.java</code>
<b>Clase:</b>	<b>XMLReader</b>
<b>Función:</b>	Clase empleada para la lectura de archivos XML en los documentos OpenDocument Draw.
<b>Secreto:</b>	El tipo de parser XML empleado.
<b>Archivo:</b>	<code>XMLReader.java</code>
<b>Clase:</b>	<b>XMLWriter</b>
<b>Función:</b>	Clase empleada para la escritura de archivos XML en documentos OpenDocument Draw.
<b>Archivo:</b>	<code>XMLWriter.java</code>

### 6.2.3. Detalles del Algoritmo de Simulación

El algoritmo del Sistema de Simulación está basado en el algoritmo descripto por Harel y Naamad en el documento [HN96] (página 312, sección "THE BASIC STEP ALGORITHM"), donde se muestra la ejecución de un único paso, entendiéndose como *paso* todo aquello que involucra la ejecución de una transición, el cambio del status del sistema y los eventos y acciones que se desencadenan en respuesta a dicho suceso.

Se discriminan tres etapas en el algoritmo de Harel y Naamad:

- Etapa 1 - Preparación de un paso: la organización de los eventos a ejecutar y el tratamiento de las acciones planificadas y eventos "timeout" (si bien no se habla de estados temporizados, estos eventos tienen un sentido semejante).
- Etapa 2 - Cálculo del contenido de un paso: se realiza un análisis de las transiciones habilitadas a fin de ejecutar las prioritarias y detectar situaciones de conflicto y no determinismo.
- Etapa 3 - Ejecución de las transiciones: se ejecutan todas aquellas acciones correspondientes al egreso e ingreso (desactivación y activación) desde y en los estados.

En el algoritmo se mencionan algunas características propias de la implementación de STATEMATE, como ser *control activities*, *static reactions* y acciones, las cuales no fueron contempladas en el actual desarrollo y en consecuencia no estarán presentes en el algoritmo que se expone a continuación.

En [HN96], si bien no se detalla en el mismo algoritmo el tratamiento del tiempo, se describe en una sección posterior los dos modelos que STATEMATE soporta: sincrónico y asincrónico. El algoritmo aquí implementado tiene más relación con el segundo enfoque, donde el avance del tiempo de la simulación deber ser explicitado por el operador o entorno.

A continuación se describe el algoritmo del Sistema de Simulación. Se detallan las etapas más importantes junto a los pasos que las componen y cada una de las funciones que los llevan a cabo. En esta descripción se hacen numerosas referencias a las clases, funciones y algunas estructuras de datos del Sistema; se recomienda fuertemente el análisis del algoritmo junto al código fuente del Sistema. A diferencia del algoritmo en [HN96] existe un número mayor de etapas puesto que comprende todo lo previo y posterior a la ejecución de un paso.

### **Entrada**

- Modelo Statecharts representado en un archivo Open/Star Office de extensión `.odg`.
- Lista de eventos a enviar al modelo (los considerados eventos externos).

### **Etapas**

#### **● Etapa de construcción del sistema**

En esta fase se construye una representación un modelo de objetos a partir del el el sistema gráfico plasmado en el archivo `.odg`.

- Se accede al archivo `content.xml` del documento `.odg` que representa al modelo Statecharts que el usuario selecciona mediante la interfaz gráfica, [`Core.setSystem()` - Ejecución de script *init*]
- *Reconocimiento de los eventos del sistema* para que el Editor de Eventos ofrezca sólo eventos válidos en la lista de eventos de entrada que el usuario selecciona. [`Core.getEventsInSystem()`]
- *Clasificación de elementos gráficos* de acuerdo a la información contenida en XML del archivo `.odg`. Se chequean errores relacionados con la gramática Statecharts simples, como elementos gráficos no reconocidos, errores en nombre de estados temporizados, tipo inválido de conector, etc. [`Core.construct_SG()`]

#### **● Etapa de validación del sistema y clasificación de elementos**

Se aplican métricas de inclusión entre cada uno de los elementos gráficos para determinar relaciones entre ellos. Se clasifican los estados dependiendo del tipo de estado y los subestados que contienen.

- *Chequeo de referencias*: Chequea la existencia de *estados referencia*. Tiene como finalidad la de evitar que el Sistema tome a los *estados referencia* como *estados root* y se produzca el error gramatical de "múltiples estados root". [Sistema.checkReferences()]
- *Inclusión de elementos gráficos*: Realiza el cálculo de inclusiones entre estados, conectores y símbolos de inicio de transiciones por defecto. Se reduce al cómputo de métricas de inclusión y la relación *contiene-a* o es *padre-de*. Tiene que ver con inclusiones desde el punto de vista gráfico. Determina cual es el *estado root* del sistema. [Sistema.setInclusions()]
- *Establecimiento de relaciones entre estados, determinación de tipos*: Establece relaciones entre los estados Statecharts, se categorizan cada uno de estos de acuerdo a los subestados que contienen y se calcula su nivel de inclusión en la jerarquía de estados. Se chequean errores gramaticales más específicos relativos a los *estados referencia* y *estados instancia*. [Sistema.statesRelations()]
- *Establecimiento de conexiones entre estados*: Se registran las vinculaciones entre estados/conectores a través de las transiciones, se detectan las transiciones por defecto. Se chequean errores relativos al número y tipo de transiciones en estados iniciales (de las transiciones por defecto) y se detectan transiciones descolgadas. [Sistema.setConnections()]
- *Chequeo de conectores*: Chequeo de validez de los conectores controlando el número y etiquetado de transiciones entrantes y salientes de acuerdo al tipo. [Sistema.checkConnectors()]

