FACULTAD DE INFORMATICA
UNLP

TESINA DE GRADO

TITULO: Entorno abierto de desarrollo y ejecución de simulaciones discretas para análisis evolutivo de entidades mediante persistencia
AUTORES: Ariel Jorge Lira – Gonzalo Luján Villarreal
DIRECTOR: Dr. Silvia Gordillo
CODIRECTOR: Ing. Marisa R. De Giusti

CARRERA: Licenciatura en Informática - Licenciatura en Sistemas

Resumen.

Las herramientas de simulación a eventos discretos ofrecen una amplia gama de opciones para realizar análisis detallados en base a los resultados arrojados luego de las corridas de cada simulación. En este trabajo proponemos una alternativa en la que se persisten todas las entidades del modelo simulado a medida que se van produciendo cambios en las mismas, registrando la evolucion de cada entidad dentro del modelo. Lo anterior sienta las bases para el desarrollo de nuevas herramientas de análisis que permitan conocer en un alto nivel de detalle la evolución de la simulación como un todo y de cada una de las entidades por separado. El desarrollo, basado en el lenguaje de simulación GPSS, consta de la implementación de todo el modelo que éste lenguaje utiliza para ejecutar simulaciones y de un conjunto de entidades básicas que permiten crear verdaderas simulaciones. Este trabajo también incluye una herramienta para crear y ejecutar simulaciones, y conocer las historias de las entidades involucradas en las mismas.

Líneas de Investigación

Sistemas y Modelos
Simulaciones a eventos discretos
Lenguajes de programación orientados a bloques.
Bases de datos relacionales y mapeos Objeto-Relacional

Trabajos Realizados

Estudio sobre simulaciones a eventos discretos
Análisis del modelo de ejecución de simulaciones de GPSS
Estudio de herramienta para persistencia de objetos
Estudio de herramientas de análisis de simulaciones propias de GPSSw.
Estudio de metodologías de generación de interpreters

Conclusiones

Generar un entorno mínimo apto para realizar simulaciones discretas es una tarea compleja que requiere la implementación de una gran cantidad de entidades propias de las simulaciones. Ejecutar estas simulaciones mientras son persistidas en una base de datos acarrea un alto costo en cuanto al tiempo total requerido. A pesar de esto, las posibilidades que esta aproximación ofrece, especialmente orientadas al desarrollo de herramientas de análisis, hacen que el esfuerzo y la penalización por "tiempo extra" valgan la pena. Adicionalmente, contar con una herramienta de simulación de código abierto puede ser muy útil para la enseñanza dentro éste área, ya que permite estudiar en un nivel de detalle muy profundo el intrincado funcionamiento de las entidades de GPSS en un lenguaje de programación familiar para la mayoría de los estudiantes.

Trabajos futuros

Continuar incorporando entidades de GPSS para completar el modelo de entidades definido.
Definir nuevas entidades no contempladas por el modelo de GPSS y agregar nueva funcionalidad a las entidades existentes.
Desarrollar nuevas y mejores herramientas de análisis evolutivo que aprovechen toda la potencialidad que esta aproximación ofrece.
Mejorar la usabilidad del entorno de programacion definido, aplicando tecnicas y herramientas open source existentes de Eclipse.

Agradecimientos:

Queremos agradecer a nuestras familias por brindarnos su apoyo y comprensión en todo momento, permitiéndonos abocarnos de lleno a este desarrollo. Junto a nuestras familias, queremos agradecer a otras personas que han hecho posible la finalización de este trabajo en el tiempo previsto:

A Germán Biagioli, que supo darnos una gran mano de amigo en el momento que más lo necesitamos, y a Virginia Rocco por su participación e interés.

A Silvia Gordillo y a Marisa De Giusti, directora y co-directora del trabajo, por su excelente predisposición y constante ayuda en todo momento y en todo lugar (literalmente hablando), su preocupación y sus consejos.

A Javier Bazzocco, profesor de la Facultad de Informática e investigador del LIFIA, por el tiempo que nos ha brindado, por su paciencia y los consejos que nos ayudaron a realizar este trabajo.

Índice de contenido

Capítulo1. Introducción

Simular comportamientos de sistemas no es una idea nueva. Ya por el año 1940, Von Neumann y Ullman dieron los primeros pasos en este area en el marco del proyecto Monte Carlo, simulando un flujo de neutrones para la construcción de la bomba atómica. En aquel momento, no existían técnicas especificas ni mucho menos lenguajes de programación, pero a pesar de esto se pudieron simular estos comportamientos. Años después, la NASA utilizó algunos de estos conceptos y técnicas y las aplicó en su programa Apollo. En esa epoca, el hardware de computadoras y el software existentes eran muy escasos, limitados y el acceso a los mismos estaba restringido a unos pocos organismos militares o instuticiones, pero ya entonces se vislumbraba cuan ventajosas podían ser estas nuevas técnicas y herramientas.

Décadas después, la situación ha cambiado de manera considerable. Las computadoras están al alcance de la mayoría de las personas; la potencia de los procesadores actuales, la cantidad de memoria disponible y el tamaño de los dispositivos de almacenamiento se multiplica año tras año, las plataformas de software y sistemas operativos son cada vez más complejos y ofrecen mayores facilidades para gestionar grandes programas de manera concurrente, y los lenguajes de programación han evolucionado diversificándose enormemente. Todas las áreas de estudio y ciencias actuales han sido favorecidas por esta expansión tecnológica, y el área de modelado y simulación de sistemas no es la excepción.

Pero, ¿qué es una simulación? Se denomina simulación al proceso de ejecutar un **modelo**, o sea la representación abstracta de un sistema descripto con un nivel de detalle suficiente pero no excesivo. Este modelo implica un conjunto de entidades, cada una con un estado interno, un conjunto de variables de entrada controlables y otras tantas no controlables, una serie de procesos que relacionan estas variables de entrada con las entidades y finalmente una o varias salidas, resultantes de la ejecución de estos procesos.

Esta definición, que ampliaremos apropiadamente en los siguientes capítulos, nos permite vislumbrar intuitivamente cuales son las principales tareas que implica realizar una simulación por computadora: se toma un sistema del mundo real y se estudia en detalle, se genera un modelo del sistema tomando todos los elementos interesantes (donde el término interesante se refiere a lo que nos interesa tomar y estudiar de nuestro modelo) incluyendo los procesos complejos que afectan a los elementos de nuestro sistema/modelo, y se lo ejecuta en una computadora, lo cual genera un conjunto de resultados que luego deberán ser evaluados apropiadamente. Cuando el modelo es lo suficientemente flexible puede ser utilizado para comparar muchas estrategias diferentes de operación, escenarios posibles, casos extremos o incluso situaciones inverosímiles.

La ejecución por computadora requiere el uso de algún lenguaje de programación que permita pasar nuestro modelo matemático y sus entidades a un conjunto de instrucciones que las computadoras puedan interpretar y ejecutar. Actualmente, existen varias decenas de lenguajes de programación que se utilizan en áreas de simulación. Podemos clasificarlos en dos grandes grupos:

- Lenguajes de propósito general (C/C++, Java, Pascal y derivados, C#/.Net, Ruby, Perl, entre otros): su principal ventaja radica en la flexibilidad que cualquiera de estos lenguajes ofrece al momento de programar simulaciones. Como principall desventaja, el programador debe programar todas y cada una de las entidades de cada sistema a ser simulado. Es tarea del programador también desarrollar herramientas de análisis y generación de reportes.
- Lenguajes de simulación (SIMULA, GASP, GERT, GPSS, SIMSCRIPT, SIMAN, SLAM, entre otros): su principal ventaja es el grado de abstracción con el que se programan los

modelos, ya que proveen herramientas para establecer una correlación directa entre las entidades del sistema que está siendo simulado y las entidades dentro de la simulación, tanto para sus características como para su estado interno, comportamiento e interacción con otras entidades. Como desventaja, la flexibilidad es mucho menor, ya que se cuenta con un conjunto acotado de instrucciones sin la ortogonalidad presente en un lenguaje de propósito general. Adicionalmente, si bien estos lenguajes no son extremadamente complejos, la curva de aprendizaje está presente y en algunas situaciones donde apremian los tiempos puede no ser la mejor solución.

La simulación de un modelo carece de utilidad si no se la puede analizar, lo cual significa el estudio de las interacciones entre las variables de entrada, entre las entidades del modelo y su peso en los valores de salida. Los lenguajes de programación de simulaciones proveen un gran abanico de herramientas para analizar estos resultados, para generar y ejecutar experimentos y para hacer análisis complejos (como por ejemplo Análisis de Varianza). Esta característica, junto a la abstracción antes mencionada, han determinado su éxito.

En este trabajo consideraremos Simulaciones a Eventos Discretos, esto quiere decir que el estado de las variables de los sistemas que están siendo simulados cambian en un conjunto de instantes de tiempo finito o infinito contable. Dicho de otra manera, los modelos discretos poseen un reloj interno que cuenta los instantes por los cuales la simulación va pasando; en cada uno de estos instantes, las entidades alteran su estado interno, se crean nuevas entidades y se eliminan otras.

Los objetos o entidades pertenecientes a una simulación de un modelo poseen un estado inicial y sufren una serie de cambios a lo largo del tiempo de simulación. Algunas de estas entidades existen desde el principio de la corrida y permanecen hasta que la misma finalice (entidades permanentes) y otras se crean y se destruyen a lo largo de la simulación (entidades temporarias). Cada entidad permanente posee un comportamiento determinado por su tipo y un uso específico representando objetos o entidades del sistema real que está siendo simulado, mientras que las entidades temporales representan a los objetos del sistema que se mueven y utilizan al resto de las entidades. Para facilitar la comprensión de estos conceptos, podemos ver a las entidades permanentes como servidores que proveen una funcionalidad específica y a las entidades temporales como clientes que utilizan a los servidores.

Las herramientas para simulación, en su gran mayoría, ofrecen una gran cantidad de datos estadísticos, reportes, gráficos y tablas una vez finalizadas las corridas. Esto es, como se dijo, sumamente importante para analizar los valores de salida de los modelos. Pero en muchas ocasiones es deseable conocer la evolución de las entidades a lo largo de los distintos momentos de simulación o incluso comparar la evolución de una o varias entidades entre distintas simulaciones. Esto es especialmente util para realizar el debug del código de la simulación ajustando el modelo de manera tan exacta como sea posible, pero también para estudiar la evolución de determinadas entidades de manera detallada y para buscar sucesos puntuales o la ocurrencia de algún evento específico.

Otro objetivo deseable es el almacenamiento de las entidades de las simulaciones junto con toda su historia. Las entidades podrán ser recuperadas y analizadas a posteriori ya sea para una simulación en particular o incluso para varias simulaciones. Este punto es particularmente complejo, ya que implica el almacenamiento de toda la simulación, lo cual incluye todas las entidades permanentes y temporales, el estado de todas las colas de la simulación, el reloj de simulación e incluso los bloques (que también son entidades permanentes); y esto debe hacerse en

cada cambio del reloj, para así saber exactamente que sucedió con cada una de las entidades en cualquier momento o incluso para concer su variación en un intervalo de tiempo dado.

El trabajo que aquí presentamos aborda dos grandes temáticas, y de cada una de ellas surge un objetivo específico:

- Desarrollo de un entorno de simulación: se debe proveer todo el funcionamiento básico para poder ejecutar simulaciones. Esto implica la gestión de entidades, el scheduling de transacciones, el manejo de colas y listas, la ejecución de bloques y el manejo del reloj de simulacion.
- Persistencia de simulaciones: las simulaciones deben ser almacenadas en un dispositivo no volátil para luego poder ser recuperadas. Esto implica guardar todos los cambios de todas las entidades en todos los instantes de tiempo de simulacion. La organización de todos estos objetos y el volumen de datos que esto genera para una simulación por más simple que sea es enorme, lo cual constituye un gran desafío de implementación.

Organización del documento.

En el capítulo 2 de este documento introduciremos el concepto de Modelos y Simulaciones, y veremos de manera muy resumida como funciona el lenguaje GPSS y que herramientas nos ofrece para desarrollar modelos y ejecutarlos. En el Capítulo 3 introducimos el concepto de Bases de Datos, de Bases de Datos Orientadas a Objetos y el mapeo Objeto-Relacional. A continuación, veremos algunos aspectos sobresalientes de la implementación JPOX de JDO, la cual hemos utilizado para desarrollar este trabajo. El capítulo 4 se centraliza en el desarrollo en sí, explicando en que consiste el modelo y cual es su alcance, así como también algunos aspectos de diseño que permitirán al lector extender el modelo provisto con nuevas entidades y bloques. Este capítulo también incluye una sección donde se explica la implementación del guardado (persistencia) de las simulaciones de manera conceptual. En el capítulo 5 presentamos un ejemplo simple y concreto, mostrando como correr una simulación pequeña para así ver como se generan y almacenan los objetos de acuerdo al modelo propuesto. El capítulo 6 tiene como objetivo la profundización en algunos detalles de implementación que consideramos particularmente interesantes. Esto incluye el Scheduler de Transacciones, la gestión de las cadenas, la persistencia del grafo de objetos completo, la jerarquia de entidades, el análisis de expresiones, entre otras. En el capítulo 7 realizamos algunas conclusiones sobre el trabajo y presentamos los problemas más sobresalientes y trabajos futuros, tanto para expandir el modelo como para solucionar estos problemas. Finalmente, en el capítulo 8 listamos la bibliografía utilizada a lo largo de todo el trabajo, que incluye principalmente al área de modelado y simulación como al área de bases de datos.

Capítulo 2. Simulación y Modelos

Sistemas, modelos y simulaciones.

La planeación e implementación de proyectos complejos en los negocios, industrias y gobierno son extremadamente costosas tanto en tiempo como en dinero; esto hace indispensable la realización de estudios preliminares que permitan asegurar su viabilidad y conveniencia. La simulación es una técnica basada en la modelización y la teoría de sistemas, que permite correr o ejecutar modelos y obtener resultados con muy bajo costo de inversión – especialmente si se lo relaciona con la inversión que requeriría el sistema real – y en relativamente poco tiempo. La realización del modelo del sistema a simular es un proceso que involucra un alto grado de abstracción, en el cual se toman solo las características más relevantes del sistema en cuestión y se realiza una descripción basada en tales características; de aquí, el modelo resultante no es una réplica exacta de la realidad, sino que es una versión incompleta pero mucho más facil de manejar. Gracias al dinamismo que caracteriza a los modelos, se pueden ajustar las entradas para estudiar el comportamiento del sistema en diferentes situaciones.

Como hemos mencionado, el proceso de correr un modelo es lo que se denomina ***simulación***. ***Thomas H. Naylor*** [WIL66] define Simulación *como "una técnica numérica para conducir experimentos en una computadora digital. Estos experimentos comprenden ciertos tipos de relaciones matemáticas y lógicas, las cuales son necesarias para describir el comportamiento y la estructura de sistemas complejos del mundo real a través de largos periodos de tiempo"*. Dentro de esta definición, se introducen los experimentos como objetivo de las simulaciones y las computadoras como herramienta básica para llevarlas a cabo. Naylor también utiliza el término "períodos de tiempo" en esta definición, lo cual nos da la idea intuitiva de simulaciones a lo largo de un tiempo determinado.

Como mencionamos en el capítulo 1, este concepto de simulación no es nuevo. El proyecto Monte Carlo incluyó un modelo simulado de flujo de neutrones, llevado a cabo por Von Neumann y Ullman en el año 1940 en el Laboratorio Nacional Los Alamos. Este proyecto sentó un conjunto amplio de métodos estadísticos para correr simulaciones, en los cuales se indica como se deben tomar las muestras, como hacer las repeticiones de los experimentos e incluso introducen conceptos de generación de números aleatorios y pseudo-aleatorios[ROB04]. En aquel momento, no existían técnicas especificas ni mucho menos lenguajes de programación, pero a pesar de esto se pudieron simular estos comportamientos. Años después, la NASA utilizó algunos de estos conceptos y técnicas y las aplicó en su programa Apollo (1961-1975) [LAU91]. El desarrollo del programa de simulación de Apollo se realizó en paralelo con el desarrollo del hardware de vuelo requerido para la creciente complejidad de la misión, evolucionando mientras las fases de las operaciones pasaban de Gemini a Apollo y mientras tanto Los vuelos Apollo progresaban de ser simples operaciones de exploración de los límites del planeta a operaciones de orbitación del espacio o incluso hasta el alunizaje, lo cual resultó en una enorme maduración en el alcance y capacidad de las herramientas de simulación que se desarrollaban por ese entonces. Las simulaciones incluían situaciones extremas durante el lanzamiento o aterrizaje, y diferentes escenarios que permitían estimar como se comportarían los sistemas bajo una gran cantidad de situaciones. Esta fuerte relación entre los simuladores de misiones y el Centro de Control de la Nasa marcaron una serie de avances muy significantes en pos de los métodos de simulación[NAS08.

En nuestra definición, la simulación es una técnica que se basa en la teoría de sistemas. Necesitamos entonces definir algunos términos que nos permitirán comprender mejor a las simulaciones:

Un sistema es un conjunto de partes organizadas funcionalmente para formar un todo conectado. Tomemos como ejemplo un astillero, el cual consiste de un conjunto de muchas partes o componentes tales como muelles, grúas, rutas de transporte, barcos, cargamentos y personas, y un conjunto de interacciones entre estos elementos que tambien forman parte de sistema.

El ***estado*** de un sistema viene determinado por el conjunto de variables necesarias para describirlo en cualquier instante temporal, recibiendo cada una de estas variables el nombre de variable de estado.

Los sistemas se pueden clasificar de acuerdo al modo en que las variables evolucionan en el tiempo. Los **Sistemas discretos** son aquellos en los que sus variables de estado cambian en un conjunto de instantes de tiempo contable (finito o infinito numerable), mientras que los **Sistemas continuos** son aquellos en los que sus variables de estado cambian de manera continua a lo largo del tiempo.

Estudiar un sistema permite establecer relaciones entre algunas de sus entidades, medir sus prestaciones, o bien predecir su comportamiento bajo ciertas condiciones nuevas. Este estudio se puede acometer de las siguientes formas:

- Realizando experimentos sobre el sistema. Estos experimentos sólo se pueden realizar cuando se dispone del sistema y cuando es posible alterar sus condiciones de funcionamiento. Por tanto, esta solución no es aplicable cuando los estudios se realizan antes de disponer del sistema, o bien cuando el coste de modificación de las condiciones de funcionamiento resulta muy elevado.

- Realizando experimentos sobre un modelo del sistema.

Un subsistema es un componente de un sistema total que puede ser tratado o bien como una parte del sistema total o bien de manera independiente. Nuestro sistema como un todo, que es de lo que nos encargaremos, depende de las relaciones internas entre los subsistemas y también de las relaciones externas que conectan al sistema con el medioambiente, esto es con el universo que está fuera.

Un Modelo es una representación de un sistema. Un requerimiento básico para cualquier modelo es que nos dé la posibilidad de describir al sistema con suficiente detalle de modo tal que provea predicciones válidas sobre el comportamiento del sistema. Más aún, las características del modelo deben corresponderse con algunas de las características del sistema modelizado.

Tipos de modelos

Modelos físicos. Estos modelos son muy usados en las industrias aeronáutica y del automóvil. En general, y dada la naturaleza del problema, este tipo de modelos tiene poco interés en la investigación de operaciones y en el análisis de sistemas.

Modelos abstractos. Estos modelos representan un sistema mediante un conjunto de relaciones cuantitativas y lógicas entre sus componentes, permitiendo estudiar cómo se comporta el modelo del sistema cuando cambia alguna de sus partes.

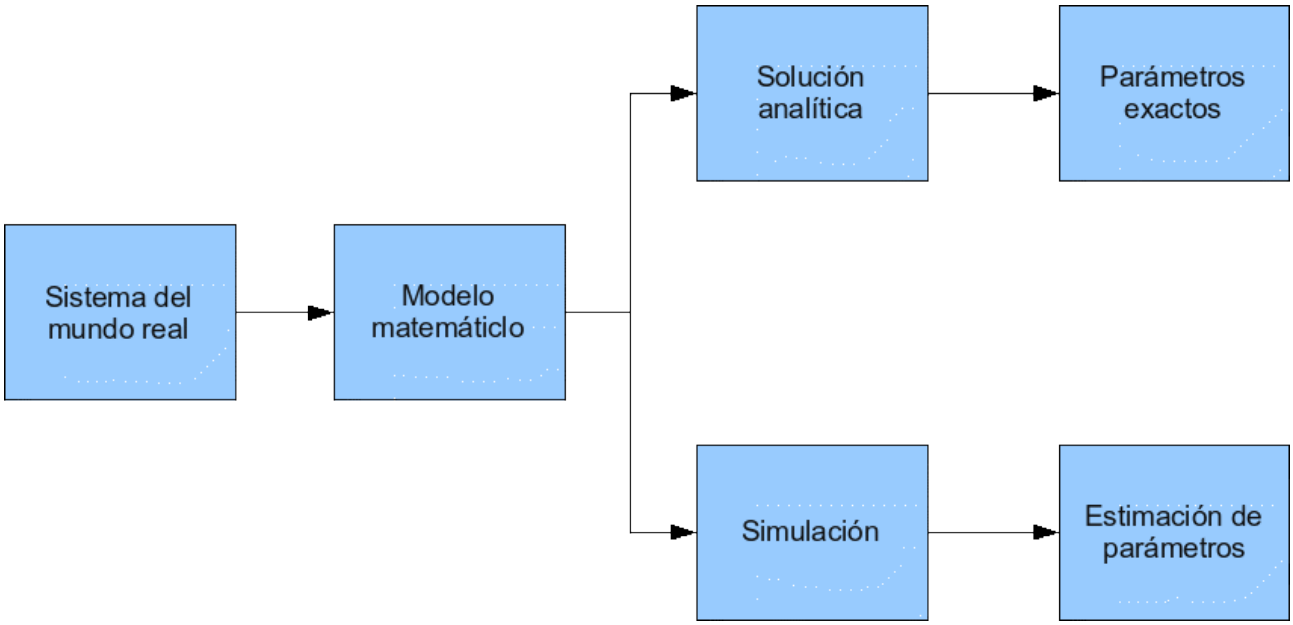

Figura 2.1. Vias de estudio de un sistema

Una vez definido un modelo matemático de un sistema, se debe realizar un primer estudio con el objetivo de determinar cómo usar este modelo para dar respuesta a las cuestiones de interés planteadas sobre el sistema que representa. Si el modelo es suficientemente sencillo, será posible obtener una **solución analítica** que relacione las magnitudes de interés. Caso contrario, el modelo debe estudiarse de forma *aproximada* recurriendo a su **simulación** (Figura 2.1). Tenemos nuevamente, dos posibles soluciones para estudiar los modelos:

• **Solución analítica**: supone analizar totalmente el modelo del sistema y obtener una solución que valdrá para todo momento y para obtener cualquier parámetro de interés. Si se puede obtener y si su coste computacional es asumible, se preferirán las soluciones analíticas. Sin embargo, la elevada complejidad de muchos sistemas reales imposibilita la obtención de modelos suficientemente ajustados con soluciones analíticas o, en caso de obtenerlas, la carga computacional que conllevan desaconseja su uso.

• **Simulación**: se recrea una o varias evoluciones temporales del modelo con el fin de estimar un conjunto de parámetros. Los modelos de simulación son modelos matemáticos que permiten obtener una estimación del comportamiento del sistema para una configuración determinada.

Tipos de modelos de simulación

Podemos clasificar los modelos de simulación de acuerdo a distintos criterios:

1. Según el instante temporal que representan:

• Estáticos: representan a un sistema en un instante determinado.

• Dinámicos: representan a un sistema que evoluciona a lo largo del tiempo.

2. Según la aleatoriedad de sus variables de estado:

• Deterministas: la representación del sistema no contiene ninguna variable de estado aleatoria. Tienen un conjunto conocido de entradas que dará por resultado una salida única.

• Estocásticos o aleatorios: la representación del sistema contiene al menos una variable de estado no determinista.

3. Según el modo en que evolucionan sus variables de estado:

• Discretos o de eventos discretos: si las variables de estado del modelo varían en un conjunto contable de instantes de tiempo.

• Continuos: si las variables de estado varían de modo continuo en función del tiempo.

En nuestro trabajo nos referiremos a los modelos de simulación **dinámicos, estocásticos y discretos**, que se denominan **modelos de simulación de eventos discretos**.

Ventajas e inconvenientes de la simulación

En no pocas ocasiones, la elevada complejidad de muchos sistemas imposibilita obtener un modelo matemático con solución analítica. En estos casos, habrá que recurrir necesariamente a la simulación. La utilización de un modelo de simulación para el estudio de un sistema presenta una serie de **ventajas**:

1. Permite estudiar el comportamiento de un sistema bajo un conjunto de condiciones de operación predeterminadas.

2. Pueden compararse diseños alternativos de sistemas y estudiar aquel que satisface mejor un conjunto de requisitos.

3. Una simulación permite mantener un mejor control sobre las condiciones de funcionamiento que el que se obtiene experimentando directamente sobre el propio sistema real.

4. Permite estudiar un sistema en un marco temporal adecuado, comprimiendo el tiempo o expandiéndolo según el sistema a estudiar. Interesa contraer el tiempo cuando la evolución del sistema sea muy lenta, y dilatarlo cuando sea demasiado rápida.

De todas formas, también presenta un conjunto de **inconvenientes**:

1. El diseño de modelos de simulación es a menudo un proceso costoso en recursos y tiempo.

2. El sistema se comportará de forma diferente en cada ejecución, ya que tenemos un modelo estocástico. De aquí que el uso de modelos de simulación para la optimización de un sistema presente problemas en su aplicación.

3. Como ocurre en cualquier modelado de sistemas, siempre se cumple que la calidad del estudio está acotada por la calidad del modelo: si el modelo no representa de manera

suficientemente aproximada el sistema, las conclusiones inferidas de los resultados de las simulaciones pueden no ser correctas, por lo que siempre será necesario validar el modelo de simulación.

Formulación de un modelo para simulación

A la hora de diseñar un modelo de simulación de estos sistemas, como se ve en la Figura 2.2, podemos seguir los pasos siguientes:

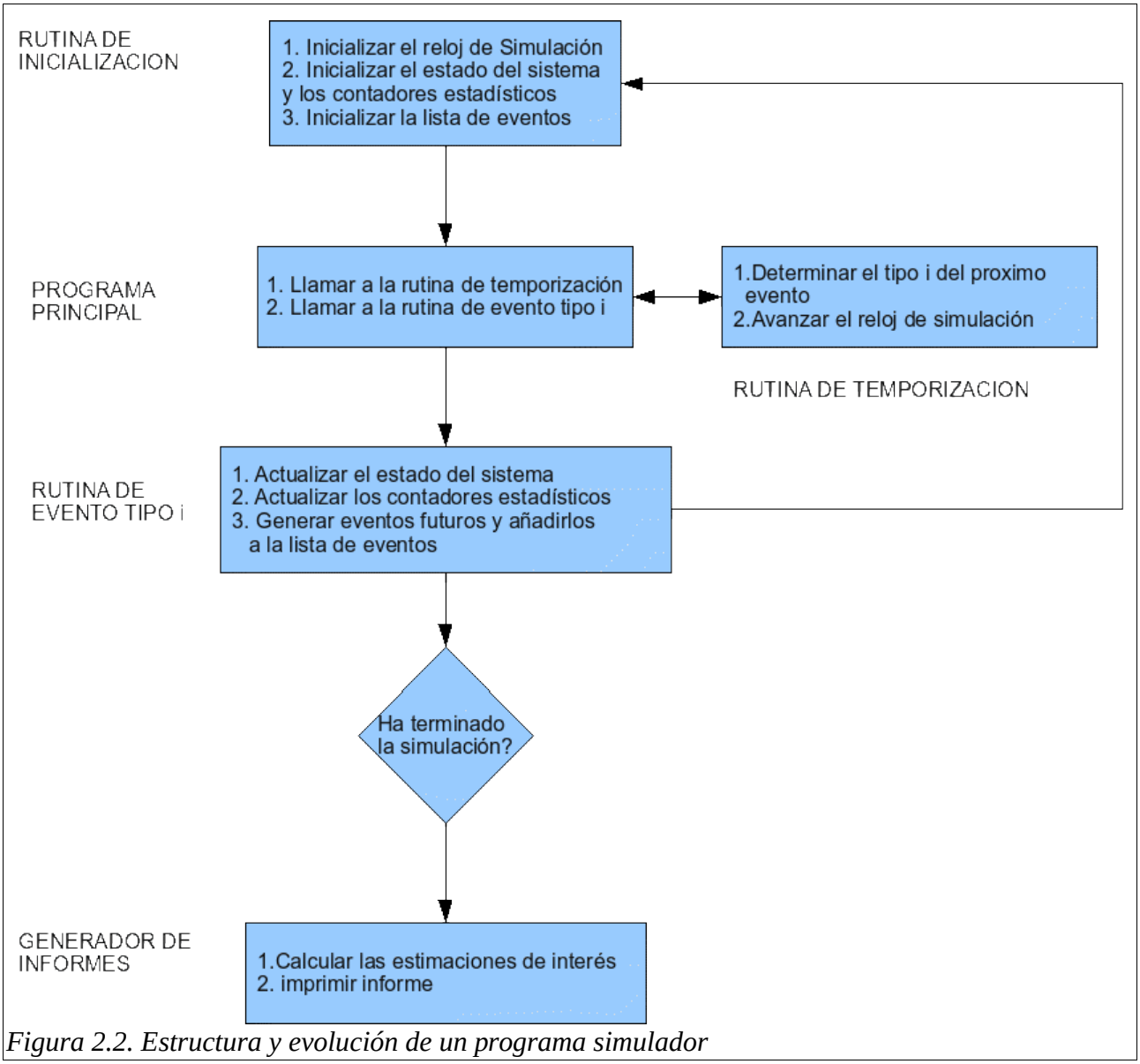

Figura 2.2. Estructura y evolución de un programa simulador

1. Identificación de eventos. Un evento o suceso será cualquier acción (instantánea) susceptible de modificar el estado del sistema.

2. Definición del mecanismo de control de tiempos. Dada la naturaleza dinámica de la simulación de eventos discretos, durante el proceso de simulación es necesario mantener

información sobre el tiempo de simulación, así como disponer de un mecanismo que permita decidir cuándo este tiempo evoluciona de un valor a otro. Llamaremos reloj de simulación a la variable que almacena el valor actual del tiempo de simulación, es decir, el instante temporal en el que está representado el sistema. En cuanto al mecanismo de avance del tiempo de simulación, pueden establecerse dos tipos de temporización:

- Temporización síncrona. En este caso el tiempo simulado (reloj) avanza a incrementos fijos.

- Temporización asíncrona. En este caso el tiempo simulado avanza sólo cuando ocurre un evento. Para el tipo de sistemas que se va a estudiar en este capítulo (sistemas de gestión de recursos) resulta mucho más eficiente esta forma de computar el tiempo.

3. Identificación de las estructuras de datos:

- Definir cuáles son las variables de estado.

- Reloj de simulación. Obviamente crucial ya que llevará el cómputo del tiempo de simulación.

- Listas o tablas de sucesos. En estas estructuras se almacenarán los instantes temporales en los que se ha planificado su ocurrencia, además de otra información relativa a dichos sucesos o eventos futuros.

- Contadores estadísticos. En estas variables se almacena la información necesaria para la medición de las magnitudes de interés. No son variables de estado sino auxiliares, es decir, no representan el estado del sistema.

4. Definición del flujo de control y de datos del programa simulador.

La aleatoriedad de algunas de las variables de estado del sistema se conseguirá mediante el uso de un generador de números aleatorios.

5. Definir la estructura del programa que realiza la simulación. Éste estará formado por los siguientes módulos:

- Programa principal.
- Rutina de inicialización.
- Rutina de temporización (manejo del reloj).
- Colección de rutinas de manejo de eventos y actualización de contadores estadísticos.
- Rutinas de generación de informes.
- Rutinas de generación de secuencias aleatorias, utilizadas para la generación de los diferentes eventos que se producen en el simulador.

6. Selección del lenguaje de programación. Tenemos dos grandes conjuntos de lenguajes para implementar un simulador:

• Lenguajes de propósito general: PASCAL, C, C++, JAVA, etc.

• Lenguajes específicos de simulación: SIMULA (Holmevik, 1994), SIMSCRIPT (Fa-yek, 1988), GPSS (Banks et al., 1995), SLAM (Pritsker, 1995), AutoMod (Stanley, 2001), ModL (Krahl, 2001), SLX (Henriksen, 2000), etc.

7. Programación.

8. Verificación del simulador. Necesaria para corregir las divergencias de la implementación respecto al modelo de simulación.

Ventajas e inconvenientes de los lenguajes específicos para simulación

La elección del lenguaje en el que se va a implementar el simulador es una decisión de diseño. El escoger un lenguaje específico para simulación presenta una serie de ventajas sobre los lenguajes de propósito general y, por supuesto, una serie de inconvenientes.

Ventajas de los lenguajes específicos para simulación:

1. Proporcionan automáticamente la mayoría de los elementos necesarios en la programación de modelos de simulación, con la consiguiente disminución del tiempo de desarrollo.

2. Proporcionan un entorno natural de desarrollo para modelos de simulación.

3. Los modelos de simulación son generalmente más fáciles de modificar cuando están desarrollados en un lenguaje de simulación.

4. Minimizan el número de errores. Por ser necesario escribir menos líneas de código, la probabilidad de cometer algún error disminuye.

5. Facilitan la detección de errores ya que existe una serie de errores potenciales, ya identificados, cuya comprobación es automática.

Inconvenientes de los lenguajes específicos para simulación:

1. Todo programador conoce algún lenguaje de propósito general, pero no necesariamente uno de simulación. Este desconocimiento implicará un esfuerzo de aprendizaje.

2. Disponibilidad. Son lenguajes usualmente caros y no suelen estar disponibles para todos los sistemas operativos.

3. Eficiencia. En general, un programa escrito en lenguaje de propósito general requiere menos tiempo de ejecución que uno escrito en un lenguaje de simulación.

4. Escasa flexibilidad. Pueden aparecer problemas si se intentan solventar necesidades diferentes a las comunes; sin embargo, los lenguajes de propósito general permiten una flexibilidad mayor.

Para cada situación concreta habrá que decidir la conveniencia o no de utilizar un lenguaje de simulación o uno de propósito general.

