Departamento de Investigación Operativa
Instituto de Computación - Facultad de Ingeniería
Universidad de la República (UdelaR)
Montevideo - Uruguay

Proyecto de Grado
Ingeniería en Computación

SimPP
Librería de Simulación en C++

Sebastián Alaggia
Bruno Martínez
Fernando Pardo

Diciembre 2005

Tutores
MSc. Antonio Mauttone
MSc. Maria Urquhart
Facultad de Ingeniería - Universidad de la República (UdelaR)
Departamento de Investigación Operativa - Instituto de Computación
Montevideo - Uruguay

1. Resumen

La simulación a eventos discretos es una técnica del área de Investigación de Operaciones. Es una técnica más que nada experimental. La idea detrás de simulación es experimentar con aquellos modelos que presentan grandes dificultades a la hora de realizar pruebas. Esto puede ocurrir porque es extremadamente costoso o porque no se tiene acceso al sistema real.

El presente informe trata sobre la investigación, desarrollo y prueba de SimPP. SimPP es una biblioteca de simulación a eventos discretos basada en componentes independientes y desacoplados, está implementada en C++ y es capaz de simular sistemas modelados en varios paradigmas de simulación a eventos discretos.

Este proyecto se dividió en tres fases. Durante la primera fase del proyecto se realizó un estudio del estado del arte en cuanto a herramientas existentes, paradigmas desarrollados hasta el momento, diversas técnicas de diseño e implementación en C++, y librerías externas que podrían sernos útiles.

Luego de realizada esta fase de investigación se definieron los requerimientos en base a los cuales SimPP sería construido. Se tuvo como requerimiento fundamental crear una herramienta capaz de correr sistemas modelados en varios paradigmas utilizando un calendario y ejecutivo simples.

Finalmente se probó a SimPP simulando el sistema de transporte público de la ciudad de Rivera. En la implementación de este sistema se utilizó una técnica de diseño e implementación estudiada anteriormente y basada en el refinamiento de componentes a gran escala.

2. Índice

Informe Ejecutivo

1. Introducción

El Departamento de Investigación Operativa de la Facultad de Ingeniería ha utilizado la técnica de simulación a eventos discretos durante años en el área de modelado de sistemas. Si bien este departamento posee varias herramientas de simulación, no tiene ninguna herramienta avanzada y libre capaz de realizar estas tareas. Desde hace varios años se tiene la idea de construir una biblioteca o lenguaje de simulación que sea del propio instituto, pero dicha propuesta no ha sido llevada a cabo; el proyecto SimPP busca cumplir con ella.

El objetivo del proyecto SimPP es la construcción de una librería de simulación a eventos discretos implementada en C++. Este objetivo parece bastante concreto a primera vista, pero dada la gran variedad de herramientas para simulación que han sido construidas, definir explícitamente los requerimientos de este tipo de sistemas no es tan sencillo como parece. Fue necesario definirlos antes de comenzar el diseño de la librería.

Para poder definir los requerimientos de la librería que se construiría, se comenzó con la búsqueda de un conjunto de funcionalidades que sirvieran para clasificar las herramientas de simulación existentes. Para ello, se investigaron varias herramientas de simulación a eventos discretos y se las clasificó de acuerdo a una clasificación que produjo el equipo en base a aquellas que fueron halladas en la literatura.

Otro punto que se abarcó y en el cual se puso mucho énfasis fue en el estudio de los diversos paradigmas de simulación que existen en la literatura. Se investigaron no solo los paradigmas clásicos de esta área como orientación a eventos y orientación a procesos; se estudió el formalismo Devs y el paradigma de Grafos de eventos. También se estudiaron técnicas para implementar simuladores paralelos.

Se hizo foco en el área de diseño e implementación, tanto para el desarrollo de la librería como para el modelado de sistemas que la utilicen. Debido a que C++ soporta mecanismos de abstracción de tipos, mecanismos de herencia, sobrecarga de funciones y despacho dinámico, es capaz de soportar patrones que utilicen tipos paramétricos. Se trató de encontrar patrones de arquitectura innovadores y que fueran elegantemente soportados por C++.

Luego de esta etapa de investigación se procedió al desarrollo de la librería. Como metodología de desarrollo se decidió utilizar una metodología ágil. Este modelo de proceso es iterativo, incremental y fuertemente basado en testing. La librería fue construida en forma incremental, refinando los componentes a medida que se iban construyendo otros. SimPP es

una librería libre y portable basada en componentes desacoplados, capaces de ser utilizados juntos o separados. Otra de las características distintivas de SimPP es su capacidad de soportar sistemas modelados con el paradigma de procesos, eventos o con un enfoque híbrido. Soporta persistencia de simulaciones, aunque limitando las estructuras de modelado. Ofrece herramientas útiles para la visualización gráfica, recolección de datos, manejo de complejidades y serialización.

Finalmente se modeló el sistema de ómnibus de la ciudad de Rivera utilizando la librería creada. Se decidió modelar este sistema a modo de prueba de la librería. La aplicación fue implementada de acuerdo con la metodología GenVoca utilizando una estructura Gestalt (ver apéndice C) diseñada por el equipo, lo que permite crear un modelo en capas cuyos refinamientos son sucesivos. Además permite que las capas no dependan unas de las otras, lo que hace que los modelos que se construyan de este modo sean más fácilmente mantenibles.

En el siguiente informe ejecutivo se tiene un resumen de todo lo realizado a lo largo del proyecto. En caso de querer un estudio más profundo sobre un determinado tema brevemente desarrollado en el informe, se debe leer el apéndice correspondiente. En la sección 2 se tratan los objetivos de este proyecto. En la sección 3 se da el marco conceptual sobre el cual se apoya este proyecto. En la sección 4 se da un resumen del estudio del estado del arte que fue realizado, esta abarca herramientas, paradigmas y técnicas de diseño e implementación. En la sección 5 se presentan los requerimientos que se le han impuesto a SimPP. En la sección 6 se explican las decisiones de diseño que fueron tomadas en la construcción de SimPP. En la sección 7 se describe el caso de estudio que se utilizó para la prueba de la librería. En la sección 8 se dan las conclusiones de este proyecto y finalmente en la sección 9 se menciona el trabajo futuro que se puede realizar sobre SimPP.

2. Objetivos

Dentro del área de simulaciones a eventos discretos hay varios temas en los que se puede hacer foco, como por ejemplo manejo de estructuras y algoritmos eficientes para correr una simulación, modelado de sistemas y tratamiento estadístico de los datos. En nuestro caso decidimos orientar el proyecto más en el modelado de sistemas y el manejo de estructuras y algoritmos. No nos focalizaremos en el tratamiento estadísticos de datos de entrada o de salida.

Más concretamente, los objetivos del presente proyecto de grado son los siguientes:

- Hallar las características de las herramientas de simulación a eventos discretos. Al tener un conjunto de características comunes en este tipo de herramientas, es posible clasificarlas de acuerdo a estás. Además, con esta clasificación es fácil elegir el tipo de herramienta que se quiere construir.
- Construir una librería de simulación a eventos discretos, focalizándose en el modelado de sistemas. Dicha librería debe incluir un conjunto de funcionalidades básicas y avanzadas. En caso de no soportar alguna característica avanzada, la librería debe ser suficientemente flexible como para dar lugar a futuras extensiones que soporten dichas características.
- La librería debe estar construida en C++. Este objetivo, aunque un tanto restrictivo, no es tal. C++ es un lenguaje que se encuentra continuamente en desarrollo y cuenta con un número importante de adeptos, además de que posee construcciones que no son brindadas por la gran mayoría de los lenguajes orientados a objetos que existen actualmente.

3. Marco conceptual

Para adentrarnos en el desarrollo de este proyecto primero se debe establecer un marco conceptual de las áreas de trabajo de este proyecto. Dada la naturaleza del proyecto, es necesario introducir al lector en la disciplina de Simulación, en la situación actual de C++ y las librerías externas que este ofrece.

Se dará una breve introducción en el área de Simulación, luego una pequeña reseña histórica y una introducción a los paradigmas clásicos de simulación a eventos discretos.

En cuanto a C++ se hablará de sus características más sobresalientes, se dará una reseña histórica y una perspectiva de desarrollo del mismo.

3.1. Simulación

La técnica de simulación de sistemas forma parte del área de Investigación Operativa. La idea detrás de la simulación de sistemas es experimentar con sistemas a los cuales se tiene poco o ningún acceso. Esto es tratar de experimentar con sistemas donde el proceso de experimentación es muy costoso o imposible debido a que no se dispone del sistema real para experimentar.

Se define la simulación de sistemas como la construcción de un modelo abstracto basado en un sistema, dicho modelo luego es llevado a un programa de computadora para poder experimentar con él. Según [BCNN00] la simulación de sistemas involucra la generación artificial de la historia de un modelo.

Se puede distinguir entre dos tipos de simulaciones de acuerdo al rol que cumple el tiempo en el sistema: la simulación continua y la simulación discreta. Cuando cada instante de tiempo es relevante para el sistema a modelar, se dice que este la simulación de dicho sistema es continua. En cambio cuando solo ciertos instantes de tiempo son relevantes para el sistema, nos encontramos ante una simulación discreta o a eventos discretos.

Como ejemplos de ambos sistemas se puede tomar el motor de un vehículo y un cruce de semáforos. En el caso del motor, cada punto del tiempo es importante para saber en que estado se encuentra este. El motor evoluciona continuamente a medida que el tiempo pasa. En cambio en un cruce de semáforos, no todos los instantes de tiempo son relevantes puesto que este sistema no evoluciona continuamente con el tiempo. El sistema del cruce de semáforos solo cambia de estado cuando cambian de color las luces del semáforo, o cuando los autos atraviesan el cruce.

Generalmente los sistemas continuos se modelan mediante sistemas de ecuaciones diferenciales, mientras que los sistemas discretos se manejan con modelos de colas de espera. Ambos sistemas son modelados generalmente en forma estocástica, o sea que dependen fuertemente de la generación de números aleatorios. A la hora de correr una simulación, los resultados que de ésta se obtengan deben ser tomados como el resultado de una muestra de una variable aleatoria. De aquí en más solo se tratará la simulación a eventos discretos.

Metodología de la simulación a eventos discretos

El proceso de construcción de la simulación de un sistema va más allá del modelado de sistemas y la programación del mismo. Las etapas que tiene el desarrollo de una simulación son las siguientes:

- **Definición del problema:** En esta etapa debe definirse completamente la especificación del problema que se quiere modelar. Esta etapa es crucial pues todo lo que se relevará y construirá depende de una correcta definición del modelo.

- **Obtención de los datos:** Los datos de entrada de una simulación son extremo importantes para la experimentación, de otro modo no se puede garantizar que se esté experimentando con el sistema correcto. Esta etapa no solo es crucial sino que generalmente es una de las etapas más difíciles y complicadas de todo el proceso
- **Modelado del problema:** Luego de especificar el problema y relevar los datos que son necesarios, se está en condiciones de crear un modelo y codificarlo. Es aquí cuando el modelador recurre a las diversas herramientas de modelado de sistemas que tiene a la mano. El proyecto SimPP se encuentra focalizado en este punto particular del proceso de simulación.
- **Verificación y validación del modelo:** Una vez que se tiene el modelo del sistema implementado, se pasa a la verificación y validación del mismo. Estos conceptos tienen el mismo significado que en Ingeniería de Software, verificación consiste en controlar si se está construyendo el producto correctamente; mientras que la validación indica si se está construyendo el producto correcto. En simulación la validación es una actividad crítica donde se ve si el modelo construido es coherente con el sistema real. De otro modo no tiene sentido experimentar con este, pues los resultados recabados no tendrán ninguna validez.
- **Recolección de datos:** Durante esta etapa se crea el plan de prueba al que se someterá al modelo. Es importante generar suficientes casos de prueba con múltiples corridas de la simulación para tener resultados estadísticamente correctos. Recordemos que las simulaciones son generalmente procesos estocásticos.
- **Análisis de resultados:** Una vez que se han recolectado los datos a partir del plan de pruebas diseñado anteriormente se pasa a la última fase del proceso, el análisis de los resultados. Aquí es donde se interpretan los datos obtenidos y se trata de explicar los fenómenos registrados en las pruebas.

Paradigmas clásicos

Los paradigmas clásicos de simulación a eventos discretos son el enfoque de eventos y orientación a procesos. Ambos paradigmas manejan conceptos muy similares, pero difieren en la visión de mundo que proporcionan así como la implementación de los lenguajes que los soportan. Una visión del mundo es una forma de estructurar las operaciones del modelo siguiendo un enfoque particular [DO89].

Se denomina evento del sistema a aquellos instantes de tiempo que son relevantes para el sistema que se quiere modelar. Por ejemplo en el caso de un banco, la llegada de un cliente al banco y la salida del mismo.

Las entidades son aquellos objetos o individuos cuyas acciones dentro del sistema se quieren modelar. En el caso del banco las entidades relevantes son los clientes del banco. Las entidades poseen un ciclo de vida dentro del sistema, que son las acciones que realizan dentro del sistema desde que entran a éste hasta que se retiran del mismo. Dichas acciones son llamadas actividades.

Se dice que una simulación está modelada utilizando en el enfoque de eventos cuando el comportamiento del sistema se especifica mediante los eventos del sistema. Si en cambio se especifica al sistema desde el punto de vista del ciclo de vida de las entidades, nos encontramos ante una simulación orientada a procesos. Cada entidad del modelo, correspondiente a las entidades de la orientación a procesos, tiene asociada una función que describe su funcionamiento a lo largo de su ciclo de vida en el sistema. Esta función puede detenerse y reanudarse a medida que la entidad deba esperar por un recurso o la finalización de una actividad. Por mayor información sobre los paradigmas clásicos de simulación, vea el Apéndice A.

Breve reseña histórica

La historia de la simulación, así como la historia de la computación en general es bastante reciente. Según [BCNN00] la historia del software de simulación a eventos discretos puede dividirse en seis periodos. Una de las cosas interesantes de esta historia es que a lo largo de la misma se puede ver que aún hay nombres que han perdurado hasta nuestros días.

La historia del software de simulaciones vio la luz en 1955. En este primer período las simulaciones estaban construidas principalmente en FORTRAN u otros lenguajes de propósito general, sin el soporte de rutinas específicas para simulación. Un primer esfuerzo por unificar conceptos y rutinas específicas para este tipo de aplicaciones fue el libro The General Simulation Program de K.D. Tocher y D.G. Owen en 1960.

Durante el periodo de 1961 a 1965 nacen los gigantes de la simulación a eventos discretos que perduran hasta nuestros días. El primer lenguaje de programación de simulaciones (SPL) construido es GPSS, desarrollado en 1961 por Geoffrey Gordon en IBM. Este paquete basado en diagramas de bloques resultó ser fácil de usar y ganó gran popularidad. En 1963 nace SIMSCRIPT de la mano del premio Nobel Harry Markowitz. Fue desarrollado por la corporación RAND bajo el auspicio de la Fuerza Aérea de los Estados Unidos. Sus versiones iniciales estaban basadas en el enfoque de eventos. En esta época también surge GASP en 1961, desarrollado por Phillip J. Kiviat. Estaba basado en diagramas de flujo, similares a los bloques de GPSS.

Entre 1966 y 1970 se revisaron los conceptos que se tenían hasta el momento y se promovieron versiones más consistentes de las visiones de mundo que se soportaban hasta el momento. GPSS fue revisado y GPSS/360 entró en servicio a cargo de las computadoras IBM 360. SIMSCRIPT evolucionó a SIMPSCRIPT II, presentando grandes avances en lo que SPLs se refiere. Una de las características fundamentales, y que conserva hasta nuestros días, es su sintaxis extremadamente similar al idioma inglés.

Otro hecho interesante es la aparición de SIMULA. Este lenguaje sería el precursor tanto de los lenguajes de interacción a procesos modernos como de los lenguajes que manejan conceptos de clase y herencia, construcciones fundamentales de los lenguajes orientados a objetos actuales.

Para 1971, GPSS fue nuevamente actualizado pero esta vez los cambios vinieron desde fuera de IBM. Julian Reitman de Norden Systems lideró el equipo que desarrollo GPSS/NORDEN, primera versión de GPSS con un entorno de visualización interactiva en línea. En 1977 surge GPSS/H de la mano de O. Henriksen de Wolverrine Software. Esta versión fue liberada tanto para mainframes de IBM como para PC. Esta versión funcionaba entre 5 y 30 veces más rápido que el GPSS estándar y es la versión de GPSS más utilizada actualmente. En 1974, GASP sufre grandes cambios y GASP IV es liberado por Alan Pritsker de Purdue. También en este periodo se hizo un esfuerzo importante por simplificar el modelado de procesos. Se trató de crear modelos ejecutables a partir de un lenguaje de muy alto nivel. Pero este enfoque falló debido a que tenía severas limitaciones en cuanto a la generalidad con la que se podía modelar.

Entre 1979 y 1986 se ve un mayor desarrollo de herramientas de simulación para PC. Muchas de las herramientas más populares crearon versiones que fueran soportadas por múltiples procesadores, sin perder su estructura básica. Aparecen SLAM II y SIMAN, descendientes de GASP. SLAM, desarrollado por Pritsker and Associates soportaba múltiples perspectivas de modelado. Por otro lado, SIMAN presenta un enfoque de modelado más general parecido a su antecesor GASP IV, agregando el modelado de bloques. Este fue el primer SPL importante capaz de correr en IBM PC, bajo MS-DOS.

Desde 1987 hasta nuestros días se ha visto un gran crecimiento de los SPLs en computadores personales y de aplicaciones basadas en modelado gráfico y diagramas de bloques. La gran

mayoría de estos entornos proveen de muchas facilidades tanto para la creación de simulaciones con salida gráfica como para el manejo de datos de entrada y salida.

3.2. C++

C++ es un lenguaje de programación fuerte y estáticamente tipado que soporta una variedad de estilos de programación, con el objetivo de brindar una representación directa y de alto nivel de las ideas del programador. Cada característica del lenguaje está pensada para no introducir costos ocultos ni en tiempo de ejecución ni en espacio de memoria, para volver viable la implementación de cualquier tipo de sistema en el lenguaje. Actualmente es uno de los lenguajes más usados, y ha sido estandarizado por ISO.

Salvo por unas pocas excepciones, C++ conserva a C como un subconjunto. A lo largo del tiempo, esto ha sido tanto positivo como negativo. Existe cierta sinergia entre los dos lenguajes. Hay librerías que se escriben en un lenguaje y son más comúnmente usadas en el otro, en ambos sentidos. Probablemente C++ no hubiera obtenido el éxito del que goza sin haberse basado y mantenido compatible con C. Por otro lado, gran parte de los problemas de C++ se pueden rastrear a C.

Características

C++ soporta una variedad de paradigmas de programación: programación procedural, programación modular, abstracción de datos, programación orientada a objetos y programación genérica. La programación procedural se refiera a descomponer el problema en tareas a realizar. En este particular, C++ ofrece las facilidades más comunes, como son sentencias de control y llamadas a función. Modularizar comprende juntar las funciones con los tipos de datos con los que operan en partes de programa llamados módulos, de tal manera de que la definición de los tipos no trascienda los límites del módulo. Generalmente no es apropiado tener todos los datos encapsulados en una entidad estática como es un módulo, porque se tienen varias copias de los mismos, por ejemplo. La abstracción de datos permite la creación de tipos de datos a los que asociar operaciones.

Muchas veces no se tienen completamente definidas las propiedades de los tipos de datos. Existen dimensiones de características en común pero también de divergencia. Mediante jerarquía, es posible expresar las partes comunes en objetos base que luego son refinados sobre la dimensión de variabilidad mediante herencia. Esto se llama programación orientada a objetos.

Por último existen dimensiones de variabilidad que no se pueden capturar mediante herencia. Por ejemplo, un puntero inteligente a un cierto tipo de objeto tiene muchas cosas en común con uno a otro tipo. Estas diferencias se pueden capturar mediante programación genérica.

Otra característica distintiva de C++ es la destrucción determinista de los objetos. Así como los objetos tienen un constructor que establece sus invariantes, las cuales se mantienen hasta la invocación del destructor correspondiente. El destructor se llama cuando termina la vida de los objetos, y éste punto lo establecen en conjunto las reglas del lenguaje para objetos automáticos, y el programador para objetos dinámicos. Este punto es muy disputado, y es citado como una de las características principales con las que otros lenguajes mejoran sobre C++. Los usuarios de C++ tienen el manejo de esta complejidad aceptada, y existen multitud de herramientas para facilitar esta labor.

La ventaja al momento de diseño es que se tiene asegurada la destrucción de todos los objetos. Los lenguajes con recolección de basura facilitan enormemente el manejo de la memoria. Existen otros tipos de recursos, como archivos, ventanas y bloqueos sobre objetos de mutua-exclusión, que también exigen destrucción. Usualmente los recolectores de basura permiten registrar funciones de finalización para los objetos, pero no garantizan siquiera que

serán ejecutadas. La solución que se utiliza es que los objetos expongan un método que debe ser invocado por el usuario al momento de dejar de utilizar el objeto, o sea, un destructor. Si un cambio de diseño necesita la incorporación de un recurso que hay que destruir en otro objeto, este último ahora necesita también destructor. Hay que perseguir todos los usos del objeto y asegurarse de encontrar un punto donde destruirlo. La ventaja que tiene C++ es que todos los objetos ya soportan esta funcionalidad, sin elección. La destrucción determinista es un aspecto que luego es muy difícil de introducir.

Historia

C++ nació a principios de los ochenta en base a las necesidades de su creador, Bjarne Stroustrup. Él trabajaba en simulaciones a eventos discretos, pero ninguno de los lenguajes existentes le servía completamente. Simula tenía todas las facilidades necesarias, pero las implementaciones eran demasiado lentas, y C era rápido pero carecía de facilidades para estructurar adecuadamente programas grandes. Además, Stroustrup trabajaba en los laboratorios Bell, en New Jersey, el propio lugar de origen de C y Unix, y era compañero de trabajo de Dennis Ritchie y Brian Kernighan. Desde un principio C y C++ tuvieron un gran aporte mutuo. C++ está basado en versiones de C que datan de principios de los 1980, mientras que C actual se basa en C++ primitivo.

Las primeras mejoras sobre C fueron la introducción de clases, clases derivadas, control estricto de tipos, expansión en línea de funciones simples y argumentos con valores predeterminados. En 1983 se adoptó el nombre de C con clases para el nuevo lenguaje, pero luego se cambió por C++, porque se empezó a referir a C como el viejo C. Se agregaron funciones virtuales, sobrecarga de funciones, referencias, constantes, new y delete. En versiones posteriores aparecieron la herencia múltiple, las clases abstractas, los miembros estáticos, los templates, las excepciones y los namespaces. Los templates se basan en construcciones similares de Ada y Clu, y las excepciones se inspiraron en Ada, Clu y ML. Hacia finales de los 1980 el uso era tan extenso que se hizo necesaria la estandarización del lenguaje. La base del documento fueron el estándar de C recién ratificado y el libro The Annotated C++ Reference Manual.

El estándar se terminó en 1997. Desde ese tiempo, se viene trabajando en un nuevo estándar a ser terminado en esta década o la siguiente. En el 2003 se liberó un estándar con correcciones, pero ninguna adición. Los últimos tiempos han visto una erosión del uso de C++ en ciertas franjas, pero el lenguaje se ha establecido como uno de los lenguajes más usados profesionalmente. Al principio el énfasis en la comunidad de C++ se hizo sobre las técnicas orientadas a objetos. Actualmente la punta de lanza la tienen los componentes genéricos. Están en revisión mejoras del lenguaje en este sentido.

4. Estudio del estado del arte

El desarrollo de este proyecto implicó un gran trabajo de investigación. Para poder construir a SimPP se investigó en varias áreas. Era necesario encontrar patrones de arquitectura capaces de explotar todo el poder de C++. También era necesario relevar los simuladores existentes y evaluar su funcionalidad para tener una idea más acabada de dichas herramientas y de las características del software que se iba a construir. El conocimiento de diferentes paradigmas de simulación es imprescindible a la hora de construir una herramienta de simulación para definir el modelado de sistemas soportado.

4.1. Paradigmas de simulación

Cada paradigma da una diferente visión del mundo a la hora de modelar un sistema. Como se describió en el punto anterior hay dos paradigmas clásicos en simulación a eventos discretos: enfoque de eventos y orientación a procesos. También se estudió el Formalismo Devs, Grafo de eventos y Simulación paralela. Debido a que los enfoques clásicos se trataron en el marco conceptual de la simulación, no se volverán a describir en la presente sección. Para profundizar más sobre los paradigmas estudiados se recomienda leer el Apéndice A.

Formalismo Devs

El Formalismo Devs fue creado por Zeigler en 1976 [Wai03]. Es un esfuerzo por crear un modelo matemático para describir la simulación a eventos discretos. En Devs se presenta un enfoque de modelado modular y jerárquico. Otra de las nociones importantes que presenta este paradigma es la separación entre modelo y simulador.

Los modelos son las representaciones de los sistemas que se desean simular. Los modelos se comunican con el exterior por medio de un conjunto de puertos de entrada y salida. Esto define una interfaz del modelo con el exterior, permitiendo que los aspectos internos al modelo queden ocultos. De este modo es posible crear modelos a partir de otros cuyas interfaces son conocidas.

En Devs se definen dos tipos de modelos: atómicos y acoplados. Los modelos atómicos son modelos básicos, a partir de los cuales se pueden crear nuevos. La construcción de modelos se hace de acuerdo a mecanismos de acoplamiento. Estos mecanismos definen como debe ser la interacción entre los modelos componentes de uno acoplado. Se pueden construir modelos acoplados a partir de otros modelos, sean atómicos o acoplados. El conjunto de los modelos es cerrado bajo el acoplamiento.

Para simular sistemas en Devs es necesario tener un autómata capaz de correr cualquier tipo de modelo. Así como los modelos son jerárquicos, los simuladores de Devs también son jerárquicos. La idea es que se tiene un simulador por cada modelo componente de uno acoplado. La simulación de un modelo se lleva a cabo por medio de un protocolo de pasaje de mensajes entre todos los simuladores que integran la jerarquía. Se tiene un simulador en la cima de la jerarquía que es quien coordina la simulación.

Grafo de eventos

Los grafos de eventos ofrecen una notación de diseño para el modelado de sistemas. Esta notación fue introducida por Schruben en 1983 [Sch83]. Consiste en un multigrafo dirigido etiquetado, donde los vértices son los eventos y las aristas indican la relación de causalidad entre los eventos.

En los eventos se describen los cambios de estado que sufre el modelo durante dicho evento. Las aristas están etiquetadas con etiquetas compuestas. En la primera componente se tiene

una condición y en la segunda un valor de tiempo. Éstas indican que el pasaje de un evento a otro se hace efectivo en el tiempo indicado si se cumple la condición.

Este método es extremadamente útil para el modelado de sistemas, pues es independiente de la notación que se utilice.

Otra ventaja de esta notación es su extensibilidad. Es posible definir mecanismos para modelar eventos que reciban parámetros y para cancelar eventos.

También es posible definir mecanismos para generar nuevos modelos a partir de grafos de eventos que modelen subsistemas del sistema que se quiere modelar.

Simulación paralela

La simulación paralela surge como la aplicación de técnicas de paralelismo a los modelos de simulación. En aquellos sistemas que son muy complejos, para acortar el tiempo que se demora en ejecutar la simulación se paraleliza. Generalmente esto es posible pues en estos grandes sistemas no todas las entidades interactúan entre sí. Además es posible partir el modelo en submodelos similares más pequeños.

Sin embargo, esta migración no es tan sencilla. Debido a que se parte el modelo en submodelos independientes, cuando un submodelo interactúa con otro agendándole eventos se puede incurrir en problemas de coherencia temporal. Un ejemplo de esto es cuando un submodelo M agenda un evento, cuya ocurrencia es en el tiempo t, a otro submodelo M'. Pero M' debe ejecutarse en un tiempo de simulación t' > t. El problema es qué se hace con todos los eventos que transcurrieron entre t y t'.

Frente a esto se tienen dos enfoques. Uno de ellos, el enfoque conservador, evita esta situación. Para ello se bloquea la ejecución de algunos de los submodelos para sincronizar sus tiempos. Pero esto tiene problemas de eficiencia ya que no se paraleliza todo lo que se podría. Por otro lado se tiene un enfoque optimista, donde no se evitan los problemas de coherencia temporal sino que se brinda un mecanismo de retroceso cuando se llega a una incoherencia.

4.2. Simuladores existentes

A lo largo del proyecto se realizó un relevamiento de diversas herramientas de simulación que el equipo de simulación tenía a su alcance. A la hora de clasificar dichas herramientas, se buscó una clasificación que hiciera hincapié en las funcionalidades que estas brindaban a la hora de desarrollar modelos de simulación.

Existe una clasificación clásica de este tipo de software, la cual define tres categorías bien diferenciadas: lenguajes de programación de propósito general, lenguajes de simulación o extensiones a los lenguajes de propósito general, y entornos para el desarrollo de simulaciones. Esta clasificación separa el software de simulaciones por nivel e integración, pero no por decisiones de diseño, que es lo que nos interesa. Por tanto se definió otra clasificación basada en parte en la clasificación definida por [BCNN00].

Se clasificaron las herramientas según las siguientes características:

- **Modelado:** Es una característica esencial de toda herramienta. Determina la visión del mundo que se brinda al usuario.
- **Manejo de números aleatorios:** La gran mayoría de las simulaciones que modelan sistemas reales dependen de la generación de números aleatorios.
- **Manejo estadístico de datos:** Luego de ejecutar pruebas sobre el modelo, es necesario un tratamiento sobre los datos de salida que son de interés para el modelador.

- **Performance:** Mide la velocidad de ejecución y el consumo de recursos. Esta propiedad atañe no solo las herramientas de simulación sino a todas las aplicaciones informáticas en general.
- **Visualización en tiempo real:** En muchos sistemas, dependiendo del objetivo de la simulación, es importante ver en tiempo real el estado del sistema durante una corrida. Es una poderosa herramienta a la hora de validar un modelo.
- **Complejidades:** Esta característica engloba la flexibilidad de la herramienta hacia comportamientos complejos de los elementos del sistema.

Las herramientas que se eligieron para ser estudiadas fueron ProModel, Simpy, C++SIM, DesmoJ, EOSimulator, y Erlang. La elección de estas herramientas se hizo para intentar cubrir una muestra del software de simulaciones disponible que permita observar la variedad de soporte de estas características. Esta decisión está documentada de forma más profunda en el Apéndice B.

4.3. Técnicas de diseño e implementación

El proyecto comprendió el diseño e implementación de diversos componentes. No todos éstos tienen la misma naturaleza. Según si el componente es fuertemente orientado a objetos, genérico o funcional, la técnica varía. Para la visualización gráfica, se usó programación orientada a objetos y la técnica de compilation firewall para lograr aislar las dependencias externas. La creación y combinación de eventos que es tan fácil bajo SimPP se logra mediante la adopción del modelo de objetos funciones de la librería de C++, para la que existen herramientas muy poderosas. Se investigó también una técnica para la generación de clases configurables estáticamente según varios parámetros de comportamiento, el diseño basado en políticas. Por último, se abordó un modelo de arquitectura posible para una aplicación estratificada, GenVoca. En base a este modelo se elaboró un diseño que facilita la descomposición en aspectos.

Compilation Firewall

Esta técnica es un idioma de implementación para C++. El objetivo es evitar tener que compilar las clases que dependan de una en particular, cuando se cambian los miembros de la misma. Las funciones miembro no tienen problema porque se puede colocar su definición en el archivo de fuente. El problema es cuando se agrega, modifica o quita un miembro de la clase. Como este cambio afecta al archivo de inclusión para la clase, las otras clases que incluyan este archivo van a ser vueltas a compilar también.