#### ● **Etapas de simplificación del sistema**

En esta etapa se contruye un nuevo sistema, el denominado internamente en el Sistema como *sistema real*, a partir del *sistema gráfico*, estructura de objetos construida a partir de la representación del archivo `.odg`. El *sistema real* es una versión simplificada en lo que respecta a variedad de elementos Statecharts, en éste no existen los conectores, los estados referencia ni los estados parametrizados. A partir de este momento las transiciones son de estado a estado.

Mediante esta etapa se evita el tratamiento de las denominadas *compound transitions* (transiciones compuestas) mencionadas en [HN96], las cuales se constituyen a partir de uno o más segmentos de transición. Aquí, gracias a las transformaciones realizadas, las transiciones son punto-a-punto, es decir formadas por un único segmento que une estados, posibilitando así la simplificación del algoritmo de simulación en las etapas correspondientes al análisis y ejecución de las transiciones habilitadas.

- *Resolución de referencias*: Se suprimen los estados referencia del modelo, toda interacción de los estados referencia con el resto de los estados se traslada a los estados referenciados. [Sistema.resolveReferences()]
- *Apertura de estados parametrizados*: Eliminación de *estados template* (cubos) y *estados instancia* (diamantes). Se crean todos los estados faltantes

dentro de los estados parametrizados para cubrir el rango de valores del parámetro especificado, se resuelven las operaciones de los estados instancia para definir nuevas transiciones. [Sistema.openParametrizedStates(), Sistema.clearParametrizedChild(), Sistema.clearDiamonds()] - Ver el documento adjunto "Tratamiento del OpenDocument" sección *Reglas de Reescritura*.

- *Eliminación de conectores*: Se quitan del modelo los conectores y se reemplazan las transiciones que unen por transiciones con etiquetas más ricas y de semántica equivalente. Dos transiciones unidas por un conector dan lugar a una única transición con una etiqueta conformada según las reglas definidas en la sección "Elementos de Statecharts considerados / Transiciones compuestas". [Sistema.removeConnectors()]
- *Descubrimiento de transiciones duplicadas*: Luego del paso anterior, la construcción de nuevas transiciones puede dar lugar a transiciones salientes de un mismo estado, con diferentes destinos, pero igual etiquetado, lo cual es una situación de potencial no determinismo. Este tipo de errores generalmente no se detectan durante la edición del modelo a simular, por tal motivo este paso debe existir. [Sistema.checkTransitions()]

#### ● Etapa de inicialización del sistema

Esta etapa corresponde al establecimiento de los estados activos del sistema. Esto se realiza en dos etapas, en la clase que implementa el algoritmo (`Algorithms`) se mantiene actualizada la lista de los estados activos, la cual está almacenada en la clase `Sistema`. Luego con la lista ya conformada, en la clase `Core` se plasman en XML dichos estados activos empleando las funciones provistas por `JDOMWorker`. Se traducen las activaciones del estado real al estado gráfico para contemplar referencias y estados parametrizados.

- Se precisa el tipo de algoritmo para la simulación. Actualmente sólo uno está implementado pero con esto se da lugar a que el usuario, usando controles gráficos, pueda elegir entre diferentes tipos de algoritmos. [`Core.initSystem()`].
- Partiendo desde el *estado root* se van agregando estados a la lista de estados activos. Para ello, desde un estado se analizan las transiciones habilitadas y se les calcula el *scope* o ámbito (es el *estado or* de mayor nivel en la jerarquía de estados que contiene a los estados destino y origen, o los estados padres o ancestros de éstos, de la transición). Luego se quitan aquellas de menor prioridad (menor ámbito) y por último se chequea si existen situaciones de no determinismo. A los nuevos estados alcanzados se les aplica una secuencia de pasos idéntica. Se hace un tratamiento diferencial entre *estados or* y *estados and*, en los primeros se estudian las transiciones por defecto, mientras que en los últimos se realizan las activaciones de todos los estados ortogonales subordinados. [`Algorithms.setActiveStates()`]

#### ● Etapa de ejecución de eventos

Esta etapa abarca las etapas 2 y 3 del algoritmo descrito en [HN96].

El Sistema consume secuencialmente la lista de eventos definida por el usuario mediante el Editor de Eventos. Los eventos se dividen en tres clases

distintivas: los lapsos de tiempo o *eventos de tiempo*, los mandatos para graficar el status actual del sistema y los *eventos comunes*.