Valor de los modelos

La función más importante de la simulación es la de predecir el comportamiento del modelo y a partir de él del sistema real, contestarse la pregunta de ¿qué pasaría si…?. Una ventaja de los modelos de simulación es que no hay límites en cuanto a su complejidad. Cuando un sistema complicado desafía el análisis matemático o cuando la inclusión de variables aleatorias con distribuciones no estándar hace difícil el trabajo matemático, la simulación se vuelve adecuada. En suma, la estructura de un modelo de simulación da acceso a todas las variables dentro del modelo, ello permite descubrir relaciones internas entre las variables. Tales relaciones deben validarse en toda su extensión más allá de que deban ser justificadas en el dominio de la simulación. Esto ayuda a quien se está dedicando a realizar la simulación a resolver el problema y a desarrollar modelos más realistas.

La simulación es aplicable al diseño de nuevos sistemas, tales como un depósito automático que consideraremos más adelante, el modelo puede utilizarse para predecir la performance bajo condiciones variables y con diferentes equipos, esto antes de tomar decisiones de compra. Durante la vida del sistema la simulación puede continuar utilizándose para testear posibles modificaciones sin alterar el sistema en su operación normal.Los datos que se usan en ese caso son los del propio sistema y no los datos iniciales cuando se realizó el diseño, la simulación pasa a ser una parte dinámica del diseño total del sistema.

Las técnicas de simulación sirven para el entrenamiento de la gente en negocios y gestión. Por ejemplo jugadores que operan en un sistema económico, toman decisiones, el modelo simula el comportamiento del sistema económico después de dadas las decisiones de los jugadores. En particular resulta interesante estudiar las decisiones desastrosas. Los juegos de guerra que simulan el comportamiento en una batalla permiten a los jugadores tomar decisiones y el programa predice las consecuencias.

Sistema de Simulación de Propósito General. Introducción a GPSS.

Este trabajo está basado en el lenguaje de simulación de propósito general GPSS, nombre que adquiere por su autor, Geoffrey Gordon (Gordon's Programmable Simulation System o Sistema de Simulación Programable de Gordon), y del cual se ha tomado como referencia la implementación de Minuteman Software[MIN05a] llamada GPSS World (GPSSW - General Purpose Simulation System for Windows). Existen asimismo otras implementaciones de GPSS, como ser GPSS/H desarrollada por Wolverine Software[WOL07], o incluso una version para internet, llamada WebGPSS [WEB99].

El objetivo de este capítulo no es iniciar un curso avanzado de GPSS, sino explicar las características principales del lenguaje para así poder comprender la complejidad y la necesidad del planteo realizado en este trabajo. Para ello, explicaremos algunas entidades (comprendidas en nuestra implementación), su objetivo y posibles usos, su funcionamiento y las sentencias que el lenguaje nos ofrece para trabajar con ellas.

GPSS define un conjunto de bloques y comandos que el programador puede utilizar para realizar sus simulaciones, interactuando y modificando el estado de las entidades del modelo

subyacente. Cada bloque posee un nombre, un conjunto de parámetros y un código asociado, el cual sirve al para acceder, utilizar y/o alterar el estado interno de una o más entidades. El código de cada bloque es una abtracción de muy alto nivel de alguna funcionalidad compleja de la o las entidades asociadas.

Una entidad en GPSS es un objeto que posee un propósito específico, un tiempo de vida determinado y sobre el cual se pueden realizar un conjunto de operaciones predeterminado. Cada uno de estos objetos permite al programador representar entidades del modelo real dentro del modelo matemático de manera fácil y trasparente. Las entidades de GPSS se dividen en dos grandes grupos, de acuerdo a su tiempo de vida:

- Entidades Permanentes: viven durante toda la simulación. El programador puede suponer que existen y solo tiene que utilizarlas.
- Entidades Temporales: se crean y se destruyen a lo largo de la simulación. Es responsabilidad del programador gestionar el ciclo de vida de las mismas. Dentro del lenguaje solo hay una clase de entidades temporales denominadas transacciones, que son las encargadas de mantener el modelo en movimiento.

GPSSW define alrededor de 50 bloques y 25 comandos que permiten interactuar con gran número de entidades, y un lenguaje de programación procedural embebido (PLUS) para enriquecer los modelos. Un modelo en GPSS consta de una lista de sentencias (bloques y comandos). Los comandos se colocan en una cola de comandos y se ejecutan uno detrás de otro (esto es, cada comando espera que finalice el anterior para ejecutarse) mientras que los bloques son activados por las transacciones de manera secuencial, invocando al código correspondiente.

Cada sentencia consta de un nombre y un conjunto de parámetros separados por coma, pudiendo haber cambos vacíos que se también separan por coma o directamente se omiten; los parámetros son expresiones que se evaluan y retornan resultados de tipo númerico (entero, entero positivo o real) o cadenas de caracteres. Veamos algunos ejemplos:

```
START 10
CLEAR
RESET
GENERATE 10,,,,X$defaultPriority
ASSIGN 1,(Exponential(1,0,4))
TERMINATE 1
```

Las primeras tres sentencias corresponden a comandos. El primer comando, START, toma el valor 10 como parámetro de entrada. Los siguientes comandos, CLEAR y RESET, no toman ningún parámetro.

Las siguientes sentencias son tres bloques. El bloque GENERATE recibe cinco parámetros, de los cuales solo poseen valores el parámetro 1 y el 5. Este último parámetro posee la expresión X $defaultPriority que devueve el valor de una entidad SAVEVALUE (una especie de variable global); estas expresiones se denominan SNAs (Standar Numeric Attributes), las cuales detallaremos más adelante. El segundo bloque, ASSIGN, recibe dos parametros. El segundo de ellos es otro caso de expresión, en la cual se invoca a una función exponencial con tres parámetros y que

utiliza un generador de números aleatorios congruencial (primer parámetro, con valor 1), que está centrada en el punto cero (segundo parámetro) y cuyo valor de lambda es 4. Por último, el bloque TERMINATE toma solo el valor 1 como parámetro, el cual no es obligatorio.

Es importante aclarar que los parámetros de los bloques y comandos son denominados de acuerdo a su posición en la lista de parámetros con una letra del alfabeto. Esto quiere decir que, por ejemplo, en el caso del bloque GENERATE, el primero parámetro (con valor 10) se lo denomina parametro A, el segundo es el parámetro B, el tercero el C, el cuarto el D y el quinto (cuyo valor es el SNA X$defaultPriority) es el E. Los parámetros B, C y D no poseen ningún valor. Al igual que GPSS, utilizaremos mayormente esta nomenclatura para referirnos a cada parámetro a lo largo de este trabajo.

Como hemos mencionado, cada bloque posee una función específica y esta pensado para interactuar con una o varias entidades. Para describir los distintos bloques, veremos algunas de las entides de GPSS, sus usos más comunes y cómo estos bloques las afectan.

Entidad Simulation.

Representa una simulación: posee dentro a todas las entidades permanentes y temporales. Esta entidad posee un reloj de simulación, el cual se incrementa a medida que los eventos se suceden, y un contador de terminación el cual permite indicar cuando la simulación debe finalizar. A medida que nos adentremos en las diferentes entidades veremos que la entidad simulación posee más elementos de los mencionados aquí; cada unos de estos será explicado en su debido momento a medida que sea necesario.

SNAs - Atributos Numéricos del Sistema.

Un System Numerical Attribute, o simplemente SNA, es una expresión que permite recuperar el valor de un determinado atributo de algún objeto de la simulación. Existen algunos SNAs que no requieren la especificación de la entidad y otros que sí lo requieren. Por ejemplo, el SNA AC1 retorna el tiempo de reloj desde el inicio de la simulación, y el SNA XN1 retorna el número de la transacción activa; por el otro lado, tenemos el SNA FN$funcionDelUsuario que retorna la función que el usuario ha definido y se llama funcionDelUsuario, también tenemos el SNA F$facilidadDelUsuario, que retorna la facilidad llamada facilidadDelUsuario. Existen 54 SNAs distintos, aquí solo veremos algunos de ellos a modo de ejemplo o para aclarar algunos conceptos.

Entidad Transaction.

Son las únicas entidades temporales de la simulación, ya que nacen y mueren a lo largo de las simulaciones (su tiempo de vida es siempre menor o igual al tiempo total de simulación). Las transacciones se crean al inicio de la simulación a partir de la ejecución de un bloque GENERATE y se finalizan luego de la ejecución de un bloque Terminate. Existen asimismo otros bloque que permiten crear y destruir transacciones pero que tienen usos más específicos, los cuales no explicaremos ya que exceden los objetivos planteados para esta introducción a GPSS. Cada transacción tiene un conjunto de atributos fijos que se setean al momento de su creación y un

conjunto de atributos variables que el programador setea, modifica y utiliza a medida que necesita. Los atributos fijos de todas las transacciones son:

- Transaction Number: número que identifica a la transacción unívocamente del resto de las transacciones.
- Priority: las transacciones poseen una prioridad, la cual determinará su momento de ejecución.
- Block departure time (BDT): tiempo de reloj en que la transacción debe abandonar un bloque.
- Assembly Set: conjunto al que pertenece cada transacción. En principio, cada transacción pertenece a un conjunto, el cual a su vez posee solo a esa transacción.

Cada vez que una transacción es creada, se la prepara para ejecutarse en algún momento de simulación, determinado por su atributo BDT. Cuando su ejecución comienza, la transacción ejecuta los bloques secuenciales que continúan al bloque GENERATE uno tras otro, hasta que finalice o hasta que por algún motivo debe detenerse y ceder el turno a otra transacción. Es importante dejar en claro aquí que, sin importar cuantas transacciones estén en la simulación ejecutandose en un determinado momento, **siempre existe una y solo una transacción activa** atravesando los bloques, que es aquella que está ejecutando los bloques uno tras otro. El lenguaje permite la ejecución de varias transacciones en un mismo momento de simulación, pero esto no indica que esta ejecución se realiza de manera concurrente o paralela. Asimismo, también debemos dejar en claro que las transacciones se ejecutan (esto es, pasan de bloque en bloque) hasta que algo las detenga. Si nada las detiene, se ejecutarán indefinidamente o hasta que finalicen.

Las transacciones que no son la transacción activa se encuentran dentro de una o más cadenas, las cuales no son más que colas ordenadas por algún criterio en particular. Una cadena siempre posee transacciones y nada más que transacciones, la entidad Simulation por ejemplo posee dos cadenas:

- CEC (Current Events Chain): esta cadena posee el listado de transacciones que deben ser ejecutadas en este momento de reloj ordenadas a partir de la prioridad de las transacciones de mayor a menor (esto quiere decir que las transacciones de mayor prioridad serán ejecutadas primero) y luego en orden FIFO (al igual que el resto de las cadenas); el reloj de simulación no avanzará hasta que esta cadena esté vacía. El scheduler de transacciones toma una a una las transacciones de esta cadena y coloca la primer transaccion de la cadena como transacción activa de la simulación. Cada vez que una transacción activa se detiene por algún motivo, el scheduler tomará la siguiente transacción de la CEC y la colocará como nueva transacción activa. Una vez que no quedan transacciones en la CEC, avisará a la simulación que debe actualizar su reloj.
- FEC (Future Events Chain): en esta cadena, las transacciones están ordenadas por su BDT en orden ascendente. Como mencionamos anteriormente, el BDT de una transacción indica cuando la misma debe abandonar un bloque y continuar su ejecución. Cada vez que la CEC se vacía, la entidad Simulation tomará el menor BDT de entre todas las transacciones de la FEC y seleccionará todas las transacciones con ese BDT, para luego eliminarlas de la FEC y colocarlas en la CEC. Esto provocará un incremento en el reloj de simulación, el cual será seteado con el BDT de las transacciones que están siendo movidas de una cadena a la otra.

Algunos de los bloques que nos permiten interactuar con las transacciones son:

GENERATE A,B,C,D,E

Crea las transacciones de la simulación.
Parámetros:
A: Tiempo medio entre generaciones.
B: Modificador del tiempo medio. Si se especifica,las transacciones se generarán cada A±B unidades de tiempo, o dicho de otro modo, siguiendo una variable aleatoria uniformemente distribuída en el intervalo [A-B,A+B].
C: Momento de generación de la primer transacción
D: Cantidad máxima de transacciones a crear a partir de éste bloque GENERATE
E: Prioridad de las nuevas transacciones

Ningún parámetro de este bloque es obligatorio, aunque existen restricciones. Por ejemplo, si no se especifica el parámetro A, debe especificarse sí o sí el parámetro D. Por ejemplo, la sentencia

```
GENERATE ,,,4
```

generará 4 transacciones al comienzo de la simulación, ya que no se especifica ningún tiempo entre creaciones. Similarmente, la sentencia

```
GENERATE ,,3,4
```

también generará 4 transacciones pero lo hará en el tiempo de reloj 3, ya que se ha especificado el parámetro C. Finalmente, la sentencia

```
GENERATE 4,1,2,,4
```

generará transacciones hasta que finalice la simulación. La primer transacción generada por este bloque será en el tiempo de simulación 2, y luego se generarán transacciones cada 4±1 instantes. Todas las transacciones generadas tendrán prioridad 4, indicada en el parametro E.

El bloque GENERATE es mucho más complejo que lo explicado aquí, y la combinación de los parámetros permite una cantidad de opciones muy amplia, pero como se mencionó previamente, en esta sección pretendemos explicar como es el funcionamiento de este lenguaje de manera muy general y sin entrar en detalles puntuales.

ASSIGN A,B

Permite asignar el valor de algún atributo (variable) de la transacción.

Parámetros:
A: Nombre o número de atributo
B: Valor a setear

Algunos ejemplos:

```
ASSIGN 1,4
```

asigna el valor 4 en el parámetro 1 de la transacción activa.

```
ASSIGN estado, "ocupado"
```

asigna el valor "ocupado" en el parametro llamado *estado.*

```
ASSIGN 2,XN1
```

asigna el número de transacción en el atributo 2 de dicha transacción. Aquí, XN1 es un SNA que devuelve el número de la transacción activa.

Como puede apreciarse, los atributos de las transacciones pueden tener un número o un nombre, y su contenido puede ser tanto numérico como string. Más adelante veremos esto con mayor detalle.

`TERMINATE A`

Finaliza la transacción.

Parametros:
A: Valor a decrementar del contador de terminación.

Ejemplos
`TERMINATE 1`
elimina la transacción y decrementa el contador de terminación en 1

`TERMINATE`
elimina la transacción, pero no decrementa el contador de terminación

`ADVANCE A,B`

Detiene la ejecución de la simulación, planificándola para algún tiempo futuro. Esto quiere decir que las transacciones que pasen por un bloque advance con un incremento positivo pasaran a la FEC conteniendo como BDT el valor de la evaluación de los parámetros A y B.

Parametros
A: Incremento de tiempo medio
B: Modificador del tiempo medio (opcional)

Ejemplos

`ADVANCE 4.7,1.3`

Primero se calcula 4.7±1.3, este resultado es utilizado para calcular el BDT de la transacción activa, el cual consitirá de dicho valor más el reloj actual de la simulación. A continuación, la transacción activa es situada en la FEC, de acuerdo al BDT que acaba de calcularse. Esto provoca que una nueva transacción sea seleccionada de la CEC como próxima transacción activa.

Con estos bloques ya podemos generar nuestra primer simulación simple. Veamos un ejemplo práctico:

```
GENERATE 2,1,,10
ASSIGN 1,“este es el parametro uno”
ADVANCE 4,2
ASSIGN terminando,“ya estoy por terminar”
TERMINATE 1
```

Este ejemplo, sin utilidad real, sirve para ilustrar el funcionamiento de las transacciones. El intérprete del lenguaje busca los bloques GENERATE y los ejecuta, lo cual provoca la generación de transacciones cada 2±1 instantes de reloj, y que se generen como máximo 10 transacciones. Cada

transacción generada, ejecutará inmediatamente su bloque ASSIGN, lo cual hace que se les asigne el atributo 1 con el texto “este es el parametro uno”. A continuación, y siempre en el mismo instante de reloj en que se generó la transacción (ya que aun nada la detuvo), se calculará el tiempo de avance, el cual estará entre 4±2 unidades de reloj. Esto provocará que la transacción detenga su ejecución y sea colocada en la FEC, preparada para ejecutarse en algún tiempo futuro. Cuando este momento llegue, nuestra transacción continuará desde donde se detuvo antes de ingresar a la FEC (o sea en el bloque ADVANCE). Esto implicara que se setee el atributo llamado “terminando” con el valor “ya estoy por terminar”, y finalmente morirá decrementando en 1 el contador de terminación.

Entidad Random Number Generator

GPSS utiliza generadores de números aleatorios (Random Number Generator o RNG) en muchos de sus bloques, comandos e invocaciones a primitivas propias. En nuestro ejemplo simple tenemos dos bloques que requieren de estos generadores para ejecutarse: el bloque GENERATE y el bloque ADVANCE. En ambos casos tenemos una expresión que retornará un valor en el intervalo [B-A,B+A], y la función del RNG es seleccionar uno de entre todos los posibles valores de dicho intervalo con **igual probabilidad** para cualquier valor; dicho en términos matemáticos, tendremos una variable aleatoria $X \sim U [B-A,B+A]$.

Existen varios RNG predefinidos dentro de GPSS, los cuales son referenciados por la expresión (llamada SNA) RNx donde x es un número entero positivo. De todos modos, el programador puede definirse sus propios generadores, o incluso puede modificar mediante comandos la forma en que los RNG predefinidos generan los valores, cambiando por ejemplo la semilla utilizada para dicha generación.

Entidades permanentes

Como ya hemos mencionado, estas entidades existen a lo largo de toda la simulación; las transacciones las utilizan y alteran a medida que se ejecutan, cambiando su estado interno y generando un gran número de estadísticas. Tanto las entidades permanentes como las temporales poseen al menos una cadena, llamada Retry Chain. En esta cadena se colocan todas las transacciones que están esperando por el cambio de algún atributo de esta entidad; esto se debe a que algunos bloques frenan a las transacciones hasta que se cumpla alguna condición específica en la cual interviene un atributo SNA de alguna entidad. Para poder continuar con la ejecución, estas transacciones son separadas de la simulación y colocadas en la o las retry chains de las entidades por las que esperan algún cambio y, cuando esto suceda, volverán a la simulación para **intentar** continuar su ejecución normal.

Entidad Facility

Las facilidades son un tipo de entidad permanente muy importante en las simulaciones, ya que permiten representar de manera totalmente transparente un enorme conjunto de entidades de sistemas reales dentro de los modelos matemáticos. Estas entidades representan objetos que pueden estar en un estado libre u ocupados, y que cuando están ocupados es porque una y solo una entidad (transacciones de GPSS, también referenciadas como owners) está utilizando el servicio que provee.

Cada vez que una transacción intenta utilizar una facilidad ocupada, deberá esperar en una cola (cadena dentro de GPSS) y competir por el acceso a esta facilidad con el resto de las entidades que se encuentren en esa cola. Para aclarar este concepto, veamos algunos ejemplos:

- Computadora monoprocesador: el procesador es la facilidad, y los programas son las transacciones. Los programas solicitan al Sistema Operativo el acceso al procesador, y esperan a que este les de permiso de ejecución. El SO tendrá una cola de programas solicitando recursos.
- Supermercado: podemos representar al cajero como una facilidad y a los clientes como transacciones. Cuando el cajero está cobradole a un cliente, el resto deberá hacer cola detrás
- Servidor web: el proceso servidor solo puede atender un determinado número de peticiones concurrentes, y el resto deberán esperar en cola hasta que puedan ser atendidas.

Como podemos observar, las colas estan estrechamente ligadas a las facilidades, ya que su ordenamiento determinará el orden de ejecución de los eventos. En cualquiera de los ejemplos, es intuivo pensar en procesos de mayor prioridad o clientes más importantes; las colas de GPSS contemplan el concepto de prioridad, y permiten el ordenamiento de las transacciones por prioridad (y FIFO para transacciones de una misma prioridad). Pero esto no siempre es así, ya que dependerá de la función de cada cola dentro del sistema; por ejemplo, la cadena FEC tiene transacciones ordenadas por BDT y independientemente de la prioridad, ya que no necesita este valor para funcionar.

A lo largo de las simulaciones, las transacciones toman facilidades, las utilizan y las liberan. Para ello, utilizan el bloque SEIZE para tomar una facilidad y el bloque RELEASE para liberarla.
Por ejemplo: supongamos que queremos simular un cajero automático, en el cual tenemos 20 clientes que intentan utilizarlo todos a la vez. Los clientes hacen cola para utilizar nuestro cajero, y a cada cliente le toma entre 2 y 8 minutos realizar la operación deseada; una vez que lo ha utilizado se retira inmediatamente. En GPSS, esto lo podemos simular del siguiente modo:

```
GENERATE ,,,20   ;generamos 20 clientes en el instante de reloj 0
SEIZE cajero     ; intento tomar el cajero
ADVANCE 5,3      ; utilizo el cajero entre 2 y 8 minutos
RELEASE cajero   ; libero el cajero
TERMINATE 1     ; el cliente se retira, decrementando el contador
de terminación en 1

START 20         ; inicializo el contador de terminación en 20
```

Con estas pocas líneas de código hemos representado nuestro modelo. Se generan todos los clientes juntos e intentan tomar la facilidad cajero (podemos observar que esta facilidad nunca se declaró, pero GPSS la reconoce como tal y la crea cuando la primer transacción la invoca). Solo la primer transacción podrá hacerlo, continuando con el bloque ADVANCE; el resto de las transacciones entrarán a una cola de espera por la facilidad (llamada Delay Chain). Aquí, se calcula el tiempo de retardo, se coloca la transacción en la FEC y quedando *dormida* hasta que le toque su turno nuevamente, de acuerdo al tiempo calculado.

Aquí, nuestro modelo quedó trabado. Tenemos 19 transacciones en la delay chain de la

facilidad cajero, y una en la FEC. Pero no tenemos transacciones en la CEC, con lo cual es el momento de cambiar el reloj de simulación. La entidad Simulation (junto con el planificador de transacciones) tomará el menor BDT de la FEC (el único que existe) e incrementará el reloj con ese valor, luego tomará la única transacción de la FEC y la colocará en la CEC. Ahora sí, esta transacción será seleccionada como transacción activa y podrá continuar su ejecución. Dicha transacción liberará la facilidad cajero (RELEASE), y entrará al bloque TERMINATE, decrementando en 1 el contador de terminación para finalmente ser eliminada de la simulación. Una vez que las 20 transacciones hayan decrementado este valor (colocándolo en cero), la simulación habrá terminado.

En base a este modelo, podemos realizar algunos cálculos simples para verificar su funcionamiento:

- ¿cuántos instantes de reloj durará la simulación?
  Lo único que provoca cambios en el reloj de simulación es el bloque ADVANCE, el cual se ejecuta de manera uniforme con media 5. Con lo cual, podemos suponer que si tenemos 20 transacciones que demoran 5 unidades de tiempo en avanzar, tendremos alrededor de 100 unidades de tiempo de simulación.

- ¿Cuál es el nivel de uso de nuestra facilidad?
  En nuestro planteo, durante todo el tiempo de simulación hay algún cliente esperando con lo cual la facilidad estará siempre ocupada. Esto nos da un nivel de ocupación del 100%

- ¿Cuál es el tiempo promedio de uso de la facilidad?
  Nuevamente, el tiempo estará determinado por el bloque ADVANCE, con lo cual es de esperar que de un valor cercano a 5.

- ¿Cómo finalizó nuestra simulación?
  No deben existir transacciones pendientes ni en la FEC ni en la CEC, y todos los bloques deben haber recibido 20 entradas, una por transacción.

La ejecución de este código en GPSSW generará un reporte, el cual nos servirá para evaluar las estimaciones que realizamos sobre nuestro modelo matemático. Veamos el reporte generado:

```
GPSS World Simulation Report - Untitled Model 1.1.1


Wednesday, February 13, 2008 15:11:52

 START TIME           END TIME  BLOCKS  FACILITIES  STORAGES
      0.000            108.653     5        1          0


    NAME                         VALUE
CAJERO                         10000.000


LABEL              LOC  BLOCK TYPE     ENTRY COUNT CURRENT COUNT RETRY
                   1    GENERATE            20             0       0
                   2    SEIZE               20             0       0
                   3    ADVANCE             20             0       0
                   4    RELEASE             20             0       0
                   5    TERMINATE           20             0       0

FACILITY         ENTRIES  UTIL.   AVE. TIME AVAIL. OWNER PEND INTER RETRY DELAY
 CAJERO              20    1.000       5.433  1        0    0    0     0      0
```

Observemos que el valor END TIME (tiempo de finalización) nos esta dando 108.6, bastante cercano a nuestra estimación. Podemos también ver que el nivel de utilización de la Facility CAJERO (UTIL.) nos da 1.00, lo que equivale al 100%, y que el tiempo promedio de permanencia de cada una de las 20 transacciones (ENTRIES) en esta facilidad (AVE. TIME) es 5.433, también muy cercano a nuestra estimación de 5. Finalmente, no existen listados de transacciones pendientes en la FEC ni en la CEC, lo cual significa que todas las transacciones han terminado como se esperaba.

En este simple ejemplo hemos simulado un sistema complejo con muy poco código (y sin entrar en detalles de implementación de gestión de colas, generación e inicialización de objetos, retardos y scheduling), y adicionalmente el entorno nos ha generado un reporte con varias estadísticas acerca del modelo. Estas dos características hacen que este tipo de lenguajes sean ideales para este tipo de tareas.

Preempción de facilidades.

Volvamos nuevamente a tomar el ejemplo del sistema monoprocesador, en el cual tenemos un único procesador y un conjunto de procesos compitiendo por acceder a este. Es de esperar que este sistema considere procesos de mayor prioridad, los cuales serán seleccionados para tomar el procesador antes que sus pares de menor prioridad. Pero, ¿qué sucede si en un determinado momento un proceso de muy alta prioridad necesita sí o sí tomar el control del procesador? Esperar que el proceso que lo posee actualmente no es una opción, ya que el momento tiene que ser ahora; el proceso que ocupa el procesador debe quitarse para permitir al nuevo proceso el uso del procesador, y cuando termine podrá entregar el procesador al proceso a quien se lo quitó.

Este concepto se denomina preempción y como es de esperar, GPSS la incorpora la funcionalidad que permite desarrollar simulaciones que sigan este modelo de comportamiento. Para

realizar preemciones GPSS ofrece dos bloques: PREEMPT y RETURN.

PREEMPT A,B,C,D,E

Sirve para tomar el control de una facilidad, desplazando a la transacción que tenía previamente el control.

Operandos:

A: Facilidad a preemptear
B: Modo Prioridad (PR)
C: Destino para la transacción que poseía la facilidad
D: Parámetro de preempción.
E: Modo remoción (RE)

Si la facilidad esta libre, el bloque PREEMPT se comporta igual que el bloque SEIZE. De lo contrario, opera en modo Priority o en modo Interrupt, según se determine en el operando B.

En *modo Priority* (B = PR), solo una transaccion de mayor prioridad puede desplazar a la que controla actualmente a la facilidad. Si la transaccion activa no posee suficiente prioridad, se coloca en la Delay Chain. En *modo Interrupt* (se omite B), si la facilidad ya habia sido preempteada por otra transacción, la transaccion activa se ubica en la Pending Chain (la cual tiene mas prioridad que la Delay Chain y que la Interrupt Chain); si la facilidad no habia sido preempteada, se ignoran las prioridades y la transaccion activa desplaza a la transaccion que actualmente controla la facilidad.

Los operandos C, D y E sirven para decidir que hacer con la transaccion que está por perder la propiedad de facilidad. **Las transacciones preempteadas no pueden existir en la FEC.** Transacciones preempteadas que aun tienen tiempo sin concluir en un bloque ADVANCE pueden ser reemplazadas automaticamente en la FEC omitiendo los operandos C y E. Alternativamente, si se elige reemplazar la transaccion de la FEC manualmente, para eventualmente completar el tiempo remanente, se debe utilizar el operando D y eventualmente enviar a la transaccion preempteada a un bloque ADVANCE.

Una transaccion preempteada puede ser removida de la competencia por la facilidad (removida de todas las cadenas de la facilidad) utilizando la opcion RE en el operando E. **La opcion RE remueve todas las restricciones sobre transacciones preempteadas debidas a preempciones sobre esta facilidad**, y hacen que cualquier intento subsecuente de RETURN o RELEASE provoquen una ERROR CONDITION. Si el parametro E vale RE, se debe utilizar obligatoriamente el parametro C.

Una transaccion preempteada no puede existir en la FEC. Cualquier transaccion preempteada en un bloque ADVANCE que se encuentra en la FEC es removida de la misma y la duración de tiempo residual es almacenada. Alternativamente, si se utiliza el operando D, este tiempo tambien es almacenado en el parámetro indicado (si no existe, se crea). El tiempo residual es utilizando para volver a planificar la transaccion una vez que logra la posesion de todas las facilidades por las cuales esta compitiendo. Alternativamente, se le puede dar a una transaccion preempteada un nuevo bloque para intentar ingresar utilizando el parametro C.

Una transaccion preempteada permanece en competencia por la facilidad incluso si se utiliza el parametro C, a menos que se especifique RE en el parametro E. Si una transaccion preempteada, que aun compite por la facilidad, intenta ingresar a un bloque TERMINATE, ocurrira un ERROR STOP. Si se desea terminar transacciones preempteadas, debera utilizarse el parametro E.

- Cuando se utiliza el parametro C al preemptear una transaccion, se le da un nuevo bloque destino a la misma y se elimina de las siguientes colas:
  FEC
  PENDING (Interrupt Mode PREEMPT) Chains
  DELAY (Major Priority) Chains
  USER Chains
  Retry Chains
  pero permanece en la CEC, en las Interrupt Chains y en las Group Chains.
  Tambien se les remueven todas las condiciones de bloqueo, ya que al no estar en las Retry Chain no tiene sentido que sigan manteniendo estas condiciones; no se limpian las preempciones sobre otras facilidades.
- Cuando no se utiliza el parametro C, una transacción que fue eliminada de la FEC retornará automaticamente a ella. Se espera que eventualmente estas transacciones (las que fueron preemteadas con el parametro C) entraran a un bloque RELEASE o RETURN. Si ingresan a alguno de estos bloques antes de volver a ganar la propiedad de la facilidad, se eliminaran de la competencia por dicha facilidad. Esto no provoca ningun error, ya que es un caso considerado en ambos bloques.

Una transaccion puede ser preempteada de cualquier cantidad de facilidades, y puede aun continuar corriendo en el modelo. Sin embargo, esta sujeta a dos restricciones:

1. Se le negara el ingreso a cualquier bloque ADVANCE con tiempo positivo
2. No podra dejar ningun bloque ASSEMBLE, GATHER o MATCH hasta que todas sus preempciones se limpien.

Una facilidad puede ser preempteada cualquier cantidad de veces, pero una transaccion que perdio una facilidad en un PREEMPT no puede luego intentar ejecutar un SEIZE sobre la misma facilidad. Obviamente, una transaccion puede ser preempteada de cualquier cantidad de facilidades. Y una transaccion no puede preemptearse a si misma.

RELEASE A

Cuando una transaccion ingresa a un bloque RELEASE se remueve asi misma de la competencia por la facilidad. Esto puede suceder de dos maneras.

- Si la transaccion activa era el dueño de la facilidad, la libera y continua al Next Sequential Block.
- Si la transaccion activa había sido preempteada de la facilidad, es removida de la Interrupt Chain de la misma. El owner de la facilidad aquí no es afectado, ya que se trata de alguna otra transaccion. Si la transaccion activa se ha liberado de todas las preempciones, puede ahora continuar libremente en el modelo.

Si no era el dueño, y tampoco había sido preempteada de la facilidad, ocurre un ERROR STOP. Si la transaccion activa abandona la facilidad, se debe elegir un nuevo dueño, el cual se toma

primero de la Pending Chain, luego de la Interrupt Chain y finalmente de la Delay Chain.

Si actualmente hay algún PREEMPT pendiente en Interrupt Mode, al primero se le entrega la facilidad; de lo contrario, se le da el dominio de la facilidad a la siguiente transacción que habia sido preempteada. Si no hay transacciones ni en la Pending Chain ni en la Interrupt Chain de la facilidad, se seleccionará la transacción de mayor prioridad de la Delay Chain. Si la Delay Chain tambien estaba vacía, la facilidad pasara a estado IDLE (libre u ociosa).
Cuando un nuevo dueño es seleccionado desde la Delay Chain o Pending Chain, este ingresa inmediatamente a un bloque SEIZE o PREEMT lo cual hace que el scheduler la coloque en la CEC.

A lo largo de este capítulo hemos descripto algunas de las entidades y sus bloques asociados de GPSS, que son precisamente aquellas que hemos incluido en este trabajo. Este lenguaje posee una cantidad mucho mayor y muy diversa de entidades, y por consiguiente una variedad enorme de bloques y comandos para interactuar con dichas entidades, pero como ya hemos dicho previamente el objetivo que nos hemos planteado en este capítulo es simplemente el de introducir estos conceptos básicos para facilitar la comprensión del trabajo. El lector interesado podrá consultar toda la documentación de GPSS en [MIN05b] y acceder a una gran cantidad de tutoriales y ejemplos en [MIN05c].

Capítulo 3. Persistencia de Objetos.

El trabajo que estamos presentando tiene como base dos grandes áreas de investigación y desarrollo: Modelos y Simulaciones por un lado y Bases de Datos por el otro. El primer tema fue introducido en el capítulo anterior, junto con algunos conceptos muy básicos de GPSS que nos ayudan a facilitar la comprensión de este desarrollo. El segundo tema que abarca nuestro trabajo se refiere al almacenamiento persistente de datos, en particular de objetos. Consideramos que es fundamental poseer al menos un mínimo conocimiento del tema para poder entender la verdadera importancia de este trabajo así como también las discusiones que se plantean en los Capítulos 6 y 7. Es por esto que, al igual que con el tema de Modelos y Simulaciones, hemos incluido esta sección especial dedicada a la persistencia de datos, donde veremos qué es una base de datos, qué tipos de bases de datos existen, cual es la diferencia entre una base de datos relacional y una base de datos orientada a objetos y que alternativas existen para almacenar objetos en bases de datos relacionales. Asimismo, presentaremos una breve introducción a JDO y JPOX, que nos han servido persistir y recuperar los datos persistidos. Desde ya, no trataremos ninguno de estos temas en profundidad, sino que solo destacaremos aquellos aspectos que consideramos esenciales para comprenderlos así como también cuál su relación con este desarrollo.