La técnica se basa en que es posible dar una declaración de una clase pero no su definición. Entonces, en lugar de una lista cambiante de miembros se coloca uno solo que es un puntero a una clase que se define recién en el archivo fuente, pero en el archivo de cabecera permanece indefinido. El gran poder de esta técnica pasa porque no solo se independiza a las clases cliente de los cambios en la implementación de la clase a la que se le aplica compilation firewall, sino que como los miembros no se mencionan, no es necesario incluir sus archivos de cabecera tampoco.

Funciones de alto orden en C++

Los eventos acarrean un cambio de estado en el modelo. Usualmente este estado se restringe a una entidad en particular, pero tampoco se quiere limitar en este sentido, porque hay situaciones de entidades cooperantes y otras complicaciones. Lo más general es tener un objeto que encapsule totalmente la acción a realizar, el patrón Comando. El problema es que se vuelve engorrosa la creación de una clase particular para cada evento.

Algunos lenguajes soportan nativamente cálculo lambda de programación funcional para combatir este problema. Aunque no es el caso de C++, se puede emular con librerías, haciendo algunas concesiones en cuanto a sintaxis y performance. La implementación de una función que pueda guardar estado arbitrario, como es el caso de una función que se aplica sobre una entidad en particular y las dos deben ser agrupadas, se realiza mediante un objeto. La sobrecarga de operadores permite redefinir la aplicación, así que es posible invocar un objeto como si fuera una función. C++ permite que una función genérica reciba un objeto de un tipo desconocido y lo invoque, con lo que se logra polimorfismo. A su vez, con el mismo principio es posible escribir funciones de alto orden, por ejemplo para aplicar parcialmente los argumentos a una función.

Existen una cantidad de componentes para la creación de objetos función. La librería estándar de C++ ofrece objetos para las operaciones aritméticas y combinadores para fijar uno de los argumentos de una función, bind1st y bind2nd. También existe una función para convertir una función miembro n-aria en un objeto a invocar con n argumentos más uno, el propio objeto. Boost, un conjunto de librerías introducido en el Apéndice E, tiene la librería Boost.Bind que permite fijar varios argumentos al mismo tiempo y dar vuelta el orden de los mismos, etc. Boost.Lambda va más lejos y sobrecarga el significado de los operadores para convertir _1 + _2 + _3 en la suma ternaria.

Diseño basado en políticas

Generalmente hay varias opciones para la implementación de una clase, y dependiendo de cual se elija se termina con una clase diferente. No hay una opción que sea en todo sentido mejor que otra, porque la valoración depende del uso que se le quiera dar.

La solución puede ser realizar todas las variaciones, separando en una clase base los aspectos comunes, pero si hay varias dimensiones de variabilidad esto ya no funciona. Otra opción es tener una clase base abstracta por cada punto donde hay opciones, e implementaciones para cada opción, de acuerdo al patrón Estrategia. Una factoría puede encargarse de encapsular la configuración apropiada a una situación en particular. El problema de este enfoque es que no se puede variar la interfaz de la clase original de acuerdo a las estrategias elegidas, se paga un costo en tiempo de ejecución y en espacio por la indirección de las estrategias y el sistema de tipos no distingue las distintas variedades de una clase.

Un diseño basado en políticas también separa cada política en una clase, pero las mismas se eligen y combinan en tiempo de compilación, así que no hay un costo mayor, lo que permite una granularidad pequeña. La clase recibe las políticas elegidas como parámetros template, y hereda de las mismas, lo que permite a las políticas agregarse a la interfaz de la clase principal.

GenVoca

GenVoca es una metodología de diseño estratificado. La idea clave es que cada capa depende solo de la interfaz de la capa inmediata inferior, de tal manera que un sistema concreto se puede crear eligiendo sus capas. Cada capa entonces es en realidad una meta función que convierte un sistema en otro sistema más complejo, y el tipo del argumento y retorno de estas funciones determina si una ordenación de capas es válida o no. Las capas simétricas son las que su función tiene igual tipo de entrada y salida, y permiten la existencia de una variedad de sistemas de un mismo tipo.

Cada capa tiene una colaboración entre los objetos del sistema. Así, es posible construir las colaboraciones independientemente, y luego generar un sistema concreto con las colaboraciones necesarias. Entonces GenVoca permite expresar separadamente cada uno de

los roles de los objetos, sin tener que mezclarlos en un solo objeto, lo que sucede automáticamente.

En C++ se puede expresar un sistema diseñado con GenVoca mediante la técnica de capas mixin. Una capa mixin es una clase template que contiene otras clases, que recibe un tipo del que hereda y a su vez todas sus clases heredan de la misma clase pero de la capa argumento.

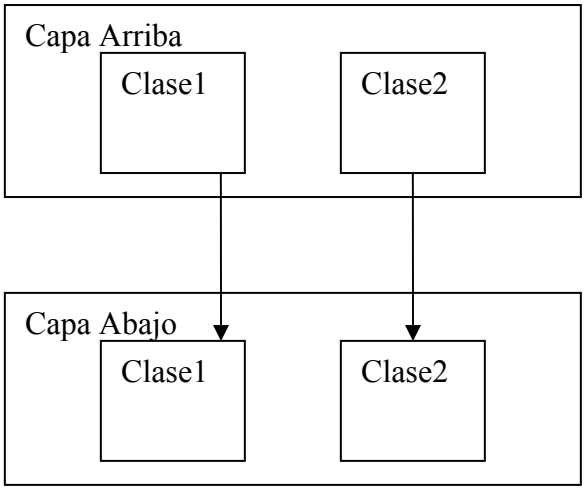


Figura 1: Capas mixin

Una capa no template, que contiene implementaciones concretas pero posiblemente vacías de todas las clases es la capa de más abajo. Luego, se puede usar el propio mecanismo de templates para instanciar un sistema de acuerdo a las capas deseadas. Una simple expresión de tipo enumerando en orden las capas y terminando en la capa base representa el sistema, y la configuración puede ser almacenada como un simple typedef.

Se desarrolló, basado en estas ideas, un enfoque de diseño bautizado Gestalt. El mismo permite que las capas sean intercambiables y todo el sistema sea accedido desde cualquier punto. Para presentarlo se describe la idea de diseño y la implementación en C++.

Por más información sobre las técnicas presentadas en este punto, ver el Apéndice C.

5. Requerimientos de la librería SimPP

La etapa de relevamiento de requerimientos fue una fase de mucha incertidumbre. Hasta avanzado el proyecto y pasada la fase de investigación del estado del arte no se tenían claros los requerimientos esperados de la librería, pero con las sucesivas iteraciones se llegó a completar un alcance de la misma:

- **Permitir el modelado de simulaciones a eventos discretos:** el área de aplicación de la librería, definido desde el comienzo del proyecto, es el modelado de las simulaciones descritas en los puntos anteriores.
- **Utilizar C++ como lenguaje de desarrollo:** la librería debe proveer las facilidades para que el usuario modele su sistema utilizando este lenguaje de programación, originalmente creado para esta tarea. Se pueden aprovechar todas las facilidades que el lenguaje brinde, tanto en diseño como en implementación. Este requerimiento también fue definido desde el comienzo del proyecto.
- **Componentes desacoplados:** los diferentes componentes de la librería deben ser capaces de trabajar lo más independientemente posible, a modo de aumentar la modularidad y la facilidad de mantenimiento, expansión y reemplazo de componentes individuales.
- **Paradigmas de modelado:** se requiere un soporte mínimo de paradigmas clásicos de modelado, y como requerimiento opcional puede implementarse la interacción entre ellos y con otros paradigmas no convencionales.
- **Manejo de datos:** una vez obtenidos datos de salida de una simulación, la librería debe proveer mecanismos para el tratamiento de los datos, siendo un requerimiento mínimo los contenedores de datos. Requerimientos opcionales incluyen reportes y ajustes estadísticos.
- **Performance:** la eficiencia de los sistemas modelados por la librería no debe ser limitada por el funcionamiento de los componentes ni por los métodos estándar de modelado.
- **Facilidad de uso:** la librería debe ser orientada a modeladores con conocimientos básicos de C++. Estos conocimientos incluyen herencia, polimorfismo, sobrecarga de operaciones y templates.
- **Visualización gráfica:** se deben proveer mecanismos de visualización gráfica en tiempo real, durante o luego de la simulación. La preferencia es una visualización simultánea, debido a la capacidad de alterar la simulación durante su funcionamiento y observar los cambios.
- **Persistencia:** se debe proveer algún sistema de persistencia de un modelo durante una simulación. Esta persistencia puede ser en disco o en memoria, siempre permitiendo diferentes replicaciones de una simulación desde un estado en particular.
- **Portabilidad:** la librería deberá ser portable entre varios sistemas operativos y compiladores, soportando al menos Windows y Linux como sistemas operativos y Visual Studio y G++ como compiladores.

6. Diseño de la librería SimPP

Teniendo en cuenta los requerimientos de la librería y las técnicas estudiadas, SimPP fue diseñado como un conjunto de componentes débilmente conexos para modelado e implementación de sistemas, frente a la opción de elaborar un framework para el desarrollo de simulaciones. Este diseño permite una mayor flexibilidad a la hora de expandir o refinar las herramientas utilizadas, aunque requiere algo más de experiencia en C++ para el modelador. La otra razón para escoger este diseño es que, dada la aplicación, el modelador elabora un modelo que conecta las herramientas como desee.

Cada una de las herramientas débilmente acopladas se relaciona con la otra mediante conceptos. Un concepto es un conjunto de operaciones que un objeto debe cumplir para pertenecer al concepto. La diferencia con una interfaz es que no es necesario heredar del concepto. Así que, por medio de mecanismos de templates que provee C++, si una clase requiere un objeto que cumple algún concepto, puede recibir cualquier objeto de la librería (o del usuario) para satisfacerlo.

SimPP trabaja sobre librerías externas para realizar varias tareas no relacionadas con el manejo de la simulación, como la elección de números aleatorios y la visualización gráfica. Se decidió hacer la librería lo más flexible posible ante los estilos de programación y las preferencias de librería de los modeladores.

6.1. Conceptos

Las distintas herramientas exigen algunos requisitos sobre las clases que reciben. Estos requisitos son denominados conceptos, y contienen un conjunto de operaciones que las clases utilizadas deben proveer en su interfaz. De utilizar una clase que no cumple con los conceptos exigidos surgirán errores en tiempo de compilación. Los conceptos son más flexibles que la herencia porque se pueden utilizar clases del usuario y librerías externas que no hay que modificar para que hereden de clases provistas por la librería.

- **Concepto function:** Este concepto es cumplido por los functores nullarios que retornan `void`, es decir aquellos objetos que pueden invocarse como una función y no devuelven un valor tras la ejecución. Utiliza la operación `operator()` provista por C++
- **Concepto timeFunction:** El concepto timeFunction es similar al function, sólo que su invocación retorna un `double` en vez de `void`.
- **Concepto schedulerRunnable:** Este concepto representa aquellas clases calendario que proveen operaciones que un ejecutivo de calendario puede usar para correr. Se supone que esta clase contiene una lista de eventos ordenados por fecha de ejecución y varias operaciones para tratarlos.
- **Concepto schedulerPushable:** Este concepto es el simétrico del schedulerRunnable, representa las clases a las cuales se les puede agregar eventos utilizando objetos que cumplan con el concepto function.
- **Concepto priorityQueue:** Este es un concepto satisfecho por las clases que proveen mecanismos de cola de prioridad para cierto elemento element_type que tiene una función de ordenación.
- **Concepto conjuntoValores:** Este concepto es cumplido por conjuntos de pares de valores abscisa-ordenada. Las clases que lo cumplen proveen operaciones para acceder a sus elementos por medio de iteradores. Los elementos iterados cumplen con el concepto valor.
- **Concepto valor:** Este concepto denota un elemento con valores de abscisa-ordenada.

6.2. Herramientas

A continuación se describen las diferentes herramientas que componen la librería. Estas herramientas cumplen las decisiones de diseño de interacción por conceptos.

- **Scheduler:** Se provee un calendario básico para eventos a dos fases que sirve de base para comportamientos más complejos y cumple con los conceptos schedulerRunnable y schedulerPushable. El mismo recibe elementos del concepto function y los agenda para ser ejecutados en un determinado lapso. Provee operaciones para ejecutar el evento más temprano y controlar el tiempo simulado.
- **Executive:** Executive provee de ejecutivos capaces de correr simulaciones utilizando enfoques de eventos a dos y tres fases. Su expresividad permite utilizarlos también para ejecutar otros paradigmas, siempre que el objeto a correr cumpla con el concepto schedulerRunnable.
- **Process / Hold:** Estas herramientas permiten trabajar con un paradigma orientado a interacción de procesos. Si bien Process puede trabajar independientemente, Hold completa el comportamiento clásico de los modelos en este paradigma, ofreciendo una función que suspende un proceso por un determinado tiempo de simulación. La base del mecanismo es el ejecutivo de eventos a dos o tres fases, por lo que es posible combinar ambos paradigmas de modelado. Process ofrece las estructuras y funciones necesarias para crear, activar y suspender procesos.
- **Feeder:** En un modelo de simulación suelen establecerse eventos de entrada de entidades al sistema. Ya sean clientes que llegan a un local o piezas que llegan a un depósito, el mecanismo es similar: Se agenda un evento que efectúa la entrada y luego se vuelve a agendar para un tiempo determinado, que es el intervalo de tiempo hasta la próxima entrada. SimPP ofrece la herramienta Feeder para automatizar la creación y agendado de este tipo de eventos, en base a la función que debe ser ejecutada para dar entrada y una función que devuelva el intervalo de tiempo hasta la próxima entrada.
- **Display:** Display es la herramienta que provee la visualización gráfica de la simulación. Su interfaz fue diseñada para satisfacer de forma simple los requerimientos más comunes de la visualización de simulaciones. Provee de una ventana con un entorno 2D que presenta el entorno del modelo, y por medio de objetos se colocan los elementos de la simulación. Estos elementos, tanto gráficos como texto, pueden ser creados y desplazados por la ventana, de modo que se muevan de acuerdo al tiempo de simulación. La permanencia de un elemento en el entorno depende de la existencia de referencias al elemento en el modelo.
- **Histogram / Stats:** Aunque son dos herramientas separadas que pueden utilizarse independientemente y no tienen ninguna relación, estas herramientas suelen utilizarse en conjunto por su propósito y papel en el modelado de simulaciones: el análisis de los datos de salida.
- **Ticket:** Ticket es una herramienta que permite modelar algunos comportamientos complejos a la hora de manejar eventos en un scheduler. Provee un objeto que agenda un evento pero sigue teniendo control sobre el, pudiendo cancelar su ejecución. Esto permite crear eventos fijos condicionados, es decir, que corren en determinado momento a no ser que suceda algo en particular. Este algo en particular es lo que cancela el evento a través del ticket. Esto permite modelar reneging, accidentes, etc.
- **Serializable Scheduler:** Se provee de esta herramienta para poder almacenar el estado de una simulación y continuarlo luego. Esto es un calendario alternativo capaz de serializar sus eventos, utilizando la librería Boost.Serialization. Impone ciertos límites al modelado y

no permite la utilización de muchas herramientas útiles, pues no cumple con el concepto schedulerPushable.

7. Un nuevo enfoque de modelado

En base a las técnicas del apéndice D de técnicas de diseño e implementación, la sección de GenVoca, se diseñó una metodología para la elaboración de modelos de simulación. Esta técnica se utilizó para el modelado del caso de estudio, con varias alteraciones debido a que se realizó durante la creación de la metodología.

Para modelar sistemas según este enfoque se divide el sistema en aspectos, cada uno implementado mediante una capa Gestalt. Se distinguen tres aspectos fundamentales de todo modelo de simulación:

- **Base:** este aspecto contiene el estado estático de la simulación. Representa, para cada entidad y recurso, el estado del mismo en un instante de la simulación. Almacena además la colección de entidades y recursos del modelo.
- **Tiempo:** este aspecto contiene el tratamiento del tiempo para cada operación del modelo. Maneja el scheduler de la simulación y hace que la clase Model (descrita luego) cumpla con el concepto schedulerRunnable.
- **Modelo:** contiene el comportamiento de cada una de las entidades y recursos del sistema. Aquí se encuentran todos los eventos y procesos del sistema, accediendo a los datos de Base y a las funciones de tiempo de Tiempo.

La metodología se basa en dividir el modelo de simulación en estas capas. En la documentación del usuario de SimPP se explica cómo elaborar esta división. También se pueden utilizar las siguientes capas adicionales, o aquellas que el usuario considere apropiado:

- **Salida:** Esta capa provee salida al usuario durante la simulación, ya sea visualización gráfica, tracer en consola o un log en disco. Se recomienda hacer una capa Salida para cada tipo de salida, de esta manera se puede seleccionar que salida ofrecer desde un sólo punto de acceso, al crear el Gestalt.
- **Estadística:** Esta capa recoge los datos surgidos durante la simulación y ofrece facilidades para obtener y analizar estos datos. Esto se hace mediante acumuladores de datos que van interceptando llamadas de funciones y acumulando los datos. En cualquier punto de la simulación, generalmente luego de una corrida, se puede obtener esta información mediante operaciones también provistas por la capa.
- **Control:** Esta capa utiliza el `step()` de la clase Model para ofrecer un punto de control accedido cíclicamente durante toda la simulación. En este punto se puede utilizar una librería externa para obtener el estado de la entrada del usuario por teclado, mouse, joystick, etc. En base a este estado pueden hacerse operaciones como modificar la visualización, crear entidades, aumentar tasas o lo que se necesite. Quizá sea necesario aumentar la interfaz ofrecida por las clases para soportar estos cambios de comportamiento.

8. Caso de estudio

A modo de estudiar la utilidad de cada una de las herramientas implementadas y los métodos de uso y demostrar la aplicabilidad de la librería para el uso requerido, se realizó el modelado del sistema de transporte público de la ciudad de Rivera utilizando SimPP.

En este informe se da una descripción y la motivación por este sistema en particular. El desarrollo de este caso de estudio se da en el Apéndice F.

8.1. Descripción de la realidad a modelar

Consiste en una serie de líneas cuyos recorridos cubren la ciudad. Dichos recorridos son una secuencia de paradas distribuidas a lo largo de toda la ciudad, pudiendo ser circulares o de ida y vuelta. Como en cualquier otro sistema de este tipo las paradas son visitadas por varias líneas.

Otro componente del sistema son los ómnibus. Estos tienen que seguir el recorrido definido por la línea a la que han sido asignados.

Finalmente están los pasajeros. Estos llegan a una parada y esperan llegar a su destino. En la parada esperan hasta que llegue un ómnibus capaz de llevarlos a su destino. Cabe destacar que este ómnibus no necesariamente es el primero que llegue a la parada, esto depende de las preferencias de cada pasajero. Luego de que el pasajero se sube a un ómnibus de su preferencia, este baja en la parada más cercana a su destino.

8.2. Motivación

Hubo varias razones para elegir este sistema a modelar:

- La realidad a modelar es bien conocida por los tutores del proyecto. Por lo tanto pueden actuar como clientes a la hora de recabar requerimientos (clientes técnicos, lo cual es mucho mejor) y suministrarnos los datos de entrada al problema, tanto los estadísticos como los determinados para el funcionamiento.
- El sistema en cuestión contiene elementos clásicos a los sistemas usualmente modelados por herramientas similares. Colas de espera, actividades con duración, recursos, estado estacionario, etc. Probar el modelado de estas características es clave en este estudio.
- Se trata de un caso real con datos reales, no un ejercicio teórico. Esto hace que sea posible comparar los resultados con la realidad.

9. Conclusiones

Al llegar a la culminación de este proyecto de grado, debemos reflexionar sobre el camino que hemos tomado. Se logró completar los objetivos que se trazaron al principio del proyecto. Más concretamente:

- Se logró hallar una clasificación de herramientas de simulación diferente a la clasificación clásica. Esta clasificación es más específica a nuestro interés que las categorías clásicas, pues clasifica de acuerdo al soporte de funcionalidades, sin considerar la interfaz que el software brinda al usuario.
- Se logró implementar una librería de simulación a eventos discretos en base a componentes desacoplados, capaz de soportar modelos diseñados en base a varios paradigmas, y con soporte de todas las funcionalidades básicas y algunas avanzadas. El hecho de que la librería sea desacoplada brinda una gran flexibilidad a la hora de utilizarla, pues solo se usan los componentes que se requieran en lugar de la librería en su conjunto. Además, soportar la coexistencia de múltiples paradigmas en forma pura no es una característica común, y ha podido ser alcanzada. Otro punto importante es que se llegó a soportar persistencia de un modelo en ejecución, lo que permite poder crear estados preparados del modelo para luego poder ejecutar múltiples corridas partiendo de un mismo estado inicial. La librería es portable a Windows y Linux, utilizando los compiladores VC7 (Microsoft) y G++ (GNU).
- Mediante la implementación del caso de estudio elegido, se pudo comprobar que es factible la utilización de la librería para el modelado de sistemas reales. Esta prueba no es menor, nos da la pauta de que es un producto que tiene utilidad fuera del ámbito meramente académico.
- Otro hito importante es la definición de una arquitectura orientada a aspectos para el modelado de sistemas. Este enfoque permite que los modelos creados sean más escalables y mantenibles. Además da una nueva cara a la implementación de modelos de simulación, cuya arquitectura es generalmente monolítica.

En el área de desarrollo de software, pudimos constatar algunos resultados importantes relacionados con la elección de las herramientas seleccionadas:

- Se constató un impacto positivo en la productividad del equipo mediante la utilización de C++ y las librerías auxiliares que fueron elegidas, especialmente Boost. Este resultado dice dos cosas, que una sabia elección de las herramientas a la hora de implementar una aplicación es fundamental y que Boost es una librería multipropósito que vale la pena considerar. Está compuesta por varias sublibrerías que pueden ser extremadamente útiles.
- También pudimos ver que C++ sigue siendo útil para el contexto en que fue creado en un principio. Es una herramienta que no ha perdido vigencia para el desarrollo de software para simulaciones.

Finalmente, relevando la investigación que se ha realizado tanto en las diversas áreas, podemos arribar a algunas conclusiones que en un principio no parecían tan evidentes:

- A lo largo de la investigación sobre algunos de los paradigmas de simulación, pudimos ver que hay más de los que se podría pensar. Y quizás solo hayamos llegado a la punta del iceberg. El problema que tienen estos paradigmas es que no están muy difundidos, en parte porque no existen muchas herramientas que los soporten, cosa que no ocurre con los paradigmas clásicos.
- Se realizó un redescubrimiento de patrones de diseño que no son muy extendidos, por lo menos a este nivel. Sin embargo estos patrones resultaron ser muy útiles, brindando algunas funcionalidades que no son tan fáciles de lograr con los patrones tradicionales.

Esto nos enseña una vez más que no es necesario reinventar la rueda, es probable que ya alguien se haya tomado el trabajo de hacerlo, es solo cuestión de buscar con el suficiente criterio y en el lugar correcto.

10. Trabajo futuro

Se pueden clasificar las posibles direcciones futuras en tres categorías. La primera son las mejoras sobre la propia librería de simulación, ya sea sobre componentes existentes o la creación de componentes nuevos. Segundo, se sugieren mejoras sobre la implementación del caso de estudio. En tercer lugar hay posibles escenarios donde probar las metodologías de diseño e implementación estudiadas.

10.1. Librería

Se pueden agregar nuevas implementaciones del componente scheduler. En su tesis de maestría, Leslie Murray [Mur03] presenta una gran variedad de implementaciones, que serían claras candidatas para formar parte de la librería, ampliando las opciones de los usuarios. Es posible crear variaciones sobre el calendario cambiando solamente la cola de prioridad subyacente. La implementación predeterminada usa la implementación de montículo de prioridad que pertenece a la librería estándar de C++, pero es posible remplazar este componente por cualquier otro que tenga la misma interfaz.

Si la aplicación tiene extremas necesidades de performance es posible hacer calendarios especializados que contemplen mejor esa realidad. Por ejemplo, si la duración de las actividades toma valores discretos, es mucho más eficiente un calendario con una lista encadenada y punteros a las posibles posiciones de inserción, que se lograría en tiempo constante. Un posible trabajo posible es identificar situaciones de esta índole y preparar componentes más adecuados a la realidad.

La salida gráfica podría ser más pulida, manteniendo su actual simpleza. Los diferentes elementos ahora se dibujan en orden de creación, pero podría implementarse que el usuario pudiera reordenar los elementos a discreción. A su vez, el movimiento desacelerado que se menciona en el apéndice de decisiones de diseño puede ser útil a algunos programas de simulación y no es difícil de lograr. También podría hacerse un caso especial del dibujado de situaciones comunes, como las colas de entidades, facilitando la implementación de muchos sistemas.

SimPP soporta actualmente los paradigmas de simulación orientados a eventos y a interacción de procesos. Descripción de procesos consistiría en la inclusión de tipos de recursos que suspendan el proceso automáticamente cuando no están disponibles. Idealmente los mismos recursos serían accesibles desde partes del modelo implementadas mediante otros paradigmas diferentes a procesos, lo que hay que considerar detenidamente. Grafos de eventos fue investigado porque su implementación se cree un reto accesible. Como se asemeja mucho a una máquina de estado, sería posible idear un lenguaje específico embebido en C++ tomando como modelo Boost.StateCharts o el caso de estudio de [AG04]. Agregar Devs sería otro camino posible a seguir.

Se podría aplicar un diseño basado en políticas a algunos componentes de la librería. Un primer candidato serían los histogramas, que aunque simples tienen un sinfín de variaciones, y en la experiencia de los cursos de simulación son uno de los objetos que dan más problemas a los usuarios. Las variaciones en este componente podrían ser determinadas para crear las políticas adecuadas y luego poder ofrecer un extenso menú de opciones a los usuarios sin repetición en la librería. Sería muy útil tener histogramas integrados a los recursos o las colas, y que de este modo realicen las mediciones automáticamente. Esto también puede ser atacado mediante políticas.

El problema de la persistencia de los objetos función no es particular de la librería, y su solución tendría mucha utilidad en otros ámbitos. Posiblemente Boost.Serialization evolucione de una forma en que la situación pueda ser revista en otro momento, pero si no se podría

intentar llevar el acercamiento que no comprometía la facilidad del usuario hasta las últimas consecuencias. Aunque falle en algunos casos, mientras los parches necesarios sean suficientes, el usuario se ahorra mucho código.

También sería concebible implementar un tipo de archivo destino para Boost.Serialization que no convierta en bytes, sino que clonase toda la estructura en memoria. En Python esta es la manera por defecto de clonar un objeto, y posiblemente dé resultados también en C++. Los usos de este archivo trascienden la librería, y serían acogidos por la comunidad de C++ con agrado.

Cabe investigar con detenimiento cómo hacer para suspender la ejecución de un hilo de programa en algún sistema operativo en particular, para al menos tener claro hasta qué punto no es posible.

10.2. Caso de estudio

En primer lugar podría adaptarse el propio programa para servir no solo como medio investigativo sino para hacer comparaciones con variaciones en el comportamiento del modelo. Idealmente esto se podría realizar con la creación de una capa GenVoca nueva, sin afectar el resto de la aplicación. La experiencia con la introducción de la capa de visualización fue muy positiva, y se realizó sin inconvenientes. Probablemente este proyecto sea una buena manera de familiarizarse tanto con la librería como con el caso de estudio para alguien nuevo al trabajo realizado.

Segundo, se podría hacer una iteración nueva sobre la implementación del caso de estudio, haciendo "refactoring". Varios de los conceptos de implementación de GenVoca no quedaron claros hasta que se intentó aplicarlos, y una mirada fresca al problema con una perspectiva más armada de las herramientas probablemente produzca mejores resultados.

10.3. Metodologías de diseño e implementación

Aunque no atañen directamente a la simulación, las metodologías de diseño e implementación tuvieron un importante peso en el proyecto. Especialmente para GenVoca, surgió la inquietud de aplicar las mismas técnicas a dominios y lenguajes diferentes, en particular las que intersecaron en el tiempo al proyecto. Sería muy interesante tratar de aplicar GenVoca al diseño de otra clase de librería, por ejemplo, una librería de base para la construcción de aplicaciones multimedia como juegos. Ese dominio ofrece una gran riqueza que es difícil de capturar con un diseño puramente orientado a objetos, en la experiencia del grupo. También sería interesante tratar de idear una implementación para los conceptos de GenVoca en nuevos lenguajes. En particular, OOHaskell es una interesante mezcla entre la estaticidad de C++ y el dinamismo de lenguajes livianos como CLOS, donde GenVoca quizá podría florecer.

11. Referencias

[Mur03] MURRAY, Leslie. El calendario en la simulación a eventos discretos. Montevideo: InCo, PEDECIBA Informática, Tesis de Maestría, 2003. p.93. ISSN: 0797-6410

[Bstch05] Boost statecharts.

http://boost-sandbox.sourceforge.net/libs/statechart/doc/index.html Septiembre 2005

[AG04] ABRAHAMS, David; GURTOVOY, Aleksey. C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond. Boston:Addison-Wesley, November 2004. ISBN: 0-321-22725-5

[FBBOR99] Fowler, Martin; Beck, Kent; Brant, John; Opdyke, William; Roberts, Don. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999. p.464. ISBN: 0201485672

[BCNN00] BANKS, Jerry; CARSON, John S. II; NELSON, Barry L.; NICOL, David M. Discrete-event system simulation. Upper Saddle River, NJ: Prentice Hall, 2000. p.594. ISBN: 0130221021

[DO89] DAVIES, Ruth M.; O'KEEFE, Robert M. Simulation modelling with Pascal. Hertfordshire: Prentice Hall, 1989. p.302. ISBN: 0-13-811571-0

[Sch83] SCHRUBEN, Lee. Simulation modeling with event graphs. Communications of the ACM, November 1983, v.26, n.11, p.957-963,.

[Str97] STROUSTRUP, Bjarne. The C++ Programming Language (3rd Edition). Addison-Wesley, 1997. p.911. ISBN 0-201-88954-4

[Str94] STROUSTRUP, Bjarne. The Design and Evolution of C++. Addison-Wesley, 1994. p.480. ISBN 0-201-54330-3.

[Str05] Bjarne Stroustrup's Home Page

http://www.research.att.com/~bs/ Diciembre 2005

[WKCpp05] C plus plus - Wikipedia

http://en.wikipedia.org/wiki/C_Plus_Plus Diciembre 2005

[Wai03] WAINER, Gabriel A. Metodologías de modelización de y simulación de eventos discretos. Buenos Aires: Nueva Librería, 2003. p.380. ISBN: 987-1104-01-4

Apéndice A - Paradigmas de Simulación

1. Introducción

Una de las cosas más importantes a la hora de construir una herramienta de simulación es definir la visión del mundo que soporta, el paradigma de simulación que utilizará. Es vital estudiar tanto los paradigmas clásicos, así como otros paradigmas que hallan sido desarrollados ya que cada uno puede brindar una nueva idea de cómo construir nuestro simulador. Cada uno de los paradigmas estudiados tuvo un aporte en las decisiones de diseño que tomó el equipo a lo largo del desarrollo de SimPP. Solo se soportan dos de los paradigmas estudiados: enfoque de eventos y orientación a procesos que son los paradigmas clásicos de esta disciplina, pero se estudiaron otros que se decidió no soportar. Todos los paradigmas estudiados serán desarrollados en esta sección.

2. Enfoque de eventos

El enfoque a eventos es uno de los paradigmas clásicos de la simulación a eventos discretos [DO89]. Fue creado por K. D. Tocher en 1962. Es una forma de estructurar la simulación pero centrándose en los eventos más importantes del sistema. A grandes rasgos consiste en elegir solo los instantes importantes de un sistema y describir todas las acciones que se realizan en dichos instantes.