Desde la clase `Statecharts` se determina el tipo de evento [`Statecharts.buttonIniciarActionPerformed()` - respuesta a la acción de iniciar la simulación con los controles de la pantalla principal del Sistema]. Una vez identificado cada evento se invocan las funciones correspondientes de la clase `Core` [`Core.sendTimeEvent()`, `Core.drawSystem()` y `Core.sendCommonEvent()`], desde estas funciones se invocan las funciones de la clase `Algorithms` que corresponden a la ejecución de eventos de tiempo y comunes.

- *Ejecución de evento de Tiempo*

Todo evento de tiempo de la forma  $\{n\}$ ,  $n > 0$  se desmenuza en  $n$  eventos de tiempo  $\{1\}$  y se ejecutan uno detrás de otro con la función [`Algorithms.tinyStepSystem()`].

La ejecución de este tipo de eventos tiene sentido sólo si en el modelo existen estados temporizados, en caso contrario estos eventos son ignorados por el Sistema.

Desde [`Core.sendTimeEvent()`] se invoca, pasándole un entero que representa un lapso de tiempo, a la función [`Algorithms.stepSystem()`], la cual ejecuta el evento. A su finalización [`Core.sendTimeEvent()`] realiza acciones para la generación de un nuevo status del sistema [`Core.doStep()`] y se crea un archivo XML con los estados activos alcanzados en el sistema [`Core.setActiveStates()`] debido a los cambios ocurridos por la posible ejecución de transiciones producidas por este evento. Estos últimos dos pasos se realizan siempre por si el próximo evento corresponde al mandato de graficar el status del sistema.

Los estados temporizados, mediante la clase `Temp_info`, poseen una variable interna denominada `stayTime` iniciada en 0 que se incrementa en uno [`Temp_info.tic()`] en cada evento de tiempo  $\{1\}$ . Esta variable es útil para determinar si no debe abandonarse el estado debido a que su cota inferior aun es verdadera [`Algorithms.computeEnabledTransitions()`] o si debe obligatoriamente tomarse la transición *timeout* al sobrepasarse la cota superior [`Algorithms.tinyStepSystem()`]. Cuando el estado se torna nuevamente inactivo `stayTime` vuelve a ser 0.

No hay un tratamiento general del tiempo del sistema, en cada estado temporizado se hace el cómputo de los tiempos de permanencia como se comentó anteriormente.

### **Ejecución**

En esta función se chequean los estados activos para identificar cuales son estados temporizados [`Algorithms.getActiveStates()`, `Algorithms.isTemporized()`]. Los estados temporizados pueden tener dos cotas de tiempo, en este paso se chequea la cota superior para determinar su validez y eventualmente establecer como habilitada a la transición *timeout* [`Algorithms.addEnabledTransition()`], es decir sumar esta transición a la lista *enabledTransitions* de la clase `Algorithms`.

Luego de que todos los estados temporizados son analizados y actualizados tiene lugar una serie de pasos generales, ver "Ejecución de Eventos - Pasos

Generales” más abajo.

- *Graficar status del sistema*

La función [Core.drawSystem()] ejecuta el script *draw* para depositar en el directorio temporal del sistema operativo los archivos correspondientes a los distintos status que el sistema alcanza. Construye un nuevo archivo .odg a partir de un archivo content.xml generado por la ejecución del evento previo.

- *Ejecución de evento Común*

La función [Core.sendCommonEvent()] le envía una cadena de texto a [Algorithms.stepSystem()] que indica el evento a ejecutar, y al igual que [Core.sendTimeEvent()] realiza todo lo necesario para poder graficar el status del sistema si es solicitado en el paso siguiente.

Por su parte [Algorithms.stepSystem()] realiza un cálculo de las transiciones habilitadas con [Algorithms.computeEnabledTransitions()] para ver si el evento recibido pertenece a alguna transición saliente del conjunto de los estados activos. Además del evento en sí debe analizarse también, si existe la condición que conforma la etiqueta de la transición, para ello ejecuta a [Algorithms.isEnabled()]. Esta última función es la que lleva a cabo la evaluación de las condiciones de las etiquetas. Toda transición que al ocurrir este evento se encuentre habilitada, y cumpla sus condiciones, se agrega a una lista denominada *enabledTransitions*.

Finalmente se ejecuta una serie de pasos generales, ver próxima sección.

- *Ejecución de eventos - Pasos Generales*

Al realizar la ejecución de los eventos tiene lugar una serie de pasos generales e independientes del tipo de evento. Estos pasos están agrupados en la función [Algorithms.step()] y constan de las siguientes funciones y acciones principales:

- Se calcula el *scope* o ámbito de cada transición habilitada. [Algorithms.setScopeEnabledTransitions()]
- Se eliminan las transiciones habilitadas de menor ámbito (prioridad). [Algorithms.clearEnabledTransitions()]
- Se chequea la existencia o no de *no determinismo*, situación que se da cuando dos transiciones de igual scope están habilitadas en un mismo estado. [Algorithms.checkNonDeterminism()]
- [Algorithms.transitionExternalEvents()] es la función que ejecuta los eventos, recorre cada una de las transiciones habilitadas y produce la información que el sistema precisa para conocer qué estados y subestados fueron abandonados [Algorithms.leaveSubstates()] y retorna una lista con los nuevos estados activos. Reune todas las acciones de las transiciones que se ejecutaron. En la clase **Algorithms** a las acciones se las agrupa en la lista **internalEvents**.
- Se establecen los nuevos estados activos en el sistema. [Algorithms.setActiveStates()]
- Se ejecutan las acciones (**internalEvents**) como si fuesen eventos enviados por el usuario antes de ejecutar el próximo evento externo indicado por el usuario. [Algorithms.executePendingActions()]