Bases de datos.

Una base de datos (BBDD) es un conjunto de datos relacionados almacenados sistemáticamente de manera estructurada. Diferentes programas y diferentes usuarios deben poder utilizar estos datos. El concepto de base de datos generalmente está relacionado con el de red ya que se debe poder compartir esta información. "Sistema de información" es el término general utilizado para la estructura global que incluye todos los mecanismos para compartir datos que se han instalado.

Una base de datos proporciona a los usuarios el acceso a datos, que pueden visualizar, ingresar o actualizar, en concordancia con los derechos de acceso que se les hayan otorgado. Se convierte más útil a medida que la cantidad de datos almacenados crece. La base de datos puede ser local a un equipo o puede ser remota, es decir que la información se almacena en equipos remotos y se puede acceder a ella a través de una red.

La principal ventaja de utilizar bases de datos es que múltiples usuarios pueden acceder a ellas al mismo tiempo sin poner en riesgo la integridad de los datos, controlando la seguridad de los mismos para usuarios de distintos niveles y maximizando el acceso concurrente tanto como sea posible.

Componentes de una Base de Datos

Una base de datos consta de tres componentes principales que trabajan proveer todas las características mencionadas. Estos componentes son: el hardware subyacente, el software gestor de la base de datos y los datos en sí.

Hardware: constituido por dispositivos de almacenamiento como discos, tambores, cintas, etc.

Software: Sistema Administrador de Base de Datos (o DMBS por sus siglas en inglés).

Datos: los cuales están almacenados de acuerdo a la estructura externa y van a ser procesados para convertirse en información.

Ventajas de las bases de datos frente a los archivos planos

Almacenar información en archivos comunes pareciera, a simple vista, ser una solución mucho más práctica y rápida si se la compara con el uso de un motor de almacenamiento, un DBMS y un lenguaje de consulta específico. Pero el uso de una base de datos ofrece una serie de ventajas que no poseen los archivos comunes. Algunas de ellas son:

- Control centralizado, realizado por el DMBS
- Reducción de redundancias
- Eliminación de inconsistencias
- Datos compartidos concurrentemente
- Ajuste a estándares.
- Mayor seguridad.
- Mayor facilidad en el chequeo de errores.
- Balanceo de requerimientos opuestos

Independencia de datos.

Se dice que una aplicación es dependiente de los datos si es imposible alterar la estructura de almacenamiento o la técnica de acceso sin afectar a la aplicación. En un sistema de bases de datos no es recomendable tener aplicaciones dependientes de los datos por dos razones principales:

a) Cada aplicación puede requerir una vista diferente de los mismos datos.

b) El administrador de la BBDD debe tener suficiente libertad para modificar la estructura de almacenamiento o las técnicas de acceso (o las dos cosas) para adaptarla al cambio de los requerimientos sin tener que modificar las aplicaciones ya existentes.

Cada aplicación extraerá los datos que necesite de la base de datos común y los mostrará en pantalla según los requerimientos de cada uno de los sitios, pero además pueden estar diseñadas en diferentes lenguajes, php, asp, java, o incluso lenguajes de escritorio como Visual Basic® o Borland Delphi®.

Es deseable alcanzar independencia de datos a tres niveles:

1. Del campo almacenado: Es la mínima cantidad de información reconocible con un nombre que se almacena (por ejemplo, utilizaremos nombres como fecha de alta o motivo de renuncia para los campos)
2. Del registro almacenado: Es un conjunto de campos almacenados relacionados entre sí que cuenta con su propio nombre.
3. Del archivo almacenado: Es el conjunto de todas las ocurrencias almacenadas para un tipo de registro

Llamaremos **registro lógico** al registro que ve el usuario, y **registro físico** al registro tal y

como se almacena en la base de datos.

Existen desde ya diferentes formas de almacenar información, lo cual da lugar a distintos modelos lógicos de organización de la base de datos: jerárquico, red, relacional y orientada a objetos. Las últimas dos son actualmente las más difundidas y utilizadas, y nos centraremos en ellas a lo largo de este capítulo.

Los sistemas relacionales son importantes porque ofrecen las siguientes características: simplicidad y generalidad, facilidad de uso para el usuario final, períodos cortos de aprendizaje y las consultas de información se especifican de forma sencilla. Los sistemas orientados a objetos ofrecen mayor transparencia al momento de almacenar y recuperar datos de aplicaciones construidas siguiendo este paradigma. A continuación, veremos algunos aspectos importantes de cada uno.

Bases de datos relacionales.

Las tablas son un medio de representar la información de una forma más compacta; es posible acceder a la información contenida en dos o más tablas mediante consultas en un lenguaje para tal fin.

Las bases de datos relacionales están constituidas por una o más tablas que contienen la información ordenada de una forma organizada, y cumplen las siguientes leyes básicas:

- Una tabla sólo contiene un número fijo de campos.
- El nombre de los campos de una tabla es distinto.
- Cada registro de la tabla es único.
- El orden de los registros y de los campos no está determinados.
- Para cada campo existe un conjunto de valores posible.

Diseño de las bases de datos relacionales

El primer paso para crear una base de datos, es planificar el tipo de información que se quiere almacenar en la misma, teniendo en cuenta dos aspectos: la información disponible y la información que necesitamos. La planificación de la estructura de la base de datos, en particular de las tablas, es vital para la gestión efectiva de la misma. El diseño de la estructura de una tabla consiste en una descripción de cada uno de los campos que componen el registro y los valores o datos que contendrá cada uno de esos campos.

Los campos son los distintos tipos de datos que componen la tabla, por ejemplo: nombre, apellido, domicilio. La definición de un campo requiere: el nombre del campo, el tipo de campo, el ancho del campo (si es aplicable), entre otros. Los registros constituyen la información que va contenida en los campos de la tabla, donde algunos pueden contener datos simples como strings o fechas, y otros, datos complejos como imágenes o documentos XML.

En resumen, el principal aspecto a tener en cuenta durante el diseño de una tabla es determinar claramente los campos necesarios, definirlos en forma adecuada con un nombre especificando su tipo y su longitud.

Una vez que hemos planificado el tipo de información que queremos almacenar, debemos establecer relaciones entre estos datos. Por ejemplo, si tenemos por un lado personas con un nombre, un apellido y un DNI, y por el otro teléfonos, con una característica de país, otra de ciudad, un número local y un número de interno, podríamos establecer la siguiente relación entre las personas y los teléfonos:

Una persona posee al menos un teléfono, y un teléfono pertenece a al menos una persona .

Se las llaman bases de datos relacionales precisamente porque los datos que contienen en tablas se relacionan entre sí mediante, valga la redundancia, relaciones. Existen diversos tipos de relaciones binarias entre tablas, como ser 1:1 (un registro de una tabla se relaciona con un registro de otra tabla), 1:n (un registro de una tabla se relaciona con varios registros de otra tabla) y n:m (igual a una relación del tipo 1:m + n:1). Existen también relaciones n-arias (con n mayor a 2), las cuales plantean una discusión que no incluiremos aquí por exceder el propósito de este capítulo[SOU96].

El lenguaje de consulta SQL.

SQL son las siglas de Structured Query Language, que significa Lenguaje de consulta estructurado y que es el estándar de facto para acceso a base de datos. En lenguaje SQL surgió a raíz de la creación en IBM durante la década de los 70 de un sistema de base de datos llamado "System R"; para manipular y recuperar datos de dicho sistema se diseñó un lenguaje llamado SEQUEL (Standard English Query Language) que finalmente se convertiría en SQL.

En principio SQL es un estándar ISO desde 1987, sin embargo la especificación SQL es bastante grande y compleja de forma que casi ninguna base de datos implementa la especificación de forma totalmente completa. Por otro lado, los grandes proveedores de bases de datos (ORACLE o SQL Server por ejemplo) han añadido diversa funcionalidad de forma no homogénea y fuera de la especificación a sus propias bases de datos.

Dentro del lenguaje SQL podemos distinguir las sentencias en dos tipos dependiendo de la función que realicen; podemos distinguir entre sentencias DML y DDL.

DML son las siglas de Data Manipulation Language (lenguaje de manipulación de datos) y engloba todas aquellas sentencias que nos permiten consultar o modificar los datos almacenados en la base de datos. Aquí aparecen las sentencias SELECT, INSERT, UPDATE y DELETE.

DDL son las siglas de Data Definition Language (lenguaje de definición de datos) y engloba todas aquellas sentencias que nos permiten definir la propia forma de los datos, es decir, su estructura. Esto, como norma general para las base de datos relacionales, quiere decir modificar la estructura de las tablas que forman la base de datos.

Consultas de base de datos

Selección.

La consulta SELECT nos permite obtener información de la base de datos. Su sintaxis

general es la siguiente:

SELECT Tabla1.campo1, Tabla2.campo2 FROM Tabla1, Tabla2 WHERE condiciones

La consulta SELECT nos permite obtener los datos de determinados campos de una o varias tablas, imponer condiciones sobre ellos y realizar otras operaciones de conjuntos (tales como uniones o intersecciones). El resultado retornado por esta consulta (que podría ser vacío) será una nueva tabla que deberá ser procesada desde donde fue requerida (un procedimiento almacenado en el DBMS, una consulta desde un front-end de bases de datos, una consulta lanzada desde una aplicación cliente-servidor, etcétera).

Inserción.

Las consultas de inserción nos permiten, como su nombre indica, introducir datos en la base de datos. Su sintaxis general es la siguiente:

INSERT INTO Tabla (campo1, campo2, campo3) VALUES (valor1, valor2, valor3)

donde tabla indica la tabla en la que vamos a insertar el registro, campo1, campo2 ... indican los campos a los que vamos a dar valores en nuestra inserción y valor1, valor2 ... indican los valores para dichos campos (valor1 para el campo1, valor2 para el campo2, etc).

La lista de campos puede ser omitida si en la lista de valores se proporcionan tantos valores como campos tenga la tabla y además en el orden correcto. Las claves primarias autoincrementales son una excepción y **deben** ser omitidas.

Merece la pena considerar algunas peculiaridades de la inserción de datos en una base de datos:

Campos con valores por omisión: En una tabla un campo puede estar definido con un valor por omisión. Esto indica que, si no se suministra ninguno, dicho valor se usara automáticamente. Igualmente los campos autoincrementales se calculan automáticamente incrementando en uno el valor del último registro introducido. Para estos campos no es necesario introducir un valor en la consulta INSERT si no se desea.

Campos NULL y NOTNULL: En una tabla un determinado campo puede también definirse como NOTNULL lo que indica que siempre deberá introducirse un valor para dicho campo o de lo contrario la consulta fallará; siempre deberemos proporcionar un valor válido para dichos campos, claro está.

Campos autoincrementales: Los campos autoincrementales no solo pueden dejarse vacíos (no introducir su valor en la consulta) sino que por lo general debe hacerse así. Lo más normal es que dichos campos pertenezcan a las claves de la tabla y sean identificadores numéricos sin otro sentido que el de poder relacionar unas tablas con otras.

Actualizaciones.

La consulta *update* permite actualizar (modificar) datos ya existentes en la base de datos. Su sintaxis general es:

UPDATE Tabla SET campo1 = valor1, campo2 = valor2 WHERE condición

Las consultas UPDATE nos permiten tan solo actualizar datos existentes, es decir, si no existe un registro que coincida con la condición (o si la tablas esta vacía aunque no haya condición).

Borrado

Para finalizar, la consulta DELETE nos permite eliminar registros de una tabla, y su sintaxis general es:

DELETE FROM Tabla WHERE condición

Si omitimos la condición estaremos borrando todos los registros de la tabla.

Esto es solo una breve introducción al SQL, pero como podemos apreciar, se ajusta perfectamente a la estructura relacional de las tablas (de hecho fue pensado con ese fin).

Bases de datos orientadas a objetos.

En la sección anterior describimos a las bases de datos relacionales junto con el lenguaje de consulta SQL. En esta sección nos concentraremos en las bases de datos orientadas a objetos; nuevamente, no nos detendremos en detalles sino que solo introduciremos las principales características y usos de las mismas.

El problema de las bases de datos relacionales y los lenguajes de programación de objetos.

Cuando aparecieron las bases de datos relacionales, surgieron como una necesidad de mejorar los sistemas de bases de datos existentes hasta el momento. El modelo relacional, donde todos se basa en tablas y relaciones, se ajustaba perfectamente a los lenguajes de programación de la época, principalmente procedimentales y con datos bien estructurados. En aquel entonces, el concepto de Objeto como lo conocemos hoy en día no existía o era solo una idea demasiado *verde* aún.

Hoy en día, los lenguajes de programación orientados a objetos han avanzado en todos los ámbitos de desarrollo de sistemas, debido al enorme e interminable listado de ventajas que presentan al momento de desarrollar grandes aplicaciones, de trabajar en grupos posiblemente distantes, de mantener y extender los desarrollos, entre otras, provocando el surgimiento de lenguajes como Smalltalk, C# y Java. Incluso muchos lenguajes procedimentales han migrado o han incorporado características de lenguajes de programación OO ya sea *para no quedar fuera* o para aprovechar sus ventajas; podemos mencionar casos como Delphi, C++ o ADA.

Esta *nueva* forma de desarrollar sistemas utilizando objetos, donde un objeto posee un estado interno y un comportamiento propio pero que también hereda un conjunto de datos y un comportamientos de otros objetos, generó un problema a la hora de almacenar (aquí se le llama persistir) los objetos en bases de datos: la metodología relacional no se ajustaba perfectamente a los objetos, y el lenguaje de consulta SQL no aprovechaba muchas de las características propias de este paradigma de programación. De este modo, surge una nueva generación de bases de datos, en la cual lo que se persiste y recupera son objetos *puros* en vez de datos que se correlacionan de algún modo con objetos, proporcionando costes de desarrollo mucho menores y un mejor rendimiento gracias a esta transparencia directa entre las entidades de un sistema y su almacenamiento.

El modelo de datos orientado a objetos

El modelo de datos orientado a objetos es una extensión del paradigma de programación

orientado a objetos [CAT90]. Los objetos entidad que se utilizan en los programas orientados a objetos son análogos a las entidades que se utilizan en las bases de datos orientadas a objetos puras, pero con una gran diferencia: los objetos del programa desaparecen cuando el programa termina su ejecución, mientras que los objetos de la base de datos permanecen. A esto se le denomina persistencia.

Las bases de datos relacionales representan las relaciones mediante las claves ajenas. No tienen estructuras de datos que formen parte de la base de datos y que representen estos enlaces entre tablas. Las relaciones se utilizan para hacer concatenaciones (join) de tablas. Por el contrario, las bases de datos orientadas a objetos implementan sus relaciones incluyendo en cada objeto los identificadores de los objetos con los que se relaciona.

Relaciones

Un identificador de objeto es un atributo interno que posee cada objeto; ni los programadores, ni los usuarios que realizan consultas de forma interactiva, ven o manipulan estos identificadores directamente. Los identificadores de los objetos los asigna el SGBD y es el único que los utiliza.

El identificador puede ser un valor arbitrario o puede incluir la información necesaria para localizar el objeto en el fichero donde se almacena la base de datos.

Hay dos aspectos importantes a destacar sobre este método de representar las relaciones entre datos:

- Para que el mecanismo funcione, el identificador del objeto no debe cambiar mientras este forme parte de la base de datos. Las únicas relaciones que se pueden utilizar para consultar la base de datos son aquellas que se han predefinido almacenando en atributos los identificadores de los objetos relacionados. Por lo tanto, una base de datos orientada a objetos pura es **navegacional**, como los modelos pre-relacionales (el modelo jerárquico y el modelo de red). De este modo se limita la flexibilidad del programador/usuario a aquellas relaciones predefinidas, pero los accesos que siguen estas relaciones presentan mejores prestaciones que en las bases de datos relacionales porque es mas eficiente seguir los identificadores de los objetos que hacer operaciones de concatenación (join).

- El modelo orientado a objetos permite el uso de atributos multivaluados, agregaciones a las que se denomina conjuntos (sets) o bolsas (bags). Para crear una relación de uno a muchos, se define un atributo en la parte del uno que sera de la clase del objeto con el que se relaciona. Este atributo contendrá el identificador de objeto del padre. La clase del objeto padre contendrá un atributo que almacenara un conjunto de valores: los identificadores de los objetos hijo con los que se relaciona. Cuando el SGBD ve que un atributo tiene como tipo de datos una clase, ya sabe que el atributo contendrá un identificador de objeto.

Las relaciones de muchos a muchos (n:m) se pueden representar directamente en las bases de datos orientadas a objetos, sin necesidad de crear entidades intermedias. En el caso de las bases de datos relacionales, las relaciones de muchos a muchos se deben representar utilizando una nueva tabla que represente esta relación entre dos tablas.

Como bien sabemos, el paradigma orientado a objetos soporta la herencia, con lo cual una base de datos orientada objetos también puede utilizar estas facilidades. En teoría, una base de datos orientada a objetos debe soportar dos tipos de herencia: la relación es un y la relación extiende

a . La relación es un , también conocida como generalización especialización, crea una jerarquía donde las subclases son tipos específicos de las superclases. Con la relación extiende , sin embargo, una clase expande su superclase en lugar de estrecharla en un tipo mas específico.

Una de los aspectos más difíciles de manejar en las bases de datos relacionales es la idea de las partes de un todo, como en una base de datos de fabricación, en la que hace falta saber que piezas y que componentes se utilizan para fabricar un determinado producto. Sin embargo, una base de datos orientada a objetos puede aprovechar la relación todo parte en la que los objetos de una clase se relacionan con objetos de otras clases que forman parte.

Integridad de las relaciones

Para que las relaciones funcionen en una base de datos orientada a objetos pura, los identificadores de los objetos deben corresponderse en ambos extremos de la relación. Este tipo de integridad de relaciones, que de algún modo es análogo a la integridad referencial en las bases de datos relacionales, se gestiona especificando relaciones inversas. Para garantizar la integridad de este tipo de relación, un SGBD orientado a objetos deberá permitir que el diseñador de la base de datos pueda especificar donde debe aparecer el identificador del objeto inverso; por ejemplo:

```
relationship set<Obra> supervisa
inverse Obra::es supervisada
```

en la clase Aparejador y:

```
relationship Aparejador es supervisada
inverse Aparejador::supervisa
```

en la clase Obra.

Cada vez que un programa de aplicación inserta o elimina un identificador de objeto de la relación supervisa en un objeto Aparejador, el SGBD actualizará automáticamente la relación *es supervisada* en el objeto Obra relacionado. Cuando se hace una modificación en el objeto Obra, el SGBD lo propagará automáticamente al objeto Aparejador.

Del mismo modo que en las bases de datos relacionales es el diseñador de la base de datos el que debe especificar las reglas de integridad referencial, en las bases de datos orientadas a objetos es también el diseñador el que debe especificar las relaciones inversas cuando crea el esquema de la base de datos.

Versiones

El control de versiones será uno de los requerimientos más importantes del modelado de datos de la próxima generación de aplicaciones de BBDD [KIM90]. Después de iniciar la creación de un objeto, nuevas versiones del objeto derivan de este, y nuevas versiones pueden derivar de estas. Las versiones generan dos nuevos tipos de relaciones:

- Derived-from. Relación entre una nueva versión del objeto y una vieja versión del objeto.
- Version-of: entre cada versión de un objeto y un objeto abstracto que representa el objeto.

En general, versiones de un objeto forman un grafo dirigido. Cualquier número de versión puede ser derivada de cualquier otra versión, y cualquier versión puede ser derivada de más de una versión más vieja (figura 3.2). Sin embargo la mayoría de los modelos de versiones propuestos restringen al grafo a una jerarquía llamada jerarquía de versiones .

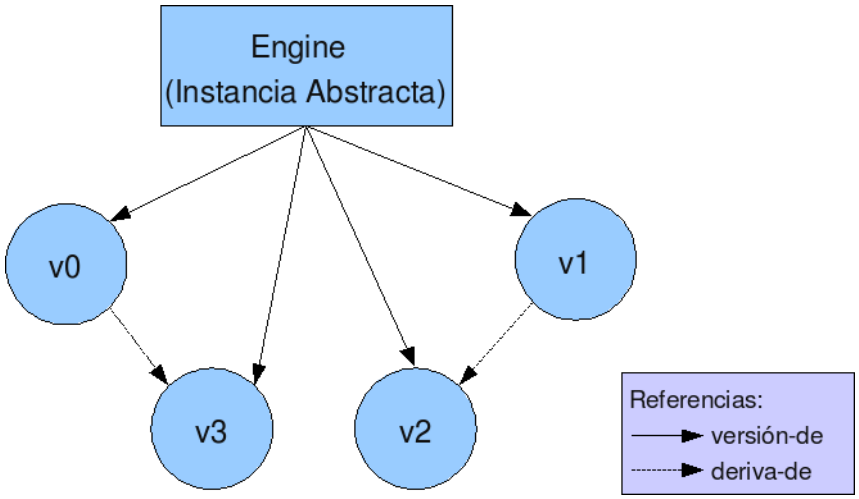

Figura 3.2. Ejemplo de versionado de un objeto Engine

Semánticas

Que el objeto pueda o no ser versionado es una de las propiedades de la clase a la que el objeto pertenece. Es decir, el usuario incluye en la definición de una clase si las instancias pueden ser versionadas. Llamaremos a un objeto que puede ser versionado un objeto versionable. Un objeto versionable puede tener una o mas versiones. Hay dos relaciones para las versiones. Existe una relación versión de entre cada versión y el objeto versionable. Además una nueva versión puede ser derivada de una versión existente de un objeto versionable. Por lo tanto existe una relación derivada de entre un par de versiones de un objeto versionable.

Cuando es creada una instancia de una clase cuyas instancias pueden ser versionadas, la primera versión del objeto versionable es creada. Una vez que esta primera versión ha sido creada, cualquier numero de versiones pueden ser derivadas de la primera y de cualquier versión siguiente, armando una jerarquía de versiones para un objeto versionable. Versiones primitivas permitían que una versión solo derive de la versión mas reciente, reduciendo la jerarquía de versiones en una cadena de versiones.

Una versión puede clasificarse en:

- Versión transient (temporal): sufre una cantidad importante de actualizaciones antes de alcanzar un estado robusto; puede ser actualizada o borrada en cualquier momento.
- Versión working (en uso): ha llegado al estado de robustez ; puede ser compartida y por lo tanto no puede ser actualizada; sin embargo puede ser borrada.
- Versión released (liberada): ha llegado al estado final de robustez y no puede ser actualizada ni borrada.

Una nueva versión, cuando es creada o derivada de una existente comienza como una versión transient. El usuario puede promoverla a una versión working una vez que llega al estado robusto y luego a una versión released. El usuario puede bajar una versión working a una transient, una versión released no se puede volver atrás.

Semántica de borrado.

Primero, si la versión a ser borrada no es una hoja de la jerarquía de versiones, es eliminada; pero su registro, en el historial de evolución de versiones, se mantiene. Por ej: en la figura anterior, la versión 3 puede ser borrada, si es una versión transient o working. Sin embargo, la información sobre la versión 3 se mantiene porque la versión 4 ha sido derivada de ella.

Segundo, si la única versión del objeto versionable es borrada, el objeto también se borra. Tercero, si el objeto versionable es borrado directamente, se borra con todas sus versiones.

En los sistemas orientados a objetos, un objeto puede referenciar cualquier cantidad de otros objetos. Un objeto puede referenciar un objeto versionable por una de dos formas: una es referenciar el objeto versionable como un objeto genérico (llamado ligado dinámico, el objeto es ligado en tiempo de ejecución a la versión por defecto del objeto versionable) y el otro es referenciar específicamente una de las versiones del objeto versionable (ligado estático). Si el usuario especifica una versión por defecto, esta es usada; sino el sistema fija por defecto la versión mas reciente.

No todas las bases de datos orientadas a objetos ofrecen versionado de objetos y/o de esquema, al igual que no todas ofrecen todas las características mencionadas en esta sección [VER08]. La selección de una BBDDOO no es una tarea trivial, y requiere tener en claros muchos de estos conceptos y tener experiencia en el uso de estas herramientas [BAR96]. Sumado a esto, la mayoría de las bases de datos de objetos están atadas al lenguaje de programación con el que se desarrolla la aplicación.

Por último, si bien las BDOO están cada vez más difundidas, no existe ni el mismo soporte ni la misma cantidad de herramientas de análisis si se las compara con las BBDD relacionales; adicionalmente, muchos desarrollos realizados sobre BBDD relacionales no pueden migrarse a BDOO tan fácilmente debido a por ejemplo, la enorme cantidad de información almacenada o a inconsistencias al momento de migrar de una base de datos a otra, con lo cual sigue siendo necesario el uso de BBDD relacionales pero aprovechando las ventajas de los lenguajes OO. Aquí es donde surgen los llamados mapeadores objeto-relacional u ORM, los cuales trataremos a continuación.

El Mapeo Objeto-Relacional.

En la programación orientada a objetos, las tareas de manejo de datos son implementadas generalmente por la manipulación de objetos, los cuales son casi siempre valores no escalares. Sin embargo, muchos productos populares de base de datos, como los productos SQL DBMS, solamente puede almacenar y manipular valores escalares como enteros y cadenas, organizados en tablas. El programador debe convertir los valores de los objetos en grupos de valores simples para almacenarlos en la base de datos (y volverlos a convertir luego de recuperarlos de la base de datos), o solo usar valores escalares simples en el programa. El mapeo relacional de objetos (ORM por sus siglas en inglés) es utilizado para implementar esta primera aproximación. Existen varias aplicaciones que realizan esta tarea, algunas gratuitas y abiertas como Hibernate [HIB08], OJB [OJB08] y JPOX [JPO08]

Desde el punto de vista de un programador, el sistema debe lucir como un almacén de objetos persistentes, pudiendo crear objetos y trabajar normalmente con ellos, los cambios que sufran terminarán siendo reflejados en la base de datos. Sin embargo, en la práctica no es tan simple. Todos los sistemas ORM tienden a hacerse visibles en varias formas, reduciendo en cierto grado la capacidad de ignorar la base de datos. Más aún, la capa de traducción puede ser lenta e ineficiente (comparada en términos de las sentencias SQL que escribe), provocando que el

programa sea más lento y utilice más memoria que el código "escrito a mano". Si bien se han desarrollado muchos sistemas de mapeo objeto-relacional a lo largo de los años, no todos han resultado ser igualmente efectivos.

Hace no mucho tiempo, una nueva abstracción ha comenzado a surgir en el mundo Java, conocido como Java Data Objects (JDO [JDO08][JOR03]). A diferencia de otros, JDO es un estándar, y muchas implementaciones están disponibles por parte de distintos distribuidores de software, como por ejemplo OJB, JPOX, FOStore, Objectivity, Orient, TJDO, Versant y muchas más, algunas de ellas gratuitas y abiertas, y otras comerciales.

En el framework de desarrollo web Ruby on Rails, el mapeo objeto-relacional juega un rol preponderante y es manejado por la herramienta ActiveRecord. Un rol similar es el que tiene el módulo DBIx::Class para el framework Catalyst basado en Perl, aunque otras elecciones también son posibles.

El trabajo que aquí presentamos fue escrito en lenguaje Java, y hemos utilizado el motor de base de datos MySQL, tanto por la gran cantidad de documentación existente como por ser open source, lo cual sigue en línea con nuestra herramienta. Siguiendo con la filosofía de código abierto, hemos seleccionado una implementación de JDO open source que se ajuste tanto como sea posible al motor de base de datos utilizado. Luego de algunas pruebas, hemos seleccionado JPOX por diversos motivos, entre los que podemos citar: estabilidad y funcionalidad, mayor eficiencia ante otros ORMs según varias pruebas ([POL07]) y por ajustarse perfectamente al estándar de JDO (motivo por el cual descartamos Apache OJB). A continuación, presentaremos las características más destacadas de JPOX, muchas de las cuales han sido aplicadas a este trabajo.

JPOX y JDO.

Alcances de JDO.

JDO define una interface (una API) para persistir objetos Java normales (también conocido como POJOs) en un almacén de datos. JDO no define el tipo de almacén (se dice que es agnóstico del almacén de datos). Se puede utilizar la misma interfaz para persistir objetos Java en RDBMS, OODBMS, en XML o en cualquier almacén que se quiera. Existen 4 ideas principales sobre JDO:

- JDO Clasifica las clases en 3 tipos. El tipo de nuestra clase define como interactuará con JDO. Algunas clases no poseen interacción con JDO, mientras que otras requieren que el programador defina su comportamiento sobre JDO.
- Controla la persistencia de nuestros objetos: Esto se realiza usando un PersistenceManagerFactory/ PersistenceManager. La persistencia de objetos Java provoca cambios en el ciclo de vida de los objetos.
- Consulta el almacén de datos para recuperar objetos
- Controla a las transacciones que gobiernan el proceso de persistencia.

Introducción a JPOX.

JPOX es una solución de persistencia heterogénea para Java. Permite al programador tomar un modelo de objetos Java y persistirlos en un almacén de datos sin tener que perder largas horas definiendo como deben ser persistidos. Esto significa que el programador puede concentrarse en desarrollar su aplicación en vez de pensar como persistir y recuperar sus objetos.

Existen 4 aspectos principales de persistencia que los usuarios deben tener en cuenta cuando utilizan JPOX. Estos son:

1. Definición de la persistencia: significa definir como serán persistidas las clases Java en el almacén de datos
2. API de persistencia: la API programática utilizada para persistencia de los objetos
3. Lenguaje de Consulta: el lenguaje de consulta por medio del cual podemos encontrar y recuperar objetos por medio de algún criterio.
4. Almacén de datos: el almacén donde se está persistiendo.

El propósito de JPOX es permitir el uso de todas las Definiciones de persistencia para ser usadas con todas las APIs de persistencia y para realizar consultas con todos los lenguajes de consulta. Esto le da a los usuarios una significativa flexibilidad ya que pueden elegir las mejores partes de estos tres aspectos. Con respecto al almacén de datos, la API de JDO provee capacidades agnósticas.

¿Cómo es el proceso de JPOX? JPOX intenta hacer de todo el proceso de persistencia de datos un proceso transparente. La idea gira alrededor del desarrollador que posee una serie de objetos Java que necesita que se persistan; con JPOX, el desarrollador define la persistencia de esas clases utilizando Metadatos (o annotations de Java5), y aumenta (enhance) el bytecode de las clases que ha definido como *persistibles*. JPOX también provee una herramienta llamada SchemaTool que permite la generación y validación del esquema antes de ejecutar la aplicación, para de este modo asegurarnos que todo será correctamente mapeado. El programador debe proveer el código para persistencia (para manejar la persistencia de sus objetos), y las consultas (para recuperar los datos persistidos). JPOX implementa todas las especificaciones JDO (1.0, 2.0 y 2.1) y también la especificación JPA (JPA1). La figura 3.2 nos muestra el proceso completo para JPOX. Gracias a estas especificaciones que definen como se deben persistir clases Java en cualquier almacén de datos, cualquier implementación de JDO (como OJB o JPOX) puede almacenar datos en cualquier banco de datos que se elija. En el caso de JPOX, el soporte para distintos RDBMS es muy amplio: MySQL, MS SQL Server, Oracle, Sybase, HSQL, H2, McKoi, PostgreSQL, DB2, y muchos más.

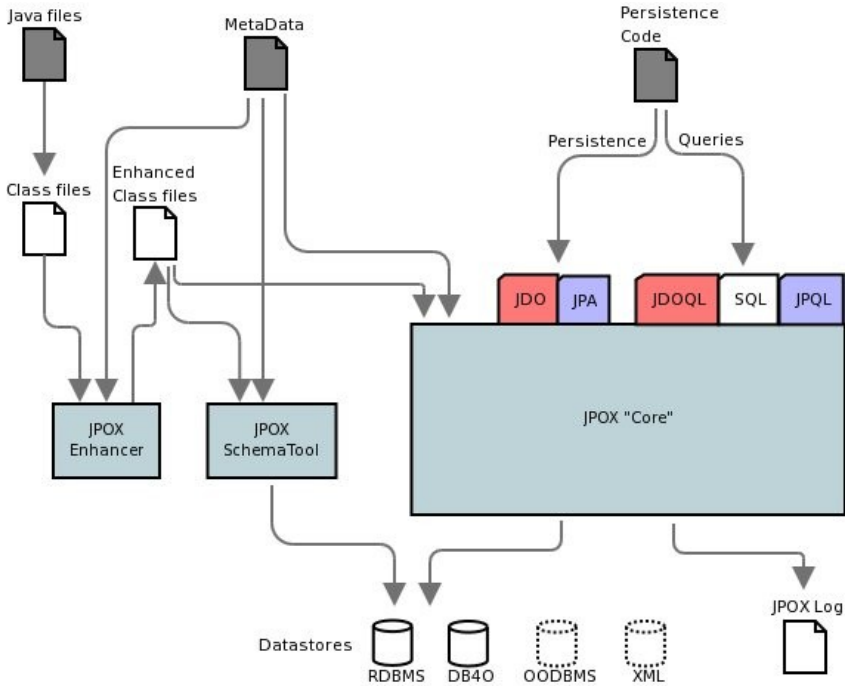

Figura 3.2. Proceso de aumento, verificación con el esquema y persistencia de JPOX

Una aplicación puede funcionar sobre JDO (se dice que es JDO-enabled) a través de muchos medios dependiendo del proceso de desarrollo del proyecto en cuestión. Por ejemplo, un proyecto podría utilizar Eclipse como IDE para el desarrollo de las clases. En este caso, el proyecto utilizaría el plugin JPOX para Eclipse. Alternativamente, podría utilizar Ant, Maven o cualquier otra herramienta de construcción de aplicaciones. El proceso de desarrollo es, para cualquier caso, bastante simple y fácil de seguir:

1. Diseñar las clases del dominio/modelo como se hace normalmente
2. Definir su persistencia utilizando Metadatos.
3. Compilar las clases, generando los .class (bytecode)
4. Aumentar las clases compiladas, utilizando el enhancer de JPOX.
5. Generar las tablas de la base de datos donde las clases serán persistidas
6. Escribir el código de persistencia dentro de una capa de acceso a datos (DAO)
7. Ejecutar la aplicación.