2.1. Descripción

Este paradigma se centra en los instantes de tiempo que son relevantes para el sistema a simular. Dichos instantes se llaman Eventos. En dichos eventos interactúan diferentes Entidades y Recursos que son relevantes para el sistema. Las entidades son aquellos objetos o individuos cuyo comportamiento es modelado. Cada entidad posee un Ciclo de Vida dentro del sistema, este ciclo comienza cuando la entidad ingresa al sistema y termina cuando lo abandona. Durante dicho ciclo las entidades realizan varias Actividades. Estas comienzan y finalizan con un evento. Los recursos en cambio restringen las actividades de las entidades. Otro elemento importante son las Colas, donde las entidades esperan a que haya suficientes recursos disponibles para que puedan continuar con su ciclo de vida.

2.2. Eventos

Los *eventos* son los instantes de tiempo relevantes al sistema. En estos instantes el sistema cambia de estado. El estado en un sistema está dado por la cantidad de recursos, la cantidad de entidades, el tamaño de las colas, entre otros. En diversos eventos del sistema las entidades abandonan o ingresan a diferentes colas, adquieren o liberan recursos, etc.

Los eventos pueden ser de dos tipos: *Fijos* o *Condicionados*. Un evento es fijo si se conoce exactamente cuando este tendrá lugar. En cambio, un evento es condicionado si no se conoce cuando ocurrirá. Los eventos condicionados ayudan a tener un sistema más modular pues en dichos eventos controlan globalmente el inicio de actividades de las entidades. Esto se da principalmente pues las actividades de las entidades comienzan con eventos condicionados y finalizan con eventos fijos. En el caso de no tener eventos condicionados, el chequeo de la condiciones inicio de actividades se realizan en múltiples eventos fijos, aquellos cuya ejecución significa la liberación de algún recurso necesitado.

2.3. Ejecutivos

Se dice que una simulación está estructurada en *dos fases* (event method) si solo se modela con eventos fijos, y en *tres fases* (three phase approach) si utiliza eventos condicionados [DO89]. El tipo de estructuración de la simulación altera el funcionamiento del *Ejecutivo* de la simulación, que es el algoritmo que corre la simulación.

El ejecutivo para una simulación en dos fases realiza dos operaciones: primero se avanza el tiempo de la simulación hasta el tiempo del evento fijo que está más próximo a ocurrir (fase A), luego se procesan todos los eventos fijos que estén agendados para ocurrir en el tiempo actual (fase B). Este esquema no contempla los eventos condicionados, por lo tanto estos deben ser implementados en los propios eventos fijos. Se aprovecha que, como los eventos fijos saben qué actividades pueden comenzar a partir de su finalización, deben implementar solo el subconjunto de eventos condicionados que pueden suceder. Tienen también la flexibilidad de seguir ejecutando luego de estos eventos condicionados, lo que puede permitir comportamientos particulares.

En cambio, un ejecutivo de tres fases realiza la fase A, la fase B y luego la fase condicional o C. Luego de la fase B, se ejecutan todos los eventos condicionados del sistema. Estos

chequean todas las condiciones que hacen cambiar de estado al sistema. El esquema de simulación en tres fases puede verse como una optimización de la vista de mundo de escaneo de actividades. En esta última se realiza continuamente una verificación sobre todas las actividades posibles, ejecutándose las que tienen cumplidas sus precondiciones. Cada vez que una actividad es ejecutada debe empezarse nuevamente la verificación sobre todas las otras actividades, porque alguna pudo haberse destrabado [Ray93].

La gran ventaja de la tercera fase es que se tiene un punto central donde asignar prioridades y coordinar las interacciones entre las tareas. En esquemas como interacción de procesos o eventos de dos fases, se deben tomar decisiones locales, y luego revisar todos estos lugares cuando se hace mantenimiento del software.

2.4. Estados de las entidades

Las entidades pueden tener tres estados: *ocupado*, *libre* y *esperando*. Una entidad está ocupada si se encuentra realizando una actividad. Esta esperando cuando se encuentra en una cola aguardando por que se cumplan las condiciones necesarias para realizar una actividad. Finalmente una actividad está libre si no está ocupada o esperando.

Los estados de las entidades están muy relacionados con la cola en la que se encuentran. Si se encuentran en el calendario entonces están realizando una actividad o están esperando por un evento fijo y su estado es ocupado. En cambio si están en otra cola del sistema, generalmente su estado es esperando.

3. Orientación a procesos

El paradigma de orientación a procesos define otra forma de ver al mundo a la hora de estructurar un sistema, centrada en las entidades del sistema. La simulación se estructura desde el punto de vista de las entidades, en contraposición al paradigma de eventos que estructura un sistema describiendo los instantes que son relevantes.

3.1. Descripción

Este paradigma define los mismos elementos que orientación a eventos. Son paradigmas iguales en cuanto a conceptos [Mit86]. Los conceptos de entidades, recursos, actividades y eventos son los mismos que fueron explicados para orientación a eventos. La única diferencia que existe entre ambos paradigmas es la forma en que estructuran la simulación.

En la orientación a procesos en lugar de describir un sistema a partir de sus eventos, se lo hace desde el ciclo de vida de las entidades. Este paradigma cambia el punto desde donde se ve el sistema. Al utilizar este punto de vista los eventos del sistema se encuentran codificados a lo largo de los ciclos de vida de todas las entidades del sistema.

El hecho de que se decida modelar el sistema a partir de las entidades que lo conforman hace que este paradigma resulte más intuitivo. Generalmente los modelos construidos mediante orientación a procesos son más compactos y fáciles de entender.

3.2. Métodos de estructuración

A lo largo de la vida de una entidad esta pasa por los estados de ocupada, esperando y libre. En el paradigma de eventos el pasaje entre estos estados se realizaba a través de varios eventos. Pero cuando se modela el ciclo de vida de la entidad, se hace necesario tener primitivas que permitan el cambio de estado. La forma en que se resuelven estas cuestiones nos definen dos tipos de simulaciones orientadas a procesos: descripción de procesos e interacción de procesos [DO89].

Interacción de procesos

En interacción de procesos las entidades interactúan entre ellas de forma explícita. Esto es las entidades cambian su estado o el de otras entidades en forma explícita. Las primitivas que se proveen clásicamente son *activate*, *passivate* y *hold*.

La primitiva *activate* se aplica sobre las entidades que se encuentran esperando. Esta primitiva no cambia de estado de las entidades, sino que deja que éstas puedan prepararse para luego pasar al estado de ocupado o libre. Típicamente esta primitiva es invocada por otra entidad que libera recursos y activa otra entidad para que los utilice. En cambio, la primitiva *hold* lleva una entidad al estado ocupado, o sea que les agenda un evento fijo visto desde la perspectiva de eventos. Finalmente la primitiva *passivate* lleva una entidad al estado esperando.

Esto hace que el ejecutivo de la simulación sea muy simple y prácticamente igual al de eventos en dos fases. Su única responsabilidad es avanzar el tiempo, y activar todas entidades que tiene agendado el fin de una actividad para el tiempo actual de la simulación. Como se puede ver este ejecutivo es muy sencillo de implementar y tiene fase A y B así como el ejecutivo de eventos en dos fases.

Descripción de procesos

En descripción de procesos las entidades cambian de estado de forma implícita. Los cambios de estado quedan determinados por las acciones que toma la entidad a lo largo del ciclo de vida.

El ciclo de vida de la entidad consiste en tomar recursos, y utilizarlos mientras dure la actividad que realiza. En caso de haber recursos suficientes, la entidad debería pasar al estado esperando, de otro modo define la duración de la actividad y pasa al estado ocupado. Por tanto los recursos deberán bloquear a la entidad. Luego de que los ha obtenido y que ha fijado la duración de la actividad, la entidad debe ser agendada en el calendario.

Como se ve el ejecutivo de estos modelos es muy similar al ejecutivo eventos en tres fases. El ejecutivo debe realizar la fase A y B, y luego debe chequear cuales son las entidades que pueden salir del estado esperando. Esta última fase es similar a la fase C, donde se procesan todos los eventos condicionados.

4. Formalismo Devs

El Formalismo Devs es un modelo matemático para las simulaciones a eventos discretos. Se separa la noción de modelo de la de simulador. Un modelo se utiliza para describir el comportamiento de un sistema. En cambio los simuladores son autómatas capaces de ejecutar cualquier modelo.

4.1. Descripción

El Formalismo Devs fue creado por Bernard Zeigler en 1976. Se trata de especificar el modelado de simulaciones a eventos discretos, que hasta ese momento estaba basado más que nada en detalles de implementación [Wai03]. Este formalismo define dos conceptos básicos: Modelos y Simuladores. Los modelos definen el comportamiento de un determinado sistema. Estos modelos pueden ser atómicos o acoplados. De este modo está definido un método de creación de modelos nuevos a partir de otros anteriores, dando un enfoque modular. Finalmente los simuladores son autómatas que se encargan de correr los modelos. Hay simuladores definidos para modelos atómicos y acoplados.

4.2. Modelos

Los modelos son representaciones de sistemas. Estos definen una interfaz con el exterior por medio de puertos, una función de transición interna, una función de transición externa, y una función de salida.

Como se menciona arriba, los modelos pueden ser de dos tipos: atómicos o acoplados.

4.3. Modelos atómicos

Los modelos atómicos se definen como:

M (I, X, S, Y, δ_{int}, δ_{ext}, λ, D)

Donde:

X es el conjunto de eventos de entrada;

Y es el conjunto de eventos de salida;

S es el conjunto de estados;

I es la interfaz del modelo, en ella se brindan puertos de entrada y salida para los eventos de X e Y;

δ_{int}: S → S es la función de transición interna;

λ: S → Y es la función de salida;

D: S → R^+ es la función de duración de un estado;

δ_{ext}: (S, t) x X → S es la función de transición externa, donde t se encuentra en [0, D(s)] siendo s el estado actual.

Los modelos tienen una interfaz con el mundo exterior, que está dada por I. Aquí se definen los puertos por donde se aceptan eventos de entrada y se producen eventos de salida. Además de los estados definidos en S, un modelo guarda dos variables de vital importancia, la fase y la σ. La fase representa el estado actual del modelo y σ se tiene al tiempo que se permanecerá en el estado actual. Siempre se ejecutan las transiciones internas, pero antes de ejecutar dicha transición se produce la salida usando λ. Dicha salida se envía como un evento de salida que sale por el puerto correspondiente.

En caso de ocurrir un evento externo, se produce un cambio de estado de acuerdo con el evento de entrada, el estado actual, y el tiempo de σ, se actualiza la fase y σ.

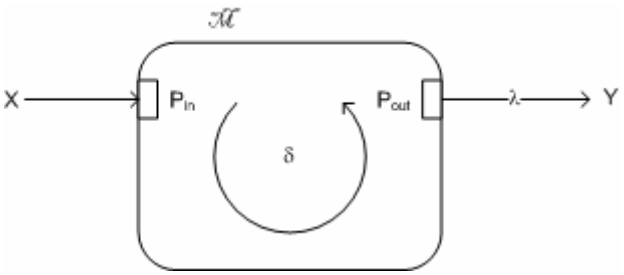


Figura A.1: Modelo atómico

4.4. Modelos acoplados

Los modelos acoplados se pueden formar a partir de acoplar modelos atómicos o acoplados para formar nuevos modelos. Estos se definen de este modo:

CM (I, X, Y, D, {M_i}, {I_i}, {Z_{ij}}, select)

Donde:

X es el conjunto de eventos de entrada;

Y es el conjunto de eventos de salida;

I es la interfaz del modelo, en ella se brindan puertos de entrada y salida para los eventos de X e Y;

D es el conjunto de índices de modelos componentes del modelo acoplado;

I_i es el conjunto de modelos componentes influenciados por M_i;

Z_{ij}: Y_i → X_j es la función de traducción de salida de M_i a M_j;

select: D → D es una función de selección de los eventos simultáneos

Básicamente en este tipo de modelo es una colección de modelos componentes interconectados. Para un cierto modelo componente M_i, este está conectado con todos los modelos cuyos índices se encuentran en I_i, y sus eventos de salida que ocurren previo a una transición interna, se traducen en eventos de entrada para el modelo M_j según Z_{ij}. La función select se utiliza para elegir cual de los componentes inminentes va a ejecutar su próximo

evento. Se entiende por componente inminente a aquel que tiene el menor tiempo para ejecutar su próximo evento.

De este modo pueden construirse modelos acoplados a partir de modelos atómicos. También es posible expresar un modelo acoplado como un modelo básico. Finalmente, tenemos un mecanismo jerárquico para construir modelos que es cerrado bajo el acoplamiento. Esta es una de las bondades del formalismo. Permite crear modelos partiendo de otros ya construidos. Los modelos están solamente acoplados con su interfaz y los eventos de entrada y salida que soportan.

4.5. Simulación

Es necesario construir ahora algún autómata capaz de correr simulaciones de los modelos. Queremos que nuestro autómata sea capaz de correr modelos en general, no modelos específicos. Para ello el tipo de simulador que se define esta caracterizado de la siguiente forma:

- **Simulador:** este componente se encarga de ejecutar los modelos atómicos. Cada modelo atómico tendrá su simulador asociado.
- **Coordinador:** este componente se encarga de controlar la ejecución de los modelos acoplados. Nuevamente se asocia a cada modelo acoplado un coordinador, este se encarga de dirigir a los simuladores de los modelos componentes.
- **Coordinador raíz:** este controla a los coordinadores de mayor nivel.

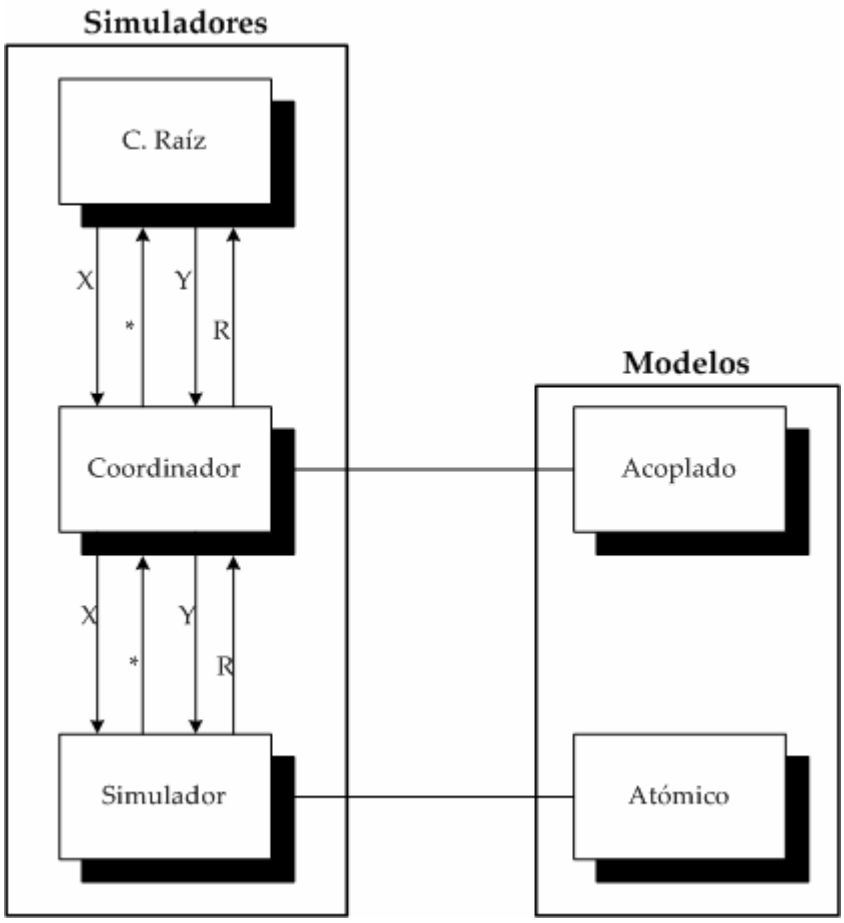


Figura A.2: Relaciones de simulación en Devs

La simulación en Devs se realiza por medio de pasaje de mensajes. Cada mensaje tiene información sobre la hora, el emisor, un contenido consistente con el puerto de destino, y el valor asociado. Hay cuatro tipos de mensaje:

X: indica la llegada de un evento externo. Incluye la hora global y viene del padre.

Y: contiene los valores resultantes de la transición externa.

*: indica la llegada del próximo evento interno.

listo (R): indica que un componente hijo terminó con su tarea.

Interacciones del coordinador

Un coordinador dirige la ejecución de un modelo acoplado. Recibe mensajes de sus componentes y actúa en consecuencia.

Si se recibe un mensaje X, se lo transmite a un hijo usando Z_{ij}.

Si llega un mensaje *, este es enviado al hijo inminente. La llegada de este mensaje indica que el próximo evento interno será llevado a cabo.

En el caso de los mensajes Y, se consulta si este mensaje debe ser transmitido al exterior (componente padre) o si debe ser transmitido a alguno de los otros componentes que forman al modelo acoplado. Un mensaje de listo, indica que un hijo ha terminado con su tarea. Cuando un coordinador haya recibido los mensajes listo de todas sus influencias o de todos sus receptores, calcula el nuevo hijo inminente, para enviarle el siguiente mensaje *.

Interacciones del simulador

En el caso de los simuladores, se deben tener varias variables para mantener el estado del simulador. Se mantiene la hora actual (t), la hora del próximo evento interno (tn), la hora del ultimo evento(tl), el tiempo transcurrido desde el último evento interno (σ), el modelo atómico asociado, y el coordinador padre.

En el caso de un mensaje X, primero se chequea si la hora que lleva el mensaje esta dentro de [tl, tn]. De cumplir con esa condición, se actualiza el tiempo transcurrido y se aplica la función de transición externa. Además se actualizan las horas locales de eventos (tn y tl). Finalmente se envía un mensaje de listo al nivel superior indicando que se ha terminado con la transición.

Si se recibe un mensaje *, chequea que su hora coincida con su tn, en tal caso aplica la función de salida. El resultado se inserta en un mensaje Y que se envía al coordinador padre. Luego se realiza la transición interna, que actualiza el estado, las horas de los eventos, y finalmente envía un mensaje listo al coordinador.

Funcionamiento general

El funcionamiento general de este autómata compuesto podría describir el comportamiento de una simulación. Esta comienza inicializando los estados de los modelos atómicos, determinando sus tn. Estos tiempos se propagan hacia arriba por medio de mensajes listo y se establece un camino de subcomponentes inminentes desde el modelo acoplado superior hasta un modelo atómico. Cuando el coordinador raíz recibe un mensaje de listo de su hijo, le envía un mensaje * con su tn. De este modo se comienza la simulación.

En lo sucesivo se siguen mensajes * hacia abajo, que originan mensajes Y que se traducen en mensajes X, culminando con mensajes listo. El último de estos mensajes es el que llega al coordinador raíz que es quien sigue con la siguiente iteración.

4.6. Conclusiones

El formalismo Devs trata de encarar la simulación de sistemas con un enfoque matemático, definiendo estructuras y autómatas que son tratados formalmente. Es de los primeros trabajos que intentan formalizar este campo [Wai03]. La simulación se ha basado principalmente en la implementación de herramientas y sistemas. En este formalismo se ven algunas características importantes:

- Se separa claramente los modelos de los autómatas que los ejecutan. Se provee de un simulador universal, capaz de ejecutar cualquier modelo Devs. Esto independiza los modelos de las estructuras y algoritmos que se encargan de ejecutarlos.
- Gracias al acoplamiento y las interfaces por puertos, se presenta un enfoque modular. De este modo es posible crear modelos nuevos a partir de modelos ya existentes. Esto trae conceptos como la reusabilidad sobre el tapete, planteando la idea de utilizar modelos que ya han sido utilizados con éxito para simular otros sistemas.

Por tanto, si bien se pueden tener discrepancias en cuanto a como definir los modelos y los simuladores, hay conceptos que son importantes que pueden dar ideas válidas a la hora de construir herramientas para simular sistemas.

5. Grafos de eventos

La vista basada en eventos es la más adecuada para simulaciones eficientes en computadora. Sin embargo, de las posibles opciones es también el arma más filosa. Para domar esta complejidad se necesita de una notación de diseño que permita una visión global de la interacción de los distintos eventos en la simulación de un sistema. Los grafos de eventos cumplen esta función.

5.1. Elementos básicos

Los grafos de eventos fueron introducidos en [Sch83]. Consisten de un grafo con los vértices y aristas etiquetados. Pueden haber aristas bucles, que vayan de un vértice a si mismo, y también varias aristas entre dos vértices. Así, el grafo es un multigrafo en realidad.

Los vértices representan los eventos. Los eventos están definidos por la transformación de estado que ocurre cuando se ejecutan. Se etiquetan con esta transformación.

Las aristas se corresponden con la relación de causalidad entre eventos. Si un evento acarrea otro, habrá una arista entre el primero y el segundo. Las aristas están dirigidas. Las aristas se etiquetan con la condición necesaria para que suceda el segundo evento.

Las condiciones de las aristas tienen dos componentes. La primera es tiempo que transcurre entre los eventos. En general será cero o una variable aleatoria. La segunda es un predicado sobre el estado del modelo que debe ser cierto al momento de suceder el primer evento. Solo si el predicado es cierto sucederá el segundo evento.

Una cualidad muy importante de este método es la total independencia con la notación que se use para etiquetar los vértices y aristas. Los procedimientos de los eventos, los tiempos, y predicados de las aristas pueden estar escritos en el nivel de detalle deseado por el modelador. De esta manera se puede evolucionar un modelo de seudocódigo hasta el lenguaje que será usado en la implementación sin abandonar los grafos de eventos.

Otra ventaja es que no se supone nada sobre la estructura del estado de simulación. Los procedimientos y predicados actúan sobre un estado que no está caracterizado de ninguna manera por la notación. Así, se puede usar la forma que le sea más natural al modelador. Otras notaciones acarrean restricciones en la estructura de los datos. Un ejemplo es el uso de canales o puertos en las metodologías que manejan mensajes, como Devs.

Veamos un ejemplo de un modelo expresado como un grafo de eventos. Sea un servidor que atiende una cola de clientes. Los clientes llegan con una separación de Tc y son atendidos en un tiempo Ts. Tc y Ts son variables aleatorias apropiadas. El estado del sistema será una bandera que marca si el servidor está ocupado. En general, si un tiempo, procedimiento o predicado es cero, no operation o true se suprime del diagrama, aunque en este caso se especificó todo.

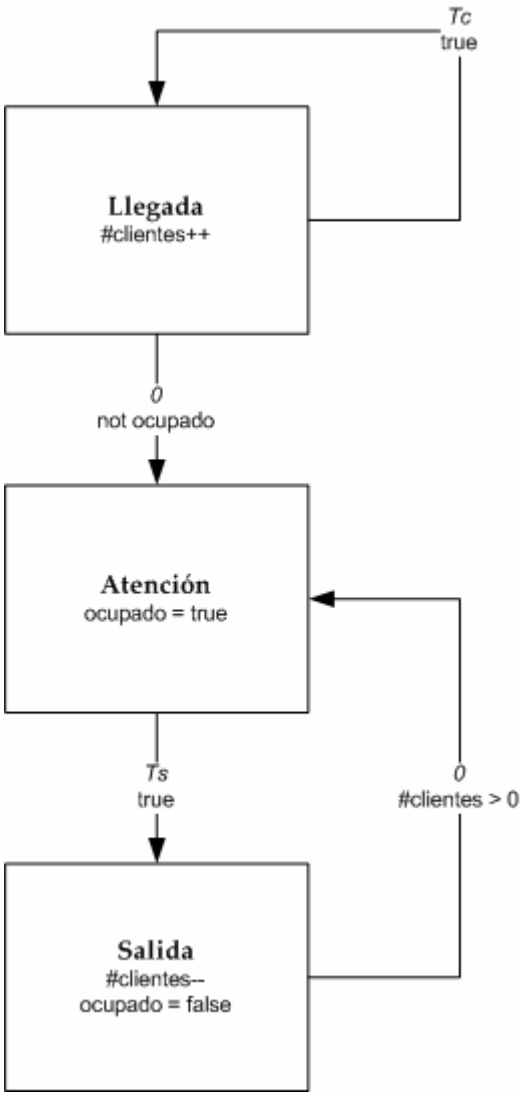


Figura A.3: Ejemplo de atención de clientes en grafo de eventos

5.2. Elementos avanzados

El artículo original presenta también la posibilidad de tener eventos con parámetros. Así en el caso de un modelo como el anterior pero con dos servidores, en lugar de replicar los eventos atención y salida, se puede darles un parámetro que signifique el servidor que se ocupa del cliente. Este parámetro se puede usar en los procedimientos y predicados del evento que los recibe, y se envían como una etiqueta extra del arco que lo agenda.

A su vez, se incluye la posibilidad de cancelar eventos. Técnicamente la cancelación de eventos no es necesaria para modelar ningún sistema, pero brinda expresividad. La cancelación de eventos se realiza con un tipo de arista especial etiquetada con un tiempo y predicado, que apunta hacia el evento a cancelar y se dispara si el predicado es cierto, el tiempo indicado luego de la ejecución del evento desde donde parte la arista.

Otras formas de cancelación han sido propuestas. [SS95] sugiere no agendar un evento hasta estar seguro de que no va a ser necesario cancelarlo. En la práctica esto requiere remplazar el evento por un evento intermedio que agenda al evento original en tiempo cero y bajo la condición de que no fue cancelado.

[IMW96] propone agregar un predicado más a las aristas, de tal manera de poder cancelar un evento al momento que iba a ejecutar. Es una notación especial para lo que se sugería antes.

Los grafos de eventos no brindan ninguna construcción estándar para encapsular modelos más sencillos como elementos de otros, en una jerarquía. Esto es una fortaleza en nuestra opinión. En general los modelos construidos por acoplamiento son muy rígidos. Un ejemplo de estos son los modelos acoplados Devs. Mientras que los modelos Devs atómicos son muy generales, los acoplados son muy inflexibles.

La forma de crear una jerarquía de modelos con grafos de eventos es mediante las construcciones que se usan en los predicados y procedimientos con que se etiqueta el grafo. Por ejemplo, si se está modelando una peatonal y se desea colocar el modelo de un banco para generar así los tiempos que pasan las personas dentro de los locales, el camino según [Sch95] es generar eventos dentro del modelo de banco por cada modelo de la calle, y viceversa, en el lenguaje usado para especificar los procedimientos en el modelo de la calle.

5.3. Conclusiones

Las construcciones de los grafos de eventos son muy apropiadas para el diseño de simulaciones a implementar en el paradigma de eventos. Permiten la visión global de lo que acontece sin complicar el pasaje al lenguaje final. A su vez, diversas extensiones permiten agregar facilidades de modelado que encajen bien con el resto de los elementos.

Los grafos de eventos no eliminan los peligros que tiene usar eventos. Sin embargo, la brevedad con que los sistemas son plasmados permite que posibles fallas salten a la vista más fácilmente.

6. Simulación paralela

La simulación paralela surge como la aplicación de técnicas de paralelismo a los modelos de simulación. Estas técnicas generalmente se aplican a simulaciones de sistemas complejos. Se busca distribuir la carga de trabajo y reducir el tiempo de ejecución a través de la ejecución concurrente.

Hay dos factores que hacen intuitiva este tipo de simulación:

- En un modelo con muchas entidades, no todas interactúan entre sí, sino que lo hacen casi siempre de manera dispersa.
- En un sistema grande se modelan varios submodelos menores que funcionan de forma similar entre sí.

La idea detrás de la simulación paralela, en particular la simulación a eventos discretos paralela, es dividir el problema en partes llamadas "Procesos Lógicos". Cada uno de estos tiene: su calendario de eventos, que respeta el tiempo de simulación global; y sus entidades particulares. Utilizan un mecanismo de comunicación para la coordinación de elementos en distintos procesos.

En un principio podría pensarse que la simulación paralela no es un paradigma sino un conjunto de técnicas elaboradas para distribuir una simulación en sistema de cómputo formado por múltiples procesadores. Sin embargo las características de los modelos de simulación imponen restricciones únicas a la paralelización. Dichas restricciones pueden solucionarse de varias maneras que involucran más directamente a conceptos de la simulación que al paralelismo. Por estas razones creemos que es válido discutir este punto en este apéndice.

6.1. Ejemplo

Se desea modelar un sistema de comunicación celular. Las entidades del sistema son las estaciones base (EB) y las estaciones móviles (EM). Una EM envía y recibe llamadas comunicándose con una EB por un canal de radio. Llamamos célula al área de cobertura de una EB, cuya forma depende de la propagación de la onda de radio.

Cuando una EM desea hacer una llamada, busca la EB con señal más fuerte e intenta conseguir un canal libre. Si no lo consigue, la llamada se bloquea. Cuando la EM encuentra una EB con señal más fuerte (por una cantidad significativa), busca un canal libre en la EB. Si lo encuentra, la llamada continúa por ese canal. Si durante un tiempo determinado la EM no encuentra una EB disponible cuya señal sea más fuerte que un mínimo determinado, la llamada se corta.

La idea es hacer un modelo de distribución de EBs tal que se reduzcan las posibilidades de bloqueo y corte.

En el modelo de simulación cada canal está modelado como un proceso lógico, pues hay un número finito de canales. Hay también un proceso encargado de generar nuevas EM en busca de señal y enviarlas al proceso lógico del canal que las va a manejar. El proceso del canal mantiene información de las EM y EB que usan el canal. La información sobre el alcance de cada EB está almacenada de forma global sólo lectura para el acceso de todos los procesos.

Durante la simulación, la movilidad de las EM hace que los procesos de los canales agenden eventos a sí mismos: actualizaciones en la posición de las EM comunicándose a través del canal. Estos eventos se dan cuando la señal de la EB tiene distinta fuerza en la nueva posición de la EM.

A intervalos regulares, cada EM verifica la fuerza de la señal de las distintas EB. Si encuentra una significativamente más fuerte que la actual con canales libres, intenta el cambio de canal enviándole un evento al otro proceso.

6.2. Coordinación temporal

Uno de los mayores problemas (si no el mayor) de la simulación paralela es mantener un tiempo global de simulación. Mantener un tiempo global en un sistema paralelo en una tarea difícil, aunque en este caso sólo se debe simular que todos los procesos lógicos corren sincronizadamente en tiempo de simulación, independientemente del tiempo real. Esto nos da mayor libertad y nos exige solamente que dado un conjunto de eventos agendados en un proceso lógico, éstos se ejecuten en el orden correcto de acuerdo con el tiempo de simulación. De esta manera se necesitan menos mecanismos de sincronización sin simular el problema equivocado.

La única causa posible de esta desincronización es la llegada a un proceso lógico de un evento externo que debe ejecutarse en un tiempo de simulación menor que el del evento actualmente en ejecución.

Existen dos enfoques para evitar esta desincronización, uno conservador y uno optimista.

Enfoque conservador

Este enfoque trata de evitar la generación de eventos para procesos lógicos que estén más avanzados. Esto se logra averiguando cuál es el tiempo mínimo simulado de los otros procesos. Pero también es necesario asegurarse que el tiempo mínimo simulado no avance más allá de los tiempos de simulación de los eventos que puedan agendarse en ese calendario. Una forma de hacer cumplir estas restricciones es por medio de bloqueos en la ejecución.

Si bien este método resuelve correctamente el problema, tiene el inconveniente de que se pierde eficiencia tanto en la sincronización como en el bloqueo especulativo a la espera de otros procesos. Además si no se implementa bien puede haber problemas de deadlock. Es por esto que esta técnica no se utiliza para la simulación paralela, excepto para benchmarks y otros procedimientos no dedicados a la investigación de aplicaciones reales.

Enfoque optimista

Este método permite bajo ciertas restricciones (posiblemente ninguna) que los procesos lógicos corran independientemente del resto. Además se tienen mecanismos de retroceso en cascada que permiten la vuelta atrás en caso de que llegue un evento anterior al tiempo actual de ejecución. Esta técnica es también llamada "Salto en el tiempo" (Time Warp).