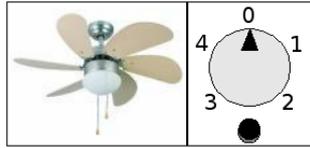


Figura 30: Ventilador y su interruptor

### **Salidas**

- Cartel que informa que la simulación de desarrolló exitosamente o un mensaje de error detallando el motivo.
- Eventualmente archivos con los *distintos* status del sistema.
- Archivo de *log* de la simulación.

## **7. Caso de estudio**

En este apartado se analiza un caso de estudio junto con sus modelos en Statecharts propuestos, se remarcan los aspectos del lenguaje comprendidos y su posterior simulación analizando los resultados obtenidos.

### **7.1. Ventilador de cuatro velocidades**

Este primer ejemplo muestra características simples del diseño y simulación de modelos basados en Statecharts.

#### **7.1.1. Descripción**

Un ventilador de techo cuenta con un interruptor para su manejo que consiste en un botón para el comando de la luz incorporada y una perilla giratoria para aumentar o reducir la velocidad de rotación de las aspas.

Al presionar el botón la luz se enciende, si se presiona nuevamente la luz se apaga, y así sucesivamente.

La perilla puede girarse a la derecha recorriendo las velocidades 0 (aspas detenidas) hasta la 4 (máxima velocidad); como contrapartida el giro a la izquierda disminuye la velocidad de rotación. La luz es independiente del giro de las aspas, pudiendo encontrarse encendida o apagada más allá del estado del ventilador en sí.

En la Figura 30 se muestra una imagen del ventilador junto con el interruptor previamente descripto.

Se pretende controlar el comportamiento de este ventilador mediante un software de control que asegure el funcionamiento correcto del mismo, por ejemplo que a igual cantidad de giros de la perilla en una dirección exista un aumento

de velocidad de rotación de las aspas proporcional, es decir, en el caso de girarse a la derecha dos veces la perilla, que el ventilador gire a velocidad 2. Otra propiedad a controlar y verificar mediante la simulación es que el pulsado del botón produzca una secuencia alternada de encendidos y apagados de la luz.

### 7.1.2. Modelado

Siguiendo la documentación [ZJ96] se modela el denominado *conocimiento del dominio*, es decir todo aquello que sabemos acerca del ventilador y del dispositivo para controlarlo, y la *especificación*, que expresa cómo el *software de control* debe restringir o administrar los eventos recibidos para que el ventilador funcione de manera correcta.

Designaciones:

- Se presiona el botón  $\approx$  *Push*
- Se gira una vez la perilla a la izquierda  $\approx$  *left*
- Se gira una vez la perilla a la derecha  $\approx$  *right*
- El sistema apaga la luz  $\approx$  *off*
- El sistema enciende la luz  $\approx$  *on*
- El sistema disminuye la velocidad de las aspas  $\approx$  *slow*
- El sistema aumenta la velocidad de las aspas  $\approx$  *fast*

Tabla de control y visibilidad

Evento	EC-S	MC-S
<i>Push</i>	X	
<i>left</i>	X	
<i>right</i>	X	
<i>off</i>		X
<i>on</i>		X
<i>slow</i>		X
<i>fast</i>		X

Las siglas presentes en el encabezado de las columnas dos y tres tienen el siguiente significado:

- EC: *Environment Controlled*
- MC: *Machine Controlled*
- S: *Shared*

Durante la manipulación de la perilla y/o el botón, el software de control captura los eventos y los traduce a eventos internos que modifican el estado del ventilador.