Nos detendremos aquí en los pasos 2 y 4, ya que involucran directamente a JPOX. Si bien el paso 5 también lo hace, afortunadamente puede omitirse ya que JPOX es capaz de crear la estructura de la base de datos por completa si es que no existe tal estructura (aunque sí debe existir la base de datos), lo cual ahorra una enorme cantidad de tiempo especialmente durante el desarrollo y debugging de la aplicación.

Metadatos y aumento de clases.

JPOX requiere (en realidad, JDO así lo define) que se le indique mediante un archivo en formato XML cuales serán las clases a persistir, y que atributos de dichas clases serán persistidos. Puede crearse un solo archivo XML para toda la aplicación, un archivo por paquete o uno por clase. Nosotros hemos utilizado la segunda opción, ya que evita la generación un solo archivo extremadamente extenso (opción uno) o de una cantidad excesiva de archivos pequeños (opción 3). Si bien no entraremos en detalles específicos sobre estos archivos, es importante aclarar algunos aspectos interesantes.

- En caso de atributos simples o referencias simples, el programador solo debe indicar el nombre del atributo a persistir.
- En caso de listas o arreglos, se debe indicar no solo el nombre, sino también el tipo de datos contenido en estas listas. Adicionalmente, si se desea que JPOX genere una tabla de join (o relación) para representar esta lista, debe ser indicado mediante el tag <join>. Si esta etiqueta es omitida, el atributo será tomado como un campo de tipo BLOB (Binary Large Object) en la base de datos.
- En caso de tablas de hash o diccionarios, donde tenemos claves y valores, debe indicarse el tipo de datos tanto de los objetos que forman parte de la clave como de aquellos que representan los valores. Al igual que sucede con listas y arreglos, se deberá utilizar la etiqueta <*join*> para *solicitar* a JPOX que genere una tabla que represente esta relación.

Dentro de este archivo se pueden también especificar restricciones de integridad, campos nulos, valores por defecto, nombres de columnas, entre otras.

Este archivo también sirve para indicar como deberán mapearse las jerarquías de clases. Aquí tenemos diversas aproximaciones: se genera una única tabla que incluye a todas las subclases,

se genera una tabla específica para cada subclase, o se generan varias tablas, algunas con datos de las superclases y otras con datos de las subclases, enlazadas con sus superclases mediante un atributo de join. Esto permite organizar mejor la información, pero requiere acceder a varias tablas para levantar cualquier objeto.

Una vez que tenemos definido el o los archivos de mapping, debemos generar los archivos .class (compilando las clases java) para luego indicarle a JPOX que debe aumentar estas clases. Para ello, JPOX incorpora un enhancer que toma los bytecodes dentro de los archivos .class y los archivos .jdo que hemos definido, y modifica estos bytecodes incorporándole código necesario para mapear las entidades de acuerdo a lo especificado por el usuario.

Consultas de JPOX.

Hasta aquí hemos discutido la persistencia de objetos. Una vez que logramos persistir nuestros objetos, necesitamos poder recuperarlos. Las especificaciones de JDO (versiones 1.0.1 y 2.0) requieren que las implementaciones provean la capacidad de consulta (Query) utilizando su propio lenguaje de consulta (JDOQL). JDOQL está orientado alrededor de los objetos que son persistidos , y provee una interface para seleccionar aquellos objetos dentro del marco de una consulta. La especificación JDO 2.0 requiere que las implementaciones provean un mecanismo de consulta SQL (para aquellas bases de datos que soportan SQL). JPOX provee estas características; adicionalmente, provee un lenguaje de consulta que toma aspectos de JDOQL y de SQL, que posee una sintaxis similar a la de SQL pero permite acceso a los campos nombrados de las clases. Este lenguaje se denomina JPOXSQL.

Qué lenguaje de consulta utilizar depende de cada desarrollador. La capa de datos de una aplicaciones podría ser escrita principalmente por un desarrollador Java, quien muy probablemente preferirá utilizar algún método orientado a objetos, con lo cual JDOQL será la mejor alternativa. Del otro lado de la capa de datos podría existir un desarrollador que trabaje directamente sobre la base de datos y que seguramente estará más familiarizado con conceptos de SQL, con lo cual podría utilizar este lenguaje para acceder a los datos. Aquí radica una de las ventajas más importantes de JPOX: provee la suficiente flexibilidad para que diferentes personas desarrollen diferentes capas de una misma aplicación con sus propios conocimientos y habilidades evitando tener que aprender conceptos totalmente nuevos.

Capítulo 4. Visión global del desarrollo

En este capítulo veremos de manera general en qué consiste el desarrollo que aquí presentamos, cuáles son sus límites y qué podemos hacer con esta herramienta. El objetivo de este capítulo es proveer toda información necesaria para ejecutar simulaciones y comprender lo que hace el modelo subyacente para que estas simulaciones sean llevadas a cabo, sin entrar demasiado en detalles de cómo lo hace. A lo largo de este capítulo se nos plantearan muchos problemas conceptuales y de implementación tan complejos como interesantes, pero sin adentrarnos demasiado ya que excede el propósito de esta sección; para ello hemos incorporado un capítulo dedicado exclusivamente a ciertos aspectos avanzados de implementación.

Entidades del modelo.

El modelo desarrollado es incluye un conjunto de entidades básico y necesario para funcionar, y algunas entidades extras que hemos seleccionado para poder realizar simulaciones simples. Adicionalmente, hemos incorporado otras entidades que agregan algún tipo de funcionalidad propia de nuestra implementación. A continuación conoceremos cada una de ellas y describiremos sus características principales y sus funciones y objetivos dentro de las simulaciones.

Cabe recordar que no hemos implementado todas las entidades permanentes presentes en GPSS debido a que lo único que se lograría es hacer más compleja la comprensión del modelo y aumentar el tiempo de desarrollo sin agregar nuevos conceptos útiles para los fines buscados en el presente trabajo.

Commands

Un comando o COMMAND sirve para ajustar determinados parámetros al inicio de las simulaciones (valor del termination count, limpieza parcial o total del sistema una vez finalizada una simulación y antes de comenzar la siguiente, condiciones de parada, y muchos otros) o para declarar algunas entidades que luego serán requeridas (entidades como FUNCTION, EQU, MATRIX o STORAGE). Son las sentencias más simples de GPSS y, como es de esperarse, son colocados en una cola de comandos de la cual son tomados uno a uno a medida que finalizan su ejecución.

Al comenzar la simulación, se localizan todos los comandos y se los ubica dentro de la cola de comandos, de esta forma el orden en que los mismos han sido declarados determinará el momento de su ejecución. Un comando particularmente importante es el comando START A, donde A es un valor entero positivo. Este comando sirve para indicar al planificador de transacciones que debe comenzar a seleccionar y ejecutar las transacciones, lo cual significa que la simulación ha comenzado, y una vez que se ha ejecutado el comando no será eliminado de la cola de comandos hasta que la simulación haya terminado. El valor del parámetro A del comando START indica el valor del contador de terminación o termination count, y determinará el momento de finalización de la simulación.

Block

El segundo tipo de sentencia que existen dentro de GPSS es el bloque o BLOCK. Al igual

que los comandos, un bloque consta básicamente de un nombre y un conjunto de parámetros u operandos, entre los cuales pueden existir algunos obligatorios y otros opcionales. Una correcta combinación de bloques y comandos es crucial para hacer que nuestras simulaciones funcionen como esperamos.

Los bloques, como el resto de las entidades de GPSS, poseen un conjunto de variables que indican su estado, como ser un código de identificación o la cantidad de transacciones que han ejecutado ese bloque. Antes de comenzar la simulación, se colocan los bloques en una lista de bloques en el mismo orden en que han sido escritos. Nuevamente, el orden en que han sido declarados estos bloques determinará la forma en que se ejecutarán, aunque aquí pueden aparecer saltos o loops que alterarán este flujo secuencia.

Al iniciar la simulación, los bloques GENERATE son seleccionados. Estos bloques son los encargados de provocar la creación de las transacciones y el comienzo de su ejecución. cada transacción pasa por los bloques uno a uno de manera secuencial, hasta que finaliza en un bloque TERMINATE o hasta que es detenida por algún motivo. Por ejemplo, si una transacción ingresa a un bloque ADVANCE será colocada en la Cadena de Eventos Futuros y tendrá que esperar a algún tiempo de simulación futuro para continuar en el siguiente bloque; otra situación común de detención de las transacciones es la ejecución de bloques SEIZE o PREEMPT. Las facilidades que las transacciones están intentando tomar pueden estar no disponibles u ocupadas, con lo cual nuestras transacciones deberán ingresar a alguna cadena de la facilidad y esperar que la misma sea liberada, para luego poder realmente realizar el SEIZE o PREEMPT.

Cada bloque contiene un conjunto de parámetro que contienen expresiones que pueden contener valores simples o expresiones compuestas y pueden incluir la invocación a atributos numéricos estándar o SNAs. La expresión asociada a cada parámetro es evaluada en el momento de ejecutar el código correspondiente a cada bloque. Esto da una gran flexibilidad pero incrementa considerablemente la complejidad del desarrollo.

Transaction Scheduler

El Transaction Scheduler o Planificador de Transacciones es la entidad encargada de seleccionar cual será la próxima transacción a ser ejecutada (transacción activa), eligiendo todas las de colas o cadenas de transacciones de la simulación. Es también trabajo de esta entidad el de indicar cuando ha finalizado el tiempo de reloj actual, de actualizar este reloj y de provocar el movimiento de transacciones planificadas a futuro a la lista de transacciones a ejecutarse en el nuevo tiempo de reloj, esto quiere decir que deberá seleccionar de la FEC cuales son las transacciones con menor BDT, el cual será utilizado para actualizar el reloj de simulación, y luego colocar estas transacciones en la CEC en orden aleatorio.

Existe una sola instancia de esta entidad para toda la simulación, fuertemente ligada a la entidad Simulation de la cual también existe solo una instancia; ambas entidades trabajan juntas para permitir que las transacciones se muevan por los bloques ejecutando su código en tiempo y forma esperado. El planificador de transacciones también informará a la simulación en caso de

algún tipo de error que le concierna, por ejemplo la ausencia de transacciones tanto en la CEC como en la FEC (en cuyo caso la simulación deberá detenerse abruptamente ya que no existen transacciones capaces de decrementar el contador de terminación y provocar una terminación "normal").

Transaction

Las entidades Transaction o Transacción son creadas y destruidas durante la corrida de cada simulación, y atraviesan por una serie de estados a medida que ejecutan los bloques. Las transacciones otorgan dinamismo a nuestros modelos, interactuando con el resto de las entidades, utilizando recursos y liberándolos y finalmente determinando la duración de cada simulación. Estas entidades temporales poseen usos muy diversos, entre los que se encuentran clientes que utilizan servicios de otras entidades, entidades encargadas de controlar otras entidades o incluso controlar el reloj de simulación, indicando el momento indicado en que la simulación debe finalizar. Veamos dos ejemplos simples para comprender lo que se acaba de mencionar:

```
GENERATE 1
*código propio de la transacción
TERMINATE                       1
START 10
```

Ejemplo A. Sin transacción controladora

```
GENERATE 1
*código propio de la transacción
TERMINATE

GENERATE ,,10,1
TERMINATE                       1
START 1
```

Ejemplo B . Transacción controladora

En el ejemplo A tenemos un solo grupo de transacciones, cada una de las cuales decrementa en uno el valor del contador de terminación; cuando exactamente 10 de estas transacciones hayan terminado, la simulación finalizará.

En el ejemplo B tenemos dos grupos de transacciones (podemos ver a cada grupo de transacciones identificado por un bloque GENERATE y su conjugado TERMINATE). En el primer grupo, las transacciones se crean cada un tiempo de reloj, ejecutan su código y finalizan, pero **ninguna de ellas decrementa el contador de terminación**. El segundo grupo de transacciones genera exactamente una transacción en el instante 10, la cual solo decrementa en uno el contador de terminación. Dado que este valor había sido inicializado en uno (por el comando START), la simulación finalizará en ese momento. En este ejemplo, esta segunda transacción está funcionando como un reloj que controla el tiempo de simulación, asegurándose que la misma finalice exactamente en el tiempo 10 sin importar lo que el resto de las transacciones estén haciendo.

Simulation

La entidad Simulation también forma parte del conjunto de entidades que existen durante la ejecución de una simulación. Esta entidad, al igual que el resto de las entidades, posee un conjunto de variables cuyos valores determinarán su estado en un momento, como ser el contador de terminación o el estado de cada una de sus cadenas de transacciones.

La entidad Simulation posee una amplia variedad de responsabilidades que sirven a fines muy diversos, entre los que encontramos:

- Gestión del reloj de simulación: junto al Trasanction Scheduler controla cuándo y cómo debe actualizarse
- Gestión de los comandos y ejecución ordenada de la cola de comandos
- Localización de entidades permanentes, ya sea conociendo su identificador interno o mediante un nombre o label.
- Creación de transacciones, lo cual implica entre otras cosas la asignación de un número de transacción, calculo del BDT y planificarla para ejecutarse en algún tiempo de reloj a partir de algún bloque GENERATE.
- Terminación de transacciones: desactivarlas, removerlas del modelo y mantenerlas en una lista de transacciones listas para ser eliminadas.
- Modificación y control del contador de terminación
- Gestión de la lista de bloques, organización y búsquedas de los mismos. Es tarea de la simulación detectar y retornar el siguiente bloque a ejecutarse para cada transacción
- Ejecución de transacciones (junto al Transaction Scheduler)

Además de la lista de bloques y de la cola de comandos, esta entidad posee las cadenas de eventos actuales y futuros, así como también el listado de facilidades, almacenes, y otras entidades existentes, y la lista de transacciones que han finalizado durante este tiempo de reloj.

System Clock

El reloj del sistema o System Clock indica en todo momento cual es el tiempo de simulación actual. Además de mantener su estado interno, esta entidad está muy ligada al planificador de transacciones, a quien le comunica que ha cambiado para que éste último realice las actualizaciones pertinentes.

Facility

La entidad Facility o Facilidad, como hemos visto, es un tipo de entidad permanente que puede ser tomada por a lo sumo una transacción y que posee un amplio conjunto de funciones complejas. Las facilidades poseen cuatro cadenas distintas, cada una de las cuales tiene un propósito específico que determina el orden en que las transacciones la tomarán. Si bien existen varias excepciones y casos especiales, estos serían los principales fines para los cuales se han creado cada una de estas cadenas:

- Delay Chain: En esta cadena se encuentras las transacciones que intentaron hacer un SEIZE pero que encontraron la facilidad ocupada por otra transacción. Posee un ordenamiento por prioridad y luego FIFO.
- Pending Chain: Aquí tendremos a aquellas transacciones que están esperando para hacer un PREEMPT, ya que al momento de ejecutarlo la facilidad ya había sido preempteada

por otra transacción. Utiliza ordenamiento FIFO.

- Interrupt Chain: En esta cadena tendremos a todas las transacciones a las que se les quitó la facilidad mediante un bloque PREEMPT y no se les indicó como ni donde continuar. Estas transacciones pueden haber tomado la facilidad tanto por un SEIZE como por un PREEMPT.
- Retry Chain: Esta cadena, que también poseen el resto de las entidades, tiene una lista de transacciones que están esperando por el cambio de estado de alguna variable interna de la facilidad; por ejemplo, en el caso de las facilidades, tendremos transacciones esperando que la misma este disponible si es que alguna transacción en algún momento la había seteado como no disponible.

En nuestro modelo, las facilidades son las encargadas de ejecutar los bloques SEIZE, PREEMPT (en ambos modos), RELEASE y RETURN; estas entidades también informan a la facilidad si la transacción activa puede ejecutar un seize o un preempt sobre la misma.

Entity References Manager

Todas las entidades de GPSS, tanto temporales como permanentes, poseen un valor de identificación; en muchos casos, las entidades permanentes también poseen un "nombre". Las referencias a entidades permanentes pueden realizarse mediante su nombre o un código. Veamos estos dos casos:

```
GENERATE 1
SEIZE una_facilidad
ADVANCE 2
RELEASE una_facilidad
TERMINATE                1
START 1
```

Ejemplo A. Referencia nombrada

```
GENERATE 1
SEIZE 10
ADVANCE 2
RELEASE 10
TERMINATE                1
START 1
```

Ejemplo B . Referencia a través de su identificador

En el ejemplo A, conocemos el nombre de la facilidad *una_facilidad* con lo cual podemos referenciarla de manera más legible, mientras que en el ejemplo B, nuestra facilidad es referenciada a partir de su código interno. En este caso, el código se genera explícitamente a pedido del usuario (código 10) pero esto no es necesariamente así. Este tipo de referenciación es muy útil para acceder a entidades cuyo nombre o id no conocemos a priori, pero cuyo valor lo tenemos dentro de un parámetro de la transacción. Por ejemplo, veamos el siguiente código:

```
GENERATE 1
ASSIGN 1,10; coloco el valor 10 en el parámetro
SEIZE *1   ; referencio al parámetro 1 de la transacción
ADVANCE 2
RELEASE 10 ;para liberar la facilidad, utilizo explícitamente el
            identificador
TERMINATE 1
START 1
```

Ejemplo C: referenciación indirecta.

El ejemplo C es exactamente igual al A y al B, con la diferencia que ahora la facilidad que se está tomando se consigue a partir del valor de un parámetro de nuestra transacción; este parámetro podría haber sido inicializado a partir de una función, variable o bloque SAVEVALUE con el fin de asignar entidades a nuestras transacciones con determinada distribución estadística.

En los ejemplos B y C, hemos forzado al intérprete de GPSS a utilizar un determinado número de identificación para nuestra facilidad, mientras que en el ejemplo A nos hemos abstraído y hemos utilizado un nombre simbólico que representa la facilidad. ¡Pero esto no quiere decir que la facilidad no tenga un identificador, y mucho menos que aún no podamos utilizar este identificador para acceder a la misma! Veamos cómo funciona esto.

GPSS implementa el concepto de valor nombrado o Named Value. Un named value no es más que un valor numérico al cual se le asigna un nombre. El programador puede realizar esta asociación explícitamente mediante el comando EQU, o puede dejar que el lenguaje asigne un identificador para cada nombre que ha utilizado (para este último caso, generará identificadores a partir del valor 10000). Esto permite al programador referenciar una misma entidad de varias maneras distintas, o incluso mantener un pool de nombres y valores asociados para un conjunto de entidades. Más aún, varias entidades de distinto tipo pueden utilizar un mismo valor nombrado, lo cual le otorga mayor legibilidad al código. Veamos un ejemplo en el cual los valores nombrados se utilizan tanto para referenciar entidades como para simplificar el código:

```
cajero EQU 29
demora_promedio EQU 40
tiempo_embolsado EQU 15

GENERATE 1
QUEUE cajero
SEIZE cajero
ADVANCE demora_promedio
RELEASE cajero
ADVANCE tiempo_embolsado
DEPART cajero

TERMINATE 1
START 1
```

En este ejemplo, simulamos un único cliente que va a la caja del supermercado para que le cobren, luego embolsa sus items y se retira. Hemos declarado 3 valores nombrados distintos: cajero, demora_promedio y tiempo_embolsado. El primero de ellos se utiliza para referenciar tanto a la cola (bloques QUEUE y DEPART), la cual colectará una serie de estadísticas interesantes de la transacción, y para referenciar a la facilidad. Al leer este código, es fácil identificar que la cola cajero mide estadísticas de la facilidad cajero; en caso de tener varios cientos de bloques, colas y facilidades, estos nombres ayudarán enormemente a comprender el código.

Los siguientes valores se utilizan para indicar cuales son los tiempos de demora para realizar un cobro (por parte del cajero) y para embolsar los items comprados. Aquí, el código queda no solo más prolijo y legible sino que también más fácil de modificar: un cambio en estos tiempos solo implicará una modificación en los comandos EQU sin necesidad de alterar el código de las transacciones.

Hasta aquí hemos visto como influyen los valores nombrados en nuestras simulaciones, pero ¿cómo se relacionan estos Named Values con nuestra entidad Entity Refereces Manager? La asignación automática de identificadores no es una tarea simple: muchas entidades "permanentes" se crean cuando la primer transacción las referencia, ya que el valor de referencia solo se conoce en tiempo de ejecución; a estas entidades se les debe asignar apropiadamente un identificador. Adicionalmente, el usuario o programador puede definir sus propias entidades permanentes con sus valores nombrados, lo cual puede generar conflictos con otras entidades. Mientras tanto, miles de transacciones son creadas y destruidas, cada una con su propio identificador, el cual no debe entremezclarse con ningún identificador existente en la simulación. Por último, y para hacer las cosas aún más complejas, una mismo identificador o valor nombrado puede ser utilizado para referenciar múltiples entidades permanentes de distinto tipo. Para gestionar la asignación ordenada de identificadores, la asociación de nombres a determinados valores, o la asignación de un valor nombrado a una o varias entidades hemos introducido en nuestro modelo una entidad con suficiente complejidad para poder lidiar con todas estas situaciones.

La entidad gestora de referencias conoce en todo momento cuál fue el último identificador de transacción asignado, el último número de bloque otorgado (recordemos que los bloques también son entidades, y por lo tanto deberán tener un identificador y posiblemente un nombre) y el último valor asignado automáticamente a un nombre o label. Esta entidad mantiene a su vez un conjunto de listas para saber que asociaciones nombre/valor existen y cuáles identificadores ha utilizado explícitamente el usuario (mediante, por ejemplo, comandos EQU).

Cadenas de Transacciones.

Como hemos podido observar, las cadenas son un elemento fundamental de las simulaciones. Esas determinan el orden en que sucederán los eventos, los cuales provocarán la ejecución de transacciones a través de los bloques permitiendo que las simulaciones avancen hasta alcanzar el termination count. El diseño de este tipo de entidades nos enfrentó a un problema muy particular, ya que requería una gestión ágil y eficiente para no aumentar los tiempos de simulación, una implementación segura ya que cualquier falla podría hacer que una simulación deje de funcionar o lo que es peor finalice "normalmente" y entregue resultados incorrectos, y ademas debía ser extensible para cuando se creen nuevas entidades con sus propias cadenas y sus propios mecanismos de ordenación de transacciones. Adicionalmente, existe en GPSS una entidad llamada USERCHAIN, en la cual el programador declara sus propias cadenas para almacenar transacciones, y estas cadenas pueden tener el orden que el programador desee: descendente o ascendente, por algún parámetro que él ha definido o incluso por algún parámetro preexistente en las transacciones.

Atendiendo a estos requerimientos, hemos creado un conjunto de clases organizadas en una jerarquía y una interfaz simple (TransactionChain) que permiten implementar fácilmente nuevos tipos de cadenas o incluso generar cadenas dinámicamente. Esta interfaz solo posee métodos simples para obtener la primera o las primeras n transacciones, y también para eliminar o remover transacciones. Luego, hemos discriminado las cadenas en cadenas ordenadas por el momento de

ingreso (FIFO y LIFO), y cadenas ordenadas a partir de cierto parámetro (SortedList); estas últimas nos han permitido implementar las clases BDTEventsChain (utilizada para la FEC) y PriorityEventsChain (utilizada en la CEC). La creación de nuevos tipos de cadenas solo requiere la implementación de esta entidad y su correcta localización dentro del árbol jerárquico correspondiente.

Persistencia de Simulaciones.

De acuerdo al planteo que hemos realizado en este trabajo, es condición necesaria que las simulaciones sean almacenadas en una base de datos para su posterior análisis. Pero para que las simulaciones funcionen como es esperado, el almacenamiento de todos los objetos debe ser totalmente transparente. Es importante aquí destacar que todos y cada uno de los objetos o entidades de la simulación deben ser almacenados junto con su estado actual, ya que luego deberán poder ser analizados individualmente, lo cual requerirá conocer exactamente en que estado estaba cada objeto en cada instante de reloj.

Para ello, hemos creado un objeto que se encarga de realizar el guardado de toda la simulación. Este nuevo objeto, denominado DAO (Data Access Object), es invocado con cada cambio del reloj de simulación para que acceda a todas y cada una de las entidades existentes a partir de la entidad Simulation y recolecte toda la información requerida para almacenarlas. La invocación a la entidad DAO se debe realizar antes que se registre el nuevo tiempo de reloj, para así almacenar el tiempo actual y no el tiempo "futuro".

Podemos ver una simulación como un grafo dirigido con ciclos y con un nodo raíz a partir del cual se pueden llegar a todos los nodos restantes: nuestra entidad Simulation. En Programación Orientada a Objetos, una simulación es precisamente un grafo de composición horizontal, lo cual nos permite recorrer y recolectar entidades a almacenar a partir de esta entidad raíz, asegurándose que no quedará ninguna sin "*visitar*", e indicarle al DAO que las almacene. Nuestro grafo de entidades, como hemos mencionado previamente, posee posiblemente muchos ciclos, relacionados principalmente con las transacciones y su localización al momento de recorrer el grafo. En el siguiente ejemplo, veremos un caso en que una misma transacción se encuentra en más de un lugar a la vez:

```
GENERATE ,,,1
SEIZE facilidad1
SEIZE facilidad2
ADVANCE 3
RELEASE facilidad2
RELEASE facilidad1
TERMINATE 1

GENERATE ,,3,1,5
PREEMPT facilidad1
PREEMPT facilidad2
ADVANCE 4
RETURN faciliad1
RETURN facilidad2
TERMINATE 1

START 2
```

Analicemos en detalle este ejemplo. Tenemos dos generaciones de transacciones, la primera genera en el tiempo cero de simulación una única transacción, la cual toma la entidad *facilidad1* y luego la entidad *facilidad2*, para finalmente ejecutar el bloque ADVANCE 3 y colocarse en la FEC para ejecutarse en tiempo 3.

Por el otro lado, se nos genera una única transacción en tiempo 3 con mayor prioridad que la anterior (5). En este tiempo de reloj (el transaction scheduler ya actualizó el tiempo actual en 3), nuestras dos transacciones se encuentran en la CEC, pero como la segunda transacción tiene mayor prioridad, será colocada al frente como transacción activa; esto hace que tome la entidad *facilidad1* por la fuerza e inmediatamente haga lo mismo con la entidad *facilidad2*, para luego colocarse en la FEC para ejecutarse en tiempo 7 (tiempo actual 3 + tiempo 4).

Nuestra segunda transacción desplazó a la primera de las entidades *facilidad1* y *facilidad2*, pero ¿qué pasó con esta transacción? Además de estar en la CEC, fue colocada en la Interrupt Chain de *facilidad1* y luego en la Interrupt Chain de *facilidad2*. Conclusión: tenemos una transacción que se encuentra en 3 cadenas al mismo momento.

Planteada la situación, veamos por qué se genera un ciclo. Comenzamos a recorrer nuestro grafo a partir de la entidad Simulation, tomamos la CEC y recolectamos la información de la única transacción allí existente. Luego tomaremos la FEC y haremos lo mismo, así finalmente continuar con la lista de facilidades. Seleccionamos la facilidad *facilidad1* y recorremos cada una de sus cadenas en busca de transacciones; cuando le llegue el turno a la Interrupt Chain, nos

encontraremos con una transacción que ya había sido analizada. Esto, en nuestro grafo, es nada más y nada menos que un ciclo.

La entidad DAO deberá también interactuar con el gestor de almacenamiento subyacente y almacenar los datos de manera segura; afortunadamente, el uso de JPOX facilita mucho las tareas aquí. Adicionalmente, nos ofrece un nivel de abstracción sobre la forma en que se almacenan los objetos, en nuestro caso en una base de datos relacional (MySQL), lo cual nos permite cambiar de DBMS sin cambiar el código de nuestro desarrollo.

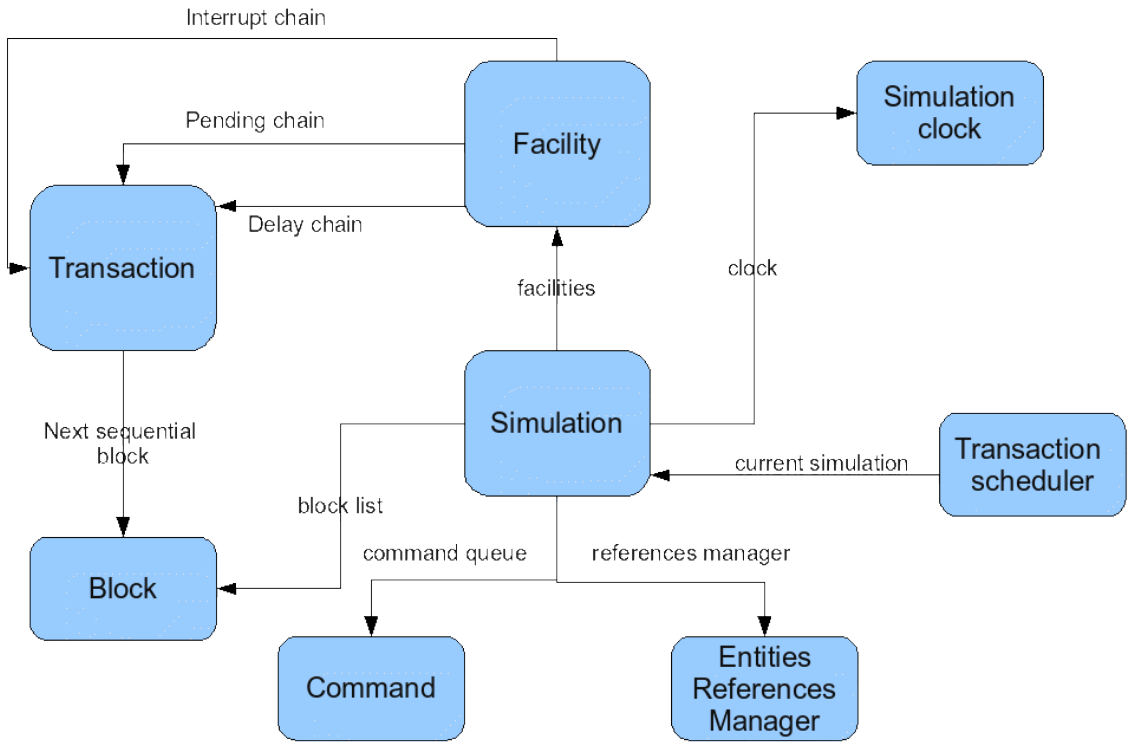

Figura 4.1 Grafo de composición horizontal simplificado

Transacciones terminadas.

Un detalle a tener en cuenta es que hacer con las transacciones una vez que han ejecutado el bloque TERMINATE. Según la especificación de GPSS, estas transacciones son eliminadas del modelo; pero al hacer esto, se pierde la posibilidad de almacenarlas en nuestra base de datos. ¿por qué sucede esto? Analicemos el siguiente ejemplo:

```
GENERATE 1
ADVANCE 1
TERMINATE 1
START 2
```

En este ejemplo simple, se generan transacciones cada un instante de reloj; cada nueva transacción es planificada a futuro, para luego simplemente finalizar. En tiempo t=1 se genera nuestra primera transacción (1), la cual queda lista para ejecutarse en tiempo t=2; dado que ya no quedan eventos pendientes en la CEC, habrá que actualizar el reloj previa invocación al DAO. En t=2, se genera la transacción 2, la cual es planificada para continuar en tiempo 3, y también continua la ejecución de la transacción 1, la cual decrementa el **termination count en uno y muere**. Nuevamente, como ya no quedan transacciones en la CEC, deberá invocarse al DAO e incrementarse el reloj de simulación. Pero, ¿que pasó con la transacción 1? A esta altura, ya la hemos eliminado con lo cual no podremos persistirla. A simple vista, parece ser que estaríamos perdiendo una única transacción, pero esto podría ser muchísimo más grave de lo que parece. El próximo ejemplo que veremos ilustrará perfectamente esta situación:

```
GENERATE ,,,1000,2
ADVANCE 2
TERMINATE
GENERATE ,,2,,1
TERMINATE 1
START 1
```

En este ejemplo, se generan mil transacciones en tiempo t=0 y se planifican para continuar en tiempo 2. Antes de cambiar el reloj de simulación, se almacenan estas mil transacciones y se registra que todas se encontraban en la FEC al finalizar el tiempo t=0. En tiempo t=2, las mil transacciones pasar a la CEC, y ya se habrá generado una transacción (1001) a partir del segundo GENERATE la cual, al tener menor prioridad, permanecerá al final de dicha cadena. Finalizadas primeras las mil, la transacción 1001 decrementará el termination count y finalizará la simulación. En este simple ejemplo, hemos perdido las 1001 transacciones de nuestro modelo, situación que es claramente indeseable.

El problema que hemos planteado se basa en que las transacciones finalizadas se eliminan de la simulación inmediatamente, lo cual en GPSS es correcto y apropiado. Si no las necesita más, no tiene sentido que las mantenga ocupando recursos. Pero nuestro modelo requiere que las transacciones terminadas sean almacenadas antes de su eliminación real; para solucionar este problema, la transacción mantiene una lista o pool de transacciones que han finalizado. Al recorrer el grafo de composición horizontal, se incluye también esta lista de transacciones; ahora sí, al finalizar el recorrido completo del grafo, se podrá vaciar la lista de transacciones terminadas y,

finalmente, actualizar el reloj de simulación.

Visualización de resultados.

Hemos presentado hasta ahora las principales entidades de nuestro modelo y la manera en que trabajan juntas para generar simulaciones discretas y persistirlas en una base de datos. Necesitamos algún medio que nos permita ver que es lo que ha sucedido, al menos para saber si nuestra simulación hace lo que queremos que haga. Si bien no es objetivo primordial de este trabajo el generar una herramienta de análisis, hemos incluido una pequeña aplicación que permite ver con cierto nivel de detalle como han sucedido los eventos a lo largo de la simulación; desde luego, esta herramienta utiliza el modelo presentado y los objetos para acceder a la base de datos y recuperar la información requerida.

Para maximizar la portabilidad y manteniendo la filosofía de código abierto tanto como sea posible, esta aplicación (también abierta) funciona como un plugin dentro del IDE Eclipse [ECL08]; y posee básicamente un menú (Figura 4.2) que nos permite acceder a dos ventanas principales: ventana de transacciones y ventana de Facilidades.