Restricciones de avance

Existen dos conceptos vinculados al enfoque optimista y las restricciones de avance temporal:

- **Agresividad:** Es la tendencia a procesar eventos sin saber si tienen el menor tiempo de ejecución del sistema.
- **Riesgo:** Posibilidad de que un proceso envíe un mensaje basado en procesamiento agresivo.

Las restricciones que se imponen intentan ajustar la agresividad y el riesgo de los procesos, minimizando el tiempo perdido por procesos bloqueados y tiempo de retroceso. Se utilizan métodos de ventanas de tiempo, tanto real como simulado para el envío y la recepción de

mensajes; procesamiento especulativo sin envío de mensajes a otros procesos; etc. Varios métodos pueden ser vistos en [HT96].

Salto en el tiempo

Cuando sucede una discontinuidad provocada por el arribo de un evento pasado, hay que deshacer lo hecho hasta el momento de ejecución del nuevo evento, luego agendarlo en el calendario y seguir con la simulación. Se requiere de información extra para guardar la historia de la ejecución. También es necesario definir anti-mensajes para cancelar aquellos mensajes que hayan sido enviados durante el período que se debe simular nuevamente. Dependiendo de la estrategia a usar se guardarán ciertos datos con determinada frecuencia.

Existen dos estrategias para el salto:

- **Almacenamiento de estado disperso (sparse state saving):** Se almacena todo estado del proceso cada cierto lapso, y en caso de discontinuidad se recupera el último estado anterior al evento pasado y se reejecutan todos los eventos. No se envían mensajes de eventos durante la recuperación. Una de las grandes ventajas de este método es que permite transparencia al usuario, porque se salvan todos los datos. Existen varias estrategias para elegir el lapso de tiempo entre almacenamientos, dependiendo de los tiempos de almacenamiento y el uso de memoria.
- **Almacenamiento de estado incremental (incremental state saving):** Tras cada evento ejecutado en un proceso se almacenan solamente los datos que cambiaron desde el evento anterior, y cuando aparece un evento anterior al tiempo actual se restaura el estado hasta ese tiempo y se reejecutan los eventos a partir de este último. Este método da tiempos de recuperación generalmente mejores que el anterior, pero lograr transparencia es mucho más difícil.

El uso de una de estas dos estrategias depende de la cantidad de variables que cambian por evento y la velocidad de recuperación. Para un estudio más refinado de estas características ver [Rön96].

7. Referencias

[DO89] DAVIES, Ruth M.; O'KEEFE, Robert M. Simulation modelling with Pascal. Hertfordshire: Prentice Hall, 1989. p.302 ISBN: 0-13-811571-0

[Ray93] Ray J. Paul, Activity cycle diagrams and the three-phase method, Proceedings of the 25th conference on Winter simulation, p.123-131, December 12-15, 1993, Los Angeles, California, United States

[Mit86] MITRANI, I. Simulation techniques for discrete event systems. Cambridge: Cambridge University, 1986. p.185. Cambridge computer science texts; 14. ISBN: 0-521-23885-4

[Wai03] WAINER, Gabriel A. Metodologías de modelización de y simulación de eventos discretos. Buenos Aires: Nueva Librería, 2003. p.380. ISBN: 987-1104-01-4

[NS02] NANCE, Richard E.; SARGENT, Robert G. Perspectives on the Evolution of Simulation. Operations Research, January 2002, v.50, n.1, p.161-172.

[Sch83] SCHRUBEN, Lee. Simulation modeling with event graphs. Communications of the ACM, November 1983, v.26, n.11, p.957-963,.

[Sch95] SCHRUBEN, Lee. Building reusable simulators using hierarchical event graphs. Arlington, Virginia, United States: Proceedings of the 27th conference on Winter simulation, December 1995, p.472-475.

[SS95] SAVAGE, Eric L.; SCHRUBEN, Lee W. Eliminating event cancellation in discrete event simulation. Arlington, Virginia, United States: Proceedings of the 27th conference on Winter simulation, December 1995, p.744-750.

[IMW96] INGALLS, Ricki G.; MORRICE, Douglas J.; WHINSTON, Andrew B. Eliminating canceling edges from the simulation graph model methodology. Coronado, California, United States: Proceedings of the 28th conference on Winter simulation, December 1996, p.825-832.

[HT96] HAMNES, Donald O.; TRIPATHI, Anand. A comparative study of adaptive risk vs. adaptive aggressiveness control in parallel and distributed simulation. Proceedings of 29th Simulation Symposium, 1996.

[Rön96] RÖNNGREN, Robert et al. A comparative study of state saving mechanisms for time warp synchronized parallel discrete event simulation. Proceedings of 29th Simulation Symposium, 1996.

[Mis86] MISRA, Jayadev. Distributed Discrete-Event Simulation. ACM Computing Surveys, March 1986, v.18, n.1, p.39-65.

Apéndice B - Simuladores existentes

1. Introducción

En la literatura clásica se encuentra una primera clasificación de las herramientas en lenguajes de programación, lenguajes de simulación y entornos de simulación [BCNN00].

Los lenguajes de programación son aquellos utilizados para la generación de programas (no sólo de simulación), como C, Pascal, Java, Haskell, Python, etc. Estos lenguajes no fueron estudiados, debido a que no representan características propias de una herramienta específicamente diseñada para la simulación.

Los lenguajes de simulación son aquellos que definen un lenguaje estrictamente para la realización de modelos de simulación. Esto puede ser un lenguaje completo o una extensión de un lenguaje de programación existente, ya sea mediante un compilador con más palabras reservadas que extienden el lenguaje o con una librería que brinda funciones y estructuras adicionales.

Los entornos de simulación son herramientas pensadas para la simulación de alto nivel de determinados sistemas especiales, como la atención de público o la manufacturación. A cambio de la pérdida de flexibilidad se gana facilidad para modelar por parte de usuarios más vinculados al sistema a modelar pero con menos habilidades de programación.

2. Características

Más allá de la clasificación clásica, existen varios aspectos que caracterizan las diferentes herramientas. Se han estudiado diferentes artículos sobre el estado del arte de herramientas de simulación, encontrando la tabla de características más completa en [BCNN00]. De todos modos, esta clasificación pasa más bien por aspectos de funcionalidad que por las decisiones de diseño de las herramientas, así que decidimos hacer nuestra propia clasificación.

Para hacer esta clasificación filtramos características de estos surveys y los complementamos con revisiones superficiales de varias herramientas existentes. Luego elegimos un conjunto de estas herramientas a modo de cubrir el mayor espectro posible en cuanto a funcionamiento, interfaz y diseño.

Las características se dividieron en básicas y avanzadas, dependiendo de la necesidad de contemplarlas al diseñar una librería. Las mismas son:

- **Modelado:** Característica esencial de toda herramienta, determina la forma en que el usuario diseña y construye el modelo de la simulación, y como lo introduce en la herramienta para estudiarlo. Esta característica fue determinada como básica.
- **Manejo de números aleatorios:** Si bien no siempre son necesarios, una simulación realista toma en consideración sucesos que pueden ocurrir en forma aleatoria, de acuerdo a diferentes distribuciones de probabilidad. En esta característica se evalúa el método para obtener números aleatorios en determinadas distribuciones. Esta característica fue determinada como básica.
- **Manejo estadístico de datos:** Luego de ejecutar pruebas sobre el modelo, hay datos de salida que son de interés para el usuario, pero que no tienen utilidad alguna si no se realiza un estudio de los mismos. Esta característica engloba todo el manejo de los datos de salida, desde la simple exposición hasta elaborados reportes con histogramas y estudios de valores medias con intervalos de confianza. Esta característica es tanto básica como avanzada, dependiendo del tratamiento realizado a los datos.
- **Performance:** Esta propiedad atañe no solo las herramientas de simulación sino a todas las aplicaciones informáticas en general, esta característica refiere al consumo de recursos de las herramientas y cuanto tiempo demoran en realizar una tarea. El estudio de esta característica en las herramientas existentes fue realizado cualitativamente, a modo de ilustración general y por falta de modelos de benchmark adecuados. De todas maneras esto no contribuye a las decisiones de diseño en este aspecto, pues siempre se busca la mayor performance. La performance es un aspecto importante en esta clase de software, debido a que se acostumbra trabajar con una carga pesada de datos procesada mediante fuerza bruta, y debido a la necesidad de obtener datos estadísticos y no puntuales deben hacerse varias ejecuciones. Esta característica (o mas bien una buena performance) fue determinada como básica.
- **Visualización en tiempo real:** En muchos sistemas, dependiendo del objetivo de la simulación, es importante ver en tiempo real el estado del sistema durante una corrida. Esta visualización puede ser de varias formas, desde un registro en formato de texto plano marcando el tiempo de los eventos hasta un despliegue gráfico en 3D. La interacción del usuario con la visualización puede ser importante, no sólo para elegir lo que se ve sino para alterar los comportamientos del modelo en tiempo real. No todas las herramientas de simulación disponen de esta característica, y debido a que es una característica no imprescindible pero deseable, se determinó esta característica como avanzada.
- **Complejidades:** Esta característica engloba la flexibilidad de la herramienta hacia comportamientos complejos de los elementos del sistema, como entidades cooperantes, accidentes, reneging, etc. Las herramientas existentes suelen no contemplar estos casos

específicamente, y pueden proveer herramientas para implementarlos o métodos para resolver casos particulares del dominio al cual está orientado la herramienta. Se determina esta característica como avanzada.

Las herramientas que se eligieron para ser estudiadas fueron ProModel, Simpy, C++SIM, DesmoJ, EOSimulator, y Erlang. La elección de estas herramientas no fue al azar, sino que fueron escogidas cuidadosamente. El motivo de esta selección es cubrir herramientas disímiles entre sí pero que a la vez fueran una muestra representativa de todas las herramientas de simulación que se pueden encontrar. Se trató de probar integrantes de distintas clases de herramientas.

Por un lado tenemos a ProModel perteneciente a los entornos comerciales para el desarrollo de simulaciones, o también llamadas herramientas para el soporte de decisiones. Antes de realizar nuestra clasificación, y guiados por la clasificación clásica, buscamos un ejemplo de la clase de los entornos de simulación. Se estudió ProModel por tener una versión de prueba a disposición. En la nueva clasificación también es un representante en varias características, como el modelado por bloques y la optimización de los datos de salida.

Por otro lado, C++SIM es una herramienta de bajo nivel que soporta interacción a procesos pero dando un soporte mínimo. En DesmoJ se tiene una herramienta capaz de soportar dos paradigmas de simulación simultáneamente, y que además soporta algunas funcionalidades que podrían clasificarse de avanzadas. También tenemos a EOSimulator, una herramienta que soporta simulación orientada a eventos a dos y tres fases. Es una herramienta que está en pleno desarrollo y que espera ser utilizada por el Departamento de Investigación Operativa de nuestra facultad. Finalmente tenemos a Simpy, que es otra librería que se apoya fuertemente en la gran cantidad de librerías que posee Python. Todas estas son librerías de código libre para la simulación a eventos discretos, programadas sobre los lenguajes más utilizados. Es el grupo con mayor diversidad que se puede encontrar en cuanto a soporte de funcionalidad se refiere, además de ser el más extendido.

Erlang por su parte es un lenguaje funcional para el desarrollo de simulaciones a eventos discretos. Es una herramienta muy interesante para nuestro estudio por las características singulares que presenta: comunicación por pasaje de mensajes, distribución y lógica funcional.

La tabla B.1 muestra como se clasifican las herramientas estudiadas según esta clasificación:

	ProModel	Simpy	C++SIM
Modelado	Modelado por bloques	Descripción de procesos	Interacción de procesos
Manejo de números aleatorios	Un sólo stream de números aleatorios, ofrece varias distribuciones	Usa un generador externo	Ofrece varias distribuciones, expansible mediante herencia. Un generador por distribución
Manejo estadístico de datos	Reporte elaborado con datos relevantes a la manufacturación. Optimización mediante métodos iterativos	Externa	Ninguno
Performance	Tiene una buena performance	Consume mucho procesador, ahorra en cambios de proceso	Según la librería de threads subyacente
Visualización	Modelado orientado a la visualización, se ve en el mismo entorno de modelado (WYSIWYG)	Externa	Ninguna
Complejidades	Relevantes al área de manufactura	Ninguna	Ninguna

Tabla B.1a: Características de las herramientas existentes

	DesmoJ	EOSimulator	Erlang
Modelado	Eventos e interacción de procesos	Eventos, dos o tres fases	Interacción de procesos
Manejo de números aleatorios	Ofrece varias distribuciones, expansible mediante herencia. Un generador por distribución	Ofrece varias distribuciones, expansible mediante herencia. Un generador por distribución	Usa un generador externo
Manejo estadístico de datos	Counters, tallies, histogramas y análisis de regresión. Se elabora un reporte con lo elegido	Histogramas	Ninguno
Performance	Consume mucha memoria, buenos tiempos de ejecución	No genera un gran gasto de memoria, tiene una estructura del calendario que no es eficiente	Dado que se usa principalmente en red, depende de la velocidad de la misma
Visualización	No provee de salida gráfica, pero puede verse la generación de estadísticas en tiempo de ejecución	Ninguna	Ninguna
Complejidades	Clases implementadas para templates de puertos	Ninguna	Ninguna

Tabla B.1b: Características de las herramientas existentes

Las siguientes secciones de este apéndice profundizan en los diferentes aspectos de las herramientas elegidas, ya no solo según la clasificación sino en general.

3. ProModel

ProModel [HGB00] es un paquete que facilita el modelado y estudio de un sistema mediante la técnica de simulación a eventos discretos. Es un software comercial del que se evaluó una versión reducida del mismo.

ProModel permite manejar modelado del sistema, recolección y tratamiento de datos y optimización sobre el sistema. Presenta un enfoque de modelado distinto al convencional, pero que no es muy difícil de utilizar. Está orientado a las simulaciones visuales y a los procesos de manufactura.

3.1. Modelado de sistemas

ProModel es una herramienta de simulación que utiliza un enfoque gráfico de modelado. Está orientado a la simulación de sistemas de manufactura donde se tiene una línea de producción definida. Generalmente las diferentes materias primas van pasando por todas las locaciones de la línea y en cada una de estas etapas se transforman en otro componente. Finalmente al terminar su recorrido las materias primas se han convertido en un producto terminado.

El enfoque de estructuración propuesto por ProModel se ajusta perfectamente a este tipo de sistemas. En esta herramienta un modelo se compone de varios elementos que se van definiendo a medida que se avanza en la construcción de modelo. Otro aspecto importante del modelado de sistemas en este simulador es la representación gráfica de los mismos. Varias de los elementos de modelado tienen una representación gráfica, y mientras que se avanza en el modelado del sistema se va construyendo la salida gráfica de la simulación. Para modelar un sistema en ProModel se deben definir cuatro elementos básicos: Locaciones, Entidades, Procesos y Arribos.

Las locaciones son aquellos sitios que son relevantes para el modelo. Las entidades recorren las locaciones a lo largo de su ciclo de vida. Es en estos lugares … donde las entidades cambian de estado. En algunos casos las entidades pueden transformarse en otro tipo de entidades al llegar a una locación. Para crear una locación se elige un ícono, que será la representación gráfica de dicha locación, se le da un nombre y se lo coloca en una posición determinada dentro del plano donde se visualizará la simulación. También se le definen la cantidad de unidades que posee (por ejemplo la locación camas puede tener 12 unidades), la regla de despacho que utiliza y el tipo de estadísticas que se toman.

El concepto de entidad manejado por ProModel es igual al convencional. Son los individuos u objetos que son relevantes para el sistema que se quiere modelar. Para crear una entidad se elige el ícono que la representa, y se le asigna un nombre y una velocidad de movimiento.

Otro punto importante es la llegada de las entidades al sistema. Se modelan con arribos. Los arribos definen el tiempo entre arribos de las entidades, la locación en donde arriban las entidades, y la cantidad que llegan por arribo. También se puede definir si hay una cantidad máxima de arribos en el sistema, entre otras cosas.

Luego se definen el ciclo de vida de estas. Aquí se definen el camino que siguen las entidades a través del sistema. Se definen las locaciones que recorren las entidades, cuanto tiempo pasan en cada locación, y que hacen al terminar su estadía. En ProModel estos ciclos se modelan separadamente por medio de dos elementos: proceso y ruteo. Los procesos describen las acciones que realiza la entidad. En los ruteos se definen las locaciones que recorren. Para construir estos elementos, ProModel da una interfaz tipo tabla donde se define para cada entidad y locación la acción que se tomará. Las acciones que una entidad puede tomar se declaran en un proceso y están definidas por un lenguaje subyacente. El ruteo se presenta también en un formato tabular y se construye junto con el proceso. Este define el tipo de entidad de salida del proceso, a que locación se dirige, que regla utiliza para moverse,

y que lógica de movimiento se usa. Las reglas definen a que locación se dirigen dependiendo de la disponibilidad que estas tengan.

3.2. Manejo de datos

Luego de una corrida, ProModel genera un reporte con todos los datos que fueron recolectados. Dichos de datos incluyen cantidad de arribos, cuantas unidades de tiempo se ocupa una locación determinada, cantidad de entidades remantes en el sistema, entre otras.

Además genera gráficos sobre los datos recabados, dando gran cantidad de opciones de cómo presentarlos. Se crean gráficos de series de tiempo, y diversos gráficos sobre la utilización de recursos.

Para el modelado de los datos de entrada, se tiene el paquete Stat::Fit que ajusta juegos de datos a distribuciones conocidas. Stat::Fit es capaz de ajustar datos de tipo continuo o discreto a diversas distribuciones. Para ello utiliza test como el Chi-cuadrado y el Kolmogorov-Smirnov.

3.3. Optimización

En algunos casos se puede utilizar la simulación del sistema para "realizar optimizaciones" sobre el mismo. La idea es encontrar valores óptimos de las variables de decisión de acuerdo a una cierta función objetivos sobre los valores de las variables de salida. Para realizar estas optimizaciones generalmente se usan procedimientos heurísticas como Búsqueda Tabú, Algoritmos Genéticos, entre otros.

En ProModel se tiene una aplicación capaz de realizar este tipo de optimizaciones sobre modelos, SimRunner. Para utilizar SimRunner es necesario declarar una función objetivo que en base a algunas variables de salida y elegir cuales variables de entrada se utilizaran para optimizar. Se utiliza un algoritmo genético para la optimización.

3.4. Conclusiones

ProModel es un paquete comercial para el desarrollo de simulaciones a eventos discretos. Si bien está orientado a simular sistemas de manufactura, también puede ser usado para otros sistemas con mayor o menor dificultad. Pero en cuanto al modelado que soporta se puede decir en principio que no es muy intuitivo o ceñido a un paradigma en particular. Posee buenas herramientas para realizar análisis de las variables de salida y ajustes de los datos de entrada. Posee visualización grafica, pero el manejo de complejidades no es muy extenso y está muy adaptado al dominio de sistemas de manufactura. En síntesis, implementa todas las características básicas y varias de las deseables de las herramientas de simulaciones, siempre para el dominio al que está orientado.

4. Simpy

Simpy [SPY05] es una librería basada en descripción de procesos para la simulación a eventos discretos en el lenguaje de programación Python. En realidad, es la única librería de dicho tipo para Python, pero no es parte de la librería estándar y se obtiene por separado.

Python es un lenguaje moderno, interpretado, dinámicamente tipado y orientado a objetos. No tiene ninguna característica nueva y original, sino que se inspira en una cantidad de lenguajes anteriores. Python define facilidades para el manejo de excepciones, módulos, overloading de operadores, herencia múltiple y generadores, entre otras. Estos últimos son la clave para la implementación de Simpy.

4.1. Generadores

Un generador es un objeto de Python para la generación a demanda (perezosa) de una sucesión de valores. La manera de crear un generador es con un procedimiento o método que use la palabra clave `yield`:

```
def gen():
    yield 1
    for i in range(3):
        yield 2
    yield 3

for i in gen():
    print i
```

```
1
2
2
2
3
```

Los generadores son muy útiles para la escritura de código de tipo callback que mantiene estado entre llamados. En el ejemplo anterior, el objeto creado por la invocación de `gen` recuerda en qué parte del cuerpo del procedimiento se encuentra, y continúa desde ese lugar cuando se le pide el siguiente valor.

La programación de entidades se ajusta al patrón que facilitan los generadores. Se ejecutan ciertas instrucciones y se pasa eventualmente a un estado durmiente, para ser reactivadas por el ejecutivo. Las directivas que tienen que ver con la simulación se codifican entonces como:

- `yield hold,self,t` - Para dormirse un tiempo t.
- `yield request,self,r` - Obtener el recurso r, quizá bloqueando.
- `yield release,self,r` - Devolver el recurso.

En las versiones primitivas de Simpy se utilizaban threads para lograr la orientación a procesos. Las threads tienen la desventaja de ser más lentas y consumir más memoria, al menos en la mayoría de las implementaciones. Disminuyen la portabilidad porque existen sistemas que no tienen una librería de threads. Además, hay librerías que no son transparentes a ser invocadas desde más de una thread, aunque se mutuoexcluya el acceso

externamente. En Windows, por ejemplo, no se pueden recibir los mensajes destinados a una ventana desde una thread diferente a la que la creó.

Los generadores tampoco son una solución perfecta. Son una implementación de semi-corutinas a nivel del lenguaje de programación. Comparadas con corutinas tienen dos desventajas, que solo se puede transferir el control al contexto inmediatamente anterior y que funcionan en un nivel solo. La primera es fácilmente sorteable, pasando siempre por un intermediario que reciba por parámetro el `yield` que otro generador activa. En nuestro caso este intermediario es el ejecutivo.

Que los generadores sean de solo un nivel es un problema más serio. Lo que esto quiere decir es que no se puede partir el proceso de una entidad en varios métodos:

```
# versión que no funciona
def proceso_mal():
    parteA()
    parteB()

def parteA():
    yield 'a'

def parteB():
    yield 'b'
```

Las invocaciones a `parteA` y `parteB` en `proceso_mal` crean generadores pero estos se desechan inmediatamente. De hecho, como `proceso_mal` no contiene `yield`, Python lo considera una función cualquiera y el intento de tratar el valor de retorno de su invocación como un generador fallará en tiempo de ejecución. La definición correcta de proceso es:

```
def proceso():
    for i in parteA():
        yield i
    for i in parteB():
        yield i
```

Como se ve, hay que volver generadores explícitamente a todos los procedimientos de un árbol de llamado, lo que obviamente no es escalable. Entonces las ventajas del enfoque basado a descripción de procesos no se ven realizadas en su totalidad en Simpy. Para procesos pequeños, se obtiene una definición concisa del accionar de la entidad. A su vez, cuando la entidad es simple, no es un gran costo definir las diferentes partes de su ciclo de vida en métodos diferentes, como se requiere con una herramienta de simulación basada en eventos. Para procesos grandes, donde la definición a eventos se vuelve un spaghetti de llamadas alrededor, los procesos de Simpy se vuelven bloques monolíticos enormes o se introduce una cantidad grande de plomería para poder separarlos.

4.2. Ejemplo de modelado

Se muestra a continuación un sistema sencillo modelado con Simpy. Se trata en este caso de la simulación del hospital simple de [DO89].

```
from SimPy.Simulation import *
```

```
from random import *

intervalo_arribo_pacientes = 4.0
duracion_internacion_pacientes = 4.0
intervalo_arribo_operados = 15.0
espera_pre_operacion = 2.0
tiempo_operacion = 2.0
espera_pos_operacion = 2.0

class PacienteFeeder(Process):
    def generar(self):
        while(True):
            yield hold, self, expovariate(1.0/intervalo_arribo_pacientes)
            p = Paciente()
            activate(p, p.internacion())

class OperadoFeeder(Process):
    def generar(self):
        while(True):
            yield hold, self, expovariate(1.0/intervalo_arribo_operados)
            o = Operado()
            activate(o, o.internacion())

class Paciente(Process):
    def internacion(self):
        time = now()
        def put(msg):
            print '%.2f %s %.2f'%(now(), msg, time)
        put('arribo comun')
        yield request, self, bed
        put('como duermo')
        yield hold, self, expovariate(1.0/duracion_internacion_pacientes)
        put('chau')
        yield release, self, bed

class Operado(Process):
    def internacion(self):
        time = now()
        def put(msg):
            print '%.2f %s %.2f'%(now(), msg, time)
        put('vine a que me corten')
        yield request, self, bed
        put('me acostaron para tranquilizarme')
        yield hold, self, expovariate(1.0/espera_pre_operacion)
        put('pa dentro')
        yield request, self, sala_operaciones
        put('no se olviden la anestescia')
        yield hold, self, expovariate(1.0/tiempo_operacion)
        yield release, self, sala_operaciones
        put('me duele!!')
        yield hold, self, expovariate(1.0/espera_pos_operacion)
        yield release, self, bed
        put('pa mi casa')

def main():
    global bed
    global sala_operaciones
```

```
bed = Resource(capacity=2)
sala_operaciones = Resource()
seed(0)
initialize()
f = PacienteFeeder()
activate(f, f.generar())
f = OperadoFeeder()
activate(f, f.generar())
simulate(until=100.0)
```

El código es muy similar a seudocódigo. Las entidades se definen directamente en el estilo de proceso, sin transformación de ningún tipo. Los `yield` marcan claramente cuales son las acciones posiblemente bloqueantes, ayudando a la distinción sintáctica de las diferentes acciones de una entidad. Debe pasarse `self` en todos los `yield`, pero esto no desentona con los otros elementos del lenguaje, porque Python requiere ser explícito en esos casos.

4.3. Conclusión

Simpy parece una alternativa sólida para el modelado de sistemas usando el paradigma de descripción de procesos. Provee las primitivas clásicas del enfoque sin perder la comodidad de un lenguaje ampliamente usado y soportado. De esta manera, es posible definir nuestro modelo naturalmente pero al mismo tiempo tener acceso a las facilidades de carácter general que todo programa necesita y que harían pesado un lenguaje específico de un dominio. El acercamiento de librería de veras brilla.

5. C++SIM

C++SIM [CXS05] es un paquete de simulación en C++ que provee simulación basada en procesos con mecanismos similares a los de SIMULA pero con más énfasis en la herencia de clases. Fue creado en la universidad de Newcastle upon Tyne y se encuentra en la versión 1.7.4 desde octubre de 1998.

5.1. Elementos básicos

Threads

Todos los procesos de la simulación corren en threads administradas por el scheduler. Éste se encarga de correrlas según se agendan, suspenderlas y retomarlas. Antes de ver el scheduler y los procesos se estudian las threads, ya que son uno de los elementos fundamentales del sistema.

Se utiliza un sistema de threads para mantener el estado de los diferentes procesos a medida que ejecutan, no para la ejecución concurrente de las mismas: en todo momento hay una sola thread corriendo, aunque se simula la ejecución simultánea de las mismas en tiempo de simulación. C++SIM no soporta ejecución paralela.

El sistema está diseñado para trabajar con cualquier biblioteca de Threads que maneje las siguientes características:

- Identificación
- Suspensión de la ejecución

```
class Thread : public Resource {
public:
    virtual void Suspend() = 0; // Suspende la ejecucion
    virtual void Resume() = 0; // Retoma la ejecucion
    virtual void Body() = 0; // Cuerpo del proceso
    virtual long Current_Thread() = 0; // Identificador de la Thread
    virtual long Identity(); // ID del objeto que corre en la Thread
    static Thread* Self(); // Thread actualmente corriendo
    static void Exit (int = 0); // Termina la aplicacion
    static void mainResume (); // Continua la ejecucion de la primer
                               // thread
    static void Initialize (); // Inicializa el paquete. Debe ser
                               // llamada una sola vez al comienzo
protected:
   Thread (); // Constructor
   virtual ~Thread (); // Destructor
   long thread_key; // ID del objeto, seteado por la subclase
};
```

Para utilizar la biblioteca se debe hacer una clase que hereda de `Thread` e implementar las operaciones virtuales en la biblioteca deseada.

Ejemplo: Sun Threads

```
class LWP_Thread : public Thread {
```

```
public:
    virtual void Suspend();
    virtual void Resume();
    virtual void Body() = 0;
    virtual long Current_Thread();
    thread_t Thread_ID(); // Informacion detallada de la Thread
    static void Initialize();
protected:
    static const int MaxPriority;
    LWP_Thread(int priority = MaxPriority); // Constructor
};
```

Como se puede ver, `Body` sigue sin implementar. Esta operación debe ser implementada por los diferentes procesos.

La clase `Resource` de la que deriva `Thread` es una forma de hacer garbage collection con reference counting. Tiene algunos bugs acerca de condiciones de carrera, los cuales no se han mejorado.

```
class Resource {
    public:
        static void ref   (Resource* resource);
        static void unref (Resource* resource);

        void* operator new    (size_t amount);
        void  operator delete (void* memPtr);

    protected:
        Resource ();
        virtual ~Resource ();
};
```

Procesos

Los diferentes procesos de la simulación se definen como clases que derivan de la clase `Process`, y tienen los siguientes estados:

- **Activo:** El proceso está siendo ejecutado
- **Suspendido:** El proceso está agendado y será ejecutado luego
- **Pasivo:** El proceso no está agendado. A menos de que otro proceso lo devuelva al calendario, no se ejecutará más.
- **Terminado:** El proceso no está agendado y ya ha realizado todos sus pasos.

```
class Process : public LWP_Thread {
public:
    static const int Never; // Si se pospone a este tiempo no se
                            // realizará

    virtual ~Process (); // Destructor

    static double CurrentTime (); // Tiempo actual de la simulación
    double Time() const; // Igual, pero no static
```

```
    const Process* next_ev () const; // Proximo proceso en el
                                     // scheduler, NULL si la
                                     // cola esta vacia

    // Funciones para manejo de agendado
    void ActivateBefore (Process&);
    void ActivateAfter (Process&);
    void ActivateAt (double AtTime = CurrentTime(),
                     Boolean prior = FALSE);
    void ActivateDelay (double AtTime = CurrentTime(),
                        Boolean prior = FALSE);
    void Activate();
    void ReActivateBefore (Process&);
    void ReActivateAfter (Process&);
    void ReActivateAt (double AtTime = CurrentTime(),
                       Boolean prior = FALSE);
    void ReActivateDelay (double AtTime = CurrentTime(),
                          Boolean prior = FALSE);
    void ReActivate ();

    void Cancel (); // Si esta en la cola, lo remueve

    double evtime (); // Tiempo hasta la activación
    void set_evtime (double); // Setea el tiempo para activación
    Boolean idle (); // True si esta agendado
    Boolean passivated (); // True si está pasivo
    Boolean terminated (); // True si está terminado
    virtual void terminate (); // Termina el proceso a la fuerza

    static const Process* current (); // Devuelve el proceso en
                                      // ejecucion

    virtual void Body () = 0; // Codigo del proceso

    virtual void reset (); // Resetea el proceso
protected:
    Process (); // Constructor
    void Hold (double t); // Remueve de la cabeza de la cola y
                          // agenda luego
    void Passivate (); // Remueve al proceso de la agenda
};
```

Los tiempos que manejan estas operaciones son tiempo de simulación, no tiempo real.

Scheduler

El scheduler coordina la ejecución de los diferentes procesos en una cola. Todos los procesos esperando ejecución (ya sea porque no es su tiempo de activación o porque se está ejecutando un proceso con el mismo tiempo de activación) se encuentran en la cola de simulación.

La cola de simulación es una lista de listas. Cada una de estas listas contiene, sin orden especial, todos los procesos que deben ser activados en un mismo tiempo de simulación. Estas

listas están ordenadas en la lista de primer nivel según su tiempo de activación, de menor a mayor.