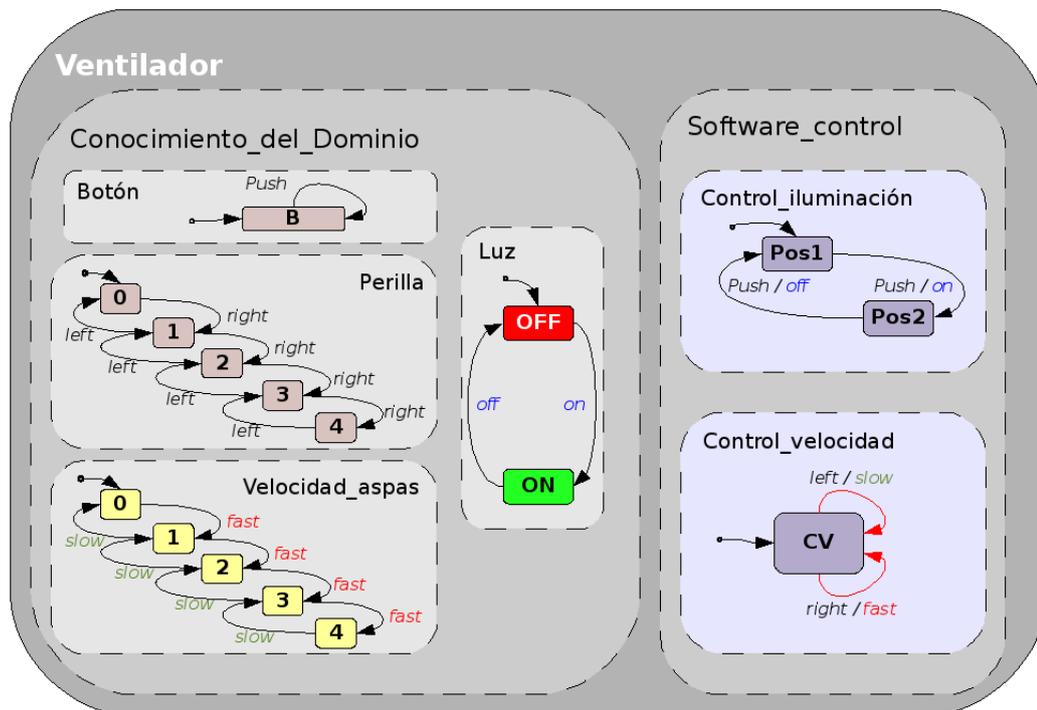


Figura 31: Ventilador: Representación 1

En la Figura 31 se presenta un *estado and* denominado **Ventilador** en el existen dos estados ortogonales, el **Conocimiento\_del\_Dominio** y el **Software\_control**, los cuales, respectivamente, modelan el estado del ventilador junto a los componentes del interruptor y el software de control en sí.

Cada uno de estos estados también son *estados and*. El estado **Conocimiento\_del\_Dominio** consta de cuatro subestados, de los cuales **Perilla** y **Botón** corresponden a los controles del interruptor y muestran cuales son los eventos posibles sobre los mismos. Por otra parte los subestados **Velocidad.aspas** y **Luz** reflejan el estado del ventilador. Puede verse a partir de estos últimos estados y sus transiciones por defecto que el ventilador inicialmente está detenido (velocidad 0) y con su luz apagada (**OFF**). Como comentario, en el estado **Perilla** los subestados numerados del 0 al 4 representan posiciones, mientras que en **Velocidad.aspas** representan velocidades.

El estado **Software\_control** está conformado por dos estados **Control.iluminacion**, que controla la luz, y **Control.velocidad** que controla la velocidad de rotación de las aspas.

Otra manera de modelar este caso es haciendo uso de los estados parametrizados para la perilla y la velocidad de las aspas. La Figura 32 lo ejemplifica, reemplazando estados por estados parametrizados con semántica

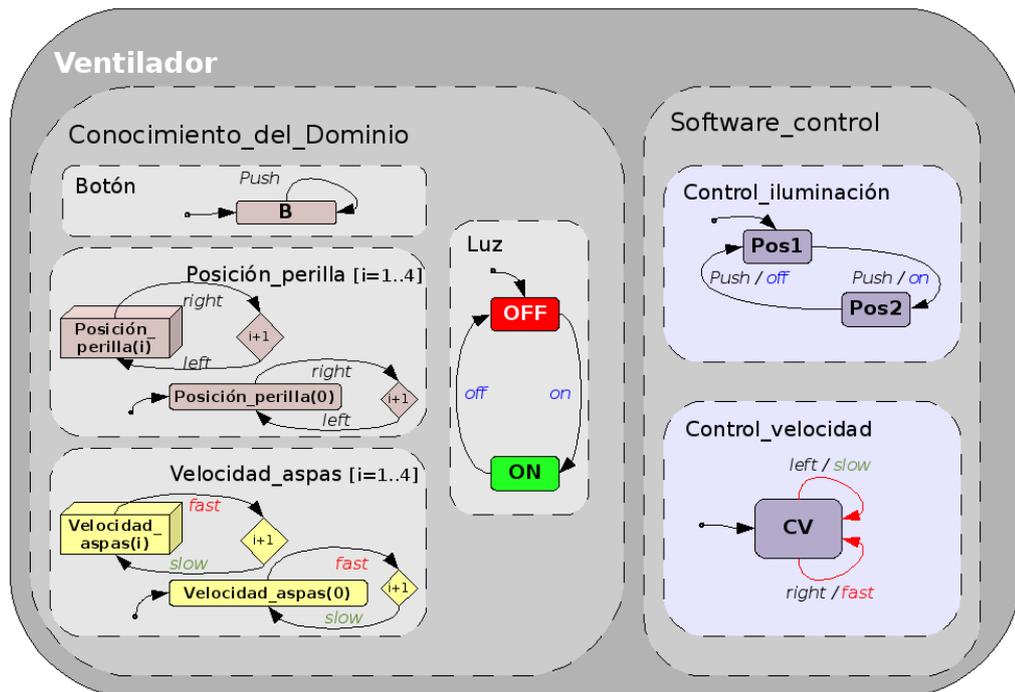


Figura 32: Ventilador: Representación 2

idéntica.