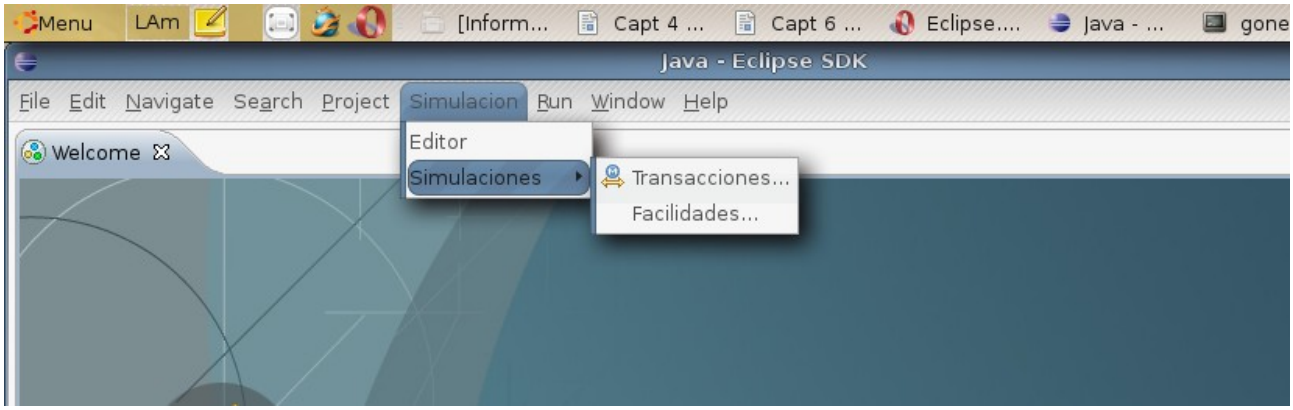

Figura 4.2. Menu de acceso al plugin

La ventana de transacciones nos permite en forma de lista ver todas las transacciones que existieron a lo largo de la simulación. Cada ítem de la lista puede ser ampliado, lo cual nos permite ver todos los tiempos de reloj de simulación en los cuales nuestra transacción seleccionada. En cada uno de estos *tiempos* podemos saber en que bloque está frenada, cual es su siguiente bloque y en que momento debe continuar, entre otros datos (Figura 4.3). Del mismo modo, la ventana de las facilidades nos permite acceder a información de todas las facilidades del modelo en cada instante de simulación (debemos recordar que las facilidades existen, en principio, durante toda la simulación).

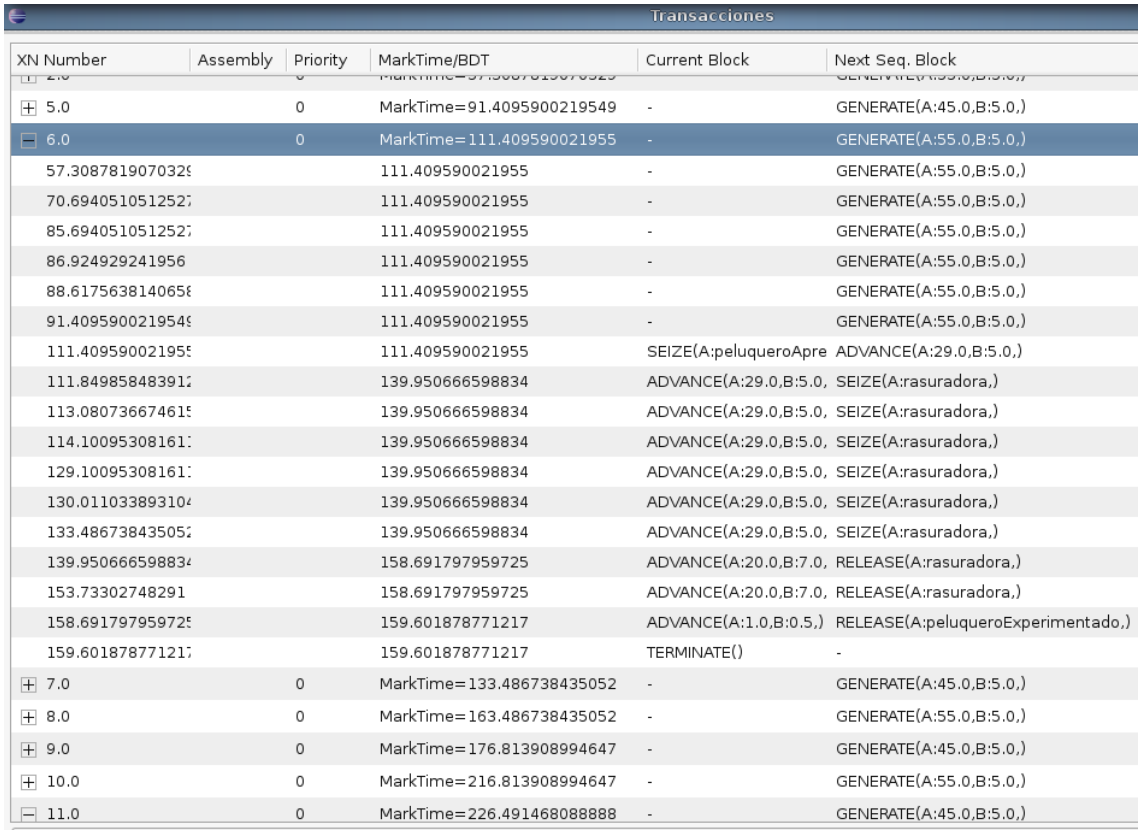

Figura 4.3. Ventana de visualización de transacciones

Con ésto tenemos al menos una idea de las posibilidades de acceso y visualización a los datos que almacenamos en la base de datos; de todos modos, en el capítulo 6 veremos con mayor detalle como se recuperan y visualizan los datos, y que tipos de análisis podríamos hacer con toda esta información.

Capítulo 5.

De la teoría a la práctica.

En el capítulo anterior hicimos un repaso genera por todo el modelo, lo cual nos permitió comprender cómo deberían funcionar las simulaciones bajo este esquema, y que entidades intervienen en una de estas simulaciones. En este capítulo aplicaremos estos conceptos en un caso práctico, en el cual tomaremos un sistema muy sencillo, generaremos un modelo y lo simularemos utilizando esta herramienta; luego, veremos que datos podemos obtener. El ejemplo estará basado en el bien conocido problema del peluquero, con algunas variantes que lo hacen más interesante para analizar.

El problema.

Nuestro sistema consiste en una peluquería, donde trabajan un peluquero experimentado y un aprendiz de peluquería. Los dos trabajadores están capacitados para realizar las tareas de corte, tintura, permanente y baños de crema. La diferencia radica en el tiempo que les demora realizar estas tareas; cómo es de esperar, el peluquero experimentado es más eficiente que el aprendiz. En la siguiente tabla veremos los tiempos (en minutos) que cada uno de ellos necesita para finalizar cada uno de los servicios ofrecidos por la peluquería:

tiempo / peluquero	Experimentado	Aprendiz
corte	22±3	29±5
afeitada	22	20±7

Como podemos observar, algunos tiempos son exactos mientras que otros están estimados en un rango A ± B. Supondremos aquí que la distribución de la variable aleatoria que determina el tiempo en cualquiera de estos intervalos está uniformemente distribuida en el intervalo [A-B , A+ B].

Los tiempos entre ingresos de clientes a la peluquería varían según el peluquero que han elegido. Los mismos han sido estimados de la siguiente manera:

peluquero	Experimentado	Aprendiz
tiempo entre ingresos (en minutos)	45 ± 5	55 ± 5

A esta peluquería, que permanece abierta durante 12 horas diarias, arriban clientes que solicitan estos servicios. Haremos aquí dos suposiciones, que simplificarán tanto el planteo como la soluciona.

- los clientes tienen un peluquero favorito y se atienden solo con ese
- si el peluquero elegido se encuentra ocupado, esperarán hasta que se libere
- los clientes solicitan primero el servicio de corte y luego el de afeitada.

Nuestra peluquería de barrio posee solo una rasuradora eléctrica, con lo cual deben compartirla entre ambos empleados; si la misma está siendo utilizada, el otro peluquero que la requiera deberá esperar a que se libere.

Una vez que finalizan los servicios, los clientes deben pagar. Solo el peluquero experimentado puede cobrar, ya que éste no confía en su compañero. Cuando los clientes del aprendiz han finalizado, solicitan al peluquero experimentado que les cobre y éste deja de hacer lo que estaba haciendo, le cobra al cliente (el cual se retira inmediatamente) y regresa con cliente. Mientras tanto, el peluquero aprendiz ya comenzó a atender a otro cliente (esto es, no espera que su cliente anterior finalice de pagar para seguir atendiendo). El tiempo requerido para cobrarle a un cliente es de **60 ± 30 segundos.**

Existe un cliente en particular, hijo del dueño de la peluquería, que requiere la atención de los dos peluqueros a la vez. Al llegar este cliente (una vez por día laboral a las 6 hs de comenzada la jornada), los dos peluqueros dejan de atender a sus clientes y atienden a este cliente especial. El tiempo de atención de este cliente varía entre 8 y 10 minutos en total, y no se le cobra; al finalizar, los peluqueros continúan con lo que estaban haciendo.

Solución propuesta.

El problema planteado parece complejo y rebuscado, y de hecho implementarlo en un lenguaje de propósito general puede resultar un dolor de cabeza. Es aquí donde los lenguajes como GPSS se hacen fuertes, ya que nos proveen todas las herramientas necesarias para realizar modelos como el propuesto con muy poco código. Veamos en que nos ayuda este lenguaje para representar nuestra peluquería:

Como primera medida, identificaremos cuales son la entidades permanentes y temporales más obvias: tenemos dos peluqueros (peluqueroExperimentado o *barber*, id 1001, y peluqueroAprendiz o *helper*, id 1002), que existen durante toda la simulación, y muchos clientes que *utilizan* a estos peluqueros; podemos también observar que los peluqueros pueden ser *ocupados* por a lo sumo un cliente. Aquí se nos presentan las primeras entidades: los clientes serán transacciones se se crean (ingresan a la peluquería), utilizan los servicios (se cortan el pelo y se afeitan) que brindan los peluqueros, pagan y se retiran. Los peluqueros serán, desde luego, las dos principales facilidades del modelo.

Tenemos una tercer facilidad que representa a la rasuradora eléctrica (rasuradora o *shaver,* id 1003). Como dijimos, existe solo una de estas en la peluquería, y los peluqueros compiten por la misma, lo cual encaja perfectamente con nuestro esquema de facilidades.

Dividiremos nuestro modelo a partir de los bloques GENERATE, encargados de crear clientes. Tendremos en principio 3 generadores de transacciones: un generador para clientes del peluquero experto, otro generador para clientes del aprendiz y un último generador para el cliente

“especial”.

Como vimos en el capítulo 4, es común utilizar una transacción especial que controle el contador de terminación, haciendo las veces de reloj que indicará cuando debe finalizar nuestra simulación. Aplicaremos este concepto aquí, y utilizaremos un cuarto bloque GENERATE para generar esta transacción, la cual colocará el contador de terminación en cero al momento indicado y finalizará nuestra simulación.

Antes de continuar explicando la solución, veamos cual es el código GPSS de esta solución. Para facilitar su lectura, hemos divido las cuatro generaciones en cuadros distintos (aunque en la realidad, es un solo archivo de texto plano):

```
* transacciones del peluquero experimentado
GENERATE 45,5 ;llega un nuevo cliente
SEIZE peluqueroExperimentado ;toma al peluquero
ADVANCE 22,3 ;tiempo de corte
SEIZE rasuradora ;toma la rasuradora
ADVANCE 22 ;tiempo de afeitada
RELEASE rasuradora;libera la rasuradora
ADVANCE 1,0.5; el cliente abona
RELEASE peluqueroExperimentado; el peluquero se libera
TERMINATE ; el cliente se retira
```

```
* transacciones del peluquero aprendiz
GENERATE 55,5
SEIZE peluqueroAprendiz
ADVANCE 29,5 ; tiempo de corte
SEIZE rasuradora ; intento tomar la rasuradora
ADVANCE 20,7 ; tiempo de afeitada
RELEASE rasuradora ;  libero la rasuradora
RELEASE peluqueroAprendiz
PREEMPT peluqueroExperimentado ; hora de pagar
ADVANCE 1,0.5  ; le cobro al cliente
RELEASE peluqueroExperimentado;
TERMINATE  ; finalizo
```

```
* cliente especial, a los 360 minutos o lo que es lo mismo a las 6 hs
GENERATE ,,360,1,2
PREEMPT peluqueroExperimentado
PREEMPT peluqueroAprendiz
ADVANCE 9,1
RELEASE peluqueroAprendiz
RELEASE peluqueroExperimentado
TERMINATE ; se retira
```

```
* reloj de simulación
GENERATE ,,720,1   ; genero una transacción que representa al reloj
TERMINATE 1 ; decremento el termination count y termino la simulación


START 1
```

Las transacciones del peluquero experimentado y las del aprendiz ejecutan casi la misma cantidad de bloques (con distintos parámetros). Podemos ver que las del aprendiz poseen un bloque PREEMPT, el cual sirve para forzar al otro peluquero a dejar de hacer lo que está haciendo para

cobrarle al cliente. Del mismo modo, el cliente *especial* utiliza dos bloques PREEMPT para apropiarse *por la fuerza* de ambos peluqueros, liberándolos inmediatamente una vez que han finalizado de atenderlo. Este cliente especial es generado con mayor prioridad (2), para asegurarse que se ejecutará antes que cualquier otro cliente, inmediatamente luego de *ingresar a la peluquería*.

El último generador provoca que una única transacción se ejecute en el instante 720 y como dijimos, su única tarea es la de decrementar el contador de terminación y finalizar nuestra simulación.

Ejecución de la simulación.

Hasta aquí ya tenemos el modelo planteado y el código de simulación. Veremos ahora como se generan y se mueven las transacciones dentro de nuestro modelo. Desde ya, no analizaremos completamente toda la simulación para evitar extender este capítulo por varios cientos de páginas. Marcaremos la ocurrencia de algunos eventos importantes, y avanzaremos sobre los mismo a medida que se requiera. Luego, el proceso será repetitivo con lo cual no tiene sentido volver a explicarlo.

Como primera medida, se identifica cada bloque GENERATE y se genera una primer transacción por cada uno de ellos; aquí tendremos nuestras 4 primeras transacciones listas para ejecutar sus bloques GENERATE. El orden ejecución de estas 4 transacciones es aleatorio, pero para simplificar un poco esta explicación supongamos que se ejecutan las 4 en orden de acuerdo a la posición del bloque generador. Con cada ejecución de este bloque, como vimos en los capítulos 2 y 4, las transacciones provocarán la creación de una nueva transacción y posteriormente se les calculará su BDT y serán planificadas a futuro. Podemos conocer a priori que sucederá con las transacciones *reloj* y *cliente especial*; el primero será planificada y puesta en la FEC, lista para ejecutarse en tiempo 720, y la segunda en tiempo 360. Este cliente especial se crea con mayor prioridad, de modo de provocar que cuando le llegue su turno, se coloque en la CEC antes que ningún otra transacción (recordemos que mientras que las transacciones de la FEC se ordenan a partir de su BDT, al pasar a la CEC se las coloca en orden aleatorio y luego se las ordena a partir de su prioridad).

Las transacciones que representan los clientes de cada peluquero no tienen un tiempo exacto de ejecución. Aquí interviene la entidad UniformFunction, la cual tiene como objetivo la generación de valores en un intervalo cerrado uniformemente distribuido. Si bien en el bloque GENERATE (al igual que sucede en otros bloques, como por ejemplo el ADVANCE) se debe colocar el centro y el radio del intervalo, esta función calculará los extremos y se encargará de generar los valores aleatorios a medida que son requeridos.

Sin importar el orden de los bloques GENERATE, sabemos que tenemos 4 transacciones listas para ejecutarse dentro de la FEC (ninguna tiene tiempo de inicio cero, con lo cual ninguna estará en la CEC al iniciar nuestra simulación). Comencemos pues con la primer transacción, aquella de menor BDT entre todas. Nuevamente, el orden de ejecución de cada cliente no puede ser previsto a priori; las transacciones para el peluquero experimentado tomarán un BDT entre 40 y 50, y las del peluquero aprendiz entre 50 y 60. Como vemos, los intervalos se superponen, con lo cual cualquier podría ser la primera.

Supondremos aquí para simplificar que comienza una transacción que representa a un cliente del peluquero experimentado (en realidad, tiene mayores probabilidades de ser primera, pero este análisis no viene al caso). Antes de salir de la FEC, esta transacción provocará una actualización del reloj de simulación. Aquí ya tenemos nuestra primer simulación persistible: tenemos un objeto Simulation, con la cadena CEC vacía y la FEC con 4 transacciones. También tendremos una lista de facilidades (esto no es tan cierto, ya que aún no han sido referenciadas, pero no entraremos en estos detalles tan finos aún), y una lista de bloques, todos con el contador de transacciones que han paso por allí en cero. Como podemos observar, nuestro primer *snapshot* de simulación fue muy simple, ya que aun no se han sucedido demasiados eventos. Generado este snapshot, podremos avanzar el reloj de simulación y pasar la primer transacción de la FEC a la CEC (Figura 5.1).

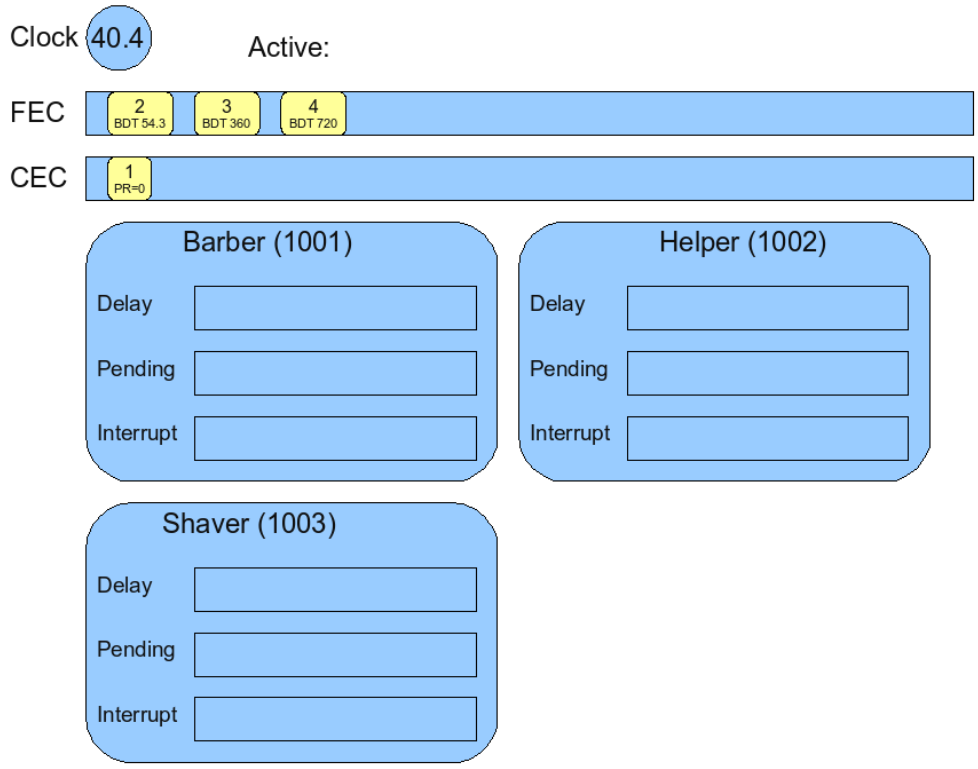

Figura 5.1: La transacción 1 esta lista para comenzar

En este punto ya tenemos una primer transacción lista para ejecutarse. Su primer bloque es, claramente, su propio bloque GENERATE, cuya ejecución es necesaria para continuar generando nuevas transacciones; así pues, esta transacción ejecutará el código de este generador, provocará que una nueva transacción sea colocada en la FEC (transacción 5) y continuará con el siguiente bloque. En el mismo tiempo de partida (aun nada la detuvo), nuestra única transacción activa entrará al SEIZE y ejecutará su rutina sobre la facilidad *peluqueroExperimentado;* al estar esta facilidad libre, nuestra transacción será colocada como dueña (owner) de la misma y podrá continuar sin ningún retraso o encolamiento. A continuación, entrará al bloque ADVANCE, se

calculará un valor uniforme en el intervalo [24,34], el cual se sumará al tiempo de reloj actual y será utilizado para planificar dicha transacción en la FEC. Esta transacción no podrá continuar, ya que fue planificada a futuro. Como ya sabemos, se buscará una nueva transacción de la CEC pero, al estar ésta vacía, se almacenará nuevamente toda la simulación y se actualizará el reloj de simulación.

Ahora le ha llegado el turno a nuestra segunda transacción, la cual será colocada en la CEC y comenzará a ejecutarse (Fig. 5.2). Su primer bloque es un GENERATE, con lo cual provoca el nacimiento de una nueva transacción (número 6) y tomará la facilidad helper. Luego, ingresará al bloque advance, se calculará su BDT en base al tiempo actual y un valor entre 13 y 27 y se la colocará en la FEC lista para partir. Luego, al no haber más transacciones en la CEC, el reloj deberá actualizarse (previa persistencia de nuestra simulación en el estado actual) y estaremos listos para el siguiente tiempo de reloj.

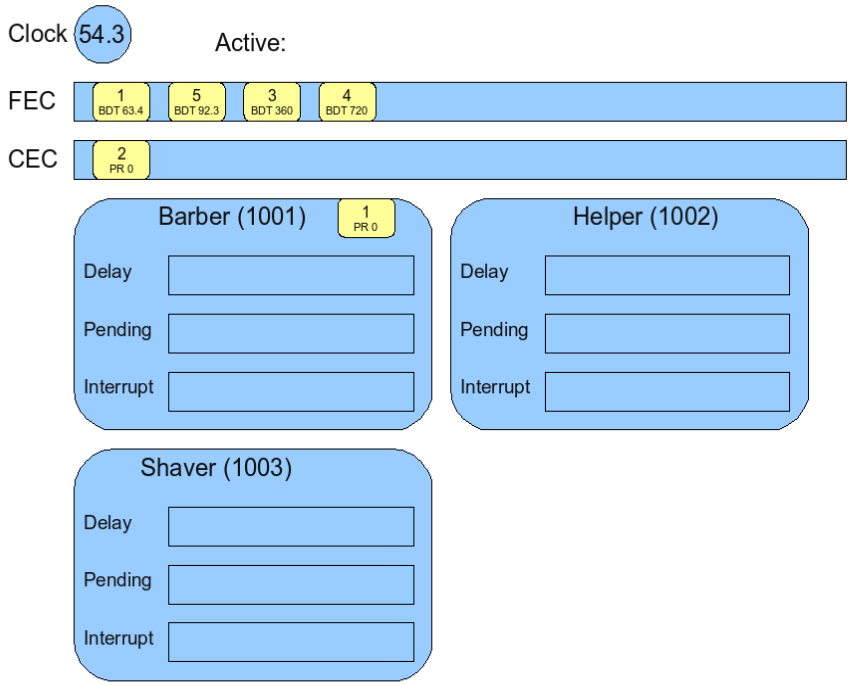

Figura 5.2. La transacción 2 comienza a ejecutarse, mientras que la 1 fue enviada a la FEC siendo el actual owner de la facilidad Barber

En el siguiente tiempo de reloj (63.4) la transacción 1 podrá tomar la facilidad *shaver*, y ejecutará un ADVANCE 22, con lo cual será colocada en la FEC con BDT 85.4 (Figura 5.3).

Hasta aquí, el flujo de transacciones no ha presentado ningún sobresalto; todas las transacciones han ingresado a sus bloques, han ejecutado sus rutinas y han continuado sin que se les niegue el acceso. Pero ahora comienzan nuestros *problemas*. La transacción 2 intentará tomar la facilidad shaver, ya que llegó el momento de afeitar al cliente 2, pero esta facilidad se encuentra ocupada por la transacción 1. Conclusión: la transacción 2 deberá esperar en la delay chain de la

facilidad shaver hasta que la misma sea liberada, cual sucederá recién en el tiempo 85.4 (Figura 5.4). Cuando ésto suceda, la transacción 2 podrá tomar esta facilidad, utilizarla y liberarla, para luego liberar al Helper e intentar tomar al Barber (el cliente debe abonarle siempre al barber).

Cuando este evento suceda, la facilidad barber ya habrá sido tomada por la transacción 5 y estará ocupada. Al ejecutar el bloque PREEMPT, la transacción 2 provocará que la facilidad barber sea liberada por la fuerza, enviando a la transacción 5 a la Interrupt Chain de dicha facilidad y continuando hacia el bloque ADVANCE. Esto la colocará en la FEC, y dejando a la transacción 5 bloqueada hasta que su facilidad sea liberada (Figura 5.5). Cuando esto suceda, la transacción 2 liberará la facilidad barber y finalizará, y la transacción 5 volverá a tomar dicha facilidad. Como aún le quedaba tiempo pendiente en la FEC, volverá allí y esperará este tiempo. Luego, simplemente continuará su flujo normal.

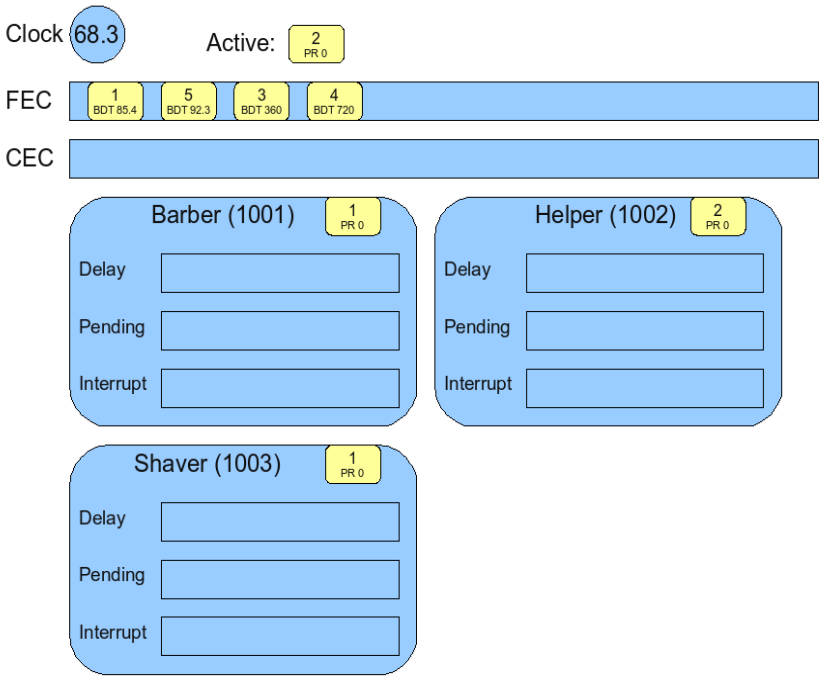

Figura 5.3. La transacción 1 es ahora el owner de la facilidad Shaver, y la transacción 2 está lista para continuar

Como podemos apreciar, esto puede volverse mucho mas complejo; tengamos en cuenta que mientras estas preempciones y desplazamientos se ejecutan, más y más transacciones se continuaron generando y han intentado tomar las facilidades barber y helper (no las hemos incluido en esta explicación para no hacerla más extensa, compleja y considerando que no incorporan ningún concepto nuevo; simplemente estarán en la FEC listas para ingresar a su GENERATE o estarán en la Delay Chain de la facilidad que intentan tomar).

La simulación continuará de este modo durante unos cuantos instantes de reloj, hasta que llegue el cliente especial (en tiempo 360, según se ha planificado). En este momento, el cliente será

removido de la FEC y colocado en la CEC y, dado que se ha generado con una prioridad mayor al resto de las transacciones (prioridad 2), la transacción será puesta al frente de dicha cadena. Una vez que todas las transacciones han sido *preparadas* para ejecutarse en este tiempo, comenzará la ejecución normal. La primera transacción en colocarse como activa es, como es de esperarse, la transacción número 3 (cliente especial), la cual tomará mediante la ejecución de dos bloques PREEMPT las facilidades *barber* y *shaver*; ambas facilidades tendrán transacciones en la cola de delay y así como también una transacción activa, pero al ejecutar este bloque, las transacciones activas pasarán a la cola de interrupción, las que están en al cola de delay permanecerán allí hasta que la facilidad de libere nuevamente (cuando las transacciones de la interrupt chain se remuevan de allí) y la transacción que está ejecutando los PREEMPT será colocada como actual owner de ambas facilidades, para continuar en el bloque ADVANCE (Figura 5.6). Las transacciones desplazadas de las facilidades estaban también demoradas de la FEC, con lo cual se calculará el tiempo que les resta (tiempo restante = BDT – reloj actual) y serán removidas de dicha cadena.

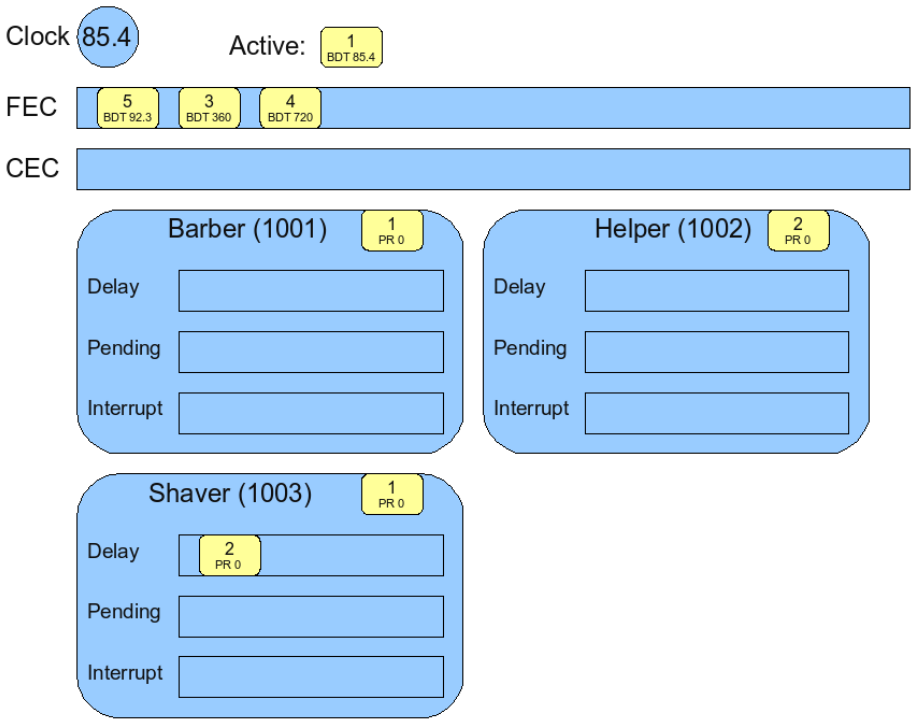

Figura 5.4. La transacción 2 debe esperar en la delay chain

Finalizado su tiempo de espera, liberará ambas facilidades mediante los bloques RELEASE y finalmente morirá. Con la ejecución de cada bloque RELEASE, se seleccionará la transacción demorada en la Interrupt Chain y, dado que no posee más preempciones, será tomada nuevamente por el Transaction Scheduler para planificar su continuación. Como ya dijimos, estas transacciones no habían finalizado su tiempo en la FEC, lo que significa que se calculará su nuevo BDT (bdt = tiempo restante + reloj actual) y serán colocadas nuevamente en esta cadena. Estas transacciones también serán colocadas como owners de las facilidades que habían perdido previamente.

Nuevamente la simulación seguirá su curso esperado hasta que llegue el tiempo de reloj 720, en el cual la transacción que representa al reloj será ejecutada. Del mismo modo que sucedió

con el cliente especial, esta transacción posee una prioridad superior al resto (prioridad 1), lo cual nos asegura que cuando llegue su tiempo de reloj se ejecutará antes que ninguna otra. En este momento, la transacción simplemente ejecutará un bloque TERMINATE con parámetro 1, lo cual decrementará el contador de terminación; dado que este contador había sido inicializado en 1, tomará el valor 0 desencadenando inmediatamente la finalización de la simulación. En este caso, todas las entidades permanecerán como están, y nuevamente se recolectarán los todos los datos requeridos para la persistencia (siempre es muy útil conocer el modo en que ha finalizado una simulación y el estado en que quedaron sus entidades). Recolectados los datos, se enviarán para su persistencia y nuestra simulación finalizará satisfactoriamente.

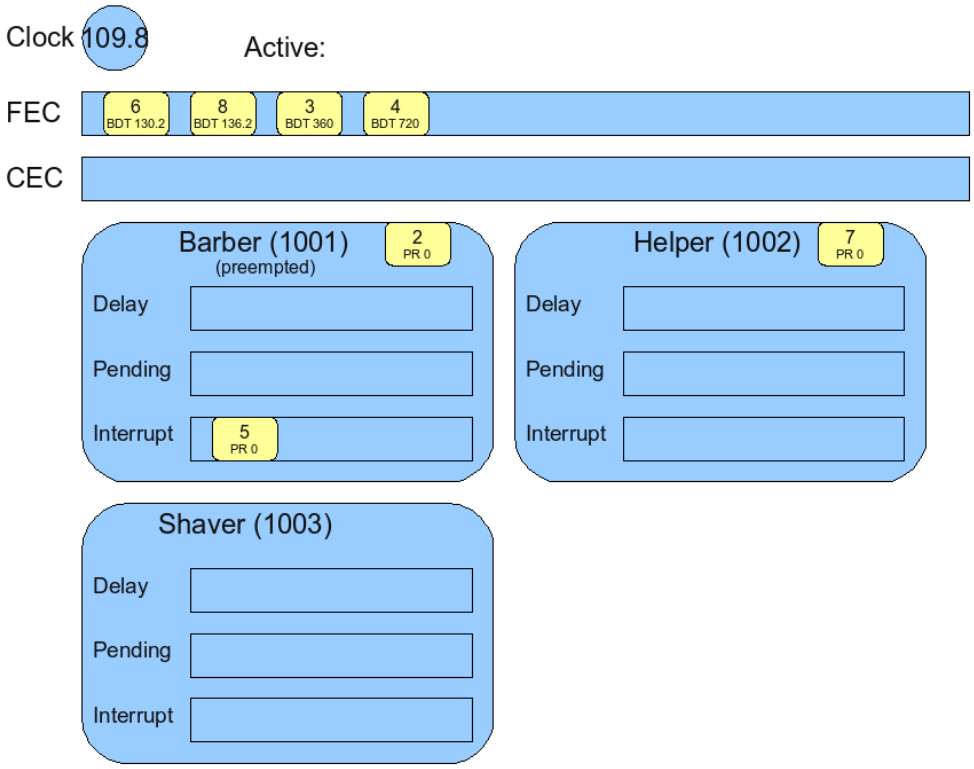

Figura 5.5. La transacción 5 es desplazada a la delay chain, mientras que la 2 toma la facilidad Barber por la fuerza

El ejemplo que hemos presentado no reviste demasiada complejidad, y su implementación en GPSS es relativamente simple. Pero realizar el seguimiento detallado de la corrida y conocer exactamente lo que ha sucedido es, por el contrario, un ejercicio bastante complejo incluso para un programador GPSS experto; consideremos que esta simulación solo posee 3 entidades Facility y menos de 30 bloques, e imaginemos cómo sería en una simulación de un sistema complejo real. El desarrollo que presentamos en este trabajo pretende, entre otras cosas, simplificar esta tarea de seguimiento y debug de simulaciones, permitiendo al programador conocer siempre lo que ha sucedido con sus entidades a lo largo de la simulación sin la necesidad de realizar un seguimiento como el que hemos hecho hasta aquí.