Cuando no hay procesos activos, se toma un elemento de la primer lista de la cola (la lista con procesos de menor tiempo de activación) y se lo activa (o reactiva). Cuando ya no hay procesos en la cola, se termina la ejecución.

```
class Scheduler {
public:
    double CurrentTime (); // Devuelve el tiempo de simulación
                           // actual
    void reset () const;  // Lleva el tiempo a 0 y reinicia procesos
    static Scheduler& scheduler (); Devuelve el Scheduler singleton
    static void terminate (); // Destruye el Scheduler singleton
    void Suspend (); // Suspende la simulacion
    void Resume (); // Reanuda la simulacion
private:
    Scheduler (); // Constructor
    ~Scheduler (); // Destructor

};
```

Siempre existe un único `Scheduler` disponible global debido al patrón singleton utilizado.

Las operaciones `Suspend()` y `Resume()` se implementan para mantener la compatibilidad con versiones anteriores, las cuales implementaban al scheduler como un thread.

`reset()` debe ser usado con mucho cuidado, puesto que los threads de los procesos solo pueden ser reseteados mientras están corriendo. Se utilizan unas macros para verificar si se reinició durante la inactividad, pero no es una práctica recomendada.

Funciones de distribución

C++SIM cuenta con varios generadores de números aleatorios de diferentes distribuciones, a la vez que permite agregar otros. La idea básica es que todos los generadores son functores subclases de `RandomStream` que devuelven un número de la distribución deseada cuando se los invoca. En el constructor se les pasa parámetros como la semilla, la media y la varianza.

Entidades

La entidad es una clase de proceso que puede ser utilizada para simular eventos asincrónicos. Y como es un proceso también puede simular eventos sincrónicos.

Además de los estados en que puede estar por ser proceso, puede estar en estos dos estados:

- **Esperando:** suspendido a la espera de un evento específico. No se encuentra agendado
- **Interrumpido:** el proceso fue interrumpido mientras esperaba un evento asincrónico.

Hay que tener cuidado al utilizar entidades para no generar deadlocks.

```
class Entity : public Process {
public:
    virtual ~Entity (); // Destructor
```

```
    // Interrumpe la entidad esperando y la coloca al comienzo de
    // la cola
    Boolean Interrupt (Entity& toInterrupt,
                       Boolean immediate = TRUE);


    virtual void terminate (); // Igual que el del padre

    virtual void Body () = 0; // Código de la entidad
protected:
    Entity (); // Constructor

    // Espera a evento asincronico
    Boolean Wait (double waitTime); // Tiempo de simulacion
    Boolean WaitFor (Entity& controller,    // Espera la terminacion
                     Boolean reAct = FALSE); // de otra entidad. Si
                                             // reAct es TRUE, va al
                                             // principio de la cola
    Boolean WaitForTrigger (TriggerQueue& _queue); // Trigger
    void WaitForSemaphore (Semaphore& sem);        // Semaforo
};
```

Semáforos

Se provee una clase `Semaphore` para la mutuoexclusión en el uso de recursos por parte de entidades.

```
class Semaphore {
public:
    Semaphore (); // Constructor
    virtual ~Semaphore (); // Destructor

    virtual void Get (Entity* attempting); // Intento de obtener
                                           // permiso, bloqueante
    virtual void Release (); // Desbloquea entidades esperando
                             // o permite la obtención por uno nuevo

    long NumberWaiting () const; // Cantidad de entidades esperando
};
```

Triggers

Los `TriggerQueue` son colas de entidades para ser disparadas en algún punto de la aplicación, como por ejemplo cuando se simula una interrupción.

Si una `TriggerQueue` abandona su scope y no está vacía, todas sus entidades serán reactivadas.

```
class TriggerQueue
{
public:
    TriggerQueue (); // Constructor
    virtual ~TriggerQueue (); // Destructor
```

```
    Boolean triggerAll (); // Activa todas las entidades
    Boolean triggerFirst            // Activa la primer entidad de
        (Boolean triggered = TRUE); // la cola. Si triggered es
                                    // FALSE, se interrumpe la
                                    // entidad en vez de despertarla
};
```

5.2. Ejemplo de implementación: Productor consumidor

Esta es la función `main` de la aplicación:

```
// Globales
long TotalNumberOfJobs = 0; // Para control
long NumberOfJobsConsumed = 0; // Para control
Queue JobQueue; // Cola de trabajos esperando

int main (int, char**) {
    Thread::Initialize(); // Inicializa el paquete de threads

    Controller c; // Crea el controlador de la aplicación
    c.Await();    // Suspende la thread de main
    c.Exit();     // Reactiva la thread de main y termina
    return 0;
}
Clase Controller:
class Controller : public Process {
public:
    Controller () {} // Constructor que no hace nada
    ~Controller () {} // Destructor que no hace nada

    void Body ();

    // Suspende la thread main
    void Await () { Resume(); Thread::Self()->Suspend(); }
    void Exit () { Thread::Exit(); } // Termina la ejecucion
};

void Controller::Body () {
    // Crea productor y consumidor
    Producer* p = new Producer(10);
    Consumer* c = new Consumer(10);
    // Aumenta el conteo de referencias
    Resource::ref(p);
    Resource::ref(c);

    // Activa los dos procesos
    p->Activate();
    c->Activate();

    // Despierta al Scheduler singleton
    Scheduler::scheduler().Resume();

    // Se suspende por 10000 segundos de tiempo de simulacion
    Hold(10000);
```

```
    // Salida de datos
    cout << "Total number of jobs present "
         << TotalNumberOfJobs << '\n';
    cout << "Total number of jobs processed "
         << NumberOfJobsConsumed << endl;

    // Reposicion del scheduler
    Scheduler::scheduler().reset();
    Scheduler::scheduler().Suspend();

    // Elimina los procesos
    p->terminate();
    c->terminate();
    Resource::unref(p);
    Resource::unref(c);

    Thread::mainResume(); // Vuelve al main
}
```

Luego hay que definir las entidades Productor y Consumidor:

```
class Producer : public Entity {
public:
    Producer (double mean) : // Constructor, inicializa distribucion
        InterArrivalTime(ExponentialStream(mean)) {}
    virtual ~Producer () {} // Destructor, no hace nada

    virtual void Body ();

    static Semaphore _semaphore;
private:
    ExponentialStream InterArrivalTime;
};

void Producer::Body ()
{
    for (;;) {
        Job* work = new Job();
        if (JobQueue.IsFull()) Producer::_semaphore.Get(this);
        TotalNumberOfJobs++;
        JobQueue.Enqueue(work);
        Consumer::_semaphore.Release();
        Hold(InterArrivalTime()); // Entre trabajo y trabajo
                                  // espera según dist. exponencial
   }
}


// Consumer es muy similar a Producer
class Consumer : public Entity {
public:
    Consumer (double mean) :
        InterArrivalTime(ExponentialStream(mean)) {}
```

```
    virtual ~Consumer ();

    virtual void Body ();

    static Semaphore _semaphore;
private:
    ExponentialStream InterArrivalTime;
};

void Consumer::Body ()
{
    for (;;) {
        if (JobQueue.IsEmpty()) Consumer::_semaphore.Get(this);
        Job* work = JobQueue.Dequeue();
        Producer::_semaphore.Release();
        NumberOfJobsConsumed++;
        Hold(InterArrivalTime());
    }
}
```

La clase `Queue` es una cola acotada común y `Job` es una clase que solo tiene un constructor y un destructor que no hacen nada, por lo tanto no se describirá aquí su código.

5.3. Resumen y conclusiones

C++SIM es una librería de simulación de bajo nivel, desde el punto de vista que hay que ir controlando referencias y punteros, sin soporte de eventos, tratamiento estadístico de datos ni visualización gráfica. Sin embargo tiene a favor que es fácil de extender debido a sus mecanismos de herencia. El control sincrónico y asincrónico de procesos en tiempo de simulación está bien hecho y sus mecanismos son intuitivos.

La documentación es simple pero informativa, con "man pages" que detallan los elementos de todas las clases utilizadas.

Parece una buena base para sacar ideas sobre la etapa de simulación y scheduling, aunque utilizando técnicas más modernas de implementación y elevando un poco el nivel de programación.

6. DesmoJ

DesmoJ es un framework para el modelado e implementación de simulaciones a eventos discretos. Se provee de dos de los enfoques más utilizados en este campo: orientación a eventos e interacción de procesos. DesmoJ esta bajo la licencia GNU Lesser GPL, por tanto se dispone de su código fuente y se permite hacerle modificaciones.

6.1. Modelado de sistemas en DesmoJ

El modelado de un sistema en DesmoJ, utilizando el enfoque de eventos, se realiza implementando clases derivadas de `Model`, `Event`, `Entity`, y `SimProcess`.

Esta librería permite crear modelos híbridos, o sea modelos que se encuentren construidos con eventos y procesos. La clase más importante en el modelado es `Model`. Es una clase abstracta que contiene el modelo del sistema a simular. En este se definen las colas de espera, las entidades y eventos, las distribuciones que rigen los aspectos aleatorios del sistema. Los modelos de DesmoJ pueden estar constituidos por otros modelos. Estos submodelos pueden ser usados para modularizar las simulaciones y como componentes reutilizables.

Las operaciones que se deben implementar para instanciar una subclase de `Model` son:

- Constructor
- `init`: en esta operación se inicializan los aspectos estáticos del modelo, se deben crear instancias de las distribuciones, las colas y las entidades globales del sistema.
- `doInitialSchedules`: en esta operación se realizan los primeros pasos para poder comenzar la simulación. O sea que se agendan los eventos que deben ponen en funcionamiento la simulación (los feeders, por ejemplo).
- `main`: Es el método principal, es aquí donde se realizan la corridas. Este método puede estar en esta clase o en otra aparte.

Otra consideración interesante de DesmoJ, es que maneja el tiempo con la clase `SimTime`. Todas las operaciones que impliquen conocer o asignar tiempo obtiene o crean un objeto de este tipo con el valor deseado.

Orientación a eventos

Esta librería soporta el enfoque de eventos en dos fases. Para el modelado de sistemas se proveen las clases `Entity` y `Event`.

`Entity` es la superclase de todas las entidades del sistema. Se debe implementar un constructor y el manejo de los atributos de la entidad (si es que tiene alguno).

`Event` es la superclase de los eventos de un sistema. En DesmoJ se definen dos tipos de eventos:

- **Internos al sistema:** son aquellos que necesitan de una entidad para ser invocados. Este tipo de eventos heredan directamente de `Event`. Para este tipo de eventos se debe implementar la operación abstracta `eventRoutine(Entity)`, que define las acciones que se toman en el evento en cuestión.
- **Externos al sistema:** son aquellos que definen acciones externas al sistema. Son los eventos que generan la entrada del sistema. Estos heredan de la clase `ExternalEvent`. En estos eventos se debe implementar la operación abstracta `eventRoutine` que es equivalente al descrito anteriormente, solo que no necesita de una entidad para ser invocada.

Para agendar eventos en el modelo se les invoca la operación `schedule` con la entidad correspondiente (en caso de haber una) y un valor de tiempo que es relativo al tiempo actual de la simulación.

Orientación a procesos

DesmoJ soporta el enfoque de interacción de procesos. Para modelar procesos se provee la clase abstracta `SimProcess`, que es una subclase de `Entity`. Esta clase contiene una `SimThread`, que es una subclase de las threads de Java. Es en esta thread donde se tiene el punto del ciclo de vida en que se encuentra la entidad. Para crear un proceso se debe implementar una clase derivada de `SimProcesss`. En ésta se debe implementar un método para la operación abstracta `lifeCycle`. Es en dicha operación donde se modela el ciclo de vida de una entidad. Se brindan las primitivas de `activate` y `passivate` para activar o suspender un proceso. También se tiene la operación `hold` para agendar a un proceso en el Scheduler. Tiene una funcionalidad similar a la operación `schedule` de `Event`.

Cabe destacar que DesmoJ incluye un conjunto de clases para modelar cierto tipo de complejidades, sobre todo relacionadas con sistemas de puertos. Dichas clases ofrecen soporte a las simulaciones de interacción a procesos, pero no para aquellas orientadas a eventos.

6.2. Manejo de números aleatorios

Todos los generadores de números aleatorios (incluidas las distribuciones) implementan la interfaz `UniformRandomGenerator`. DesmoJ provee la clase `DefaultRandomGenerator` que es el generador por defecto. Si se quisiera utilizar otro se puede reemplazar adecuadamente (y en los constructores las clases que sean necesarias) el `DefaultRandomGenerator` con la clase deseada. O también se puede cambiar la implementación del generador por defecto.

Manejo de distribuciones

El manejo de la aleatoriedad en DesmoJ se realiza por medio de 2 clases básicas: `Distribution` y `DistributionManager`. Las instancias de la primera se declaran y utilizan en el `Model`. Forman parte del sistema que se quiere modelar. Pero las instancias de `DistributionManager` se encuentran dentro de un `Experiment`. Forman parte de la infraestructura necesaria para la ejecución de una simulación.

En la clase `Distribution` se encapsula el comportamiento de una distribución (real o discreta). Es la superclase de todas las distribuciones que provee DesmoJ. Las instancias de `Ditribution` contiene dentro un generador de números aleatorios, con el cual muestrean una variable aleatoria.

En cambio el `DistributionManager` es utilizado por un `Experiment` para manejar el conjunto de las distribuciones utilizadas. Contiene generador de números aleatorios, `seedGenerator`, y el conjunto de todas las distribuciones definidas en un modelo. Con el generador se inicializa a las distribuciones del modelo con una semilla.

Cuando una instancia de `Distribution` se crea, ésta se registra en el `DistributionManager` y se le asigna una semilla. Esto significa que se tienen tantos streams como distribuciones declaradas en el modelo.

El `DistributionManager` es un control central de todas las distribuciones del modelo. Si se quiere repetir una corrida, se le asigna al manejador la misma semilla que en la corrida anterior, por medio de la operación `setSeedGenerator` del `Experiment`.

6.3. Ejecución de una Simulación

Para correr una simulación se necesitan dos clases, un `Model` y un `Experiment`. En el modelo se encapsula el comportamiento del sistema a simular. O sea, en esta clase se definen las colas del sistema, las distribuciones que modelan la aleatoriedad del sistema, las entidades globales al sistema y (aunque no directamente) las entidades y eventos del sistema.

En el experimento se definen las herramientas que controlan la ejecución correcta de la simulación. Estas son el calendario, clase `Scheduler`, y el manejo de las distribuciones aleatorias brindado por la clase `DistributionManager`.

El calendario consiste en una lista de tipo `EventList` que contiene objetos de la clase `EventNote`. Estos que contienen una entidad y un evento. En el caso de orientación a eventos se tiene una entidad y un evento o un proceso para la orientación a procesos, que se encuentra en el mismo miembro que la entidad puesto que la superclase de `SimProcess` es `Entity`.

El proceso de ejecución de la simulación se realiza de la siguiente manera:

1. Se crea una instancia del modelo a simular
2. Se crea una instancia de experimento
3. Se conecta el modelo con el experimento por medio de la operación `conectToExperiment`. En este paso se ejecuta el método `init` definido en el modelo a simular.
4. Desde el modelo se invoca la operación `start` del experimento. En este método se prosigue de este modo:
   i. El experimento invoca el método `doInitialSchedules` del modelo.
   ii. Se invocan varios métodos intermedios (`start(SimTime)` y `proceed`) del experimento para poder comenzar con la simulación.
   iii. En el método `proceed` se le invoca al `Scheduler` el método `processNextEventNote`, que procesa el siguiente evento o proceso de la simulación mientras sea posible.

El método `processNextEventNote` del Scheduler realiza las siguientes acciones:

1. Se invoca la operación `firstNote` al calendario. Esta retorna una instancia `EventNote` que es la que debe ejecutar.
2. Luego se realizan los chequeos necesarios para saber si se trata de una entidad y un evento o de un proceso, y se invocan las operaciones abstractas `eventRoutine` o se despierta al proceso.

Luego de cada corrida se generan 4 archivos HTML:

- **Report:** brinda un reporte completo de la corrida
- **Trace:** brinda un reporte con todos los eventos (que han sido previamente seleccionados en el constructor)
- **Debug:** permite imprimir información para remover bugs del modelo
- **Errors:** brinda un reporte de los errores cometidos en el manejo interno de DesmoJ (sacar elementos de colas a las que no pertenecen, mal manejo de estructuras, etc.)

6.4. Conclusiones

DesmoJ es un simulador que soporta dos paradigmas de simulación. Esto es debido a la estructura de su calendario, que soporta procesos o eventos, pero no restringe a que un modelo sea solo de un tipo. Aunque no lo hace de forma pura, y está más desarrollado el soporte para procesos que para eventos. Sin embargo es una herramienta de fácil uso. Provee de un buen manejo de variables de salida, pero no tiene salida gráfica. Posee una documentación de usuario completa que consiste en un tutorial y API. Implementa todas las características básicas que se definieron, pero le faltan varias de las características deseables.

7. EoSimulator

Esta herramienta es un desarrollo reciente del Departamento de Investigación de Operaciones de la Facultad de Ingeniería. La idea de EoSimulator es crear una herramienta capaz de reemplazar al simulador Pascal_Sim para los cursos de Simulación a Eventos Discretos dictados por dicha facultad. Comparte algunas características similares con DesmoJ y Pascal_Sim.

7.1. Arquitectura

EoSimulator soporta el paradigma de orientación a eventos en dos y tres fases. Es una librería implementada en C++ y está compuesta de cuatro módulos:

- `Core`: En este modulo se encuentran las clases utilizadas para el modelado de sistemas y para correr simulaciones. Para el modelado de sistemas se tienen las clases `Model`, `BEvent`, `CEvent`, `Entity` y `EntityComparator`. Para correr simulaciones se tiene la clase `Experiment`, que encapsula el ejecutivo de simulación y el calendario.
- `Dist`: Este modulo se encarga del manejo de números aleatorios y el muestreo de distribuciones. Si bien se soportan varios tipos de generadores de números aleatorios, solo se implementa uno. Se proveen las funciones de distribución más comunes.
- `Statics`: Este modulo brinda acumuladores de datos para las variables de salida. Se provee al usuario de histogramas, tanto de series de tiempo como de tiempos ponderados.
- `Utils`: En este modulo se encuentran colecciones de entidades, eventos B, eventos C y distribuciones. Para las colecciones de entidades se tienen colas en orden FIFO, LIFO, y por prioridad definida por el usuario.

7.2. Modelado

Para modelar un sistema en EoSimulator se debe crear una subclase de `Model`. Esta clase encapsula todos los aspectos de un modelo de simulación. Posee los eventos del sistema, las distribuciones utilizadas, las colas de entidades, los recursos, etc. `Model` es una clase abstracta, por tanto se deben implementar métodos para las operaciones abstractas `init()` e `initialize()`. Estas operaciones inicializan el modelo y lo llevan a su estado inicial respectivamente.

Los eventos del sistema pueden ser de dos tipos: fijos o condicionados. Los eventos fijos son aquellos de los que se conoce su ocurrencia. Mientras que la ocurrencia de los condicionados es incierta, esta depende de ciertas condiciones del modelo, típicamente disponibilidad de recursos. Para modelar eventos en EoSimulator se deben implementar subclases de `BEvent` y `CEvent` para eventos fijos y condicionados respectivamente. En ambos casos es necesario codificar un método para la operación abstracta `eventRoutine`, que en el caso de los `BEvents` tiene como parámetro una entidad. En el caso de los `CEvents` esta operación no recibe parámetros ya que actúa sobre el modelo en su totalidad. Otra diferencia entre estos eventos es que los `BEvents` deben poseer un nombre que sirve de clave para agendar entidades. Sin embargo una particularidad que tienen ambos eventos es que no tienen estado, deben ser tomados como funciones que no mantienen un estado. Para que los eventos sean accesibles es necesario registrarlos en el modelo. Se utilizan las operaciones `registerBEvent()` y `registerCEvent()`.

Para poder realizar cambios sobre el modelo, todos los eventos conservan una referencia al modelo al cual pertenecen, accesible a través del atributo protegido `owner`. Para agendar una entidad a un evento se recurre a la operación `Model::schedule` que es invocada con un

puntero a una entidad, un desplazamiento tiempo relativo al tiempo de simulación actual, y el nombre de algún evento registrado.

En contraposición a los eventos y los modelos, las entidades en EoSimulator no son clases abstractas. Poseen una implementación por defecto que es muy limitada en cuanto a atributos. En caso de necesitar más atributos, es necesario crear una subclase de `Entity`.

Para modelar recursos se proveen las clases `NonRenewable` y `Renewable`, que modelan recursos no renovables y renovables.

7.3. Manejo de números aleatorios

Eosimulator está diseñado para soportar varios generadores de números aleatorios. Cada generador brinda un stream independiente. Esto es aprovechado por las distribuciones, que heredan de la clase abstracta `Distribution`. Cada distribución posee un generador, por tanto un stream independiente. Y debido a que en principio no hay una cota para la cantidad de generadores que se pueden crear, no hay un límite para la cantidad de streams de números aleatorios que se pueden utilizar.

También se brinda un manejo centralizado de streams en la clase `Experiment`. Utilizando la operación `setSeed` se pueden setear diferentes semillas a cada generador contenido dentro de una distribución. Para esto es necesario registrar las distribuciones en el modelo, con la operación `registerDist()`.

7.4. Corridas

Para correr una simulación se debe conectar el modelo con una instancia de la clase `Experiment`. Esto se realiza mediante la operación `Model::conectToExp()`. Luego de esto es posible setear semillas para obtener replicaciones independientes. Para correr la simulación se utiliza la operación `Experiment::run()`.

7.5. Conclusiones

EoSimulator es un simulador orientado a eventos. Su arquitectura es bastante simple pero suficientemente funcional como para desarrollar modelos simples. Implementa todas las características básicas que se definieron. Actualmente se encuentra en fase de pruebas, pero se espera que llegue a ser la biblioteca utilizada para dictar cursos de simulación en la facultad.

8. Erlang

Muchos problemas tienen solución más elegante en lenguajes declarativos. Además, a veces este enfoque brinda una comprensión del problema que la programación imperativa no da. Al preguntar en *comp.lang.haskell* sobre trabajos en el área de la simulación a eventos discretos con lenguajes funcionales, una de las sugerencias que se nos hizo fue investigar el lenguaje Erlang.

Erlang es un lenguaje funcional, estricto, tipado dinámicamente y concurrente.

8.1. El lenguaje

Erlang es un lenguaje funcional con soporte intrínseco de concurrencia. Se llama así en honor al matemático Agner Erlang y fue desarrollado por Ericsson en la década de 1980, como resultado de una investigación sobre las facilidades que se necesitaban (o daban problemas) para la implementación de sistemas de telefonía.

Erlang es un lenguaje simple. El tipado es dinámico. Hay números, átomos, tuplas y listas. Las convenciones lexicográficas y parte de la sintaxis se parecen a las de Prolog, ya que Erlang era al principio implementado con un intérprete Prolog. Las variables empiezan con mayúscula y se cuenta con pattern-matching para (des)componer estructuras complejas. Los archivos son una sucesión de funciones, cada una con varias cláusulas. Se realiza pattern matching sobre las cabezas de las cláusulas para saber cual tomar. Erlang es determinístico, así que se toma la primera cláusula posible.

La función reverse en Erlang se escribe así:

```
-module(reversemod).
-export([reverse/2]).

reverse(L) ->
      reverse(L, []).
reverse([], Ac) ->
      Ac;
reverse([H|T], Ac) ->
      reverse(T, [H|Ac).
```

Las facilidades de concurrencia de Erlang se basan en pasaje de mensajes. Rompen con la pureza del lenguaje, aunque se puede argumentar que pasaje de mensajes es la primitiva de comunicación entre procesos más pura. Erlang implementa procesos livianos, a nivel del lenguaje, lo que hace que la escalabilidad sea mucho mejor que en el caso de procesos o threads a nivel del kernel del sistema operativo. Los procesos se crean con `spawn`:

```
Pid = spawn(Module, Function, [Arg1, ...])
```

Los mensajes se envían con:

```
Pid ! Message
```

Y se reciben con:

```
receive
      Pattern1 -> Seq1
      Pattern2 -> Seq2
      ...
end
```

El pasaje de mensajes es asincrónico. Si no hay ningún pattern que se corresponda con los mensajes de la cola del proceso `receive`, se bloquea al proceso. A su vez, ciertos mensajes pueden quedar en la cola porque no hay pattern para ellos pero si para mensajes posteriores.

Erlang brinda ciertas facilidades que no son comunes en otros lenguajes. Es posible remplazar código en ejecución. Así, los procesos que haya corriendo con el código viejo no se modifican pero los creados de ahí en más pasan a usar las versiones nuevas de las funciones. Además, como es imposible que procesos compartan memoria porque las variables son inmutables, es muy fácil distribuir programas sobre varios computadores diferentes.

8.2. Simulación con Erlang

Con las primitivas vistas anteriormente es relativamente sencillo implementar simulaciones basadas en procesos. Los propios procesos de Erlang sirven para modelar los ciclos de vida de las entidades. Para suspender un proceso se le puede dejar bloqueado en un `receive`.

Los objetos de otros lenguajes se pueden simular en Erlang con procesos. La llamada a un método se hace con un par `send-receive`. El estado del objeto se mantiene en una tupla o término más complicado que se pasa de función en función. La optimización de Tail Calls que hace Erlang se encarga de que el espacio ocupado por el objeto-proceso no crezca desmesuradamente. Se incluyen `receive`s para atender las llamadas a métodos de clientes. Así, los recursos y el ejecutivo se vuelven procesos en sí mismos.

Como las entidades corren en procesos, no corrutinas, es posible que lo hagan en simultáneo sin mayores modificaciones.

8.3. Conclusiones

Se puede argumentar que la concurrencia en Erlang rompe con la pureza declarativa del lenguaje. Hay que tener en cuenta el estado de los procesos con los que se interactúa, que pueden progresar independientemente. Sin embargo, la diferencia con los lenguajes imperativos corrientes también es notable. La programación de cada uno de los procesos si es declarativa, aunque su interacción no lo sea.

Erlang es destacable como lenguaje para la implementación de simulaciones porque es un lenguaje sencillo pero poderoso, con las primitivas justas que se necesitan. El pasaje de parámetros y los varios procesos alcanzan tanto para la implementación de las entidades, que son activas, como de los otros objetos pasivos pero con estado.

9. Referencias

[BCNN00] BANKS, Jerry; CARSON, John S. II; NELSON, Barry L.; NICOL, David M. Discrete-event system simulation. Upper Saddle River, NJ: Prentice Hall, 2000. p.594. ISBN: 0130221021

[DO89] DAVIES, Ruth M.; O'KEEFE, Robert M. Simulation modelling with Pascal. Hertfordshire: Prentice Hall, 1989. p.302 ISBN: 0-13-811571-0

[HGB00] HARRELL, Charles; GHOSH, Biman K.; BOWDEN, Royce. Simulation using ProModel. Boston, MA: McGraw-Hill, 2000. p.603. ISBN: 0072341440

[LM93] LITTLE, M. C.; MCCUE D. L. Construction and use of a simulation package in C++. Computing Science Technical Report, July 1993. University of Newcastle upon Tyne; 437.

[CXS05] C++SIM Home Page

http://cxxsim.ncl.ac.uk Junio 2005

[DMJ05] DesmoJ Home Page

http://www.desmoj.de/ Abril 2005

[EOS05] EOSimulator Home Page

http://www.fing.edu.uy/inco/cursos/simulacion/paginaEosim/index.htm Julio 2005

[SPY05] Simpy Home Page

http://simpy.sourceforge.net/ Julio 2005

[AVW92] ARMSTRONG, J.; VIRDING, S.; WILLIAMS, M. Use of Prolog for Developing a New Programming Language. London, England: The Practical Application of Prolog, April 1992.

[Erm95] ERMEDAHL Andreas. Discrete Event Simultation in Erlang. Sweden: Uppsala Master's Thesis, 1995.

[Eks00] EKSTROM, Ulf. Design Patterns for Simulations in Erlang/OTP. Sweden: Uppsala Master's Thesis, 2000.

[ERL05] Erlang Home Page

http://www.erlang.org Junio 2005

Apéndice C - Técnicas de diseño e implementación

1. Introducción

En este apéndice se tratan diversas técnicas, patrones y modismos del lenguaje C++. Por diversas razones, generalmente por estar acostumbrados a un método tradicional de elaborar programas, en el ámbito local no se tienen muy presentes estas técnicas, y una debida comprensión del trabajo realizado requiere cierta familiaridad con los conceptos.

Distinguir cual es la parte de diseño y cual de implementación de estas técnicas es difuso, pues los conceptos se mezclan e intersecan en varios puntos. No son puras de diseño porque traen consigo métodos para implementar las mismas, y no son puras de implementación pues la óptica desde la que se observan no es la algoritmia ni la eficiencia sino la representación de abstracciones y elementos de diseño.

Pese a que las ideas a continuación trascienden la metodología y el lenguaje, se prestó especial atención a los aspectos relacionados con orientación a objetos y C++.

2. Compilation Firewall

Uno de los anacronismos más grandes del modelo de archivos de C++ son los archivos de cabecera. Un gran problema de los archivos de cabecera es que al incluir uno se incluyen también todos los que éste incluía a su vez. Así se incrementan los tiempos de compilación, porque tiene que rehacerse cuando cambian no solo las dependencias, sino las dependencias de las dependencias. La técnica llamada Compilation Firewall [Sut99] da una solución parcial a este problema.

2.1. Idea

Técnicamente, los archivos de cabecera no forman parte de la especificación de C++, pero aunque existan implementaciones que no los tengan, son una realidad en el trabajo cotidiano con C++ o C. El que ve los diferentes archivos fuentes, de cabecera y de implementación, es el precompilador. Al compilador le llega una sucesión de tokens que es llamada *Translation Unit*. Por motivos históricos, gran parte de C++ está diseñado para que pueda ser procesado por un compilador de una sola pasada, por eso existen los conceptos separados de definición y declaración. Para muchos usos de un símbolo alcanza una declaración, y la definición puede aparecer luego. En el caso de una función, para llamarla alcanza con tener una declaración y la definición puede incluso estar en un fuente independiente. Es el linker el que resuelve esta dependencia. Incluso si todo el programa está en un solo archivo y le llega al compilador una sola *Translation Unit* es necesario usar declaraciones de funciones para el caso de funciones mutuamente recursivas.

Para el caso de clases y estructuras también existen declaraciones. Mientras no haya definición se dice que el tipo es incompleto. Las declaraciones de clases se usan menos que las declaraciones de funciones, porque no hay problema con tener la misma clase definida en dos *Translation Units*, así que se ponen en archivos de cabecera. Del mismo modo que para las funciones, sin embargo, para ciertos usos es indispensable las declaraciones, como clases que se conocen mutuamente.

Una definición es una declaración, pero las dos no se dan en el mismo punto. Una función o clase está definida cuando termina su definición, pero está declarada cuando comienza. Por esta razón no se requiere trabajo extra para funciones recursivas o nodos de listas encadenadas. El uso de una clase cuando se cuenta solo con una definición está limitado. Como no se sabe el tamaño, no se pueden crear objetos de la misma, pero si se pueden formar punteros a la clase. También pueden introducirse declaraciones para funciones que reciban o devuelvan objetos de esa clase, aunque no podrán definirse hasta que también esté la definición de la clase.

La idea central del Compilation Firewall, o muro de fuego de compilación, es limitar la cantidad de archivos de cabecera que se incluyen en el archivo de cabecera de una clase, usando mientras sea posible solo declaraciones de las clases de las que se depende, para cortar con la cadena de inclusión. En lugar de incluir el archivo donde está la definición de una clase, se hace una declaración adelantada de la misma. La ganancia no está tanto en el texto que se evita procesar por la clase en cuestión, sino el de todas sus dependencias a su vez. En el peor de los casos, un archivo de cabecera termina incluyendo por transitiva a todos los otros en una aplicación, por lo que se requeriría volver a compilar todos los fuentes si cambia cualquiera de los archivos de cabecera.