### 7.1.3. Simulación

Para realizar la simulación se efectúan los siguientes pasos:

- Ejecutar el Sistema, como se describe en la sección **Manual de Usuario - Ejecutar el Sistema**.
- Seleccionar el modelo denominado *caso1\_modelo1.odg* o *caso1\_modelo2.odg* de la *Suite de Modelos Statecharts*, ver **Manual de Usuario - Elegir el sistema/modelo**.
- Cargar con el Editor de Eventos del Sistema siguientes eventos:  
 [GRAFICAR.STATUS]  
 Push  
 right  
 right  
 right  
 right  
 right  
 right  
 [GRAFICAR.STATUS]  
 left  
 left  
 Push

[GRAFICAR\_STATUS]

En la *Suite de Modelos Statecharts* existe un archivo denominado `caso1.evs` con los eventos listados.

La tabla siguiente muestra los diferentes *status* donde se encuentra el modelo, los eventos recibidos y el estado del ventilador.

Status	Evento	Descripción
0	-	<i>Estado inicial.</i> Ventilador sin girar y luz apagada
	[GRAFICAR_STATUS]	Graficando el <i>estado inicial o Status 0</i>
1	Push	Ventilador sin girar y luz <i>encendida</i>
2	right	Ventilador en velocidad 1 y luz <i>encendida</i>
3	right	Ventilador en velocidad 2 y luz <i>encendida</i>
4	right	Ventilador en velocidad 3 y luz <i>encendida</i>
5	right	Ventilador en velocidad 4 y luz <i>encendida</i>
6	right	Ventilador en velocidad 4 y luz <i>encendida</i> , evento sin efecto
	[GRAFICAR_STATUS]	Graficando el <i>Status 6</i>
7	left	Ventilador en velocidad 3 y luz <i>encendida</i>
8	Push	Ventilador en velocidad 3 y luz <i>apagada</i>
	[GRAFICAR_STATUS]	Graficando el <i>Status 8</i>
9	left	Ventilador en velocidad 2 y luz <i>apagada</i>
10	Push	Ventilador en velocidad 2 y luz <i>encendida</i>
	[GRAFICAR_STATUS]	Graficando el <i>Status 10</i>

- Al finalizar la simulación se encontrarán en el directorio temporal los archivos de los distintos *status*: *nombre\_modelo\_status\_0.odg*, *nombre\_modelo\_status\_6.odg*, *nombre\_modelo\_status\_8.odg* y *nombre\_modelo\_status\_10.odg* y el archivo de *log*.

Claramente puede observarse que en todo momento la velocidad de rotación de las aspas se condice con la posición de la perilla del interruptor, además también se evidencia que una secuencia de pulsaciones del botón genera una alternancia estricta entre encendidos y apagados de la luz.

Se introduce ahora una restricción de tiempo al modelo. Se desea que la luz del ventilador se apague automáticamente al cumplirse 14hs de encendida sin interrupción. Veremos, mediante una simulación, si luego de transcurrido dicho lapso de tiempo, esta nueva restricción se respeta. En primer lugar se aplican modificaciones al modelo (se empleará la primer representación) obteniendo modelo de la figura 33.

En el modelo el único estado que sufrió cambios fue ***Control.iluminacion***, correspondiente al software de control. Obsérvese el estado ***Pos2***, éste es ahora un *estado temporizado* en el cual el sistema se encuentra cada vez que la luz está encendida. Posee una cota superior de tiempo, a partir de la cual inmediatamente al transcurrir 14 unidades de tiempo (horas en este caso) superando el valor 13, se ejecuta la transición *timeout* apagando la luz del ventilador.