Cabe destacar que la solución que hemos propuesto, si bien funciona y refleja exactamente

lo que hemos planteado en el enunciado, carece de uso real ya que no contiene ninguna entidad útil que recolecte estadísticas de, al menos, las tres facilidades que intervienen en el modelo; por ejemplo, hubiese sido ideal utilizar entidades QUEUE para recolectar estadísticas de uso y permanencia, entidades TABLE para obtener diagramas de frecuencia, entidades MATRIX para registrar determinados eventos interesantes, SAVEVALUES para registrar la ganancia del día, etcétera. GPSS nos ofrece una gran variedad de entidades capaces de recolectar estadísticas, generar gráficos, tablas o incluso archivos de salida; también nos ofrece bloques y entidades para agrupar y desagrupar transacciones, dividir y juntar transacciones, definir funciones específicas, y otras tantas opciones para obtener casi cualquier dato a partir de un modelo simulado. Desde ya, aconsejamos al lector interiorizarse en este punto no solo para comprender mejor este trabajo, sino porque el profundo conocimiento de este lenguaje y sus entidades es tan interesante como enriquecedor.

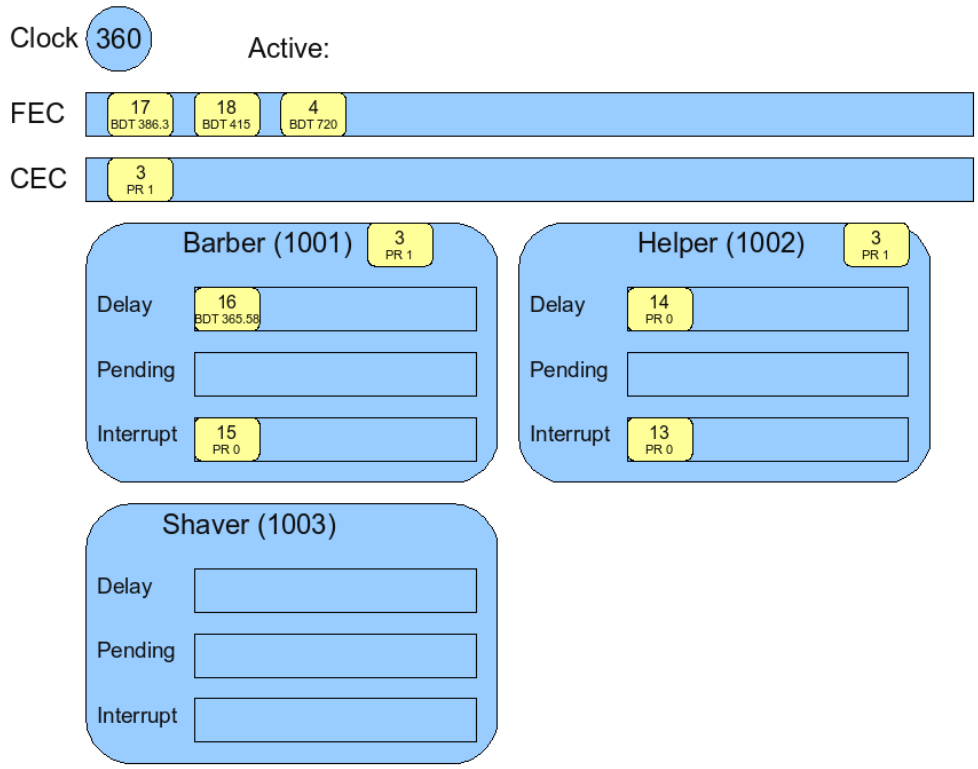

Figura 5.6. El cliente especial ha tomado ambos peluqueros por la fuerza

Capítulo 6. Tópicos avanzados y detalles de implementación

En los capítulos 2 y 3 hemos repasamos algunos conceptos necesarios para comprender en que consiste este trabajo; luego, en el capítulo 4 hemos presentamos el modelo desarrollado, las principales entidades y hemos introducido algunos problemas complejos a los que nos hemos enfrentado durante este desarrollo, sin hacer demasiado hincapié en los mismos ya que el objetivo de este capítulo era simplemente plantear el desarrollo para poder pasar al capítulo 5, donde presentamos un ejemplo concreto.

En este capítulo retomaremos muchos de los conceptos vistos en los capítulos 2 y 3, para analizar en detalle la problemática que plantean algunas de las entidades descriptas en el capítulo 4 y cuáles fueron las decisiones que hemos tomado para poder implementarlas de acuerdo al modelo y a los objetivos planteados.

El Planificador de Transacciones y el inicio de la simulación.

Como ya hemos visto previamente, el planificador de transacciones es el encargado de decidir cual será la próxima transacción a ejecutarse a partir de, principalmente, el estado de las cadenas CEC y FEC. Aunque esto es muy cierto, en la realidad es sustancialmente más complejo (como se ha visto con los bloques SEIZE, PREEMPT y RELEASE). Al comenzar la simulación las cadenas están completamente vacías, lo que significa que el Transaction Scheduler no tendrá nada que hacer hasta que alguien cree y coloque una transacción, y para que esto suceda debe haber transacciones que provoquen los cambios necesarios para que otras transacciones se creen y pasen de unas cadenas a otras. Planteado de este modo, esta situación nos lleva a un bloqueo mutuo (deadlock), por lo cual debemos implementar algún mecanismo que nos permita inicializar apropiadamente las entidades para poder correr simulaciones. Veamos cuáles son los principales pasos requeridos para crear e iniciar una simulación (ver figura 6.1):

Como primera medida se crea el objeto simulación, junto a sus cadenas FEC y CEC, System Clock y el Transaction Scheduler, entre otros. A continuación se ejecutan todos los comandos de la cola de comandos hasta que aparezca un START, el cual indicará que la simulación debe iniciarse. Luego se buscan todos los bloques GENERATE del modelo, y por cada uno de ellos se genera una primera transacción, a la cual se le coloca su propio bloque GENERATE (el que provocó su creación) como NSB y se las ubica en la FEC. Si poseen un tiempo de incremento positivo (operando C), este se utilizara como BDT; de lo contrario, se computan A y B para calcular el BDT de la transacción (el operando C solo es utilizado en la primer transacción de cada GENERATE). Hasta aquí, hemos transitado la etapa de inicialización y tenemos todo listo para comenzar: un conjunto de transacciones y un objeto simulación con un contador de terminación mayor a cero.

Una vez que se han creado y preparado las primeras transacciones comienza a ejecutarse la simulación, para lo cual el Transaction Scheduler toma todas las transacciones con el menor BDT de la FEC y las coloca en la CEC. Cada una de ellas, al convertirse en Transacción Activa, ejecutará el código del bloque GENERATE que poseen como NSB, provocando la creación de una nueva transacción que irá a la FEC como corresponde. Aquí, la simulación comienza a moverse normalmente ya que son las mismas transacciones las que provocan la creación de otras transacciones (siempre y cuando no se halla alcanzado el limite de creación especificado en el parámetro D del bloque GENEARATE), generando un efecto en cadena que resultará en una ejecución de un modelo.

La simulación completa finalizará cuando:

1. Se ha alcanzado el valor cero en el contador de simulación
2. Ha ocurrido algún STOP ERROR, como por ejemplo la ausencia de transacciones en la FEC y en la CEC, o la detección de un bloqueo insalvable o deadlock

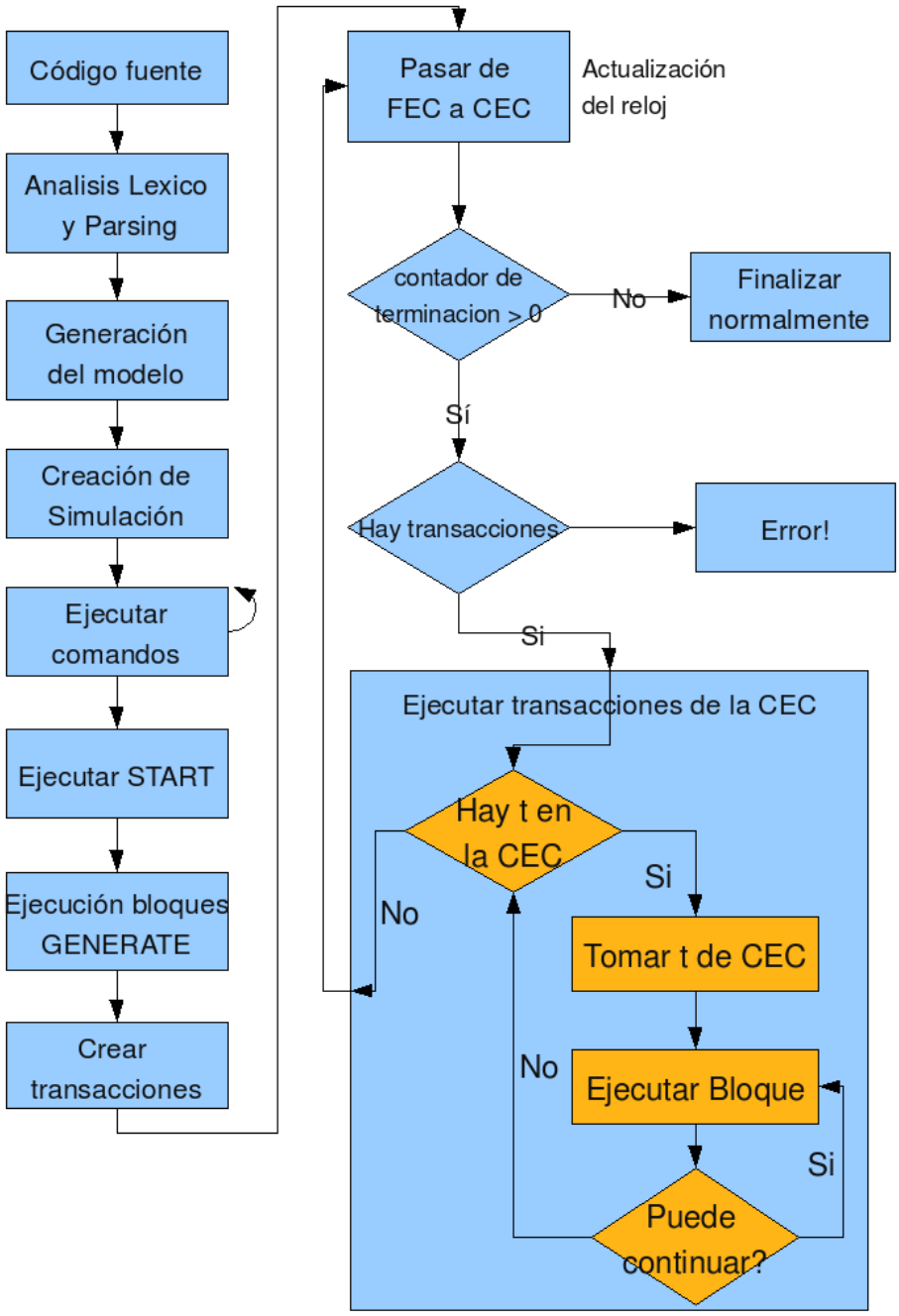

Figura 6.1. Ciclo de vida de la simulación

Persistencia de simulaciones.

En el capítulo 4 explicamos en que consistía la persistencia, sin analizar detalladamente como funciona realmente. En ese capítulo analizamos a su vez el problema que surge al finalizar las transacciones, para así introducir la lista adicional de transacciones que hemos incluido en nuestro modelo. A lo largo de esta sección veremos que la persistencia de transacciones no es un asunto menor, sino que nos ha enfrentado a números problemas, los cuales serán detallados junto con la solución que hemos hecho en nuestra implementación.

Objetos reales vs Objetos persistentes.

Como vimos a lo largo de los capítulos anteriores, cada simulación esta compuesta de un conjunto de entidades; en el modelo subyacente que hemos desarrollado, estas entidades son básicamente objetos Java que deben ser almacenados en la base de datos de acuerdo al planteo inicial que nos hemos propuesto. En el capítulo 3 indicamos que gracias al uso de JPOX podemos abstraernos del almacenamiento de la aplicación, adicionando también otras ventajas como por ejemplo el uso de un motor de bases de datos relacional como MySQL para almacenar objetos, dejando que JPOX realice todo el trabajo de mapeo de objetos a tablas relacionales y viceversa. Pero esta simplificación que ofrece el ORM implica un mecanismo de funcionamiento que en la mayoría de los sistemas es ventajoso y deseable, pero que en nuestro desarrollo se convirtió en un obstáculo a sortear: la actualización de los objetos "persistibles". Analicemos el siguiente ejemplo, para comprender el problema:

```
GENERATE ,,,1 ; solo genero una transacción
ASSIGN 1,10   ; coloco el valor 10 en el parámetro 1
ADVANCE 1     ; referencio al parámetro 1 de la transacción
ASSIGN 1,11   ; ahora cambio el valor del parámetro 1 por 11
TERMINATE 1   ; terminó la simulación

START 1
```

Este ejemplo tan sencillo sirve perfectamente para describir nuestro problema. Como ya sabemos, se generará una única transacción (XN#1) en tiempo t=1, a la cual se le asignará el valor 10 en su parámetro 1 y luego será colocada en la FEC, lista para salir en tiempo t=2. Cuando llegue este momento, el parámetro 1 pasará a tomar el valor 11 y finalmente terminará, decrementando el contador de terminación y finalizando la simulación completa. Hasta aquí, todo esta tal como se esperaba: la transacción se crea, se planifica a futuro, termina y la simulación finaliza correctamente. Pero ¿cómo se almacena esta simulación tan simple en la base de datos? El siguiente gráfico (Figura 6.2) nos ayudará a comprender este punto:

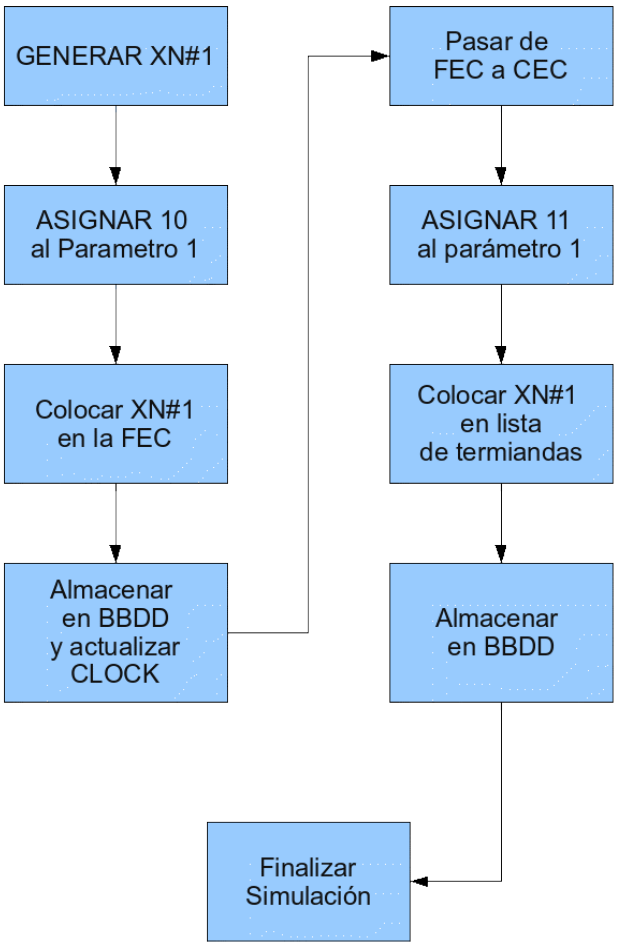

Figura 6.2. Secuencia de ejecución y almacenamiento

En este gráfico podemos ver claramente el momento en que se realiza el guardado de la simulación. Aquí, esto sucede al momento t=1 cuando la transacción XN#1 pasa a la FEC, y al momento t=2 cuando ésta transacción termina y finaliza la simulación. En nuestro modelo, esta transacción representa un objeto en memoria, con un identificador de objeto único (OID) el cual sirve, entre otros, para que el ORM identifique al objeto al guardarlo y recuperarlo. Esta transacción posee siempre su mismo OID, y esto es así ya que sin importar el tiempo de reloj, la localización en las cadenas o su estado interno, la transacción XN#1 es siempre la misma entidad.

El hecho de mantener un mismo OID no es menor; cada vez que nuestra transacción cambie su estado y sea almacenada, el ORM notará que un objeto que ya existía previamente ha sido modificado con lo cual actualizará su estado. Como hemos dicho, esta situación es esperable en la mayoría de las aplicaciones, pero en nuestro caso al actualizar el objeto perderemos el estado anterior del mismo; esto quiere decir que con cada cambio del reloj de simulación, todos los objetos se actualizarán, lo cual significa que no se conocerá su historia.

El planteo anterior nos coloca ante un serio problema, el cual puede ser atacado desde

ángulos muy distintos implementando soluciones que en ningún caso resultan triviales. En nuestro trabajo, cada objeto *persistible* posee un método que le permite clonarse, y será precisamente este clon el que se almacenará en la base de datos. Los clones son objetos casi idénticos a los originales, ya que poseen el mismo estado interno, las mismas referencias y, obviamente, el mismo tipo. Pero son distintos objetos, con lo cual el OID será distinto provocando que cada vez que se almacenen se los considere como nuevos objetos. Y gracias a que cada entidad del modelo posee su propio identificador interno (distinto al OID), siempre se podrá saber a que entidad real corresponde cada entidad clonada. Ahora, con cada cambio de reloj, se genera un clon de la entidad Simulation y se provoca una generación en cascada de todas las entidades alcanzables desde la simulación, y como hemos visto en la figura 4.1, esta entidad sirve como punto de partida ideal para recorrer el grafo de composición horizontal correspondiente al modelo, ya que permite llegar a todas y cada una de las entidades, lo cual nos asegura que no quedará ningún objeto perdido sin ser persistido.

Clonación de objetos.

Clonar un objeto simple (donde por simple nos referimos a un objeto que posee un estado interno pero no posee referencias a otros objetos) es una tarea sencilla que se reduce a crear un objeto, copiar el estado interno del objeto creador y retornarlo. Más aún, clonar un grafo acíclico de objetos es una tarea relativamente simple, que requiere recorrer el grafo clonando cada uno de los vértices y asignándolos a las nuevas aristas. Pero la clonación de un grafo con ciclos no es tan trivial, ya que presenta una problemática muy particular. Veamos la siguiente situación:

Sean dos objetos A y B cualesquiera, donde A posee una referencia hacia el objeto B y éste último posee una referencia al objeto A (figura 6.2a).

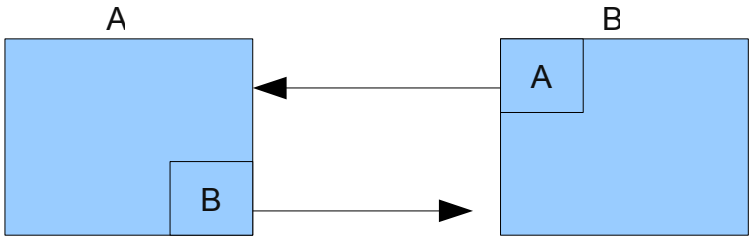

Figura 6.2a. Objetos originales

Supongamos ahora que la operación de clonación comienza a partir del objeto A; esto significa que A se debe clonar y debe provocar la clonación en cascada de todos los objetos que referencia. En principio tendremos un objeto A', clon del objeto A (figura 6.2b).

A continuación y como es esperable, A' (el clon de A) solicitará a B (su única referencia) que se clone, y actualizará su referencia. La situación quedará como muestra la figura 6.2c.

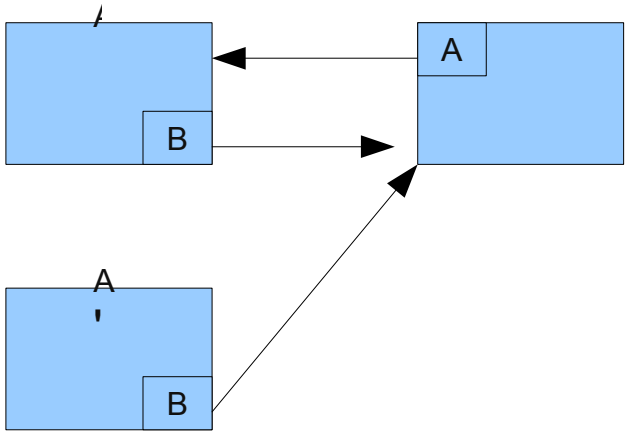

Figura 6.2b. El clon de A mantiene las mismas referencias originales

Los objetos A y B permanecen intactos, como debe ser. El objeto A se ha clonado en un objeto A', y ha requerido la clonación de su objeto referenciado, B, para así poder actualizar su propia referencia. El objeto B recibe la solicitud de clonación y se clona en un nuevo objeto, B'. B' posee una única referencia al objeto A, el cual ya ha sido marcado como clonado (para evitar ciclos en el grafo), con lo cual no continúa solicitando clonaciones, pero esto implica que tampoco actualice sus referencias, ya que supone de antemano que esta referenciando objetos clones cuando, en realidad, continúa referenciando objetos originales .

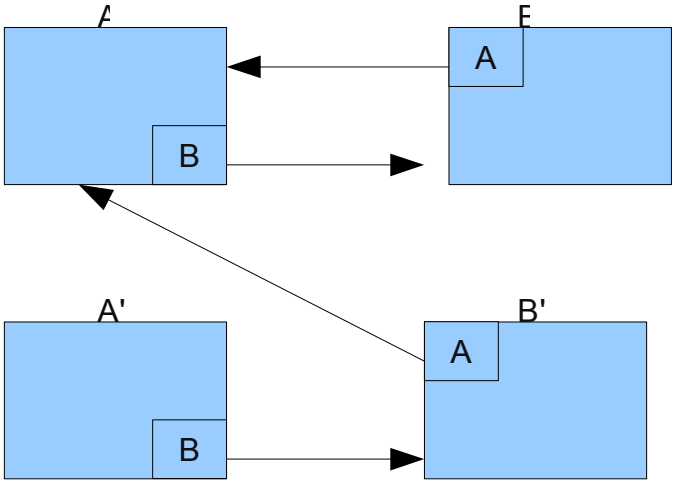

Figura 6.2c. A' actualiza sus referencias, pero B' mantiene las originales

Como podemos apreciar, marcar un objeto como ya clonado (o su equivalente visitado en cualquier algoritmo de recorrido de grafos) no es suficiente para asegurar que el recorrido se realizará debidamente. La clonación completa del grafo requiere que en todo momento se sepa si se está trabajando con un clon o con el objeto real, y que siempre los objetos sepan a que objeto deben

referenciar. Para ello, hemos ideado una solución simple y a la vez eficiente, que permite sortear esta dificultad manteniendo los tiempos de clonación en un nivel muy eficiente y aprovechando el uso de la memoria.

Nuestro modelo posee una tabla de Hash ([PIE93] , [VIT86]) donde el conjunto de claves esta conformado por referencias a los objetos originales y el conjunto de valores por referencias a los objetos clonados. Esta tabla se va completando a medida que se clonan los objetos, y en cada invocación a la función *clone()* los objetos sabrán si deben referenciar al objeto original o a la copia, ya que siempre sabrá de cual se trata. Por ejemplo, el pseudo-código en Java para la clonación de objetos de tipo Transaction queda como sigue:

```
public Transaction clone() {
      //verifico si soy un clon o un original
      if (cloned_objects.containsKey(this))
            return (Transaction) cloned_objects.get(this);
      //genero la copia y registro la equivalencia
      Transaction new_xn = (Transaction) super.clone();
      cloned_objects.put(this, new_xn);
      return new_xn;
}
```

La figura 6.4 nos muestra gráficamente cuales son los pasos requeridos para realizar la clonación de una transacción

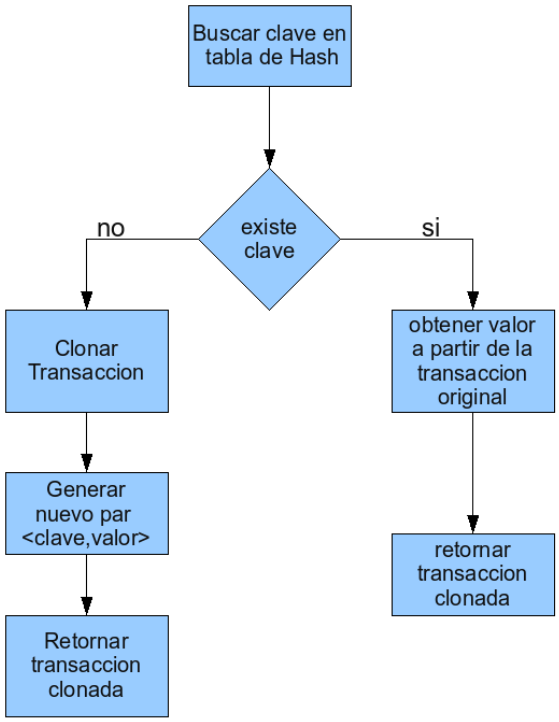

Figura 6.4. Clonación de una transacción

Como podemos apreciar en el ejemplo anterior, cada objeto sabe cómo localizar su par y evitar clonar objetos ya clonados. Si este par no existe, el mismo objeto es el encargado de generar su clon y de registrar la nueva clave-valor en la tabla de Hash. Al finalizar la clonación de todo el grafo, se limpiará esta tabla de Hash de modo que quede lista para utilizar en el siguiente tiempo de simulación.

Tiempo de persistencia.

Una vez que tenemos un grafo de simulación completo listo para persistir, debemos indicarle al DAO que realice esta tarea. Esto genera otro problema, ya que las simulaciones deben correr de manera ágil y fluida, y la persistencia en disco implica tiempos que superan en varios órdenes de magnitud a la gestión en memoria [STA00]. Adicionalmente, el guardado de toda una simulación no consta simplemente de la escritura de miles de objetos en disco, sino que requiere dividir estos objetos en tablas, mantener los OIDs para generar las referencias, insertar registros en tablas de join y mantener las tablas de catálogo (de MySQL y de JPOX) actualizadas. Claramente, esto es mucho mas que escribir en disco , y la demora que esto genera provoca tiempos de espera durante la ejecución de la simulación considerablemente grandes (para cada actualización del System Clock).

Con el fin de mantener los tiempos de simulación tan bajos como sea posible, hemos decidido implementar el guardado de las simulaciones utilizando un thread (PersisterThread) para tal fin y siguiendo la técnica productor-consumidor [AND99]. Este thread se inicia con la simulación, y permanece ejecutándose en background esperando tener datos para almacenar. Cada vez que se genera una nueva entidad Simulation, se coloca en una cola de listos para almacenar (recordemos que nuestra entidad Simulation engloba a todas las restantes entidades de la simulación). Al ingresar a esta cola, el objeto PersisterThread tomará uno a uno los elementos en el mismo orden en que fueron insertados, y los almacenará en la base de datos junto con todas las entidades alcanzables por éste (Figura 6.4.1). Este thread tiene una prioridad menor que el thread principal (aquel que ejecuta las simulaciones), para asegurar que no intervenga más de lo necesario. Con esta solución, el guardado de una simulación no retrasa la ejecución de la misma, aprovechando los ciclos de CPU restantes (que no utiliza la simulación) para su almacenamiento.

Gestión de expresiones.

En el Capítulo 2 vimos muy brevemente el concepto de expresión y de SNA; un Atributo Numérico del Sistema (SNA por sus siglas en inglés) es una expresión que hace referencia a un atributo de alguna entidad y que retorna un valor numérico. Existen alrededor de 50 SNAs, algunos propios de la entidad Simulation, otros del reloj y otros del resto de las entidades que se mueven por nuestro modelo.

Una expresión de GPSS esta compuesta por números, funciones básicas (suma, resta, multiplicación y división), invocaciones a funciones definidas por GPSS y referencias a estos SNAs; las expresiones pueden mapearse a un árbol de expresión (Figura 6.5), lo cual facilita mucho su evaluación cada vez que se requiere. En los nodos hojas del árbol tendremos elementos simples como valores mientras que en los niveles superiores habrán expresiones SNAs, funciones del

sistema, operaciones binarias, entre otras.

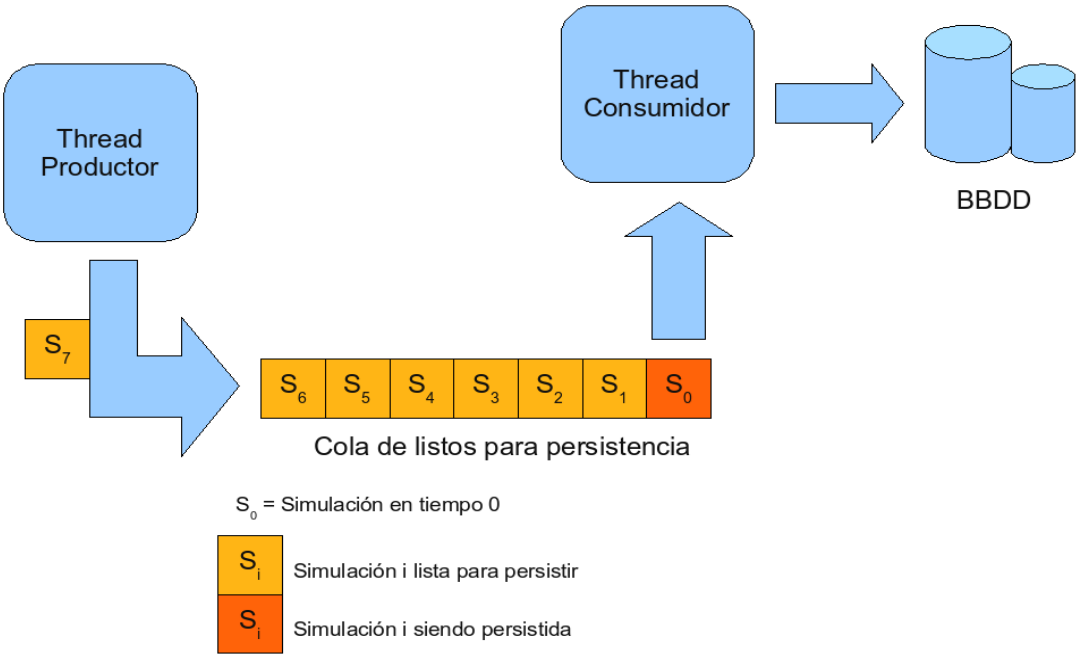

Figura 6.4.1. Esquema productor-consumidor para persistencia

El mayor problema de la evaluación de expresiones de GPSS es la evaluación de los SNAs. Cada uno de estos atributos del sistema consta de un código que lo identifica, usualmente de uno o dos caracteres, un separador (que puede no existir) y una referencia a la entidad sobre la que debe evaluarse (que también puede no existir). Veamos algunos ejemplos para cada caso:

- Sin referencias a entidades (SNAs atómicos)
  - XN1: Retorna el número de la transacción activa
  - AC1: Valor absoluto del reloj del sistema, desde la ejecución del último comando CLEAR.
  - C1: Valor relativo del reloj del sistema, desde la ejecución el último comando RESET.
- Con referencias a entidades (SNAs directos):
  - FN$unaFunción: FN hace referencia al SNA Function, con lo cual esta invocándose a la función unaFuncion.
  - P2 o P$2: Parámetro 2 de la transacción activa.
  - Phola o P$hola: parámetro nombrado como hola de la transacción activa.
  - F$unaFacilidad: Facilidad ocupada. Si la facilidad unaFacilidad se encuentra ocupada, retorna 1. Caso contrario, retorna 0.
  - FC$unaFaciliad: Contador de capturas. El número de veces que la facilidad unaFaciliad ha sido tomada, ya sea mediante un bloque SEIZE o mediante un bloque PREEMPT.
- Con referencias indirectas a entidades (SNAs indirectos):
  - *1: se refiere a la entidad cuyo número es indicado por el parámetro 1 de la transacción. Dependiendo del contexto en el que se ejecute, sera una referencia a una facilidad, un bloque u otra entidad.
  - F*barber: chequea que la facilidad que tiene el número del parámetro “barber” este libre. Nótese la diferencia con F$barber, en este ultimo caso, barber es un named value, mientras

que en el primero es un parámetro.