2.2. Aplicación

Para convertir una clase al modismo de Compilation Firewall hay que analizar el uso de cada una de las otras clases que se hacen desde el archivo de cabecera. Mediante sucesivas

refactorizaciones del código se logra ir achicando la cantidad de dependencias explícitas que se tienen.

El primer paso es asegurarse de que cada uno de los archivos de cabecera que se incluyen es auto contenido, y de que no se incluyen archivos de los que no se usa ningún símbolo. Parecería tonta esta situación, pero la realidad es que en una base de código en evolución es casi garantizado que algún cambio en una decisión de diseño o implementación vuelva sin necesidad la inclusión de un archivo. Sin prestar mucha atención, no es posible darse cuenta si un cierto cambio es el que elimina el último uso de una clase o función de cierto módulo. Existe por el contrario una manera fácil de mantener los archivos fuente auto contenidos, que consiste en incluirlos primero que nada en el archivo fuente correspondiente.

Por cada otra clase se buscan los usos que requieran la definición y se trata de moverlos al archivo de implementación. Un tal uso podría ser en la implementación de una función en línea, que sería fácil de subsanar haciendo que la función deje de ser en línea. Cómo comúnmente la velocidad de compilación importa más durante el desarrollo, donde la velocidad del programa no es tan importante, se puede idear un esquema tal que dependiendo de un parámetro de configuración las funciones sean en línea o no. Para esto se crea un nuevo archivo, donde van estas funciones, y mediante el uso del preprocesador se dispone para que el archivo en las compilaciones de depuración se incluya en el archivo de implementación y las funciones no se marquen inline, y en las compilaciones de liberación a la inversa. Algunos compiladores ofrecen facilidades para la optimización global de los programas, por lo que estos malabarismos se pueden evitar ya que coloca las funciones en línea si lo considera apropiado aunque no se las haya marcado así y la definición no esté disponible.

Otro caso que aparece es que una clase es argumento para la instanciación de un template. El comportamiento en este caso depende del componente genérico en cuestión. En particular, un programa presenta comportamiento indefinido si se instancia algún componente de la librería estándar de C++ con un argumento de tipo incompleto. Este comportamiento indefinido se presenta en las estrategias comúnmente utilizadas por los compiladores como una falla de compilación, así que no hay que preocuparse por un comportamiento oscuro en esta situación. Además, las librerías de los compiladores de Microsoft y GNU sí permiten el uso de componentes genéricos aunque sus argumentos no estén definidos, mientras la definición aparezca antes de que sea de veras requerida. Los componentes de boost documentan individualmente su comportamiento en este caso. Boost.SharedPtr requiere una definición solo al momento de la asignación de un apuntado, y capturan este conocimiento internamente para lograr flexibilidad. Boost.ScopedPtr, por el contrario, requiere la definición al momento de la destrucción. Destruir un objeto cuando el tipo de éste es incompleto trae comportamiento indefinido, por esto la restricción en el uso de ScopedPtr. Igualmente, la implementación de Boost se previene de un uso no intencionado mediante aserciones estáticas.

El caso más difícil de erradicar es la presencia de un objeto de otra clase como miembro de la clase que está siendo examinada. La idea es entonces convertir este uso en alguno de los otros. Para esto lo que se hace es en lugar de incluir una instancia del objeto, hacerlo solo con un puntero. Como pueden haber varios miembros en esta situación, se inventa una clase nueva que los contenga, y a ésta se la accede mediante un puntero. Así, la clase original reduce sus miembros a uno solo, que contiene a los originales. La clase nueva se declara en el archivo de cabecera de la clase que se está examinando, pero se define recién en un archivo de implementación. Así, mediante un simple truco, se reduce notablemente la cantidad de archivos que deben incluirse para conformar el archivo de cabecera y las dependencias son mucho más holgadas.

La clase que se crea en este modismo por lo general se declara dentro de la clase original, para no añadir símbolos al espacio de nombres que contiene a la clase original y reducir los conflictos. Como no hay dos clases de compilation firewall para cierta clase, se le da el nombre

de `impl`, y al puntero, `pimpl`, por puntero a la implementación. Este es otro nombre por el que es conocida esta técnica, bastante nemotécnico e incluso gracioso para los angloparlantes.

El puntero `pimpl` puede ser a su vez un puntero inteligente, como Boost.SharedPtr o Boost.ScopedPtr. No puede ser auto_ptr, de la librería estándar por la restricción de que no sea instanciada con tipos incompletos, que es precisamente lo que se quiere hacer. ScopedPtr ofrece la semántica más apropiada para este tipo de clases pesadas, porque evita la copia, la transferencia y que se comparta lo apuntado. SharedPtr puede servir si compartir la implementación es lo que se quiere.

3. Programación con funciones

Una de las características más sobresalientes del C++ moderno es la cada vez mayor influencia de paradigmas funcionales. La punta de lanza de este movimiento es la STL, y se encuentran ejemplos en Boost y otros lugares. Aunque no se entrará en mayor detalle sobre el paradigma funcional, se reseñará brevemente las maneras principales en que se implementa sobre C++.

3.1. Representación de funciones

Los lenguajes funcionales brindan gran sencillez, poder y elegancia al trabajo con funciones. Se pueden recibir funciones como argumento de otras, e incluso devolver una función desde otra. En C++ las únicas operaciones válidas sobre funciones son la invocación y tomar su dirección. A su vez, los punteros a función pueden usarse para invocar la función original o compararse entre si, con ciertas restricciones. En C esto es todo lo que se puede hacer, y para imitar estilos funcionales se usan comúnmente trucos como pares de punteros a función y a objetos de tipo void, con la convención de que uno de los argumentos de la función es el otro objeto. Así, se pierde totalmente el beneficio de la abstracción y la seguridad de los tipos, aunque es de destacar que esta solución brinda el poder suficiente para implementar todo lo que se desee, aunque no con la comodidad esperable.

En un ámbito orientado a objetos la solución a este mismo problema se puede encontrar por varios nombres, llamados patrones, como Estrategia, Comando, Observador, etc.. Aunque apuntan a objetivos diferentes, todos estos patrones tienen una solución similar, que pasa por la invención de una clase específica, con un método con varias implementaciones en las subclases. Esta solución como cohesiona el estado y la función no tiene el problema de falta de tipado, pero se vuelve engorrosa la creación de infinidad de clases derivadas para diferentes situaciones. Hay un problema de interoperabilidad entre las diferentes clases bases, ya que aunque el método principal tenga la misma firma, son dos clases separadas e incompatibles.

En C++ estos problemas tienen una solución por convención. Gracias a la posibilidad de sobrecargar el significado del operador paréntesis se puede invocar cualquier objeto. Con esto se estandariza el nombre del método. Una vez logrado esto, es posible implementar funciones de alto orden para adaptar una función, por ejemplo fijándole algunos de los argumentos. La librería estándar de C++ ya trae de este tipo de adaptadores, las funciones bind1st y bind2nd. Permiten convertir una función binaria en una unitaria, proveyendo el otro argumento. Boost tiene utilidades para lograr el mismo cometido mucho más directa, fácil y flexiblemente, que por suerte verán aparición en la próxima edición del estándar de C++.

Técnicamente es posible implementar también adaptadores de este estilo para functores representados con la técnica orientada a objetos, pero la multitud de clases bases incompatibles hace que la empresa no tenga una buena relación costo beneficio cuando una solución funciona con una sola de las clases. Esta apreciación trae un cuestionamiento sobre la implementación convencional en C++; en ningún punto se mencionó que tipo toman estos objetos. La solución a esta paradoja es que no se requiere ningún tipo en especial. Como los templates trabajan sobre el protocolo, la interfaz sintáctica, de sus argumentos, no es necesario que dos objetos tengan el mismo tipo para ser compatibles. Todos los algoritmos de la librería estándar que son configurables en parte mediante un functor recibido, por ejemplo, lo toman como un objeto de un tipo de sus argumentos templates. Gracias a que este tipo se deduce automáticamente por C++, el usuario no se ve infortunado con esas preocupaciones, y todo fluye sin mayores inconvenientes.

Que los tipos no sean determinados sirve cuando los functores creados se usan para invocar funciones. Cuando los functores se devuelven, eventualmente se choca con un punto en que el

tipo debe ser borrado y ajustado a una clase conocida y preestablecida, o sino no se logra compilación separada. Esto se puede resolver con el método orientado a objetos. Se crea una clase que hereda de la clase base que sea y que delegue al functor del tipo arbitrario que implementa la funcionalidad requerida. Gracias nuevamente a los templates, esta clase puede ser implementada una sola vez para cada clase base y no para cada combinación de clase base y functor, así que el costo no es tan oneroso.

3.2. Aplicaciones

La aplicación más notoria es por lejos la STL. En este caso los functores ven dos usos. En el primero son argumentos para algoritmos, como por ejemplo el criterio de parada para `find_if`. Además de este predicado, `find_if` recibe también el rango de elementos a examinar. Existen restricciones entre los tipos de los miembros del rango y los argumentos del predicado, pero no hay que decirlos porque serán develados por el compilador al momento de producirse la instanciación del template.

El otro uso es para configurar los propios objetos de la librería. Por ejemplo, la clase `map`, que representa un conjunto de pares de claves y valores ordenada por clave se puede configurar para que use una lógica de ordenación alternativa al comparador de menor de la clave. Por ejemplo, si la clave es una cadena de caracteres existen diferentes ordenaciones regionales que consideran caracteres especiales como los acentuados en distintas posiciones relativas a los demás, por lo que la ordenación lexicográfica por el valor numérico del carácter no es la más apropiada. En este caso se debe explicitar el tipo del functor en cuestión, porque este a su vez determina el tipo de todo el mapa. En este caso lo más común es escribir una clase a medida para la aplicación, para minimizar toda posibilidad de pesimización por parte del compilador.

En Boost existen más componentes que hacen uso de este enfoque. Boost.Signal es una librería para disparo y registro de eventos. Como se tienen varios eventos de tipos diferentes es indispensable borrar esta diferencia, que se hace con la técnica de mapeo a orientación a objetos ya mencionada. Internamente se usa Boost.Function para esto, aunque la elección de esta estrategia también es configurable mediante argumentos template. Por fuera se incentiva el uso de Boost.Bind para confeccionar los functores que se pasan a Boost.Signal, pero su uso no es obligatorio y las herramientas pertenecen mayoritariamente desacopladas. libsig++ es el competidor de Boost.Signal, y componente de manejo de eventos de varias librerías de interfaz de usuario, notoriamente GTKmm. La arquitectura es muy similar, aunque los componentes están más acoplados entre sí.

4. Diseño basado en políticas

Diseñar librerías de componentes reutilizables es más exigente que escribir componentes para una aplicación en particular. Los posibles clientes de la misma tienen diferentes necesidades, que en muchos casos son encontradas, e incluso si pueden abarcarse simultáneamente vuelven la interfaz de los componentes demasiado extensas e intratables. Históricamente se esperaba la solución a estos problemas mediante el mecanismo de herencia múltiple, pero luego se vio que no alcanzaba. El diseño basado en políticas revive este enfoque apoyado en técnicas de implementación novedosas que no estaban disponibles antes.

4.1. Descripción

Existe una regla general para el diseño de clases en cuanto a la encapsulación, que dice que tiene que haber por lo menos dos maneras de implementar una cierta interfaz. En general, y sobre todo para un escritor de librerías, esto es más que cierto. El desafío consiste en controlar todas estas opciones. Por lo general existen varias dimensiones de variabilidad. La idea detrás del diseño basado en políticas es identificar estas dimensiones y brindar diferentes implementaciones para cada una, de tal manera que muchas combinaciones se puedan obtener sin mayor esfuerzo. Las dimensiones deben ser lo más ortogonales entre si como sea posible. En caso contrario, dos políticas interactúan entre si de manera indeseable, y se pierde seguridad en la composición. Se puede pensar que la separación en políticas es un caso particular del diseño orientado a aspectos.

Idealmente, sería posible combinar el comportamiento de clases mediante herencia múltiple. La herencia múltiple sirve para refinar el comportamiento de una clase partiendo de un modelo. Entonces debería ser posible usando herencia múltiple lograr una clase que refine simultáneamente varios comportamientos dispares. Por ejemplo, una clase que sea una lista encadenada con mecanismos de sincronización frente al acceso concurrente de varias threads sería obtenible mediante la creación de una nueva clase, `ListaConcurrente`, que herede tanto de `Lista` como de `Concurrente`. El mecanismo de composición usado por la herencia múltiple es fijo y no demasiado intrincado. Incluso C++, que ofrece grandes mejoras en potencia frente a lenguajes posteriores como Java y C#, no puede expresar toda la gama de posibilidades. Si bien la herencia múltiple es una de las herramientas necesarias para implementar un diseño basado en políticas, por si sola no basta para realizar la composición deseada.

La otra pata sobre la que se apoya el diseño basado en políticas es el mecanismo de templates de C++. Una de las características soportadas por los templates en C++ y que no existe en otras facilidades similares de otros lenguajes es la especialización. La especialización de un template remplaza su definición original por una nueva, cuando los argumentos son los indicados. Por ejemplo, se puede especializar un contenedor cuando el argumento es un booleano para que aplique compresión de varios booleanos en una misma palabra. Entonces en lugar de ocupar un byte para cada elemento, se hacen caber ocho elementos en un mismo byte. A su vez, una especialización puede ser un template a su vez. Esto sucede cuando se especializa para un conjunto de tipos al mismo tiempo. Un ejemplo puede ser nuevamente la especialización de un contenedor, ahora para cuando los elementos son punteros. Como los punteros se comportan todos igual no importa su tipo, una misma implementación funciona para todos, lo que el tamaño del código objeto final, que si no tiene partes repetidas por cada instanciación del contenedor para un tipo puntero. La especialización se puede aplicar a una función miembro de una clase o a toda la misma, pero no a una parte nada más. Además, de por si el mecanismo no ofrece la posibilidad de que la librería ofrezca varias opciones al usuario; existe una original que el usuario puede cambiar, pero no varias originales entre las que elegir.

4.2. Implementación

La manera en que el diseño basado en políticas combina la herencia múltiple y los templates se basa en la posibilidad de herencia paramétrica que ofrece C++. Esto quiere decir que una clase template puede heredar de alguno de sus argumentos. Entonces una vez que se identifican todas las dimensiones de variabilidad se crea una clase que recibe un argumento, una política más específicamente, por cada dimensión y hereda de todas ellas. De esta manera las políticas inyectan miembros en la clase original, que se ve enriquecida de acuerdo a que argumentos le son pasados.

El ejemplo original de la `ListaConcurrente` podría modelarse entonces como un diseño con dos políticas. La primera política indicaría que contenedor subyacente utilizar. Posibles opciones pueden ser una lista encadenada, un vector o alguna especie de árbol. Existe un contrato que define este tipo de política y especifica los miembros que debe contener. La segunda política variaría en el rango de las posibles elecciones en cuanto a protección contra el uso concurrente. Implementaciones para esta dimensión serían ninguna protección o alguna protección implementada con cierta librería de concurrencia. Incluso podrían pensarse políticas que ordenen el acceso con diferentes grados de equidad entre las threads.

De alguna manera, las políticas son estrategias, en el sentido del patrón Estrategia. La diferencia está en que no hay polimorfismo en tiempo de ejecución en la solución con políticas. Todas las funciones se resuelven en tiempo de compilación, y el compilador no tiene impedimentos para crear código objeto rápido y compacto, ya que tiene acceso inmediato a todas las partes. Además, como la composición es estática puede agregarse información de tiempo de compilación, como tipos y funciones miembros extras. En el ejemplo anterior, una política que implemente acceso concurrente puede, además de las funciones que está obligada a suministrar, agregar nuevas, que también son heredadas y accesibles desde afuera de la composición. Lo mismo sucede para las declaraciones de tipos dentro de las políticas. Esto puede servir para exponer una interfaz que permita mantener la unicidad del acceso a través de varias operaciones, con lo que bastaría definir un tipo para el candado correspondiente y una función que lo cree.

En general las políticas no alcanzan con que sean simplemente clases. Por ejemplo, una política sobre la destrucción de los elementos de un contenedor depende de cual sea el tipo del elemento contenido. Es plausible que gran parte de las políticas necesiten esta información de hecho. Si las políticas son pocas, se pueden escribir políticas que sean a su vez templates y componer la clase de varias políticas ya instanciadas. Esto es repetitivo porque los argumentos de las mismas son siempre el elemento contenido en la colección. Una mejora es pasar la clase template sin instanciar y que la clase que se está componiendo las instancie con el elemento en cuestión al momento de heredar. Una tercera solución es pasar clases que tengan clases template dentro de ellas. Esta solución es la más complicada de hacerse a mano, pero es la que mejor escala si se aprovecha la infraestructura de metaprogramación ya existente en Boost.

5. GenVoca

GenVoca es una metodología de diseño de software basada en la programación orientada a objetos, pero ampliándola con capas. Una capa contiene un conjunto de clases, un componente, que se construye sobre las clases de las capas inferiores, brindando servicios a eventuales capas superiores. GenVoca procura separar cada capa para poder luego combinarlas de maneras diferentes sin esfuerzo desproporcionado. Es una técnica orientada a aspectos.

5.1. Diseño

En GenVoca cada característica de un sistema obtiene su propia capa, donde es encapsulada. Las capas son objetos de primera clase para la metodología. Una instanciación de un sistema elige un conjunto de capas y las combina. Cada capa es parametrizada por las capas que tiene debajo, y a su vez es parámetro para otras. Un sistema concreto se describe mediante una expresión lineal de los tipos de las capas en cuestión.

Se puede pensar que cada capa es una máquina virtual que expone ciertas instrucciones a sus clientes, y que para funcionar a su vez necesita de máquinas subyacentes. Los frameworks en las librerías orientadas a objetos son parecidos, pero tienen ciertas desventajas. En un framework hay generalmente dos capas de clases, una capa que pertenece a la librería y otra que es de clases creadas por el usuario. En una librería de simulación, por ejemplo, la librería proveería clases para los eventos, entidades, histogramas, etc., y el usuario heredaría de las mismas para introducir el comportamiento apropiado a sus necesidades.

El primer problema en este caso es que mientras las clases de la librería están relacionadas explícitamente, lo mismo no se da entre las clases derivadas. Las clases derivadas se comunican a través de las bases, así que se pierde información a nivel estático. Esto trae desventajas de performance, porque la elección de las operaciones se hace en runtime mediante despacho dinámico, y de comodidad, porque las referencias a objetos deben convertirse a sus clases de usuario a partir de clases de librería.

En segundo lugar, no es posible intercambiar capas en un framework. Las clases tienen fijas cuales son sus clases base, por lo que una vez que se plasmó en el código un diseño estratificado luego no puede ser deshecha la combinación. Esto se conoce como el problema de escalabilidad de las librerías. Dadas 4 características a implementar, de tal manera que una característica puede o no estar, existen 16 combinaciones. Mediante herencia se puede aprovechar la implementación de un prefijo de las características deseadas agregando solo la nueva, pero una vez que se desea remover una característica, hay que empezar una jerarquía nueva.

Cada capa en GenVoca tiene un tipo, y hace requerimientos sobre los tipos de sus parámetros. Como una capa es un conjunto de clases, su tipo son sus miembros y los tipos de éstos. Existe, al igual que en la orientación a objetos la oportunidad de realizar este tipado de varias formas. Por ejemplo, si dos capas con los mismos miembros son directamente compatibles, la situación se asemeja a los lenguajes dinámicamente tipados, donde el protocolo de un objeto determina su interfaz y la manera en que sus clientes acceden a sus métodos. El polimorfismo estático mediante templates en C++ funciona de la misma manera. Otra posibilidad es hacer que las capas declaren explícitamente su interfaz, de la misma manera que una clase declara sus clases bases, y los objetos de la misma pueden ser accedidos mediante referencias a objetos de la base. Lo primero es más permisivo, pero vuelve compatibles entidades que por cuestiones semánticas no deberían serlo.

Ciertas capas tienen la propiedad de ser de un tipo dado pero también tomar un parámetro de ese tipo. Se dice que son capas simétricas. Lo bueno que tienen es que se pueden intercalar

junto a una capa de ese mismo tipo en cualquier expresión de tipos sin violar el tipado de la misma. Un ejemplo de componentes simétricos son los filtros de la línea de comandos de Linux. En la consola se puede configurar una secuencia de programas para que la salida de uno sea la entrada del siguiente. El orden de los programas puede ser cualquiera, porque todos tienen la misma interfaz y son simétricos. Conociendo unos pocos comandos es posible configurarlos de muchísimas maneras, para lograr variados objetivos. Un orden apenas diferente puede lograr resultados muy diferentes, tanto en semántica como en performance, pero son todos válidos.

Una generalización al planteo visto es permitir que las capas tomen varias otras como parámetros. Así, en lugar de una estructura lineal se obtiene un árbol como la expresión de tipos de un sistema. La versión lineal tiene la ventaja de que es más fácil de comprender y de diagramar en una documentación, pero la arborescente es más potente. Adicionalmente, cada capa puede estar parametrizada por fuera de lo que es la propia estructura de GenVoca. Por ejemplo, una capa puede recibir un argumento con el cual configurar cierto aspecto del manejo de memoria sin que eso repercuta en el resto de las capas. Esto se puede pensar como que en lugar de capas se tiene una función que devuelve capas.

Queda por responder como aplicar estas ideas sin romper por completo con un modelo orientado a objetos. Mediante las técnicas vistas, se puede aplicar un refinamiento a un conjunto de clases simultáneamente. En un diseño orientado a objetos se dice que varios objetos están dentro de una colaboración, y mediante un protocolo de mensajes realizan una labor. Dentro de una colaboración cada objeto presenta un rol. La clase de un objeto es la unión de todos los protocolos requeridos por todos sus roles en distintas colaboraciones. Bajo esta óptica, GenVoca permite expresar separadamente las colaboraciones y por lo tanto los roles de los objetos. La combinación de todos los roles en una clase final la hace el compilador, lo que es un caso de enlace tardío, análogo al despacho dinámico pero a otro nivel. La separación permite la creación, mantenimiento y actualización de roles sin impactar los otros roles de una clase, mejorando la modularidad del programa.

5.2. Implementación

Así como hay una gran variedad de formas de estructurar un lenguaje orientado a objetos, existen diferentes maneras de implementar GenVoca. Varios lenguajes especializados existen, que adaptan algún lenguaje popular con construcciones específicas para la realización de diseños hechos con GenVoca. Esto se ha hecho para C++ y Java [BLS98], por ejemplo. La ventaja de una herramienta especialmente realizada con este cometido es que los conceptos se plasman directamente en el código, y el entorno puede ofrecer mucho mejor soporte para el programador, como mensajes de errores y chequeo de tipos. La desventaja es que se debe modificar la cadena de herramientas, y las herramientas de propósito específico no congenian con las de soporte, como debuggers, profilers y otros.

Una segunda opción sería un lenguaje dinámico, en el que las clases existan como entidades de primer orden. Así, la combinación de las capas en un sistema concreto se realizaría en tiempo de ejecución, mediante técnicas de meta programación. Lenguajes con las características adecuadas para una solución de este carácter son Ruby, Python, Smalltalk y CLOS, entre muchos otros.

La primera desventaja de este enfoque es que al realizarse la composición en tiempo de ejecución, no se pueden realizar muchas de las optimizaciones necesarias para equiparar en eficiencia una solución con un alto grado de modularización como es GenVoca a una monolítica como es la orientación a objetos tradicional en comparación. A este problema se lo denomina comúnmente penalidad por abstracción. Seguramente con el soporte de un entorno optimizado para un uso como el que se le pretende dar estas dificultades podrían ser

subsanadas, mediante técnicas de optimización como son los compiladores justo a tiempo o similares.

Otro problema es que se pierde la colaboración del compilador para determinar la correctitud de un programa antes de ser ejecutado. Según los que promueven el uso de este tipo de lenguajes, el tipado dinámico es una característica positiva, que permite mayor flexibilidad y productividad al programador. La correctitud debe ser testeada mediante pruebas unitarias de funcionamiento de componentes. El uso de un modelo estratificado como GenVoca ayuda al testeo porque ya se tiene subdividido el programa en unidades separadas desde el principio, con lo que hay que instrumentar menos para probar funcionalidades. A su vez, quizá la granularidad sea demasiado poca para que los problemas surjan en las corridas.

5.3. Implementación en C++

C++ posee también facilidades para la meta programación. A diferencia de otros lenguajes, fueron descubiertas por accidente y son consecuencia de la creciente flexibilidad que se hizo necesaria para la implementación de librerías genéricas. Aunque tiene sus lados filosos, la meta programación con templates hoy en día se encuentra suficientemente entendida como para ser utilizada en aplicaciones reales. Existen libros acerca del tema, y un creciente volumen de conocimiento vertido en foros de noticias e implementaciones concretas de librerías [AG04].

Gráficamente, un sistema estratificado se representa como en la figura C.1:

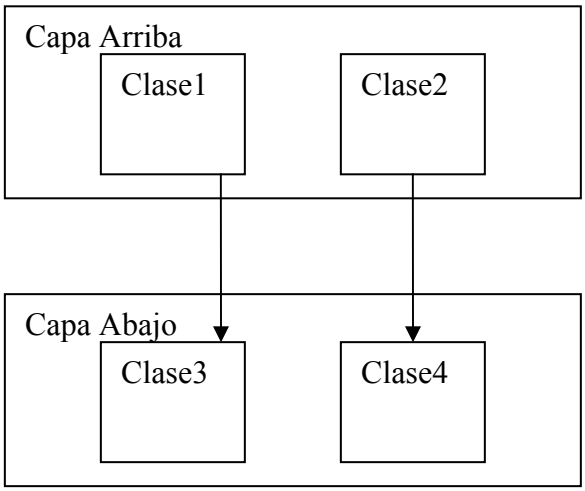


Figura C.1: Sistema en capas

Sin embargo, se quiere expresar cada capa por separado. En la figura anterior la capa de arriba referencia explícitamente a la capa de abajo. Se está en el mismo caso de los diseños estratificados comunes hoy en día en los sistemas orientados a objetos, aunque se favorece la composición a la herencia ahí. Para poder expresar la figura siguiente apropiadamente es necesario hacerse de algunas facilidades del lenguaje C++ que no son de uso cotidiano.

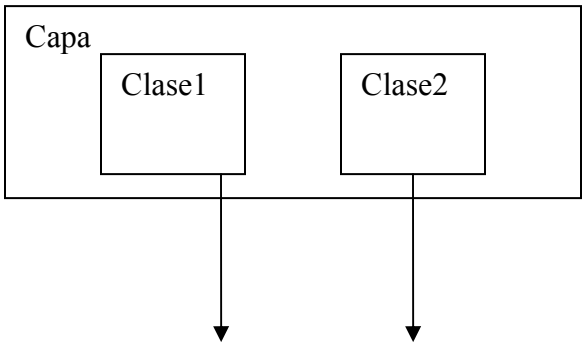


Figura C.2: Capa Mixin

Pensando en cada una de las clases que componen una capa, lo que sucede es que no se tiene fija la clase correspondiente de la capa de más abajo porque la propia capa de más abajo no está determinada. Existe en la orientación a objetos una construcción que cumple con lo que se necesita, llamada mixin. La idea de un mixin es hacer simétrica la idea de subclase y clase base. Comúnmente la clase base no conoce a las clases que heredan de ella, pero si se especifica la clase base al definir la derivada. En C++ un mixin se puede realizar mediante templates, heredando de un tipo parámetro:

```
template <class SuperClass>
class SubClass : SuperClass {
};
```

Ésta es una particularidad del enfoque de genericidad que ofrece C++. En Java y C#, que hace poco tiempo ofrecen construcciones parecidas a templates lo anterior no es posible. En Eiffel, un lenguaje que precede a C++ en la materia tampoco se puede.

Nótese que por las reglas de búsquedas de nombre en dos fases dentro de los templates, el acceso a un miembro de la clase base dentro de un mixin es más complicado de lo que sería hacerlo si no se tratara de un template. Supóngase el siguiente caso:

```
int bar;

template <class SuperClass>
class SubClass : SuperClass {
      void foo() {
            bar = 0;
      }
};
```

Dependiendo de la definición de `SuperClass` con la que sea instanciada, `bar` puede querer decir cosas diferentes. En un arreglo de clase-subclase común, y bajo las reglas normales de C++, el `bar` en la superclase tiene preferencia sobre la global. Sin embargo, es muy inconveniente que el `bar` dependa de la `SuperClass` en cuestión. Este problema es muy parecido al de los símbolos higiénicos en lenguajes de la familia de los Lisp, como Scheme, lo que no sorprende dado el carácter símil macro expansión que tienen los templates muchas veces. La solución pasa por tener que especificar claramente uno de los usos, y hacer que el otro sea el predeterminado. En este caso es más seguro decantarse por tomar los símbolos ya

definidos al momento de la definición de los templates. Cuando se quiere utilizar un símbolo de la superclase cuando esta es un parámetro template ha de usarse la notación `this->bar = 0` o `SuperClass::bar = 0`.

Otro problema a resolver es la elección de la representación para un componente. Usualmente un componente se encapsula en un namespace en C++. La construcción análoga en Java, el paquete, incluso otorga privilegios especiales de acceso entre las clases de un paquete. Pero en ninguno de los dos casos es posible manipular un paquete como se desearía. Por ejemplo, no es posible que un namespace en C++ actúe de parámetro para un template.

Una salida para este caso es agrupar todas las clases de un componente dentro de una otra clase en lugar de un paquete. En principio esto tendría la gran desventaja de que todas las clases de un componente deben definirse dentro del mismo archivo físico, lo que traería enormes problemas de mantenimiento. Esto es reconocido en el lenguaje C#, por ejemplo, que en su versión 2.0 provee facilidades para definir clases parciales, que son una sola clase que puede tener su definición repartida en varios fuentes. En C++ una solución que se presenta es separar la clase-componente en varias clases para luego unirlas mediante herencia múltiple. Como se trata de un recurso puramente sintáctico y no se usarán de ninguna manera las superclases que componen la clase-componente, no se viven ninguna de las contrariedades comúnmente asociadas con la herencia múltiple. Otra posibilidad es definir las clases miembro de un componente separadamente e importarlas a la clase-componente mediante la utilización de la facilidad de alias de tipos de C++, el typedef.

La segunda figura puede entonces ser realizada como en el siguiente trozo de código:

```
template <class CapaAbajo>
class CapaArriba {
  class Clase1 : CapaAbajo::Clase3 {
  };
  class Clase2 : CapaAbajo::Clase4 {
  };
};
```

Hay dos apreciaciones para hacer en este punto. Actualmente no se cuenta con capas genuinamente reemplazables y opcionales. La razón es que `CapaArriba` codifica internamente los nombres `Clase3` y `Clase4` de la capa subyacente con la que será instanciada. Normalmente cada capa traduce lo que en una etapa de diseño más temprana es una colaboración, y cada mixin dentro es el rol que juega alguna clase en la colaboración, como ya fue visto. Por lo tanto no es natural que se pongan nombres diferentes a los roles que conformarán una misma clase. Lo más común es estandarizar en que un rol lleva el nombre de la clase sobre la que aplica. Nombrar clases con nombres aplicativos no es nada nuevo, visto el uso de las interfaces en Java, por ejemplo. El nombre de la capa si puede, y es mejor que sea el de la colaboración que representa.