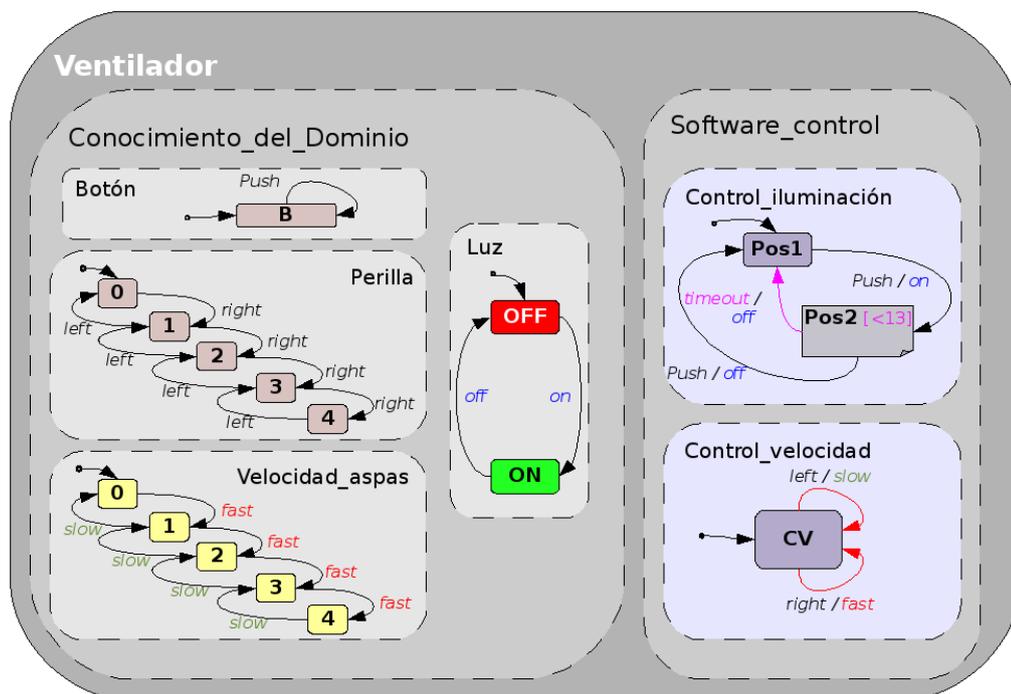


Figura 33: Ventilador: Restricción de tiempo

En el modelo, una vez más abstrae ciertas características del sistema, nada sabe éste que se están considerando horas y el simulador, al interactuar con el modelo, sólo sabe que al recibirse la decimocuarta unidad de tiempo debe abandonarse el estado temporizado.

En la tabla que sigue a continuación se muestran los eventos que permiten verificar mediante simulación la restricción temporal impuesta.

Status	Evento	Descripción
0	-	<i>Estado inicial.</i> Ventilador sin girar y luz apagada
	[GRAFICAR_STATUS]	Graficando el <i>estado inicial</i> o <i>Status 0</i>
1	Push	Ventilador sin girar y luz <i>encendida</i>
2	13	Ventilador sin girar y luz <i>encendida</i>
	[GRAFICAR_STATUS]	Graficando el <i>estado inicial</i> o <i>Status 2</i>
3	1	Ventilador sin girar y luz <i>apagada</i>
	[GRAFICAR_STATUS]	Graficando el <i>estado inicial</i> o <i>Status 3</i>

Al finalizar la simulación se obtendrán los archivos

- Archivo de stauts *nombre\_modelo\_status-0.odg*
- Archivo de stauts *nombre\_modelo\_status-2.odg*
- Archivo de stauts *nombre\_modelo\_status-3.odg*

- Archivo de *log nombre.modelo.log*

Analizando los resultados conseguidos puede concluirse que la nueva restricción sobre el estado de la luz es implementada correctamente por el software de control.

## 8. Conclusiones y trabajos futuros

El espíritu de este trabajo final fue el de ofrecer una herramienta operativa para la simulación de modelos Statecharts, colaborando con el dictado de las clases sobre sistemas reactivos de la materia *Análisis de Sistemas* de la carrera Licenciatura en Ciencias de la Computación de la Universidad Nacional de Rosario.

Originalmente este trabajo iba a tener categoría de *prototipo* y finalmente se convirtió en una aplicación que contempla gran parte del lenguaje Statecharts, sin embargo tiene mucho por delante.

Con el Sistema de Simulación es posible simular una amplia variedad de modelos de diversa complejidad, los suficientes como para que el alumnado pueda probar los sistemas desarrollados en las clases prácticas. No hay muchas limitaciones respecto a lo que se puede simular aunque existen casos particulares no soportados debido a su complejidad, como son los *estados parametrizados multinivel* (donde cada uno de los estados instancia de un estado parametrizado es a su vez un estado parametrizado).

El proceso de simulación es simple y consta de pocos pasos. Es un proceso ágil y el tiempo que consume (algunos segundos, dependiendo de la complejidad del modelo) es producido en mayor porcentaje no por cálculos propios del algoritmo de simulación sino por la generación de los archivos de salida, pues obviamente esto precisa acceso a disco. En futuras versiones se podrían definir distintos niveles de *logging*.

La selección de las herramientas utilizadas para la implementación fue acertada, Java se complementa muy bien con XML gracias a la librería JDOM. Open/Star Office Draw cumplió con creces el rol de “editor” de modelos, fue la herramienta indicada luego de probar varias herramientas para tal fin, las cuales fueron siendo descartadas por imponer un trabajo excesivo para el usuario en la confección de los modelos y en el análisis de los archivos XML generados para el desarrollo.

Se pudo emplear en general sólo tecnología de libre distribución y fuente abierta, lo que constituía un objetivo personal.

Con la simulación de un modelo es posible, como fue demostrado, determinar el cumplimiento o no de ciertas propiedades sobre el mismo.

## 8.1. Trabajos futuros

La siguiente lista presenta algunas funcionalidades que pueden agregarse al Sistema en futuras versiones:

- Este Sistema de simulación diferencia entre los estados correspondientes al entorno (conocimiento del dominio) y el software de control. Contando con dicha segregación es posible evitar el *broadcasting* entre estos dos *submodelos* y así poder emplear nombres de eventos idénticos tanto para el entorno como para el software que controla el sistema.
- Ofrecer una sintaxis más rica para las condiciones y acciones soportadas para el etiquetado de las transiciones, agregando nuevos conectivos lógicos.
- Soportar varios algoritmos de simulación diferentes. El usuario podría escoger cual aplicar en cada proceso de simulación. Un algoritmo interesante a sumar es aquel en el que el tiempo no lo induce el usuario a través de lapsos de tiempo explícitos sino que es tomado del sistema operativo.
- Editor incorporado para edición y visualización de modelos que se ajuste a la descripción de OpenDocument. Un editor integrado implica que la construcción de modelos respeten de manera natural la gramática de Statecharts, trasladando el control de errores básicos que actualmente hace el Sistema al editor, prohibiendo todo aquello que vaya en contra de dicha gramática. Este editor debería poder realizar las gráficas mediante *drag and drop* y ofrecer algunas de las características de formateo de elementos (colores, formato de texto, etc.) que ofrece Open/Star Office Draw y contar con la funcionalidad de poder abrir los archivos de status generados para su inspección.
- Admitir los conectores históricos  $H$  y  $H^*$ , [HN96]. Estos conectores describen la "memoria" de un *estado or*, recordando el último subestado visitado de dicho estado. Gráficamente son representados mediante círculos y existen dos versiones:
  - Conector  $H$ : Se denomina *conector histórico - history connector*. Recuerda el último subestado visitado del super estado que lo contiene.
  - Conector  $H^*$ : Se conoce como *conector histórico profundo - deep history connector*. Posee la misma función que el conector  $H$ , pero además se aplica a cada uno de los subestados del super estado que lo contiene.

## Bibliografía

- [BJ] Britt, James. *Interactive Java & JDOM online tutorial*.  
JDOM Tutorial - TopXML, <http://www.topxml.com>
- [HA87] Harel, David. *Statecharts: A Visual Formalism for Complex Systems*.  
Elsevier Science Publishers B.V. (North-Holland)  
Science of Computer Programming, Vol. 8, pp. 231-274, Junio 1987
- [HA88] Harel, David. *On Visual Formalism*.  
Communications of the ACM, Vol 31, No 5, Mayo 1988
- [HN96] Harel, David; Naamad, Amon. *The STATEMATE Semantics of Statecharts*.  
ACM Transactions on Software Engineering and Methodology  
Vol. 5, No. 4, pp. 293-333, Octubre 1996
- [ZJ96] Zave, Pamela; Jackson, Michael *Four Dark Corners of Requirements Engineering* ACM Transactions on Software Engineering and Methodology  
Vol. 6, No. 1, pp. 1-30, Enero 1997
- [HP99] Harel, David; Politi, Michal. *STATEMATE MAGNUM - Modeling Reactive Systems with Statecharts: The Statemate Approach*.  
I-Logix Inc, 1999
- [MB01] McLaughlin, Brett *Java and XML, 2nd Edition*.  
O'Reilly
- [EB02] Eckel, Bruce. *Thinking in Java - Third Edition*.  
MindView Inc., 2002
- [LL02] Lammport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* Addison-Wesley Professional, 2002
- [GHJV03] Gannam, E.; Helm, R.; Johnson, R.; Vilssides, J. *Patrones de Diseño. Elementos de software orientado a objetos reutilizable*.  
Addison Wesley, 2003
- [RE03] Ray, Erik. *Learning XML - Second Edition*.  
O'reilly, September 2003.
- [STMPB] STATEMATE I-Logix. Product Brochure [http://www.spectrum-systems.com/vendors/telelogic/statemate\\_brochure\\_v4.pdf](http://www.spectrum-systems.com/vendors/telelogic/statemate_brochure_v4.pdf) 2004
- [UML] UML - Unified Modeling Language <http://www.uml.org/>
- [CM06] Cristiá, Maximiliano. *Statecharts Parametrizados*.  
Materia Análisis de Sistemas, Licenciatura en Ciencias de la Computación  
FCEIA - Universidad Nacional de Rosario - 2006
- [ILTL] Telelogic Acquires I-Logix, Secures Leading Position in Embedded Market <http://www.embeddedstar.com/press/content/2006/3/embedded19695.html>  
Junio 2006
- [RRRT] IBM Rational Rose RealTime *A guide for evaluation and review*,  
<http://www.ibm.com/developerworks/rational/library/622.html>.

- [VST] visualSTATE <http://www.iar.se/website1/1.0.1.0/371/1/index.php>
- [RHPS] Rhapsody [http://www.ilogix.com/products/rhapsody/rhap\\_inc.cfm](http://www.ilogix.com/products/rhapsody/rhap_inc.cfm)
- [LBVW] LabView <http://www.ni.com/labview/statechart>
- [EDIT] An Interactive Editor for the Statecharts Graphical Language  
<http://www1.cs.columbia.edu/~sedwards/sc/index.html>
- [CHSM] CHSM - Concurrent Hierarchical State Machine  
<http://chsm.sourceforge.net>
- [JDOM] JDOM <http://www.jdom.org>
- [WWWC] World Wide Web Consortium <http://www.w3.org>
- [DOM] DOM - Document Object Model <http://www.w3.org/DOM>
- [SAX] SAX - Simple API for XML <http://www.saxproject.org>
- [XRCS] Xerces Java Parser <http://xerces.apache.org/xerces-j>
- [HA07] Harel, David *Statecharts in the Making: A Personal Account*.  
The Weizmann Institute of Science, Junio 2007
- [MC07] Cristiá, Maximiliano *Comentarios sobre diseño de software* Materia In-  
geniería de Software, Licenciatura en Ciencias de la Computación  
FCEIA - Universidad Nacional de Rosario - Julio 2007