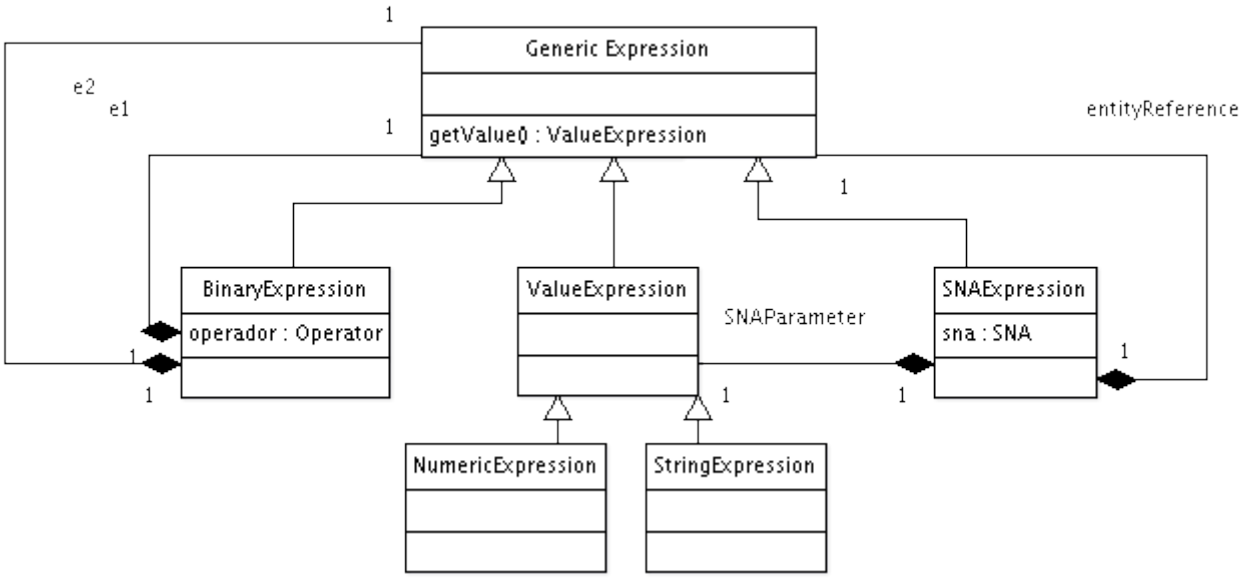

Figura 6.5. Árbol de expresiones

Como podemos ver, la evaluación de un SNA puede retornar un valor preexistente (como por ejemplo el tiempo de reloj o el valor de un parámetro) o requerir la invocación de algún tipo de función para realizar el cálculo (como por ejemplo, para conocer el estado de una facilidad). Para clarificar un poco las posibles estructuras de una expresión SNA citamos parte de la gramática utilizada:

```
SNAExpression : DirectSNA | IndirectSNA ;

IndirectSNA: code=EntitySNAClass '*' ref=(PosInteger | ('$')?  name=Name);

DirectSNA: AtomicSNA | SimpleSNA ;
AtomicSNA: code=AtomicSNA ;
SimpleSNA : code=EntitySNAClass reference=(PosInteger | '$'  name=Name) ;

EntitySNAClass : 'RN'|'P'|'F'|'FC'|'FI'|'FV'|'N'|'W' ;
AtomicSNA : 'A1'|'M1'|'PR'|'XN1'|'TG1'|'AC1' ;
```

Más allá del tipo de SNA y de la forma de calcular el resultado, podemos observar que un SNA siempre tiene:

- Un código (P, F, AC1, XN1)
- Un parámetro (unaFacilidad, unaFunción, etc).

El problema aquí es que, por ejemplo, unaFacilidad es una instancia específica de un objeto de tipo Facilidad, que se debe recuperar en tiempo de simulación (recordemos que pueden ser

referenciadas a partir de un valor nombrado, por su nombre o por su valor) y que el código del SNA referencia a una función, que también se conoce en tiempo de ejecución, y que se debe ejecutar sobre la entidad específica en cuestión.

El primer problema aquí es conocer el tipo de la entidad sobre la cuál se está invocando el SNA, ya sea Simulation, Facility, Transaction o la que fuere. Para solucionar este problema, hemos creado un objeto Mapeador de SNAs llamado SNAMapping, el cual simplemente asocia un elemento enumerativo (que representa el SNA) con un tipo o clase (que representa la entidad). Por ejemplo, el SNA P será asociado al tipo Transaction, al igual que los SNAs PR (prioridad) y XN1; los SNAs C1 y AC1 se asociarán con el tipo Simulation, los SNAs F y FC se asocian al tipo Facility, y así sucesivamente.

Solucionado el problema de la clase de la entidad, ahora se debe saber si se trata de una entidad referenciada directamente por su nombre o por su valor (identificador) o indirectamente a través de un parámetro de la transacción activa. Como hemos mencionado en el capítulo 4, existe un objeto EntitiesReferencesManger que *sabe* localizar cualquier entidad de cualquier tipo (siempre y cuando conozcamos el tipo) en nuestra simulación. Este objeto nos permite conseguir exactamente la instancia del objeto buscado, a partir del tipo de objeto obtenido anteriormente y al identificador que acabamos de conseguir.

Una vez que tenemos nuestro objeto, necesitamos saber que propiedad o función queremos conseguir del mismo, y aquí entran en juego las entidades del sistema. Cada entidad conoce los SNAs a los que puede responder, y como es de esperar sabe que debe responder en cada caso. Gracias a esta implementación, solo resta solicitarle a la instancia de la entidad que hemos conseguirlo anteriormente que ejecute el SNA que le estamos enviando, y este objeto simplemente calculará y retornará el valor obtenido.

En esta solución hemos tomado un problema complejo y lo hemos divido en un conjunto de problemas más simples, creando objetos simples que solucionan algunos de estos problemas o delegando responsabilidades a los objetos que les conciernen, lo cual facilita enormemente la extensión del modelo. Con esta implementación, incorporar un nuevo SNA para una entidad cualquiera requerirá simplemente agregar dicho SNA en el mapeador de SNAs a clases, e implementar el código del SNA dentro del objeto que corresponde.

Jerarquías de entidades.

La cantidad de entidades que existen en GPSS es muy amplia y variada. En nuestra implementación solo nos hemos propuesto trabajar con 3 de ellas (Simulation, Transaction y Facility), pero como hemos visto fue necesario implementar otras entidades (Command, Block, Chains, System Clock, Random Number Generator, entre otras) y algunas herramientas auxiliares fundamentales como Entities References Manager, Transaction Scheduler.

Con el objetivo de facilitar la implementación y mantenimiento, pero también buscando que una futura ampliación de nuestro modelo que incluya la creación de nuevas entidades (con todo lo que ello significa) hemos organizado las entidades en una jerarquía de clases en la cual, como es esperado, las características más comunes se encontrarán en las clases superiores del árbol jerárquico, y las más específicas se encontrarán en las clases inferiores. A continuación (Figura 6.6) veremos un diagrama de clases que abarca las entidades principales y los métodos y atributos más importantes que deben ser tenidos en cuenta ante cualquier extensión. Del mismo modo, en la figura 6.7 veremos la jerarquía utilizada para implementar las distintas cadenas.

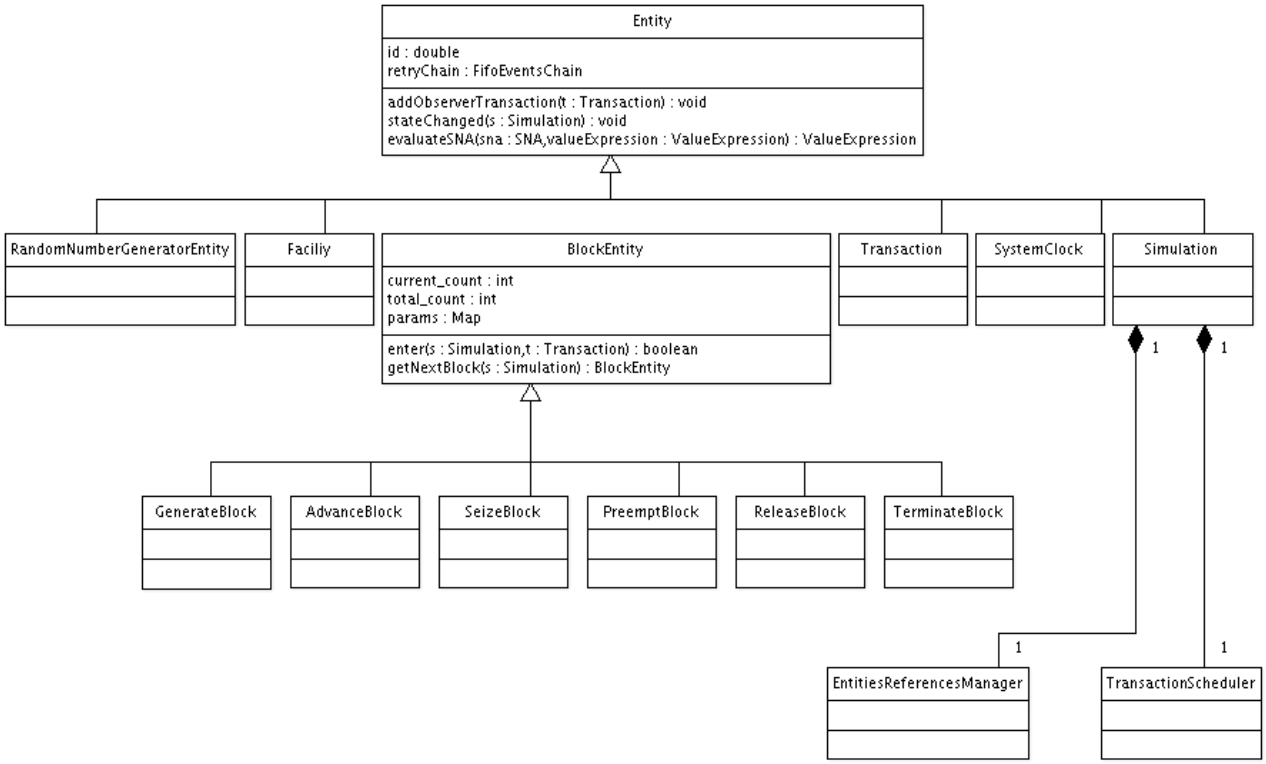

Figura 6.6. Jerarquía de entidades con sus respectivas abstracciones.

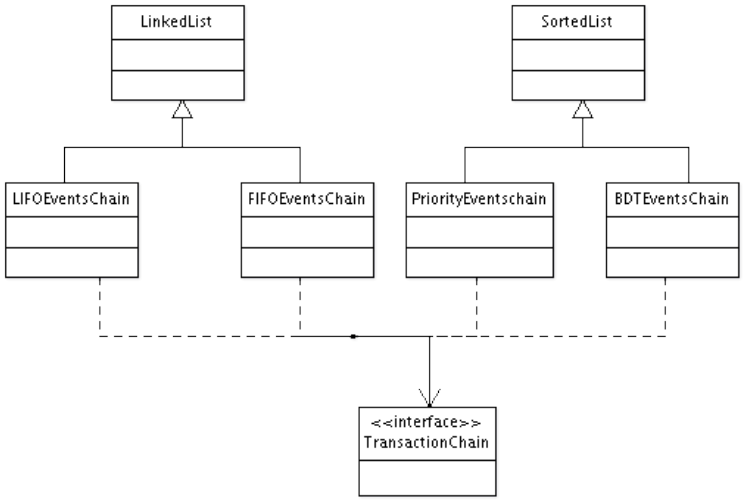

Figura 6.7. Jerarquía de cadenas

Accediendo a los datos almacenados.

Hemos visto hasta aquí de manera muy simplificada todos los eventos que se suceden para ejecutar un modelo relativamente sencillo como el descripto más arriba. Ahora, aprovecharemos las ventajas que nos trae esta implementación para poder acceder a algunos datos interesantes.
En el capítulo 4 presentamos una pequeña herramienta que nos permite acceder a todas las

transacciones y facilidades de nuestra simulación; en esta sección veremos que objetos nos permiten recuperar estos datos y mostrarlos en ésta aplicación.

Hemos incluido un objeto especial llamado SimulationRun el cual representa precisamente una corrida de simulación. Este objeto posee toda la lógica necesaria para recuperar todas las historias de una simulación en particular; del mismo modo, este objeto es capaz de recuperar la historia de cualquier facilidad y de cualquier transacción que se le solicite. Para ello, debe analizar todas las historias de simulación y recolectar a partir de ellas toda la información requerida para saber en que momento cada entidad comenzó a *existir* dentro del modelo y cuando, si corresponde, abandonó la simulación. En el caso de las transacciones es distinto al de cualquier otra entidad, ya que no existen durante toda la simulación; para obtener su historia, el objeto SimulationRun asocia cada instancia de una misma transacción con el tiempo de reloj en el que aparece, analizando las historias de simulaciones y las transacciones que intervienen en cada historia; de este modo, nos permite averiguar (y mostrar) de que tiempo de reloj estamos hablando cuando vemos los datos de un instante de una transacción en particular.

Al momento sólo hemos implementado la recuperación de historias sobre estas tres entidades a modo de ejemplo, pero siguiendo los mismos criterios de implementación, la adición de nuevas historias no será un problema mayor.

El debug de este programa puede ser extremadamente complejo, ya que para una simulación muy simple pueden intervenir muchos objetos y eventos en un mismo “tiempo” de reloj. Para facilitar esta tarea hemos desarrollado también un objeto SimulationPrinter, cuya única función es imprimir *reportes* de entidades del modelo en una consola. Este objeto nos permite acceder muy rápidamente a toda la información que recupera nuestro objeto SimulationRun de una manera organizada y legible, sin la necesidad de abrir una herramienta gráfica. Estos dos objetos (SimulationPrinter y SimulationRun) combinados apropiadamente resultan muy prácticos para realizar los primeros análisis y detectar la mayoría de los problemas, tanto de programación de entidades como de programación de la simulación; a modo de ejemplo, hemos creado un conjunto de tests capaces de crear simulaciones (entre ellas, el ejercicio planteado como ejemplo a lo largo de este capítulo, llamado TestBarberExample) y un test para recuperar rápidamente una simulación (RetrieveSimulationTest); recomendamos al lector ejecutar estos tests (en ese orden) para poder apreciar los resultados.

ANTLR y el intérprete de GPSS.

A lo largo de estos capítulos hemos visto varios aspectos que nos permiten comprender como funciona una simulación, en que consiste el modelo, que entidades debemos tener en cuenta y como se persisten las entidades, entre otros temas. En esta sección introduciremos de manera muy breve el framework ANTLR y veremos cual es su relación con el desarrollo.

Ejecución del código fuente.

GPSS, al igual que cualquier otro lenguaje de programación, requiere que el usuario (programador) escriba un código respetando cierta sintaxis (propia de cada lenguaje) para que luego este código sea leído e interpretado o compilado. En la interpretación, el código es ejecutado a medida que es leído (como sucede con lenguajes como Smalltalk o PHP); el código que se interpreta depende directamente de la línea de ejecución.

A diferencia la interpretación, en la compilación el código fuente es leído y utilizado para generar un código objeto, el cual puede ser ejecutado directamente por una computadora (como sucede en lenguajes como C/C++ o Pascal). Existe un tercer caso, cuyo principal exponente es el lenguaje JAVA, donde lo lenguajes son primero compilados a un lenguaje intermedio (bytecode) el cual es luego interpretado por una Máquina Virtual (VM) en la computadora destino (esto permite lograr independencia de la plataforma al momento de compilar ya que el bytecode es igual para cualquier plataforma y lo que se ajusta a la plataforma es la VM de la computadora destino).

Sin importar el método utilizado para la traducción, los traductores de código fuente poseen una serie de pasos bien definidos que permiten ejecutar de algún modo el código que el usuario ha escrito. Como mínimo, estos pasos son:

- Análisis Léxico: Consiste es analizar el código fuente e identificar un conjunto de tokens (componentes léxicos), que no son más que secuencia de caracteres que tienen algún significado dentro del lenguaje. Aquí se comprueba también la correcta escritura de algunos símbolos del lenguaje, como palabras claves y operadores.
- Análisis sintáctico: el analizador léxico identifica los tokens y los envía al analizador sintáctico, el cual los agrupa jerárquicamente en frases gramaticales. Aquí se comprueba que lo obtenido en la fase anterior cumple con la gramática definida para el lenguaje; esta gramática es también utilizada para generar la estructura jerárquica en base a un conjunto de reglas recursivas que allí se definen.
- Análisis semántico: en base a la estructura jerárquica generada en el paso anterior, se buscan errores semánticos y se reúne información de los tipos para luego poder generar un código intermedio. En este paso se hace el chequeo de tipos.
- Representación intermedia: en base a toda la información recogida en los pasos anteriores, se genera una representación intermedia del código, la cual puede luego ser manipulada tanto para su ejecución como para optimización, búsqueda de patrones o incluso de algunos errores de programación específicos (ciclos interminables, código inalcanzable, casteos de tipos inseguros, etcétera).
- Dependiendo del lenguaje en cuestión, pueden existir otras etapas como ser optimización del código y síntesis, para generar un código objeto.

El intérprete del lenguaje GPSS, al ser un lenguaje de programación, debe cumplir también con estas etapas para que el código del usuario se pueda ejecutar. Y dado que este desarrollo funciona sobre dicho lenguaje, debe conocer e interpretar sus componentes para luego poder generar el modelo descripto anteriormente. Para ello, hemos utilizado el framework ANTLR [ANT08] [TER07]

¿Qué es ANTLR?

ANTLR es un generador de analizadores, también conocido como compilador de compiladores (dado que ayudar a implementar compiladores es su uso más popular), aunque en realidad tiene muchos otros usos. Posee implementaciones en Java, Python y C bajo licencia GPL, como el resto de las herramientas utilizadas en este trabajo. ANTLR puede generar un analizador léxico, sintáctico y semántico a partir de uno o más archivos escritos en un lenguaje propio (con extensión .g), en el cual se define la gramática mediante reglas EBNF y algunas construcciones adicionales que incorpora.

Esta herramienta nos permite generar la gramática inicial, la cual se incluye en el Apéndice III, y luego extenderla a medida que se generan nuevos bloques, comandos, SNAs o cualquier otro agregado al código GPSS, ajustándose perfectamente a la idea de continuar extendiendo y ampliando nuestro trabajo.

ANTLR utiliza Árboles de Sintaxis Abstracta (Abstract Syntax Tree o AST) para manejar la información semántica del código fuente. Los ASTs pueden intervenir en varias fases del análisis: como producto del análisis sintáctico, como elemento intermedio en sucesivos análisis semánticos y como entrada para la generación de código. Junto a la construcción de árboles, ANTLR nos ofrece facilidades para recorrer estos árboles y ejecutar operaciones en consecuencia.

ANTLR es mucho más amplio de lo descripto hasta aquí, posee una gran cantidad de opciones y es altamente configurable. No entraremos aquí en detalles sobre todo lo que este framework nos ofrece, sino que veremos cómo ha sido utilizado para implementar nuestro analizador léxico y sintáctico.

ANTLR y GPSS

Para generar nuestro analizador de código GPSS, hemos definido un conjunto de reglas que permiten pasar de un código fuente escrito en GPSS al modelo descripto en este trabajo, en el cual tenemos principalmente un objeto Simulation compuesto por una lista de bloques y otra lista de comandos que serán ejecutados generando el resto de las entidades tanto permanentes como temporales. Estas reglas permiten interpretar una cadena de caracteres y generar un conjunto de elementos descriptos en nuestra gramática, contenida en el archivo GPSS.g, mediante la aplicación de una serie de reglas recursivas que, a partir de estos caracteres, generan un AST que tiene como raíz un programa GPSS y como hojas las expresiones que forman parte de los bloques y comandos.

Una vez armado el archivo con la gramática, debemos generar las clases Java que apliquen estas reglas gramaticales a un archivo de texto o cadena de caracteres de entrada, lo analicen y generen los objetos necesarios. ANTLR nos permite generar por un lado un Lexer (analizador léxico) y por el otro un Parser (analizador sintáctico); el primero se instancia con el código fuente y es utilizado por un objeto TokenStream, el cual provee un flujo de Tokens originados en el Lexer a partir del código fuente.

Una vez que tenemos el flujo de tokens, el Parser toma uno a uno los elementos de dicho flujo y verifica con cuál o cuáles reglas de la gramática coinciden (hacen *matching*), lo cual permite luego generar entidades propias del modelo GPSS. Por ejemplo, si recibe un token que representa un string START junto a otro token que representa el valor 1, el Parser sabrá que se hace referencia al comando START con parámetro 1, y generará una entidad Command con primer parámetro una SimpleNumericExpression(1).

La definición de bloques y comandos dentro de la gramática es muy general; sólo indica que un bloque (o comando) consta de un label (no obligatorio), el nombre de un bloque (o comando) y una lista de parámetros separados por comas. Si bien esta definición es en principio correcta, no se ajusta exactamente a lo definido en GPSS. Por ejemplo, un bloque GENERATE puede tener hasta 5

parámetros, mientras que un bloque ADVANCE solo tendrá 2; este tipo de restricciones no se han incluido en la gramática, ya que como dijimos, cada bloque o comando es el encargado de asegurarse que sus parámetros sean correctamente asignados y sean del tipo esperado. Pero es deseable que, en tiempo de traducción, el programador pueda saber por ejemplo que un determinado bloque no puede instanciarse con una cantidad incorrecta de parámetros.

Para adicionar esta posibilidad, hemos incorporado un objeto Interpreter, quien está también encargado de provocar la secuencia de traducción que hemos descripto, creando primero un Lexer a partir del código fuente, luego un TokenStream que utilice este Lexer, y finalmente el Parser que consume el flujo de tokens y genere un objeto Simulation. Este Interpreter realiza también la validación de los parámetros que hemos mencionado; este tipo de validación posee características tanto sintácticos como semánticos, y es específica de cada bloque o comando de GPSS. Para ello, cada subclase de la clase BlockEntity o de la clase Command posee un método validate(), que se encarga precisamente de verificar que los parámetros sean correctos. De este modo, una vez que el objeto Interpreter ha obtenido una entidad Simulation a partir del Parser, recorrerá la lista de comandos y bloques invocando a cada uno de los métodos validate() de cada uno de estos objetos.

En caso que alguna validación sea incorrecta, el objeto Interpreter agregará el mensaje de error y la excepción a una lista interna y continuará revisando el resto de los bloques y comandos existentes; al finalizar la revisión, en caso de existir algún tipo de error, informará al usuario de todos los errores encontrados (a diferencia de lo que hace GPSSW, que informa los errores a medida que son encontrados haciendo que el programador traduzca constantemente su código hasta que no contenga errores).

En determinados casos, el mecanismo de validación semántica realizado por el interprete puede ser insuficiente; por ejemplo si se posee una expresión del tipo “1 + P1”, donde P1 referencia implícitamente al parámetro 1 de la transacción activa, y P1 contiene un String, entonces se generara una excepción en tiempo de ejecución que indicara el error. Este tipo de validación no puede ser realizada en tiempo de interpretación debido al elevado grado de ortogonal que provee GPSS.

Capítulo 7. Conclusiones, críticas y trabajos futuros

Conclusiones.

Simular el comportamiento de un sistema mediante un modelo matemático no tiene prácticamente utilidad si no se puede evaluar lo que ha sucedido. Esta evaluación puede realizarse a partir de los datos de salida que nos entrega el entorno de simulación una vez que la corrida ha finalizado o a mientras se encuentra aun en curso. Analizar en detalle lo sucedido dentro de cada simulación es también una opción interesante y viable pero aún poco explorada en los lenguajes de simulación típicos.

Al ya amplio conjunto de herramientas de análisis de simulaciones de GPSS le hemos incorporado una nueva alternativa: mantener en un medio persistente el estado de toda la simulación en cada instante de reloj. Lo anterior no es en sí una herramienta de análisis, sino que resulta en un medio para realizar análisis específicos y comparaciones puntuales con un alto nivel de detalle; el potencial que esta nueva alternativa nos brinda es realmente inmensa, debido a que podemos conocer el estado de cada una de nuestras entidades en cualquier momento de la simulación, podemos saber cuándo nace y cuándo muere cada transacción, podemos conocer detalladamente el estado y las componentes de cada una de las cadenas, no solo de la simulación en sí, sino también de cualquiera de cadenas de entidades que participen en una simulación. Más aún, podemos parametrizar y ejecutar varias simulaciones, incluso en paralelo (si el hardware subyacente lo soporta) y luego analizar los resultados de cada una, o escribir nuevas herramientas que sean capaces de sacar conclusiones en base a comparaciones de cada simulación.

No es objetivo de este trabajo desarrollar todas estas herramientas, ya que requeriría una gran inversión de tiempo y un equipo de desarrolladores dedicado para tal fin. Por el contrario, aquí nos hemos planteado mostrar que, almacenar y posteriormente recuperar coherentemente toda la información generada en cada momento de reloj de una simulación y utilizarla para realizar diversos análisis es posible y viable. Luego, el modelo podrá ser extendido incluso por futuros desarrolladores, quienes también podrán incorporar herramientas de consulta y análisis basadas en el historial de corridas de simulaciones.

Además del beneficio de tener una herramienta de estas características para crear simulaciones, podemos adicionar una ventaja no tan obvia pero muy importante: su uso en entornos académicos para facilitar su aprendizaje. La enseñanza de los lenguajes de simulación es una difícil tarea, ya que requiere incorporar por un lado una nueva forma de ejecución de código "*secuencial"* donde existen objetos que entran a bloques, ejecutan una rutina asociada y salen de los mismos, y por el otro lado simplifica la comprensión del funcionamiento de muchas entidades muy distintas y, en algunos casos, extremadamente complejas. Aquí contamos con un desarrollo en un lenguaje de programación altamente difundido y de código abierto (específicamente Java), lo cual permite a alumnos del área modelos y simulación acceder y estudiar *en un idioma que comprenden y al que están acostumbrados* cómo funcionan estas entidades, qué significan realmente algunas operaciones o rutinas complejas, qué sucede cuando se ejecutan determinadas funciones, y otros tantos mecanismos "ocultos" del entorno de simulación.

Pero el uso de esta herramienta para simulaciones tiene también su costo; si bien las simulaciones no se detienen durante su ejecución, la persistencia en una base de datos de semejante cantidad de información trae aparejada una penalización en cuanto al tiempo del proceso completo, lo cual genera una demora que en ocasiones puede llegar a ser más que considerable. En algunas

pruebas realizadas, la simulación con alrededor de 2000 transacciones demoró entre 15 y 20 segundos en ejecutarse, pero su persistencia superó los 10 minutos, lo cual muestra claramente a que nos estamos refiriendo con la frase "una considerable penalización". Este tiempo depende de la cantidad de transacciones, de la cantidad de "tiempos" de simulación (esto quiere decir, la cantidad de veces que el reloj de simulación se actualiza), del número de entidades permanentes y de su uso, y de la forma en que las transacciones se crean y permanecen en el modelo (no es lo mismo que haya 5 transacciones en cada tiempo **t** a que haya 1000 transacciones en diversas cadenas en cada tiempo de simulación).

Sumado a la demora incorporada con este modelo de ejecución de simulaciones, debe considerarse la gran cantidad de recursos de hardware que se requieren. Mantener en memoria varios miles de objetos que se comunican entre sí y clonarlos manteniendo su copia también en memoria para cada cambio del reloj insume una gran cantidad de memoria; hacer esto para una simulación básica donde el reloj se actualiza como mínimo unas 100 veces implica multiplicar por este número la cantidad de memoria requerida. De todos modos, el consumo de memoria no ha representado un problema, ya que si bien se trata de varios miles de objetos, son objetos muy simples, nada que un equipo medianamente actual no pueda manejar.

Críticas y mejoras posibles

Es por cierto criticable el método en que se persisten los objetos. Para cada cambio de reloj, se clonan todos los objetos existentes y se colocan en una cola de pendientes para persistir. En cuanto a tiempo de procesamiento, no implica demasiado costo ya que la operación de clonado de objetos en memoria es altamente performante. Pero bajo este esquema, si un objeto no ha cambiado será igualmente almacenado por completo, y si un objeto considerablemente complejo o "pesado" solo ha alterado uno de sus valores, también se generará una copia completa. Un mecanismo de versionado de objetos [BAL01][KIM88] parece ser una solución ideal, ya que demandaría mucho menos espacio en disco (especialmente en momentos donde los objetos no varíen considerablemente) y también mucho menos tiempo para persistencia, ya que solo se guardarían los cambios. Si bien mantener versiones de objetos puede traer aparejadas muchas ventajas, también tendrá sus desventajas, como por ejemplo, conocer el estado de una entidad en un determinado tiempo **t** ya dejará de ser algo trivial, pues la entidad podría no existir en la base de datos en ese **t** (en caso de no haber cambiado), pero sí existir en un en un tiempo anterior y un tiempo posterior. Este planteamiento podría abrir una nueva rama de desarrollo para probar qué método es mejor. Posiblemente con una Base de datos que funcione de forma similar a los sistemas de versionado de código tales como Subversion, se podría plantear una solución altamente viable al problema de versionado de simulaciones que presentamos aquí en este trabajo.

Otro punto a criticar en este desarrollo es el modo en que se gestiona la lista de bloques. En una simulación con varios cientos de bloques, el acceso secuencial a los mismos podría no ser la mejor solución. Si bien la secuencialidad debe mantenerse, ya que es comúnmente deseable conocer cual es el siguiente bloque secuencial, también podría proveerse de un mecanismo que permita indexar esa lista para así poder localizar un bloque sin tener que recorrer toda la lista desde el principio. Una alternativa a la lista secuencia seria una tabla de hash que permita acceso directo y recorrido secuencial (esta alternativa se plantea mas en detalle en la sección de trabajos futuros).

Como hemos visto en el capítulo 6, la persistencia se realiza mediante un thread de menor prioridad. En la realidad, esto no ahorra demasiado tiempo al ejecutarse en un mismo equipo, ya que una vez finalizada la simulación, aún se debe esperar un tiempo considerable para que finalice el guardado; pero este thread podría perfectamente ejecutarse desde otra máquina en red, utilizando

una arquitectura de memoria distribuida, y logrando de este modo mejores tiempos de persistencia. También podría considerarse una mejora en la implementación de JDO utilizada; actualmente JPOX impone ciertas restricciones indeseables en lo que refiere a acceso concurrente sobre objetos persistibles.

Existe otra tarea que podría ser realizada utilizando concurrencia, tal es el caso de la gestión de las Retry Chains de las entidades. Como se ha explicado en el capítulo 2, las transacciones pueden bloquearse en espera del cambio de estado de una entidad cualquiera, agregándose automáticamente en la Retry Chain de la entidad que corresponde. Cuando dicha entidad cambia su estado, todas las transacciones frenadas en la Retry Chain son enviadas a la CEC para que chequeen nuevamente la condición que las bloqueó originalmente. Esta acción debería hacerse de forma concurrente mientras la transacción activa continúa su ejecución (siempre evaluando que ninguna de las transacciones que ingresan a la CEC tengan mayor prioridad que la activa). El uso de threads puede ser muy ventajoso, pero a la vez mucho más propenso a generar problemas o nuevos errores por sincronización[CAR05][GOE07], con lo cual deben tomarse especiales recados si se piensa utilizar esta alternativa.

Trabajos Futuros.

El editor de texto de nuestro desarrollo es un editor simple que no provee ningún tipo de facilidades al programador, como puede ser asistencia de código, sobresaltado de código fuente, corrección de errores gramaticales al vuelo, entre otros. Existen varios desarrollos que funcionan sobre la plataforma eclipse como plugins que permiten lograr este tipo de editores. TCS (Textual-Concrete-Syntax) por ejemplo, permite generar un editor en base a dos archivos de definiciones, uno .tcs que contiene información sintáctica y otro .km3 que contiene la definición del metamodelo subyacente. Otro plugin de Eclipse un poco mas sencillo es xtext, el cual genera en base a una gramática EBNF, muy similar a la de antlr, un parser, un metamodelo y un editor de texto con las comodidades típicas de un editor de eclipse (figura 7.1).

Actualmente existen muchas aplicaciones donde la interacción entre el usuario y el entorno de programación se realiza mediante objetos gráficos que son arrastrados y colocados donde se necesita mientras la aplicación genera por debajo un código ejecutable. La programación de simulaciones se adapta perfectamente a este tipo de herramientas, y de hecho ya existen algunos lenguajes en el mercado que funcionan de este modo, como por ejemplo ProModel [DUN06] o incluso MathLab con SimuLink [SIM08], en los cuales los usuarios seleccionan las entidades y organizan los modelos conectando de objetos gráficos mediante flechas y líneas. Un posible y muy interesante trabajo futuro para esta herramienta sería el de generar una interfaz icónica capaz de ofrecer este tipo de funcionalidades. Utilizando tecnología RCP junto con el entorno de desarrollo de plugins de Eclipse, se podría implementar un módulo gráfico que permita al usuario programador implementar modelos abstractos de manera mas sencilla [RCP08]. En la figura 7.2 se puede observar un ejemplo practico de interfaces iconicas junto con el código fuente correspondiente.

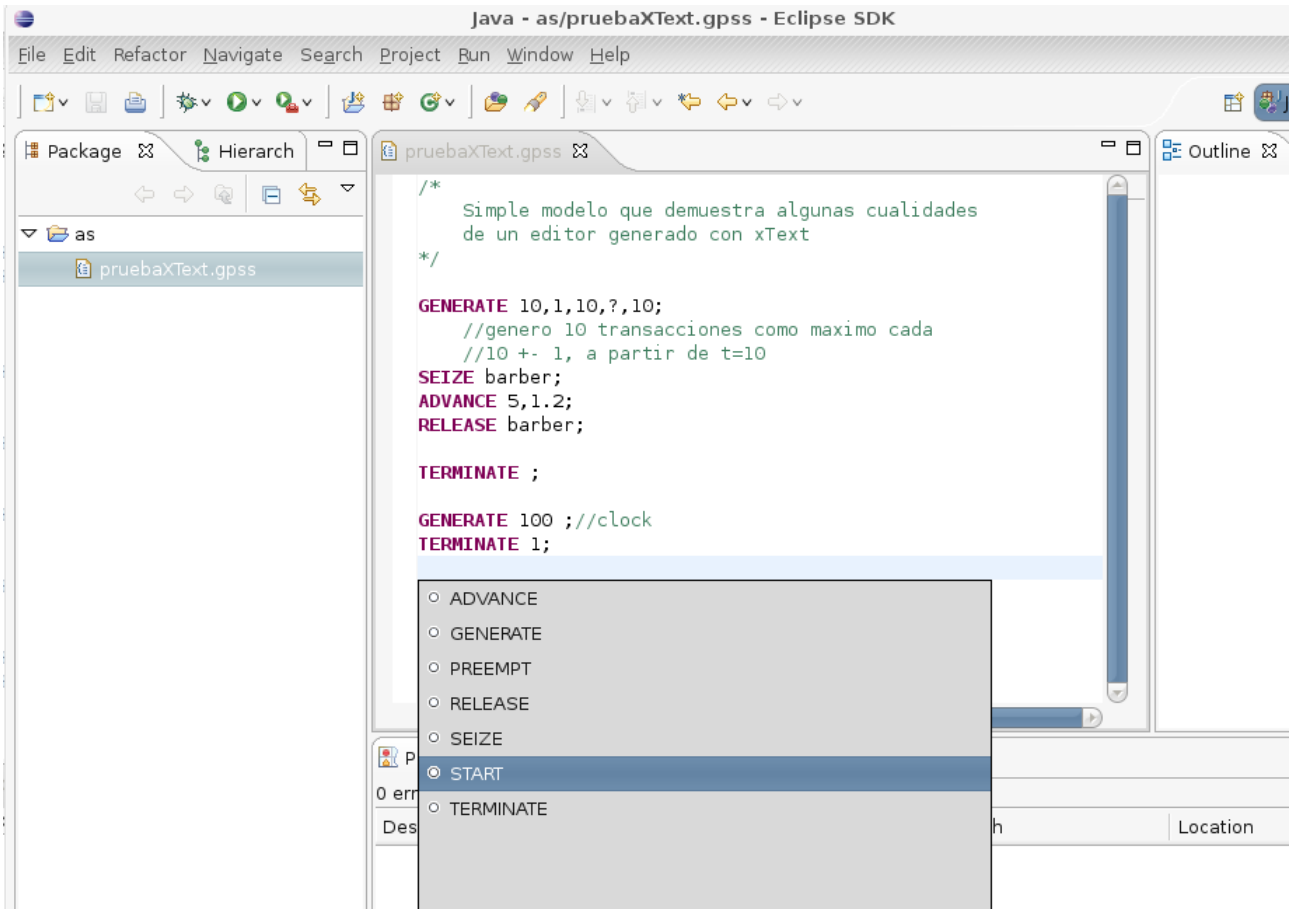

Figura 7.1 Editor ejemplo con xText/OpenArchitectureWare [OAW08]

Una última extensión sobre el entorno de desarrollo que podríamos considerar es el uso de herramientas de análisis mas complejas, como ser gráficos de torta, de dispersión, de barras, curvas de frecuencias, de áreas, entre otros. Ademas de gráficos, se podrían agregar tablas comparativas y herramientas de exportación de los datos generados. La información estadística podría ser generada al finalizar o incluso durante la ejecución de una simulación. La implementación de dichas herramientas no plantea un reto tecnológico, sino temporal y económico, ya que en el mercado existen plugins sobre Eclipse que proveen ciertas facilidades de este tipo. La figura 7.3 muestra un ejemplo de lo que podría hacerse con RCP, Eclipse y plugins OpenSource, tomado del proyecto MyTourbook [MYT08].