El segundo punto es volver a notar el uso de herencia para componer los roles. La herencia está un poco mal vista cuando hay alternativas, y se prefiere la asociación o la composición si es posible. Para los roles de una clase no es tan fácil esa decisión, porque se vuelve muy pesada la cantidad de funciones que deleguen a mantener. Incluso, como no se sabe los roles que se tiene por debajo, es imposible delegar a funciones de las que no se conoce su existencia. Ahora, no todas las colaboraciones son sobre el mismo grupo de clases, aunque se puede esperar cierta intersección. Pero entonces, una capa cuya colaboración no involucre a cierta clase no la expone a las capas superiores, para que estas sigan su refinación. La solución es hacer que las propias capas sean mixins. Esto es bastante simple de ver en el

código, pero difícil de plasmar efectivamente en un diagrama. El acercamiento se conoce con el nombre de Mixin Layers. Con estas dos mejoras, el ejemplo de trabajo se ve así:

```
template <class CapaAbajo>
class CapaArriba : CapaAbajo {
  class Clase1 : CapaAbajo::Clase1 {
  };
  class Clase2 : CapaAbajo::Clase2 {
  };
};
```

Hay que prestar atención a cuales son las capas y clases de arriba y abajo. Por lo general, las superclases se diagraman como que están sobre sus subclases. En el escenario de herencia simple, una jerarquía queda como un árbol con la raíz arriba del todo. El sentido de la dependencia baja. Totalmente al contrario, las capas se dibujan una arriba de la otra, con la capa de más bajo nivel abajo de las que utilizan sus servicios. Cuando las capas son clases, la confusión no termina y se debe ser muy riguroso en cuanto a que es arriba, abajo y super.

Por último, considérese nuevamente la expresión del tipo de una composición concreta. En C++, para el ejemplo de las dos capas es así:

```
CapaArriba<CapaAbajo>
```

Un objeto puede crearse con ese tipo, o el tipo puede almacenarse mediante un `typedef`. En general se tiende a lo segundo, porque la composición variará y no se quiere tener que revisar todos los usos del componente ensamblado. Lo más cómodo es tener el tipo con un nombre claro en algún lugar centralizado. Esto es un análogo en tiempo de compilación del patrón factoría. Otra opción puede ser hacer que una clase no template herede de la composición. También se tiene el beneficio de que los errores de compilación se expresan sobre el nombre de la clase que hereda, y no de la composición, que tiene un nombre mucho más largo y complicado:

```
class Componente : CapaArriba<CapaAbajo> {};
```

Existe un patrón de implementación que casi es un truco mágico y que permite definir tipos de manera recurrente. Se llama CRTP o Curiously Recurring Template Pattern. Recurring es por que se ha inventado muchas veces, pero a veces se cambia por recursive, dada la naturaleza del mismo. Generalmente se usa para separar la parte de la implementación de una clase para que pueda ser reutilizada para otras clases, pero sin perder el tipo de la clase derivada. Se puede pensar como una implementación del patrón plantilla.

Consiste en pasar el tipo de la propia clase a una clase de la que se hereda. De esa manera, un objeto de la clase base tiene conocimiento estático de su verdadera clase, y puede ofrecer sus servicios de acuerdo a la clase derivada, con lo que no obliga a hacer operaciones de conversión de tipos a sus clientes:

```
template <class Sub>
class Sumable
```

```
{
public:
    Sub operator+(Sub rhs) {
        return rhs += static_cast<Sub&>(*this);
    }
};

class Number : public Sumable<Number>
{
public:
    Number(int arg) : i(arg) {}
    int i;
    Number& operator+=(Number rhs) {
        i += rhs.i;
        return *this;
    }
};
```

`Sumable` es una clase base que brinda la función binaria suma (+) en base a la unaria (+=). También se podría haber hecho al revés, pero es bien conocido que es más eficiente de esta manera. La idea es que clases clientes puedan brindar solo el operador unario y heredando de `Sumable` obtengan automáticamente la binaria. La implementación de la operación binaria es igual a la que se da en los libros de texto, salvo por un detalle que salta enseguida a la vista, la operación de conversión de tipos. Se debe convertir la clase base en la derivada para poder invocar a la suma unaria. La operación de conversión es necesaria pero segura, porque los tipos tienen que estar al menos relacionados para que `static_cast` compile, aunque pueda luego fallar en tiempo de ejecución. Si la clase con la que se instancia `Sumable` no hereda a su vez de la instanciación, los tipos no están relacionados y la sentencia no pasa por el compilador. Véase como el impacto en `Number` es mínimo, y la adición de la operación tiene un fuerte tinte declarativo.

Volviendo a GenVoca, el mismo patrón es aplicable allí. Como ya se cuenta con una expresión de varios niveles, la composición entera debe alimentar a la capa más inferior. Eso se logra de la siguiente manera:

```
class Componente : CapaArriba<CapaAbajo<Componente> > {};
```

Una vez hecha esta modificación, la construcción crece en potencia. Cada capa estaba parametrizada por sus capas inferiores, lo que permitía que ciertos tipos y meta-data fluyeran ascendentemente por el componente. Si el componente entero parametriza la capa de más abajo, entonces también es posible pasar información de las capas más superiores a las de abajo. Las capas superiores heredan de las inferiores, si se recuerda, así que lo que sucede es como las funciones virtuales pero a un nivel más estático. Comúnmente esta facilidad se conoce con el nombre de tipos virtuales.

El uso principal que tiene esto es para manejar las clases totalmente ensambladas desde las capas inferiores. Como las clases de las capas superiores, que son las que van a ser instanciadas, heredan de los de las capas inferiores, no existe problema para recibir objetos de los tipos derivados; una referencia a un objeto de una clase derivada es convertida automáticamente en una referencia de la clase base. Por el contrario, no se da la conversión inversa, por lo que las capas bajas pierden información de los objetos que pasan a través de sus funciones. Si el tipo de la composición está disponible al momento de la definición de

todas las capas, se pueden declarar todos los argumentos, tipos de retorno y objetos como del tipo final, evitándose el inconveniente.

5.4. Enfoque Gestalt

A modo de llevar todos estos conceptos a tierra y ya atados a la implementación en C++, el grupo diseñó un enfoque de diseño basado en GenVoca llamado Gestalt. Gestalt quiere decir forma en alemán y está asociado a la idea de un todo conformado por la suma de sus partes y sus interacciones. Esta idea es lo que se pretende en este enfoque de diseño.

Cabe destacar que este enfoque no tiene nada que ver con el diseño Round-Trip Gestalt propuesto por Booch.

Idea

Un Gestalt está compuesto por un grupo de capas, donde cada capa proporciona un aspecto de una funcionalidad. Una de estas capas es el fondo y por encima se colocan las superiores. Una capa superior conoce solo a la capa por debajo de ella, mientras que la del fondo conoce al Gestalt como un todo, indivisible e indistinguible. La del fondo provee entonces una funcionalidad asGestalt, que devuelve un punto de acceso al Gestalt completo. Cada una de las capas superiores, empezando por la inmediata superior al fondo, recibe esta funcionalidad de la capa inferior y la ofrece hacia arriba, y de esta manera todas las capas pueden acceder al Gestalt.

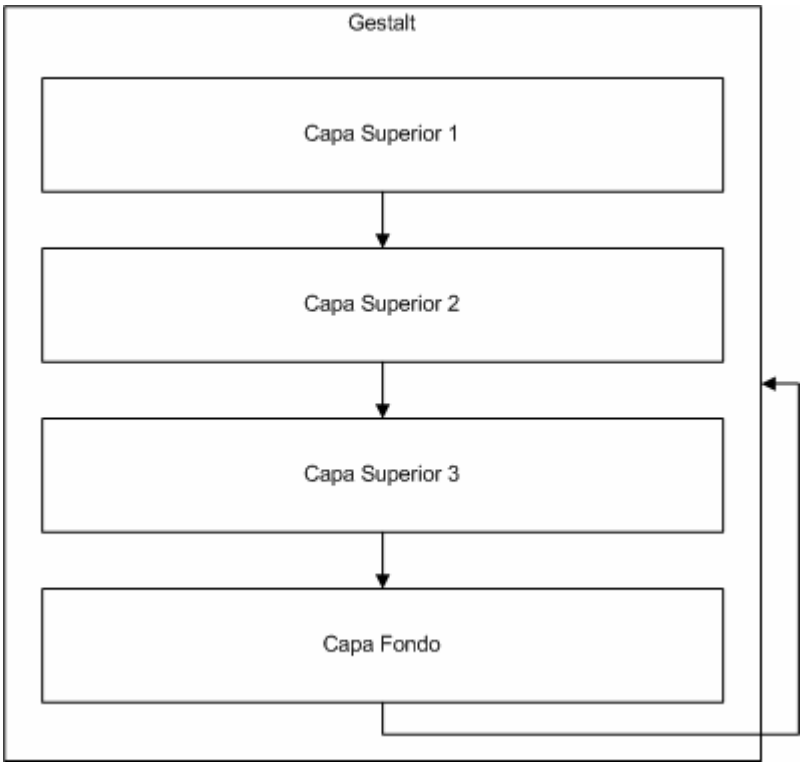


Figura C.3: Capas Gestalt

Dado nuestro sistema de clases orientado a objetos, tomemos una clase `A` con atributos y operaciones que afectan diferentes aspectos. A efectos de la explicación se supondrá que cada operación afecta a un solo aspecto y en cada aspecto interviene un grupo de atributos, luego se levantará esta restricción. Esta clase se separa en clases `A`, una en cada capa del Gestalt, donde cada una tiene las operaciones y atributos que afectan el aspecto de esa capa, más una

clase A en la capa fondo, que por ahora estará vacía. Luego se hace que cada clase herede de las clases en las capas inferiores. De esta forma, la clase A que se encuentre en la capa de más arriba será equivalente a la clase A original. Esta clase será llamada la clase A de Gestalt.

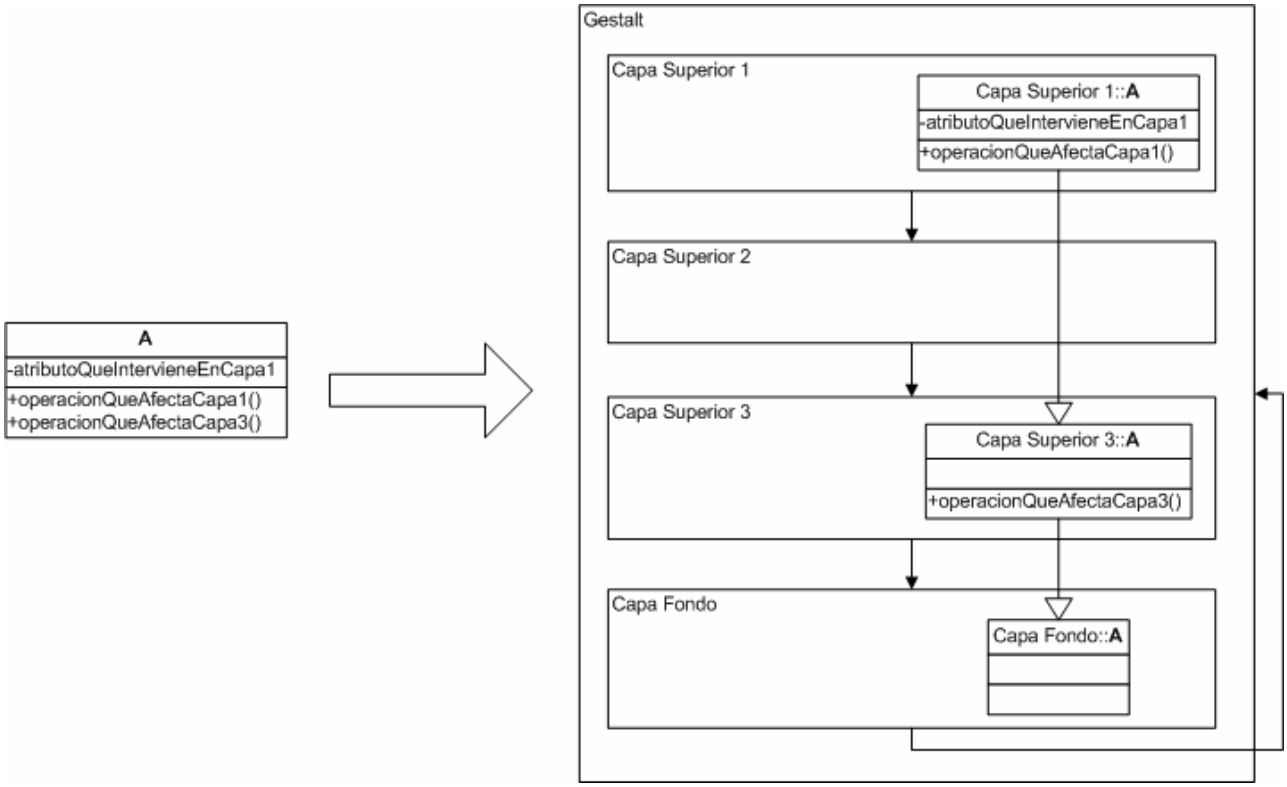


Figura C.4: Transformacion a Gestalt de una clase bien separable

Si alguna operación afecta varios aspectos, la operación se coloca en todas las clases afectadas. Los métodos de estas operaciones deben ser modificados para que trabajen sólo sobre este aspecto y llamen a la operación de la clase A inmediatamente inferior. Para satisfacer la llamada de la clase más inferior, se coloca esta operación en la clase A de la capa fondo, cuyo método no tiene efecto. Estas operaciones se denominan operaciones terminales.

Los atributos que afectan varias capas se colocan en la capa más significativa, en la que lo use más o en la capa fondo, a criterio del diseñador.

Se realiza esta separación para cada una de las clases del sistema en las diferentes capas.

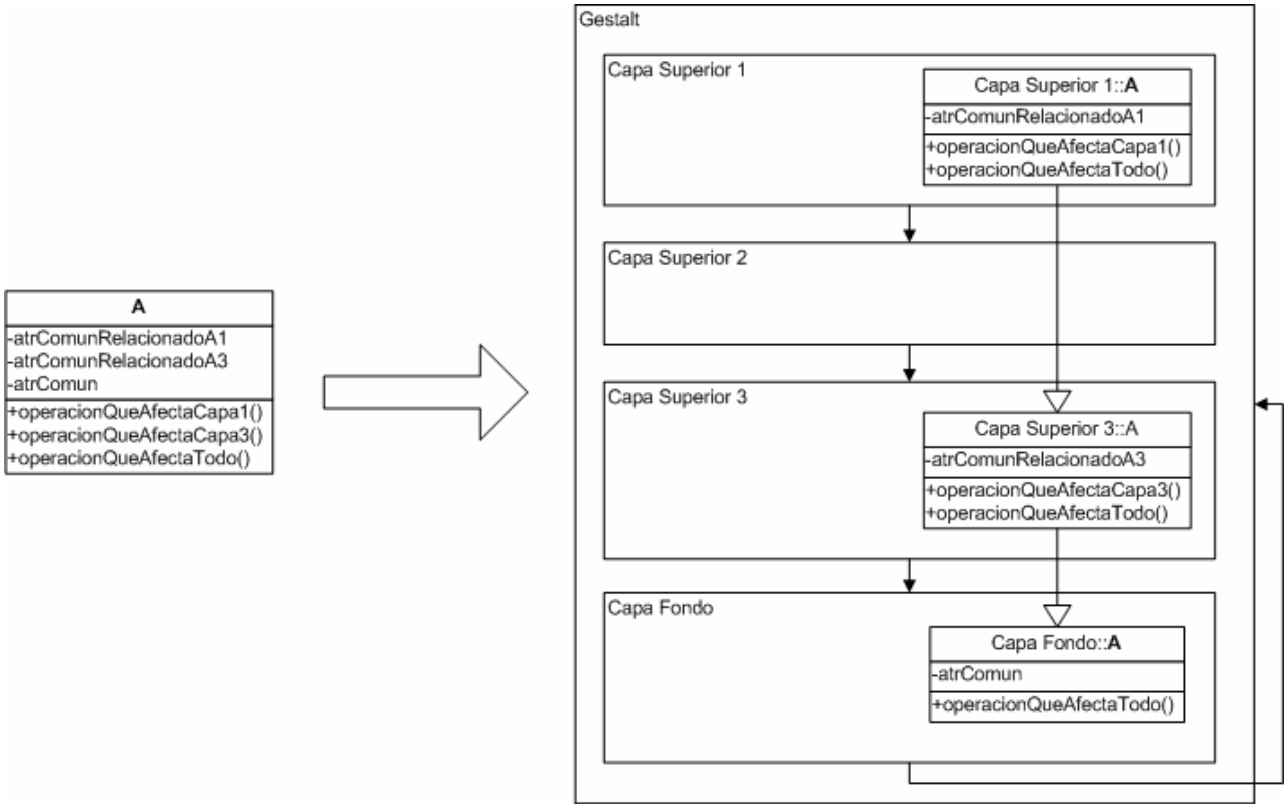


Figura C.5: Transformacion a Gestalt de una clase no tan separable

Se agrega a cada una de las clases de la capa fondo la operación `asGestalt()`, que devuelva la referencia a la clase pero convertida a la clase de Gestalt. El método de esta operación utiliza el acceso de la capa fondo al Gestalt para obtener el tipo y realizar la conversión.

Por último, toda referencia a alguna clase en el sistema debe ser cambiada por la referencia a la clase del Gestalt, y toda operación invocada a un objeto (salvo las llamadas a las capas de abajo) debe ser invocada a `objeto.asGestalt()`. Tras este paso, se puede ver que el orden de las capas superiores no es relevante.

Esta nueva estructura tiene la misma funcionalidad que la original, con la diferencia de que los comportamientos están separados en clases independientes por aspecto. Esto aumenta la modularidad y flexibilidad del sistema, con la desventaja de que a veces no es simple separar las clases en los aspectos deseados. También puede ser necesario crear más operaciones auxiliares para separar los comportamientos.

Para facilitar el desacoplamiento de las capas, se define para cada capa un contrato con lo que requiere y provee con las clases que contiene. Todo Gestalt compuesto por capas que satisfagan sus contratos entre sí es un Gestalt válido.

Implementación de Gestalt en C++

En C++ se utilizan los mecanismos de templates, subclases y CRTP para implementar un sistema bajo este enfoque, aplicando varias técnicas de GenVoca. Cada capa se implementa mediante un struct template con una clase como argumento. El struct de la capa fondo recibe Gestalt en su argumento y los de las capas superiores heredan de sus argumentos. De esta manera se construye un Gestalt (vacío) con la siguiente sentencia:

```
struct Gestalt : Capa1< Capa2< Fondo< Gestalt> > > {}
```

Para simplificar el acceso a los elementos, en cada capa se utiliza un `typedef` que define el Gestalt públicamente. Debido a que cada capa lo hace, las capas superiores pueden obtener el tipo de Gestalt de la capa de abajo. La implementación de las capas es entonces:

```
template<class Gestalt_> struct Fondo {
    typedef Gestalt_ Gestalt;

};
```

para la capa fondo y

```
template <class CapaAbajo> struct Capa1 : CapaAbajo {
    typedef typename CapaAbajo::Gestalt Gestalt;

};
```

para una capa superior.

El `typename` permite obtener el nombre del tipo de Gestalt de la clase `CapaAbajo`. De esta manera se le puede asociar el nombre `Gestalt` local.

Queda por definir como se implementan las clases del sistema en las diferentes capas:

- Para realizar la herencia de las clases, toda clase que se encuentre en una capa superior debe heredar de la misma clase en `CapaAbajo`.
- Las clases de la capa fondo deben contener las siguientes funciones:

```
typename Gestalt::ClaseA& asGestalt() {
    return *static_cast<typename Gestalt:: ClaseA*>(this);
}
typename Gestalt::ClaseA const& asGestalt() const {
    return *static_cast<typename Gestalt::ClaseA const*>(this);
}
```

  Estas operaciones hacen (por medio de la herencia) que cualquier clase, sin importar el nivel de capa en la cual se esté accediendo, se pueda convertir en la clase en Gestalt, con todos sus atributos y operaciones.

- Todas las instancias donde se utilice una clase `A` deben ser reemplazadas por `typename Gestalt::A`, a fin de usar su versión de Gestalt. Para facilitar la lectura se puede hacer un typedef dentro de la clase que la utiliza.
- Los constructores de las clases representan un problema, y es que cada clase requiere argumentos especiales para cada capa. El método es crear un datatype por clase del Gestalt. Este datatype contiene todos los parámetros necesarios para construir esa clase y

todas sus clases inferiores. En el constructor de cada clase (menos las de la capa fondo) se recibe el datatype y se construye el padre (la clase inferior) con este mismo datatype. Esto acopla los parámetros necesarios de todas las capas, pero es la solución que permite la mayor flexibilidad a la hora de agregar, quitar, sustituir o intercambiar capas.

6. Referencias

[Sut99] SUTTER, Herb. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Addison-Wesley Professional, November 1999. p.240. ISBN: 0201615622

[Lak96] LAKOS, John. Large-Scale C++ Software Design. Addison-Wesley Professional, July 1996. p.896. ISBN: 0201633620

[VJ02] VANDEVOORDE, David; JOSUTTIS, Nicolai M. C++ Templates: The Complete Guide. Addison-Wesley Professional, November 2002. p.552. ISBN: 0201734842

[GTK05] gtkmm Home Page

http://www.gtkmm.org/ Diciembre 2005

[Bst05] Boost Home Page

http://www.boost.org Mayo 2005

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis, JTS: Tools for Implementing Domain-Specific Languages. 5th Int. Conf. on Softw. Reuse (ICSR '98), IEEE Computer Society Press, 1998

[Str97] STROUSTRUP, Bjarne. The C++ Programming Language (3rd Edition). Addison-Wesley, 1997. p.911 ISBN 0-201-88954-4

[Ale01] ALEXANDRESCU, Andrei. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, February 2001. p. 352. ISBN: 0-201-70431-5

[BOM92] BATORY, Don; O'MALLEY, Sean. The design and implementation of hierarchical software systems with reusable components. New York, NY, USA: ACM Press, ACM Transactions on Software Engineering and Methodology, October 1992. v.1, n.4, p.355 – 398.

[BS02] BATORY, Don; SMARAGDAKIS, Yannis. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. New York, NY, USA: ACM Press, ACM Transactions on Software Engineering and Methodology, April 2002. v.11, n.2, p. 215 - 255.

[Sma99] SMARAGDAKIS, Yannis. Implementing Large-Scale Object-Oriented Components. University of Texas at Austin: Ph.D. Dissertation, Dept. of Computer Sciences, December 1999.

[BSST93] BATORY, Don; SINGHAL, Vivek; SIRKIN, Marty; THOMAS, Jeff. Scalable software libraries. ACM SIGSOFT, 1993.

[AG04] ABRAHAMS, David; GURTOVOY, Aleksey. C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond. Boston:Addison-Wesley, November 2004. ISBN: 0-321-22725-5

Apéndice D – Decisiones de diseño

1. Introducción

Desde un comienzo la idea de la estructura se pensó como los componentes más básicos posibles y refinamientos superiores que los usan. Cuando llegó la hora decidir cual sería este componente básico en cuanto al paradigma, se vio que todo paradigma en mayor o menor medida está basado en un enfoque a eventos en dos fases:

- El enfoque a eventos con tres fases es igual al de dos fases con un ejecutivo más desarrollado que ejecuta eventos condicionados entre los fijos.
- El enfoque a descripción de procesos tiene procesos que, si bien interactúan con los recursos, deben esperar en las actividades un tiempo determinado. Esto es equivalente a agendar su despertar como un evento fijo.
- El enfoque a interacción de procesos es similar al anterior. Además de la interacción entre procesos deben esperar tiempos fijos, lo que se logra agendando eventos fijos.
- En Devs se puede utilizar un scheduler de dos fases en cada modelo atómico, donde se agendan los eventos de entrada. Los eventos de salida se agendarían en los schedulers de los modelos a los que estén vinculados.
- Para grafos de eventos se pueden convertir los nodos del grafo en eventos fijos, agendados por los nodos antecesores en el grafo según las condiciones que marcan las aristas.
- Con simulación paralela se tienen muchos schedulers que se comunican entre sí. La refinación sobre el scheduler normal de dos fases es la capacidad de sincronización o de rollback, dependiendo del enfoque elegido.

Por lo tanto se decidió que el encare inicial hacia la ejecución de modelos era construir un scheduler a dos fases con eventos fijos, de tal manera que se permitiera su expansibilidad. Se busca también otro de los objetivos decididos, que es poder generar modelos híbridos entre paradigmas, de tal modo que cada parte utilice el enfoque más apropiado y utilicen un scheduler en común.

2. Eventos

Existen dos enfoques a la hora de decidir qué debe ser agendado en un scheduler. Una posibilidad es asociar una entidad con un evento y agendar las entidades en el scheduler. La otra es agendar instancias de un evento al scheduler, asociando a esta instancia los elementos necesarios (entidades, recursos, tiempos, etc.).

Defendiendo el primer enfoque se encuentra el hecho de que resulta intuitivo que los únicos elementos instanciados del sistema sean las entidades, debido a que cada una de estas instancias representa a un objeto tangible de la realidad. No hay un mapeo simulación-realidad similar con instancias de eventos.

Por otra parte, estas instancias de eventos se pueden ver como notas "pinchadas" en un calendario, donde cada nota tiene, además de la operación a realizar, con qué elementos aplicarla. Este enfoque es más flexible que el otro, debido a que si cada evento recibe una instancia de entidad y no se agendan dos eventos con la misma entidad, se puede representar el otro enfoque. Además, el poder asociar un conjunto arbitrario de elementos a un evento permiten hacer cosas más complejas que solo pudiendo asociar una entidad. Un ejemplo es una actividad con entidades cooperantes, donde los eventos tienen asociados más de una entidad.

Nótese que, sin embargo, se puede representar el segundo enfoque con los elementos del primero, a través de entidades y/o eventos dummy. Esta solución es mucho menos elegante y flexible.

Así que se decidió utilizar un scheduler que agenda eventos, los cuales en principio no están obligados a cumplir con ninguna estructura en cuanto a parámetros y operaciones utilizadas.

Esto supone un problema al establecer la interfaz que debe ofrecer un evento. Para tratar este problema se separa el evento en dos partes:

- La función que se ejecuta cuando ocurre el evento y que es implementada por el usuario.
- La estructura que se almacena en el scheduler, el evento propiamente dicho. Estos deben estar ordenados por tiempo de ejecución, y cuando llegue el tiempo indicado invocar la función del punto anterior.

Las instancias de evento deben ser solamente accedidas a través del scheduler, dado que es el que tiene los elementos para manejarla debido a su conocimiento de los tiempos del sistema. La interfaz que deben ofrecer al mismo es sencilla:

- Un constructor en base a una función a ejecutar y un tiempo de ejecución
- Un método para ejecutar la función
- Un método para obtener el tiempo
- Un método que permita ordenarlos de menor a mayor (el operador > del tipo)

De esta manera el scheduler puede saber todo lo que necesita del evento para disponer del mismo y de la función relacionada.

En cuanto a la función, esta estructura genera dos problemas. Por un lado cual sería la manera de pasarla al constructor del evento, y por otro lado como hacer para invocarla con los parámetros necesarios. Para arreglar estos problemas imponemos que la función a ejecutar sea en realidad un functor nullario. Elegimos llamar function al concepto que cumplen todas las clases que definen su operador paréntesis y no tienen valor de retorno. Los parámetros de la función pueden ser enunciados en el constructor del functor y almacenados como miembros, para luego ser utilizados en la operación de invocación. El evento recibe este functor y cuando necesita ejecutar la función lo ejecuta sin parámetros.

Cabe destacar que no encontramos este modo de trabajar con eventos en ninguna otra herramienta, lo cual es extraño pues parecería la manera más correcta y elegante.

3. Scheduler

Una vez establecido este mecanismo de eventos se debe crear un scheduler que los maneje. Se busca mantener la minimalidad establecida anteriormente.

En primer lugar el scheduler debe contener todos los eventos en un contenedor cuyas operaciones más eficientes deben ser la inclusión (cuando se agenda) y la obtención/eliminación del evento de ejecución más temprana (cuando se deben ejecutar). Por lo tanto se decidió utilizar una cola de prioridad para almacenar los eventos, donde la prioridad sea inversa al tiempo de ejecución.

Hubo varias opciones de donde poner el reloj del sistema. Debía ser en un objeto que fuera único y permaneciera vivo durante toda la ejecución del sistema. Eso nos deja tres opciones posibles:

- **El modelo:** hacer que el propio usuario de la librería mantenga el reloj del sistema y lo mueva a discreción. Esta opción fue descartada porque además de complicar la interacción, podía llevar a comportamientos extraños de "viajes al pasado". Como el usuario controla todos los tiempos, nadie se lo impide, y errores de modelado quedan ocultos.
- **El ejecutivo:** es un caso similar al anterior, solo que del lado de la librería. En este caso se restringe un poco más el manejo del tiempo, el cual solo puede avanzar a medida que se dan los eventos.
- **El scheduler:** igualmente restrictivo que en el ejecutivo, proporciona una ventaja adicional que no proveen los otros dos. Cuando se debe agendar un evento fijo, lo importante no es el tiempo de finalización sino el tiempo de duración. Si el scheduler controla el reloj del sistema, entonces sin necesidad de conocer otra clase puede recibir eventos agendados en tiempo relativo, sumarles el tiempo actual y almacenarlos según su tiempo absoluto. Esto aumenta la independencia del scheduler respecto al resto de la librería, lo cual es deseable. Se decidió tomar esta opción.

En cuanto a la interfaz que ofrece al usuario de la librería (y a los módulos que lo refinan), se optó por el siguiente conjunto de métodos:

- `push(offset_time, function)`: Crea un evento para que se ejecute function luego de transcurrido `offset_time` en tiempo de simulación y lo almacena en su cola de prioridad. offset_time debe ser positivo.
- step(): Obtiene el siguiente evento a ejecutar, actualiza el reloj al tiempo de ejecución del evento y lo ejecuta. Tiene como precondición que la cola no esté vacía.
- lastEventTime() y nextEventTime(): devuelven el tiempo de ejecución del último evento ejecutado y el siguiente evento a ejecutar respectivamente. Esto es útil para varios otros módulos de la librería y para la propia implementación del usuario.
- noMoreEvents(): devuelve si quedan eventos por ejecutar. Esto sirve para que el ejecutivo detecte condiciones de finalización en sistemas terminales.

4. Executive

Ya no buscando la base de los paradigmas sino la realización de los mismos, se comenzó con el diseño de dos ejecutivos, para la ejecución de modelos orientados a eventos a dos y tres fases.

4.1. Executive para eventos a dos fases

Pese a que uno pensaría el ejecutivo como una clase, vemos que no hay necesidad para que no sea una simple función, por las siguientes razones:

- Tiene una sola función, ejecutar.
- No debe conservar ningún estado, este se mantiene en el scheduler y el modelo a ejecutar, así que no tiene miembros.
- No es preciso conservarlo sin ejecutar, copiarlo, pasarlo, etc.
- La función ejecutar es más bien un algoritmo, los cuales según el diseño estándar de C++ se hacen mediante funciones libres.

El algoritmo de esta función es sencillo y no hay mucho criterio de decisión. Se ejecuta step() del scheduler hasta que se llegue a un tiempo límite o hasta que el scheduler esté vacío.

Lo que queda por decidir es el tipo de los parámetros de entrada. El primer parámetro es el tiempo durante el cual debe correr la simulación. Este se representa mediante un `double`. El segundo parámetro es el scheduler que se utiliza durante la corrida. Se decidió no exigir que este sea un scheduler de nuestra librería, y dado que el executive es una función podemos utilizar un parámetro template para implementar esto. Esto permite utilizar cualquier clase que cuente con las operaciones del scheduler (salvo `push(...)`). Decimos que toda clase que tenga estas operaciones pertenece al concepto schedulerRunnable. Es útil si el usuario quiere hacer sus propios schedulers o ejecutar directamente un modelo que contenga un scheduler y sirva de adaptor.

4.2. Executive para eventos a tres fases

Antes de hablar del ejecutivo observemos el tema de la tercera fase. Según el paradigma, la tercera fase se compone de una serie de eventos condicionados que se ejecutan secuencialmente cada vez que el reloj de la simulación vaya a avanzar.