Continuando con la persistencia de simulaciones, podría decirse que almacenar simulaciones en cada cambio del reloj de simulación no refleja exactamente una simulación. Si bien no es objetivo de este trabajo poder conocer a ese nivel de detalle una corrida de una simulación, podría perfectamente pensarse en una extensión para tal fin. En cada tiempo de reloj pueden suceder muchísimos eventos, y estos no estarían siendo reflejados de acuerdo a nuestro esquema; por ejemplo, una misma transacción podría ser movida de la CEC a una delay chain y luego nuevamente a la CEC, y todo esto no se verá. La solución ideal sería almacenar, con cada cambio de *transacción activa,* toda la simulación nuevamente, pero esta solución sí sería excesivamente costosa; un mecanismo de versionado se hace ahora mucho más interesante que antes ya que sólo registraríamos los pocos cambios ocurridos a partir del cambio de la transacción en cuestión.

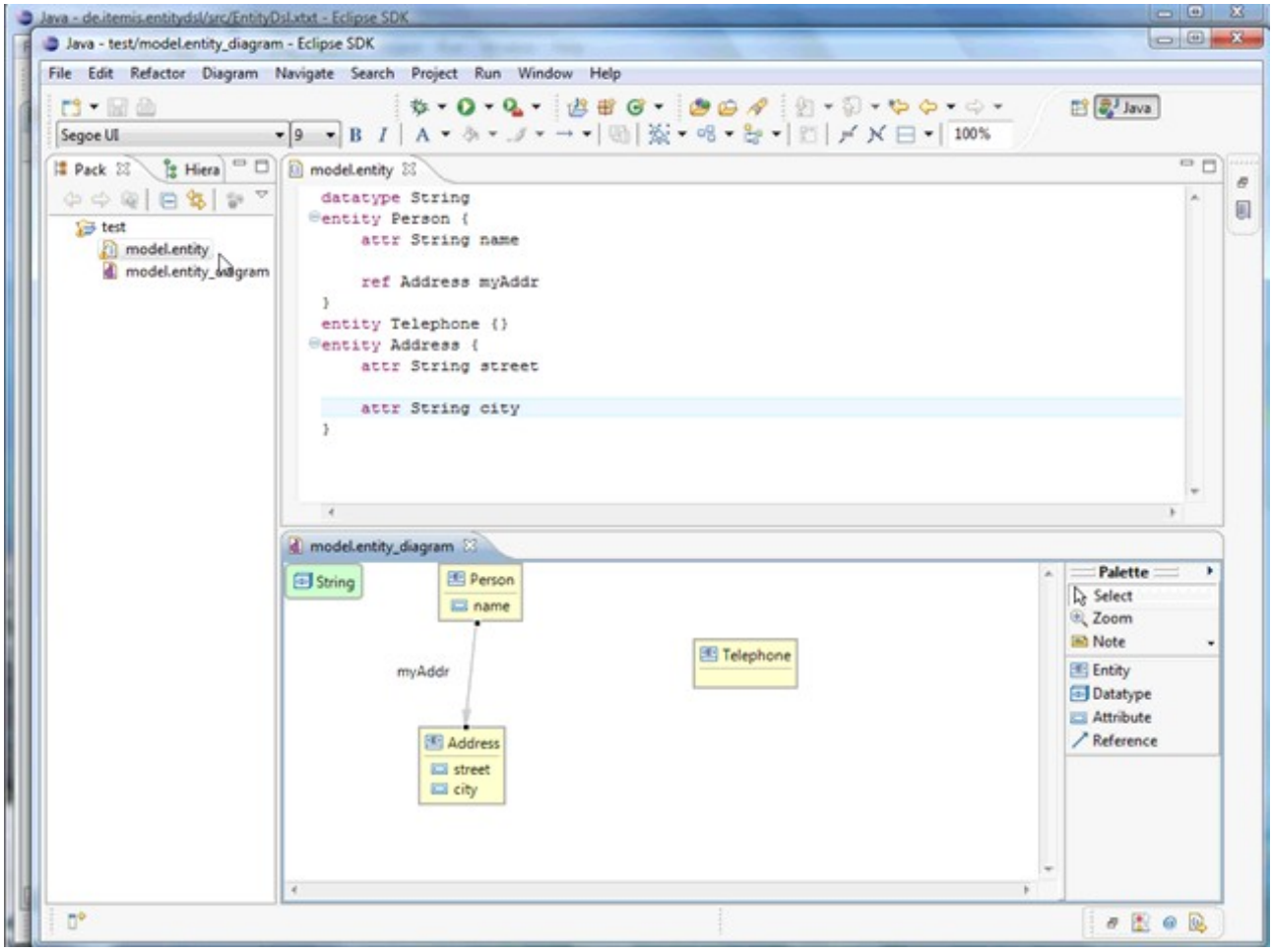

Figura 7.2. Construcción de modelos mediante interfaces icónicas

Como se indico en el capítulo 6, algunas de colecciones de entidades utilizadas poseen aun algunas falencias en lo que respecta a eficiencia en tiempo de ejecución. Por ejemplo, la lista de bloques que controla la simulación, es una lista enlazada que puede ser accedida en forma secuencial o en forma directa, aunque en realidad, este ultimo tipo de acceso es altamente costoso ya que el tiempo requerido por el algoritmo de búsqueda es de orden lineal. Una posible solución a este problema sería implementar una colección en base a una tabla de hash, para así lograr un acceso directo en tiempo lineal, con la característica especial de que cada nodo de la lista conoce al nodo siguiente, donde la secuencialidad de los nodos esta definido por algún mecanismo de ordenamiento en particular.

Entre los trabajos futuros, debe considerarse principalmente completar esta herramienta con los bloques y entidades faltantes definidos en GPSS. También se puede pensar en definir nuevos bloques y entidades que agreguen nuevas abstracciones no contempladas en el modelo de GPSS. Para ello, se deben tener en cuenta muchos aspectos particulares propios de nuestra implementación, por lo cual hemos incorporado un Apéndice donde se profundiza este tema apropiadamente.

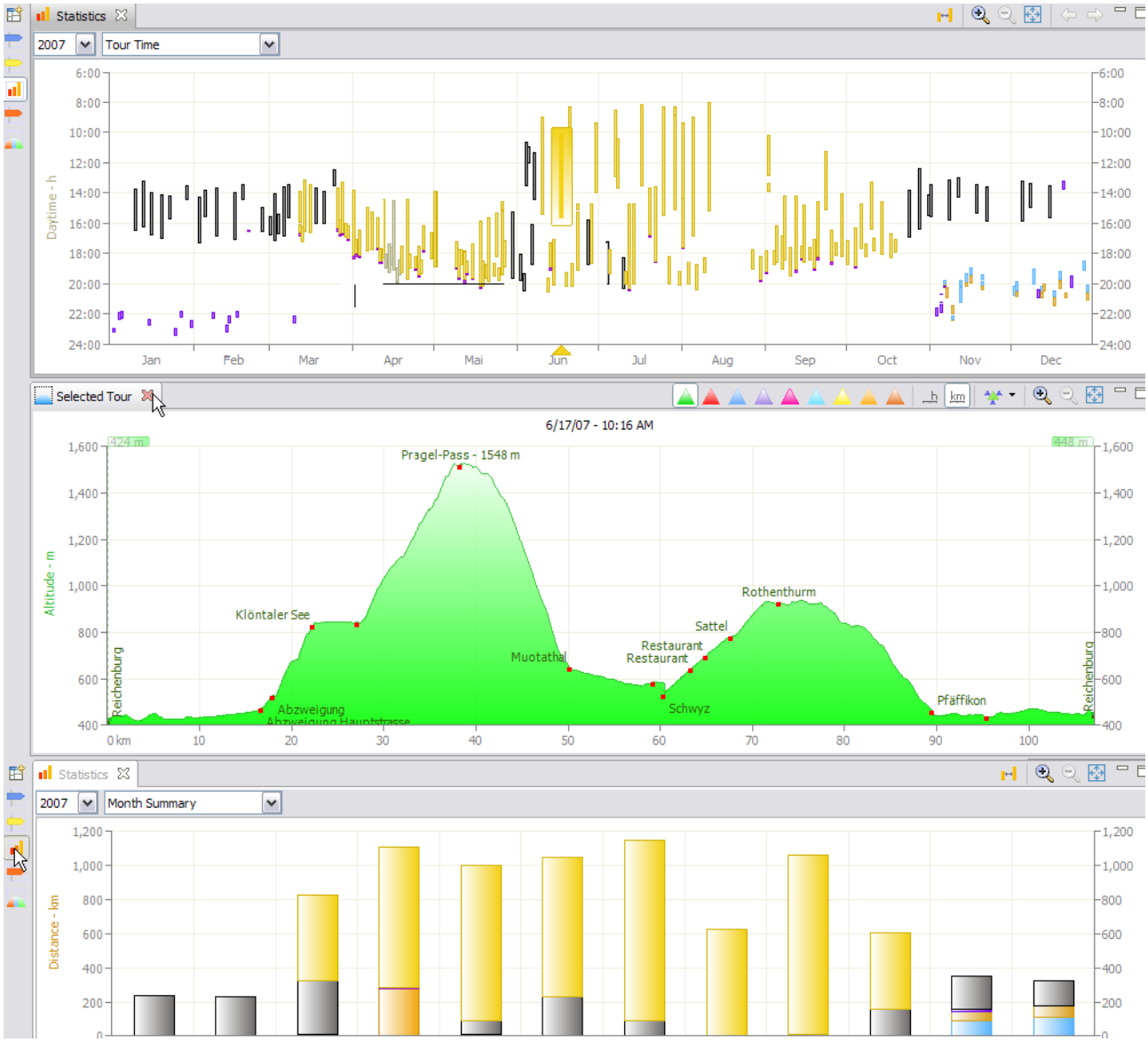

Figura 7.3. Las posibilidades gráficas que nos brinda Eclipse y RCP

Capítulo 8. Referencias

[AND99] Foundations of multithreaded, parallel and distributed programming. Gregory R. Andrews. Addison Wesley. 1999.

[ANT08] ANTLR v3. Another Tool for Language Recognition http://www.antlr.org/

[BAL01] Database Schema Evolution and Meta-Modeling: 9th International Workshop on Foundations of Models and Languages for Data and Objects FoMLaDO/DEMM 2000 Dagstuhl. by Herman Balsters, Bert de Brock. Stefan Conrad. Springer; 1 edition. 2001.

[BAR96] Douglas K. Barry. The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases. Wiley. 1996.

[CAR05] Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C ++/Pthreads/Win32 Programs. Richard H. Carver. Wiley-Interscience 2005.

[CAT00] The Object Data Standard: ODMG 3.0. R. G. Cattell, Douglas K. Barry, Mark Berler y otros. Morgan Kaufmann. 2000.

[DUN06] Simulación y Análisis de sistemas con ProModel. Eduardo García Dunna. Heriberto García Reyes. Leopoldo E. Cárdenas Barrón. Pearson. 2006.

[ECL08] http://www.eclipse.org

[GOE07] Java Concurrency in Practice. Brian Goetz. Addison Wesley. 2007.

[GRA94] Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. Addison-Wesley. 1994.

[HIB08] Hibernate ORM http://www.hibernate.org/

[JDO08] Java Data Objects http://java.sun.com/jdo/

[JOR03] Java Data Objects. David Jordan, Craig Russell. O'Reilly Media, Inc. 2003.

[JPO08] Java Persistent Objects http://www.jpox.org/

[KIM88]Schema versions and DAG rearrangement views in object-oriented databases (Technical report. University of Texas at Austin. Dept. of Computer Sciences). Hyoung Joo Kim. University of Texas at Austin, Dept. of Computer Sciences. 1988

[KIM90] Introduction to Object-Oriented Databases. Won Kim. The MIT Press. 1990.

[LAN85] Teoría de los sistemas de información. Langefors, Börje. 2a. ed. (1985)

[LAU91] The Role of JSC Engineering Simulation in the Apollo Program. James A. Lawrence. NASA/ISC Houston, Texas. Simulation Councils Inc. (1991).

[MIN05a] http://www.minutemansoftware.com

[MIN05b] http://www.minutemansoftware.com/reference/reference_manual.htm

[MIN05c] http://www.minutemansoftware.com/tutorial/tutorial_manual.htm

[MYT08] MyTourbook software. http://mytourbook.sourceforge.net

[NAS08] NASA Office of Logic Design. http://klabs.org/

[OJB08] Abache DB Object/Relational Bridge http://db.apache.org/ojb/

[PIE93] Design of Hashing Algorithms. Josef Pieprzyk, Babak Sadeghiyan. Springer-Verlag. 1993

[POL07] PolePositions benchmark results. http://www.jpox.org/docs/polepos/results/html/ . 2007

[RCP08] http://wiki.eclipse.org/index.php/Rich_Client_Platform

[ROB04] Monte Carlo Statistical Methods. Christian P. Robert, George Casella. Springer (2004).

[SIM08]. Simulation and Model-Based Design http://www.mathworks.com/products/simulink/

[SOU96] Extracting n-ary relationships through database reverse engineering. Christian Soutou. Springer Berlin / Heidelberg. 1996.

[STA00] Organización y arquitectura de computadores. William Stallings. Prentice Hall, 5ª edición, 2000.

[TER07] The definitive ANTLR reference. Terrence Parr. Raleigh. 2007.

[VER08] Versant Database. http://www.versant.com/

[VIT86] Design and Analysis of Coalesced Hashing. Jeffrey Scott Vitter, Wen-chin Chen. Oxford University Press, USA. 1986.

[WEB99] WebGPSS web site. http://www.webgpss.com

[WIL66] Computer simulation techniques. Wiley; 1st corr. printing edition (1966)

[WOL07] Wolverine Software web site. http://www.wolverinesoftware.com/

APENDICE I. Extensión del modelo

Uno de los objetivos que nos hemos planteado en este trabajo fue el de desarrollar las bases necesarias para una herramienta de modelado y simulación con los mismos principios que utiliza el lenguaje GPSS, pero con aspectos adicionales orientados principalmente a mejorar a futuro las posibilidades de análisis y debug de las simulaciones, así como también para facilitar la enseñanza de este tipo de lenguajes. Estas bases debían proveer todo aquello que las entidades requieren para integrarse a las simulaciones, interactuando con las transacciones y con las entidades permanentes, de manera de permitir que otros desarrolladores puedan incorporar nuevas entidades, crear sus propias entidades o incluso mejorar todo lo que se ha implementado.

A lo largo de este documento hemos hecho énfasis en muchos aspectos de diseño, tanto a nivel de modelado como de funcionamiento, propendiendo a la comprensión del modo en que fueron pensados todos y cada uno de los módulos que forman parte de esta herramienta. El objetivo que nos hemos propuesto es simple: si se comprende como funciona cada componente y qué justifica su intrincado comportamiento, será más sencillo para cualquier desarrollador generar nuevas componentes que continúen con la misma línea de trabajo. En esta sección, describiremos algunos aspectos para que futuros desarrolladores tengan en cuenta al momento de extender el modelo, propendiendo a mantener una metodología unificada de integración y adición de componentes.

Nuevas entidades.

Como nos muestra la figura 6.6, todas las entidades son subclase de la clase Entity. Esto es muy importante, ya que esta clase posee la implementación de la retry chain de cada entidad, utilizando una versión personalizada del patrón de diseño *Observer* [GRA94] donde cada entidad es puesta como *observadora* de otra entidad (se coloca en su retry chain) esperando que ocurra algún cambio. Este mecanismo es el mismo para cualquier entidad y para cualquier atributo de las entidades, y debe ser mantenido en nuevas entidades. La clase entity también provee los mecanismos para asociar nombres y valores a entidades, lo cual resulta en un ahorro de trabajo muy importante tanto en la implementación de nuevas entidades como en la localización de las mismas en el modelo.

En caso de incluir en el modelo una entidad que ya existía en GPSS, recomendamos fuertemente mantener su nombre original, así como también el nombre de los SNAs y bloques que la acompañan. Esto nos permite trasparentar el código GPSS entre las distintas implementaciones (incluida ésta), pero principalmente evita la duplicación de trabajo ya que, si se utilizan nuevos nombres para entidades ya existentes, los programadores podrían no notar su existencia y volverán a implementarla. Esto no quiere decir que los programadores deben limitarse a las entidades de GPSS; con este trabajo proponemos que nuevas entidades se generen, documenten e incorporen al modelo, a fin de proveer de más y mejores herramientas para crear simulaciones complejas.

Cada nueva entidad tendrá sus propios SNAs; si bien puede tener varias decenas de ellos, al menos un SNA que permita identificar a la entidad debe existir (del mismo modo que existe el SNA F para identificar una facilidad, XN1 para la transacción activa o X para un Savevalue). Los nuevos SNAs deben colocarse dentro del objeto SNAMapping, ya que de otro modo no se lo podrá utilizar, definiendo su código y la entidad sobre la que se ejecutará (capítulo 6: Gestión de expresiones). Desde luego, además de agregarlo al mapeador de SNAs, deberá implementarse el evaluador del SNA dentro de la nueva entidad (la clase abstracta Entity obliga la implementación del método

evaluateSNA), ya que de lo contrario el SNA existirá pero no estará asociado a ningún valor o función de la nueva entidad.

Nuevos bloques.

La adición de nueva entidades de simulación permanentes debe estar acompañada *al menos* de un conjunto de bloques para que las transacciones utilicen los servicios que la nueva entidad agrega al modelo. Del mismo modo, la funcionalidad de entidades ya existentes puede ser extendida mediante la incorporación de nuevos bloques.

Si bien los bloques también son entidades (subclases de Entity), existe una clase abstracta llamada BlockEntity que debe ser superclase de cualquier bloque. Esto hará que los nuevos bloques mantengan la interfaz mínima requerida: un método tryEnter, que ejecutarán las transacciones cada vez que intenten ingresar a un bloque y que retornará un valor de tipo EnterResult (ACCESS_DENIED, BLOCKED, CONTINUE) indicando si se le denegó el acceso, si accedió al bloque y quedó bloqueada en el mismo, o si accedió, ejecutó su código y continuó adelante sin ningún bloqueo.

Los bloques también deberán implementar el método getNextBlock para una determinada simulación. Este método es muy importante, ya que determinará la forma en que las simulaciones continuarán ejecutándose cuando hayan finalizado la rutina del bloque actual. Existen básicamente dos formas de continuar ejecutándose: siguiente bloque secuencial, o mediante un salto. Para el primer caso, la clase BlockEntity provee lo necesario para que nuevos bloque simplemente invoquen a este método (con la sentencia super(), desde luego) sin tener que escribir código extra. Para el segundo caso, la implementación dependerá de la forma en que se realizan los saltos (condicionales, incondicionales, mediante una función de probabilidad, mediante intentos o reintentos, etcétera); solo recordaremos a los desarrolladores que los bloques son también entidades, con lo cual el modelo ya provee lo necesario para localizarlos ya sea por su nombre (aquí será un label, pero en GPSS el label de un bloque determina la asociación valor-nombre) o por su identificador (como se explicó en el capítulo 4, Entity References Manager, la asignación de IDs a los bloques no sigue la misma secuencia del resto de las entidades, con lo cual en principio no deberá generar conflictos). Al igual que las entidades permanentes, promovemos con este trabajo la creación de nuevos bloques tanto para nuevas entidades como para entidades propias de GPSS.

Asignación de tareas.

Cada nueva entidad implicará seguramente un intrincado comportamiento interno y una manera de relacionarse con el modelo no menos compleja. Sumado a esto, estará acompañada de un conjunto de bloques que permitirán a las transacciones utilizar esta nueva facilidad, lo cual provoca que con cada nueva componente se deba agregar una gran cantidad de comportamiento. Pero, ¿a qué componente le corresponde realizar cada una de las tareas?

Esta pregunta no tiene una única respuesta correcta, ya que dependerá en gran medida de la componente que está siendo agregada y de los bloques asociados. Pero podemos marcar algunas líneas generales que hemos seguido hasta aquí:

- La validación de los parámetros es trabajo de cada bloque, lo cual hace que cuando un bloque interactúe con una componente, lo haga de manera segura ya que las validaciones correspondientes ya habrán sido realizadas. De este modo, la componente no se hace excesivamente compleja ya que no requiere conocer y validar todos y cada uno de los

parámetros de todos los bloques que la utilizan.

- Resulta muy apropiado el uso de mecanismos de logueo, tanto en bloques como en métodos de otras entidades. Hemos utilizado log4j para la gestión de logs, lo cual simplifica enormemente la tarea, y alentamos a los desarrolladores a mantener esta política.
- El estado interno de las entidades debe ser accedido por los bloques mediante métodos solamente. Corresponde a cada entidad conocer y modificar su estado interno, siempre. Esto evita muchos inconvenientes y mantiene seguridad en el desarrollo; adicionalmente, de este modo es más fácil incorporar cierta funcionalidad de manera transparente (por ejemplo, pasaje automático de transacciones de la FEC a la CEC al cambiar el reloj de simulación, o incluso el logueo mencionado en el punto anterior).
- Las entidades deben resolver sus propios SNAs, y los SNAs deben involucrar solo a elementos de la entidad a la que pertenecen.
- Si la nueva entidad posee una o más cadenas, será la entidad la encargada de mover transacciones desde y hacia sus cadenas. Los bloques no deberían interactuar con estas cadenas, sino delegar la responsabilidad a la entidad en cuestión. Por ejemplo, el bloque SEIZE en su método tryEnter realiza básicamente 3 tareas: obtener la facilidad (por el nombre o su valor), invocar al método seize() de dicha facilidad y evaluar el resultado (success, failure). El método seize() es quien analiza si la transacción puede hacer seize o si debe esperar en la delayChain, y en tal caso la coloca y retornará false.
- Debido a la diversidad de bloques, es tarea de cada bloque la de mantener contadores de transacciones que se encuentren detenidas y de transacciones que han pasado por allí.
- Recomendamos la implementación del método toString() con cada nueva entidad que se agregue al modelo, así como también una apropiada implementación del mismo. Este método nos permite imprimir en una consola el estado interno de los objetos, y es muy importante tanto para debug como para el logueo, Recordemos también que existe un objeto capaz de imprimir en consola el estado interno completo de cada entidad. Este objeto implementa el método print(Entity e); es recomendable aprovechar dicho objeto e implementar este método para cada nueva entidad.

Clonación.

Como hemos mencionado en el capítulo 6, las entidades que se persisten deber ser clonadas para evitar que se actualice su estado. Esta clonación debe necesariamente incluir todas las variables que hacen al estado de cada entidad, y en caso de poseer referencias, cadenas o listas de entidades, deberán recursivamente provocar la clonación de cada una de las entidades que referencian.

Debemos aquí recordar que existe un objeto que está por encima de todos los objetos (al que hemos llamado God) que mantiene todos los cambios que han sufrido las simulaciones y todas sus entidades a lo largo del tiempo, pero que también mantiene registro de cuales son aquellos objetos que han sido clonados en cada cambio del reloj. Debe tenerse muy en cuenta este concepto, ya que la implementación del método clone() debe poseer una invocación al objeto God para que registre el nuevo clon.

Persistencia de nuevas entidades

Clonar entidades no asegura que las mismas se almacenen en la base de datos, sino que permite que en caso que se almacenen, lo hagan apropiadamente. Para que las entidades sean realmente persistibles, deben estar incluidas en el archivo .jdo del paquete al que corresponden, y las clases de dichas entidades deben ser aumentadas mediante el enhancer de JPOX.

APENDICE II. Herramientas utilizadas.

A lo largo de todo el documento hemos hecho especial énfasis en la filosofía abierta que sigue este desarrollo. También hemos realizado algunos comentarios sobre tiempos, velocidades, recursos del sistema y otros factores pertinentes. Hemos decidido pues incluir este apéndice, en el cual simplemente se listan las herramientas de software y hardware utilizados, destacando en el primer caso que solo se utilizaron aplicaciones abiertas, y en el segundo el tipo de hardware utilizado para el procesamiento de las distintas pruebas.

Herramientas open source

El trabajo que hemos desarrollado y presentado aquí fue desarrollado enteramente con herramientas de código abierto. Nuestra herramienta es en sí un desarrollo de código abierto. Las herramientas open source que hemos utilizado para este desarrollo son:

- Linux Mint 4.0 (Darnya) como Sistema Operativo para desarrollo y testing
- SUN Java versión 1.6.0_03 como lenguaje de programación
- Eclipse 3.3.1 como entorno de desarrollo
- MySQL como motor de bases de datos
- JPOX como implementación de la interface JDO
- OpenOffice.org Writer para escribir este documento
- ArgoUML y Umbrello como herramientas para gráficos UML
- OpenOffice.org Draw para los gráficos
- Firefox, PhpMyAdmin y Apache para análisis sobre base de datos
- Inkscape para el diseño de la tapa, y uniconv para convertir del formato cdr al svg.

Hardware utilizado.

Todas las pruebas fueron realizadas sobre computadoras con configuraciones típicas, corriendo el software descripto en el punto anterior. A continuación detallamos los principales componentes de hardware utilizados, haciendo énfasis en la velocidad de los mismos (lo cual es un factor determinante para poder estimar realmente los requerimientos de las pruebas):

- Procesador AMD Turion64 X2 1.6 Ghz
- Memoria: 2 GB DDR2 de 667 Mhz
- Disco rígido de 160 GB SATA2 5400 rpm
- Linux kernel 2.6.22-14-generic 32 bits

APENDICE III. Gramática.

A fines del capítulo 6 hemos realizado una breve introducción a ANTLR, la herramienta de generacional automática de análisis utilizada. A continuación incluiremos la gramática utilizada en la sintaxis requerida por ANTLR con algunos comentarios marginales. El objetivo de este apéndice es permitir al lector visualizar la estructura básica del lenguaje implementado. La Figura A3.1 nos muestra gráficamente cuales son las interdependencias entre estas gramáticas.

Gramatica Antlr

```
grammar GPSS;

simulation: verb+ EOF;

commandSMT:
      CommandName (genericExpression((',') genericExpression)* )? ';' ;

verb: blockSMT | commandSMT;

blockSMT:
      (IDENTIFIER)? BlockName
      (genericExpression ((',') genericExpression)* )? ';';

genericExpression:
      nullExpression | identifierExpression | valueExpression;

valueExpression: additive_expression ;

snaExpression: directSNA | indirectSNA;

directSNA: atomicSNA | simpleSNA;

simpleSNA:
      EntitySNAClass (simpleNumericExpression | '$' identifierExpression);

atomicSNA: AtomicSNA;

indirectSNA:
      EntitySNAClass '*'
      (simpleNumericExpression |('$')? identifierExpression);

additive_expression:
      multiplicative_expression (('+'|'-') multiplicative_expression)?;

multiplicative_expression:
      primary_expression  (('/'|'#') primary_expression)?;

primary_expression:
      snaExpression | simpleNumericExpression | '(' valueExpression ')' ;

simpleNumericExpression: POS_INTEGER_LITERAL  | DOUBLE_LITERAL;

identifierExpression: IDENTIFIER;

nullExpression: ('?');
```

```
RULE_SL_COMMENT :
  (':'|'//') ~('\n'|'\r')* '\r'? '\n'
;

STRING :
   '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') | ~('\\'|'"') )* '"' |
   '\'' ( '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') | ~('\\'|'\'') )* '\''
;

WS : (' '|'\t'|'\r'|'\n')+;

ML_COMMENT : '/*' (.)* '*/';

BlockName:
      ('GENERATE'  | 'SEIZE' | 'RELEASE' | 'TERMINATE'
       | 'ADVANCE' | 'ASSIGN' | 'PREEMPT' | 'RETURN');

CommandName: ('CLEAR' | 'EQU'| 'RESET' |  'START');

EntitySNAClass: 'RN'|'P'|'F'|'FC'|'FI'|'FV'|'N'|'W';

AtomicSNA :'A1'|'M1'|'PR'|'XN1'|'TG1'|'AC1';

IDENTIFIER: LETTER (LETTER|'0'..'9')*;

fragment
LETTER:     'A'..'Z' | 'a'..'z' | '_';

STRING_LITERAL:  '"' ( EscapeSequence | ~('\\'|'"') )* '"';

POS_INTEGER_LITERAL : ('0' | '1'..'9' '0'..'9'*) ;

DOUBLE_LITERAL:
      ('0'..'9')+ '.' ('0'..'9')* Exponent?
    | '.' ('0'..'9')+ Exponent?
    | ('0'..'9')+ Exponent?
;

fragment
Exponent : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
EscapeSequence:   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');
```

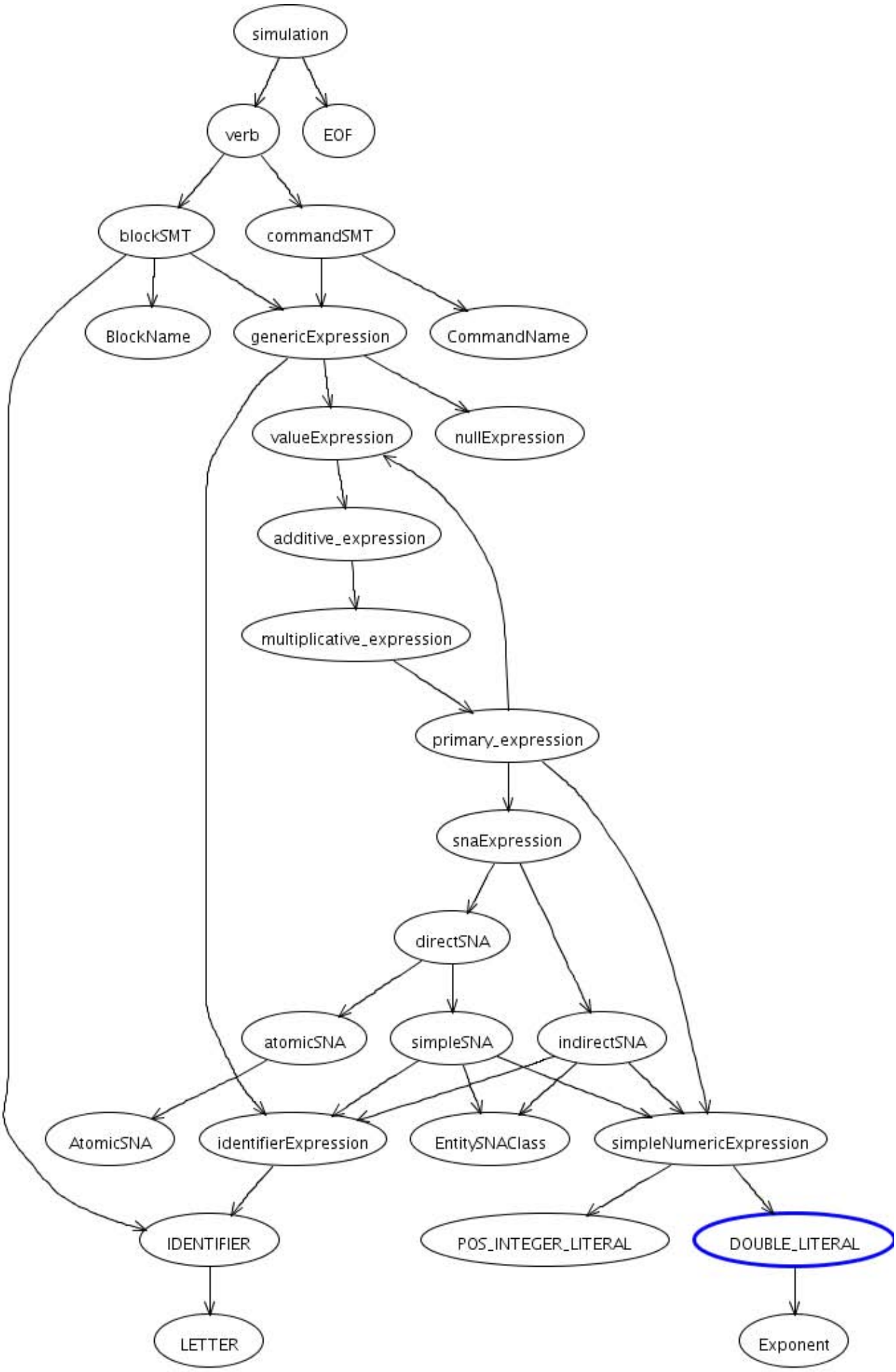

Figura A3.1: Grafo de dependencia