Hay un elemento básico que escapa los análisis normales de esta fase, y es que el orden en que se ejecuten estos eventos importa. En el orden equivocado el sistema puede quedar en deadlock u ocasionar problemas de falta de atención para deteminada cola. Quien es encargado de crear estos eventos condicionados y determinar su orden de ejecución es el propio modelador. Con este orden puede jugar con comportamientos de prioridades y manejo de recursos.

Dado que el modelador debe organizar estos eventos, puede implementarlos en una sola función, o hacer una función que los llame en el orden deseado. Así que podemos reducir el problema al caso con un sólo evento condicionado, que es la ejecución ordenada de todos.

Otro elemento es cuando ejecutar este evento. Como esta ejecución gasta tiempo de ejecución y se busca maximizar la performance del sistema, se debe ejecutar las veces que sea estrictamente necesaria.

Este evento es idempotente, debido a que la primer ejecución modifica todo lo haya para modificar y sucesivas aplicaciones dejan el estado igual. Por lo tanto no se debe ejecutar el evento repetidamente.

Por otra parte el evento debe ejecutarse cada vez que pueda modificar algo, es decir cada vez que el estado del modelo cambie. Una primera aproximación al problema diría que hay que ejecutar el evento condicionado luego de cada evento fijo.

Sin embargo, un conjunto de eventos fijos asignados al mismo tiempo no cambian su comportamiento por la ejecución de eventos condicionados en el medio. Estudiemos el caso desde dos puntos de vista:

- **Desde los eventos fijos:** los eventos fijos ya tienen reservado todo lo que necesitan para ejecutar. La ejecución de eventos condicionados no afectan el funcionamiento de los eventos ya agendados, ni el modo en que estos alteran el sistema.
- **Desde los eventos condicionados:** los eventos fijos no van a extraer ningún recurso que los condicionados necesitan. Si se debe ejecutar un evento condicionado para un determinado tiempo, da lo mismo que se posponga para después que los fijos para ese tiempo. Como el tiempo no avanza, si el evento condicionado agenda eventos fijos en tiempo 0 (o mayor), aún puede agendarlos. Sin embargo, no se puede adelantar el tiempo en que se ejecutan, pues los eventos fijos pueden liberar recursos que los condicionados necesitan.

En un principio se pensó que se podían relajar las condiciones e invocar el evento condicionado luego de cada fijo, debido a que se estudiaron mecanismos para que luego de cada fijo se ejecuten sólo aquellos condicionados cuya condición pueda ser cumplida tras la ejecución del fijo.

Consideremos lo siguiente: para poder reproducir adecuadamente una corrida de simulación, el comportamiento tiene que ser determinístico. Si se toman las precauciones de repetir las secuencias de números aleatorios, no parece haber una situación donde esto no sea así. Sin embargo, considérese el caso de que se simuló una parte de un sistema y ahora se quiere simularlo completamente pero con el subsistema anterior comportándose de la misma manera. Además de los eventos que genera el subsistema, hay nuevos eventos que se disparan por el resto del sistema completo y la interacción del subsistema con partes ajenas. Es necesario que los eventos que antes aparecían en la simulación del subsistema aparezcan en el mismo orden relativo en la simulación mayor. Otra cosa muy distinta es que el programador pueda depender del orden en que se disparan los eventos. Lo mejor es que el orden sea determinístico pero indeterminado. No se garantiza FIFO, LIFO ni ningún otro, pero si se garantiza que siempre es igual.

Si el programador agenda determinísticamente dos eventos para el mismo instante de tiempo y depende del orden relativo con que se disparan, ¿por qué no agenda un evento solo, que realice las dos acciones? La razón puede ser que la metodología de diseño usada no permitía expresar esto fácilmente, por lo que se necesita definir prioridades entre los eventos o incrementar la cantidad de información que se maneja a mano. Cuando se usa un lenguaje de programación general, este impedimento no existe. De hecho, agendar dos eventos al mismo tiempo es igual de difícil que usar un combinador de eventos, que puede escribirse una sola vez y guardarse en la biblioteca.

Recorrer los eventos condicionados es un gran desperdicio de tiempo computacional. Por eso, se han ideado trucos para acelerar este procedimiento, como la simulación celular [DO89]. En este esquema los eventos de la segunda fase marcan de alguna manera los eventos de la tercer fase que pueden correr, asumiéndose que los no marcados siguen bloqueados, y solo verificándose así los marcados. Ahora, si se eliminan los eventos concurrentes premeditados, solo quedan los raros eventos que coinciden por cuestiones del carácter discreto de la representación por computadora, ya que los tiempos son sacados de generadores aleatorios. No tiene sentido optimizar para este caso infrecuente, así que no es una gran pérdida en lugar

de marcar los eventos condicionados a recorrer, directamente recorrerlos y disparar los prontos.

Eliminar la tercera fase como una fase separada e incorporar la misma vista global como último paso en cada evento trae beneficios en la programación en C++. Ya no es necesario mantener una lista explícita de eventos condicionados posibles, aunque todavía se puede hacer si el programador lo desea. Alcanza con que cada evento referencie a los que pueda ocasionar a su vez. Se evita tener que guardar como datos la información que traiga el evento, que puede ser completamente arbitraria. La decisión de qué evento condicionado disparar se puede hacer con toda la información disponible en el evento, sin necesidad de prever las estructuras necesarias para reflejar toda la gama de eventos posibles. La inferencia de tipos funciona en C++, gracias a los templates, solo hacia adelante, es decir, puedo llamar a una función sin tener que especificar los tipos de los argumentos, pero no en sentido inverso. Como me evito la ida hacia atrás, se me abren las puertas a estás técnicas de programación. Por último, se puede explotar la facilidad que tiene C++ de colocar los objetos en el stack, ya que no retorno hasta que es de veras necesario.

De todos modos, aunque todo esto sea implementable y redunde en un beneficio, tras discutirlo con los tutores se llegó a dos conclusiones. Por un lado, la manera de combinar eventos concurrentes es complicada para el modelador, quien puede no querer ser consciente de la concurrencia determinística de actividades de diferentes módulos. Por otro lado, los mecanismos de selección de los condicionados por fijos requiere más habilidades de programación debido a que estos son relacionados al lenguaje de propósito general y no al lenguaje de simulación. El modelador no tiene por qué tener estas habilidades. Sin estos dos puntos, la consideración anterior queda descartada.

En cuanto a ejecutar todos los eventos condicionados luego de cada fijo, se observó que la pérdida de eficiencia es significativa. En caso de que algún tipo de evento se agende deterministicamente (como en nuestro caso de estudio, lo que nos hizo tomar una decisión definitiva), ejecutar el condicionado luego de cada fijo disminuye mucho la performance.

Por lo tanto, lo mejor es que se ejecuten todos los fijos para un determinado tiempo y luego una vez el condicionado. Si el condicionado agenda eventos fijos en tiempo 0, estos se toman como si fueran para un nuevo tiempo sin aumentar el reloj.

Para crear este ejecutivo, teniendo en cuenta la decisión de que sea una función por las mismas razones del punto anterior, se vieron dos maneras:

- Crear otra función totalmente independiente que tuviera el mismo comportamiento que la otra, pero que cada vez que el reloj vaya a avanzar llamara a la función del evento condicionado.
- Crear un refinamiento del scheduler básico que recibe el evento condicionado en el constructor y redefina el `step()`. Luego de ejecutar el `step()` básico se chequea si el tiempo de ejecución del próximo evento es mayor al actual. En caso afirmativo, se ejecuta el condicionado.

Pese a que la segunda alternativa es un poco más elegante, utilizar un mismo scheduler facilita el trabajo en conjunto con otros paradigmas, como el de interacción a procesos. Además, en caso de expandir el scheduler para otra cosa (como serialización), se necesita expandir dos schedulers, o utilizar alguna técnica de programación que agrega complicación. Por lo tanto agregar una función extra parece la mejor alternativa.

5. Complejidades del modelado

Se desea proveer soporte para el manejo de complejidades del modelado, como entidades en varias actividades y la cancelación de eventos fijos ya agendados. La principal herramienta para conseguirlo es la flexibilidad en el manejo de los eventos. Un evento no está restringido a una entidad en particular, ni se asocian las entidades a sus eventos, por lo que puede haber varias entidades por evento, la misma entidad en varios eventos o combinaciones.

Está definido un concepto para el calendario, pero pueden concebirse calendarios que sobrepasen esta especificación y ofrezcan acceso a su interior. No es el caso del calendario básico provisto por la librería porque se base en la estructura opaca de la cola de prioridad estándar de C++, pero un calendario sencillo con una cola encadenada, por ejemplo, perfectamente podría exponer la misma para que el cliente realice fetching si lo necesita.

Aquí se vislumbra una de las ventajas de usar conceptos en lugar de clases bases como en la programación orientada a objetos tradicional. Se puede asignar conceptos a objetos de manera no intrusiva. Los objetos no pierden sus otros conceptos al ser manipulados a través de uno en particular y la información completa se puede recuperar sin necesidad de conversiones de tipo.

Se dice a veces que un nivel extra de indirección puede resolver todos los problemas de la ingeniería de software. Resulta que para el caso de fetching, esto es así. Una solución es, en lugar de sacar el evento de la lista de eventos del calendario, dejar el evento sin efecto, para que cuando ejecute no haga nada. Para cada caso se puede idear una solución particular, por ejemplo con campos especiales en las entidades para marcar si el evento todavía se debe ejecutar o no.

La solución adoptada fue, aprovechando que se usan para los eventos un objeto función, escribir la solución una sola vez, y poder reutilizarla en variados escenarios. La librería incluye una clase para este cometido que se llama `ticket`. La idea es que en lugar de agendar un evento, se crea un `ticket`. Este `ticket` agenda el evento y tiene una función para cancelar el evento. De esta manera, este `ticket` puede ser almacenado según desee el usuario y neutralizar el evento cuando lo considere. Se tomaron precauciones para con la robustez del mecanismo, cosa de que cancelar un evento ya disparado no tenga efecto, lo que requirió el uso de contadores de referencias suministrados por Boost.

6. Procesos

Se quería que la librería soportara además del paradigma a eventos, la interacción de procesos. A diferencia de otras librerías, no se quería favorecer uno de los dos paradigmas por sobre el otro a menos que fuera indispensable, y se deseaba permitir la combinación de ambos paradigmas en una misma aplicación, simultáneamente. Por estas razones, la clase process de SimPP es bastante particular.

6.1. Diseño

Subyacentemente, en SimPP todo se maneja mediante el paradigma de eventos, así que hubo que mapear los procesos a eventos. Cuando un proceso es activado ejecuta hasta suspenderse o terminar, modificando el estado del modelo. Entonces cada proceso además de tener un método para activarlo sobrecarga el operador de invocación, con lo que es un objeto función propiamente dicho, que cumple con el concepto function. Cada vez que se invoca el proceso como función, se activa.

Los procesos pueden ser copiados libremente, para meterlos en colas y manipularlos como se desee. Se usa un mecanismo de contadores de referencia para saber todos los objetos process que existen para un proceso dado, y cuando la cuenta llega a cero el proceso se cancela. Esto está hecho así para poder escribir procesos que entraran en un ciclo infinito, lo que es natural, pero poder terminarlos eventualmente. Las interfaces de las librerías de concurrencia cuentan con primitivas para la destrucción de una thread desde fuera, pero son muy pocas las situaciones en que esto sirve, porque no se realiza ninguna limpieza de los objetos que la thread había creado, de los que el sistema operativo no tiene conocimiento. Sin embargo, en una simulación siempre que una thread va a ser cancelada se sabe que la misma está suspendida y en espera de que se le deje proseguir su ejecución. Entonces se la activa marcándole en una bandera que debe destruirse, lo que ocasiona que se levante una excepción en la propia thread, desenrollándose el stack y terminando limpiamente. Esto no hubiera sido posible si C++ no contara con destructores deterministas.

Además de activarse un proceso, se puede suspender la ejecución del propio proceso y obtener el propio objeto, para colocarlo en una cola antes de suspenderse. Estás dos últimas funciones son estáticas, porque afectan al proceso en ejecución, pero deben tener cuidado de no usar variables globales. Si bien nunca hay dos procesos de una simulación ejecutando concurrentemente, puede haber dos simulaciones corriendo en hilos diferentes, cada una con procesos. Se tomó el cuidado de que este escenario funcionase. La solución fue utilizar variables locales a los hilos para guardar el objeto actual, y no una tabla global.

6.2. Ordenamiento de procesos

En el paradigma de interacción a procesos los procesos son capaces de activarse entre si, para informarse de la disponibilidad de recursos, por ejemplo. Generalmente un proceso activa a otro antes de suspenderse a si mismo, por lo que solo hay un proceso pronto para correr en un momento dado. En situaciones especiales, sin embargo, un proceso puede activar varios otros al mismo tiempo, o activar un proceso y seguir corriendo a su vez.

Hubo que definir la disciplina con que los procesos se ejecutan. Si la disciplina no se conoce se debe asumir siempre lo peor. Cuando un proceso es activado es porque se descubrió la disponibilidad de lo que antes lo trancaba, sin embargo, puede haberse entrometido algún otro proceso y tomado ese recurso, por lo que se debe hacer el control de disponibilidad aunque el que activó al proceso ya lo realizara. Al activar un proceso se debe asumir que este ejecuta inmediatamente, por lo que también deben volverse a verificar las condiciones de ejecución sobre el modelo. Esto no solo es pesado para el programador, sino que genera más código a ejecutar. Los dos órdenes consistentes son una disciplina de stack y una de cola. Esta misma

decisión debe tomarse para el patrón de sincronización monitor. Ahí los procesos se despiertan unos a otros dentro de un ámbito protegido, pero nuevamente se puede decidir tomar una cualquiera de las dos alternativas. La decisión se tomó tratando de minimizar el acoplamiento entre los componentes de la librería.

La manera de lograr una disciplina de cola es agregando la activación del proceso como un evento con tiempo nulo, para que se ejecute último antes de que avance el reloj. En ese caso se necesitaría conocer al scheduler para implementar la función activate, que en el otro caso no es necesario.

Por lo tanto se decidió utilizar una disciplina de cola. Cuando un proceso activa a otro, se suspende hasta que se retorne del proceso llamado, ya sea porque termina o se suspende. Como el mecanismo para suspender en este caso es el del stack de llamadas y no el de los procesos, la ejecución del scheduler puede suspenderse de esta manera y esto permite seguir tratando igual la operación de activación de procesos. De todos modos no se soporta reentrancia de procesos, debido a que el comportamiento de la reentrancia en los modelos puede dar resultados inesperados y complica la arquitectura interna.

En la forma clásica de interacción a procesos existe una función más para estos, que permite suspender la ejecución un tiempo determinado. Sin embargo, para SimPP esto no es una operación fundamental, porque se corresponde a agregar al calendario la propia instancia de proceso, que al ser invocada activa al mismo. Este patrón común se encapsuló en una función separada del núcleo de interacción a procesos.

7. Visualización

Las decisiones tomadas en esta sección corresponden al diseño e implementación de la clase Display, la cual ofrece funcionalidades de visualización para los modelos de la librería. Al tratarse de una herramienta más en el toolkit, es importante destacar que es totalmente independiente del resto de las herramientas. Esto permite tanto utilizar SimPP sin este módulo de visualización, utilizar otro módulo de visualizacion o incluso utilizar este módulo para otros propósitos, si bien la interfaz y muchas decisiones de diseño fueron tomadas para facilitar la aplicabilidad en modelos de simulación.

Ante el aspecto de la visualización, el primer factor a decidir es si la simulación será modelada a través de la representación gráfica, o si la representación será un producto de lo modelado. Lo primero podría ser una opción, siguiendo un enfoque similar a ProModel, pero consideramos que esto sería más bien un paradigma a implementar y decidimos no ofrecer esta funcionalidad. Por lo tanto la representación será un producto de lo modelado, y las herramientas de modelado permitirán establecer su funcionamiento.

7.1. 2D vs. 3D

Pese al auge del modelado 3D, consideramos que una representación 3D requiere un modelador más experto, y que la expresividad que ofrece, en cuanto al análisis y estudio de un modelo, no es significativamente mejor que una representación 2D. La representación 2D ofrece además dos ventajas importantes relevantes al objetivo:

- Elementos 2D expresivos son más sencillos de crear que sus contrapartes 3D. Mientras que para los primeros basta un editor de imágenes, para los últimos se necesita una herramienta de modelado 3D.
- El tiempo de proceso de una representación 2D es inferior al de una representación 3D. Se sabe que la visualización representará un cuello de botella en cuanto a tiempo de simulación, y se busca que este sea el menor posible.
- La representación 2D tiene una perspectiva única. En un modelo 3D, con una cámara móvil, se pueden perder detalles al no observarlos desde el ángulo correcto.

7.2. Texto vs. gráficos

Desde el aspecto del modelado, modelar una visualización 2D con caracteres o con gráficos requiere de la misma habilidad. Se diseña un entorno y se trabaja con coordenadas 2D para colocar las entidades. Si bien el modo texto es más performante que el modo gráfico, el gráfico es más expresivo debido a que:

- Se representa una posición de entidad más exacta, debido al menor tamaño de los pixels frente a los caracteres.
- Un dibujo de la entidad es más claro que una letra que la simbolice.

Por lo tanto decidimos implementar una capa de visualización de simulación utilizando *sprites* en un entorno gráfico 2D.

7.3. Efectos de dibujo

Subpixel accuracy, antialiasing, alpha blending, double buffering, todos estos efectos mejoran la calidad estética de la visualización mientras que la capacidad de modelado no se ve afectada [FDFHP93]. Existe un balance entre calidad y performance, que es el que se busca alcanzar. La elección de estas características queda en manos de la eficiencia de las mismas en las librerías elegidas.

7.4. Sprites

Hay varios tipos de elementos visualizables dentro de este marco. El tipo más básico es el *sprite*, una imagen extraída de algún archivo que se dibuja en la posición deseada. Se utiliza principalmente para representar entidades y elementos que interactúan en el sistema. Tiene una posición y una imagen fuente. Para su interfaz se pensaron cuales serían las operaciones de interés para el modelado de sistemas:

Constructor

Al momento de crear un sprite se recibe una imagen y posición inicial y se dibuja inmediatamente en el display. La imagen seguirá viéndose hasta que su posición la deje fuera de la pantalla (y no vuelva) o deje de ser referenciada por el sistema.

Cambiar imagen

En determinados puntos de la simulación es deseable que la representación de un elemento cambie. Se provee de una interfaz que cambie la imagen instantáneamente. Se pensó en ofrecer una operación de cambiar imagen en la cual el cambio se dé luego de cierto tiempo de simulación (posiblemente 0), pero con el objetivo de hacer la interfaz lo más minimal posible y viendo que esto se puede lograr mediante eventos intermedios (además de que los cambios de imagen suelen ocurrir en los eventos del modelo), no se ofrece esta facilidad. Tampoco se ofrece la carga de gifs animados debido a la falta de soporte por parte de las librerías a utilizar en la implementación.

Mover

Determina la posición del sprite en un tiempo determinado. La forma más simple de esta operación es recibir unas coordenadas y mover inmediatamente el sprite a la nueva posición. Esto no permite un movimiento gradual a lo largo del tiempo de un sprite hacia su nueva posición, y si bien varias implementaciones de esta operación en otros lenguajes tienen esta política, se decidió extender un poco más la funcionalidad para aumentar el poder expresivo.

La siguiente mejora es recibir las coordenadas del próximo destino y un tiempo de recorrido. En cada tiempo intermedio, el sprite se encuentra en una posición interpolada entre el origen y el destino. Esto permite representar un movimiento continuo del elemento a lo largo de su intervención en el modelo, pero tiene la desventaja de que solo permite movimientos lineales y con velocidad constante. El siguiente paso es, en vez de especificar un destino y tiempo de viaje, especificar una lista de destinos y el tiempo de recorrido entre cada uno de ellos. El sprite recorre cada uno de los destinos en el tiempo estipulado y se detiene en el último. De esta manera se puede controlar el recorrido en tramos y la velocidad de cada tramo.

Un siguiente paso, para eliminar la linealidad de los recorridos, es agregar algún método de curvado de recorrido como líneas de Bezier o B-splines, pero decidimos que esto era una complejidad innecesaria para el modelador.

Se decidió también que si se especifica un movimiento mientras un sprite se está moviendo, el sprite se detiene en su camino anterior y comienza inmediatamente el nuevo recorrido. Esto permite al modelador un control más estricto en el movimiento.

Existe una cuestión a decidir en cuanto a los recorridos de duración 0. Si bien un recorrido de duración menor a 0 no está permitido, un recorrido de duración 0 no sólo es válido, sino que además es utilizado a menudo. Estos movimientos pueden ser representados de dos formas.

- Mover instantáneamente el sprite del origen al destino (teletransporte)

- Pausar la representación de la simulación y mover sólo los sprites que se mueven en tiempo 0, tal que la duración de su recorrido sea un tiempo determinado (estasis)

Cada una de estas alternativas tiene sus ventajas respecto a la otra.

Apoyando el teletransporte se encuentra que detener la simulación cada vez que ocurre un movimiento de tiempo 0 (lo cual es a menudo, pues esta duración suele ser determinística) no solo demoraría demasiado una corrida, sino que el flujo de tiempo entrecortado puede dificultar el entendimiento del sistema. Por otra parte, un sprite cuyo tiempo de movimiento es positivo cercano a 0 se moverá en un instante muy breve mientras que un sprite de tiempo de movimiento 0 se tomará su tiempo. Esto aparenta comportamientos antinaturales.

Apoyando el estasis está el hecho de que a veces un teletransporte no permite ver con claridad lo sucedido. Esto ocurre generalmente cuando se tiene un servicio de atención de colas, en el cual una entidad deja de ser atendida y la siguiente entidad de la cola se mueve a ser atendida. Si los movimientos de la cola a la atención y de la atención al próximo destino son determinados con tiempo 0 y las entidades tienen los mismos sprites, no se verá como el sprite atendido deja la atención y el siguiente sprite avanza a ser atendido, sino que parecerá que el mismo sprite sigue siendo atendido y el sprite de la cola desaparece. Un movimiento en estasis temporal, aunque no simule tiempo real, puede ayudar a la comprensión en ciertos casos.

Luego de hacer un balance entre estas dos posibilidades (incluyendo la elaboración de prototipos y tests para ambas alternativas), nos volcamos al teletransporte. El factor decisivo fue el hecho de que generalmente un movimiento de tiempo 0 es precedido o seguido por un período en el cual el sprite está estático o deja el sistema. Si el teletransporte da lugar a confusiones, se puede reemplazar el movimiento de tiempo 0 sustrayendo un tiempo pequeño al período estático (o de eliminado) y realizar el movimiento en ese tiempo. La pérdida semántica es mínima.

Visibilidad

Hay veces que se quiere ocultar un sprite. El atributo `visible` (con métodos para obtenerla y modificarla) permite setear si se dibuja o no.

Operaciones de movimiento Z

Pese a estar en un entorno 2D, el orden en el que aparecen los sprites por encima o por debajo de otros es importante. Por ejemplo, resulta interesante que en una cola de sprites donde los sprites se intersectan, cada sprite se encuentre por encima de los siguientes. A este ordenamiento se le llama orden Z. Por cuestiones de tiempo se decidió no implementarlas.

7.5. Otros elementos para visualizar

Si bien todos los elementos a visualizar son emulables mediante sprites, a veces la utilización de los mismos es ineficiente en comparación con utilizar elementos más especializados, además de representar una sobrecarga en el diseño de imágenes. Realizamos entonces un resumen de los diferentes tipos encontrados:

Fondo

Se trata de una imagen grande y estática, utilizada para representar el ambiente del modelo. Ocupa toda la pantalla, no tiene transparencia y se dibuja siempre debajo de todo el resto de los elementos. Si bien es representable como un sprite al fondo, al no moverse genera cálculo innecesario previendo su movimiento. Además, en cada frame se limpia la pantalla (lo cual puede ser muy costoso) para luego dibujar el fondo, cuando dibujar el fondo es suficiente. Por

lo tanto, la imagen de fondo se trata como un caso especial y se ofrece una operación para determinarlo en base a una imagen. También se puede especificar un color de fondo, aunque para mantener una interfaz minimal mantenemos el color de fondo fijo, y si se quiere cambiar se puede utilizar un fondo de color sólido, ya que esto tiene la misma eficiencia.

Texto

Tiene las mismas funciones y características del sprite, solo que en lugar de tener una imagen tiene un texto a escribir. Implementable con sprites, utilizando imágenes que sean el texto o segmentos del mismo y moviéndolas a la vez. En los casos en que el texto es fijo esto no representa una ineficiencia (y generalmente suele incluirse en el fondo), pero cuando el texto es muy variable, como un contador, tener una imagen por letra puede ser tanto molesto al modelar como ineficiente al correr.

Por lo tanto se decidió implementar un clón del sprite (textSprite), con las mismas funciones pero que en vez de recibir una imagen recibe un texto y un tamaño de fuente. Se pensó en recibir el nombre de la fuente también como parámetro, pero no pareció relevante.

Barras de estado

A lo largo de un modelo de simulación hay niveles de ocupación, tiempos restantes de actividad, porcentajes completados, etc., que pueden ser desplegados mediante barras de estado. Con las mismas funciones y características del sprite, en lugar de una imagen tiene un largo de barra llena y un porcentaje de completado. Cuando se dibuja, se dibuja la barra hasta el porcentaje indicado, completándola en 100%. Son factores determinables la dirección en la cual se llena la barra, la trama, el color y el ancho. Se decidió no implementar este dibujable por cuestiones de tiempo.

7.6. Caché de imágenes

Durante una simulación se suelen cargar muchos sprites que utilizan la misma imagen fuente, y el tener que acceder a disco todas las veces resulta costoso. Así que se decidió tener un mapa de path a imagen y cada vez que se intenta cargar una imagen con un nombre de archivo determinado, la imagen ya se tiene procesada. Esto hace que no se puedan cambiar las imágenes externamente para que cambien durante la simulación, pero no vemos por qué esto sería deseable.

7.7. Interacción con el usuario

Se busca lograr la mejor interacción posible con el usuario, no solo para decidir lo que se muestra sino para cambiar aspectos del modelo en tiempo de ejecución. A modo de lograr la mayor flexibilidad posible, se buscará una librería que permita una interfaz sencilla de teclado y mouse con la ventana de visualización. El cómo afecten estos dispositivos a la simulación se dejará en manos del modelador.

8. Persistencia

Para diversas técnicas especializadas de la simulación, y como rasgo positivo en aplicaciones de índole general, está la posibilidad de guardar el estado de un programa en una forma de la que luego sea recuperable. Esta funcionalidad se conoce con el nombre de serialización, persistencia o marshalling. Durante la charla de presentación de la librería EOSimulator, producto de la pasantía de uno de los integrantes del grupo fue que surgió este requerimiento.

La primera aplicación consiste en poder suspender el estado del programa para seguir en otro momento, o poder reiniciar el sistema sin perder el progreso. Los programas de simulación por lo general consumen muchos recursos y demoran en ejecutar, porque la técnica más sencilla de reducir la varianza pasa por incrementar la cantidad de mediciones. De todos modos, y con cierto trabajo, se puede estructurar un sistema para que haga varias corridas pero las mismas sean invocaciones nuevas de un programa, y no siempre la misma haciendo repeticiones. Así se puede interrumpir entre dos corridas el experimento y seguir en otro momento, lo que equivale a un esquema de migración por puntos de control hecho manualmente. Este procedimiento también permite la migración del experimento de simulación a una máquina diferente, siempre que el sistema pueda ser recompilado, lo que no es demasiado trabajo.

Una segunda técnica que se hace posible si se dispone de mecanismos para persistir una aplicación es grabar el estado actual para luego levantar el mismo varias veces cambiando alguna variable cada vez. Por ejemplo, hallar el estado estacionario de un sistema puede ser costoso. Además, el mismo se puede alcanzar en diferentes tiempos dependiendo de los torrentes de números aleatorios, así que no es posible calcular a priori cuanto esperar. Como el estado estacionario aún con métodos como el de promedios acumulados se alcanza en un punto subjetivo, se está en una difícil situación. Generalmente se estima el tiempo para llegar al estado estacionario con unos torrentes y se asume que para otros éste se conserva. Con persistencia, ahora se puede llegar al estado estacionario una sola vez con unos torrentes determinados, congelar ese estado y seguir varias corridas, con torrentes independientes, desde ese punto.

Cabe hacer una apreciación sobre el último uso de persistencia. En el fondo, alcanza con menos que persistencia para lograrlo. Si pudiera hacerse una copia de las estructuras del programa en la propia memoria, luego las ejecuciones se pueden hacer divergir como apetezca. Es más fácil esto porque, aunque igual se deben considerar una variedad de aspectos de implementación, no entran en juego la diferencia entre máquinas, compiladores ni entornos que si juegan un papel cuando se debe guardar a archivo, ni deben reificarse tanto las diferentes estructuras.

8.1. Precisión del requerimiento

Hay dos clases de objetos a considerar para realizar la persistencia del programa, los que son de la librería y los del usuario. En principio la librería no exige ninguna estructura particular para los objetos de usuario, así que dependerá enteramente del usuario como realizar su persistencia. Nótese que esto no es un problema con la librería, sino que es mejor de esta manera porque no se restringe la libertad del usuario cuando no está usando persistencia, ni se le dificulta en principio la labor cuando la usa.

No será posible brindar soporte para persistencia para algunas clases de la librería. La más dolorosa de estas excepciones es la clase process. Internamente el mecanismo para lograr los diferentes contextos de ejecución es la facilidad de multihilado inherente al sistema operativo en cuestión. En Windows se usan las threads de la API Win32, en Unix Posix Threads y para Macintosh existe una implementación específica también. Todo esto es encapsulado por la librería Boost.Thread. No es común que los sistemas operativos tengan soporte para la captura del estado de un hilo de ejecución de manera que pueda ser luego restaurada, y

menos concebible es todavía poder intercambiar un proceso de simulación entre diferentes máquinas. Entonces queda descartada la persistencia de simulaciones que utilicen interacción de procesos.

El otro gran componente excluido de la persistencia es la salida gráfica. En principio no existe ningún impedimento técnico. Si bien se debe restaurar el contexto gráfico cada vez, lo que involucra un cierto trabajo, el estado de todos los objetos no tiene mayor misterio. Este componente de la librería quedó fuera del alcance porque no es claro que exista un caso de uso que involucre tanto persistencia como visualización con un peso tal que amerite el esfuerzo. La visualización disminuye muy importantemente la velocidad con que corren los programas por lo que la aplicabilidad de las técnicas específicas de simulación que necesitan persistencia se ve debilitada. A su vez, como la visualización brinda una perspectiva global del sistema no importa tanto conservar un estado con todo el detalle; la simulación puede ser reiniciada simplemente en otro contexto de ejecución.

Focalizándose entonces en una simulación por eventos, el reto está en persistir principalmente el estado del calendario. La simulación corre en un ciclo brindado por el ejecutivo, pero como es una función aparte al calendario, se asegura que las invariantes del calendario están intactas en todo momento. Se puede hacer persistencia de un sistema desde un evento, entonces. Lo más común, sin embargo, es que se salga del ejecutivo para hacer persistencia, porque el ejecutivo puede ser vuelto a largar luego sin ningún inconveniente.

Aporta repetir aquí una característica clave de la librería. Si bien la librería ofrece una implementación concreta del concepto de calendario, la verdadera interfaz está dada por el concepto mismo. El resto de los componentes que interactúan con el calendario lo hacen a partir del concepto calendario, y no de la instancia concreta del mismo. No se modificó el calendario predeterminado de SimPP agregándole funcionalidades para la persistencia, sino que se provee de una implementación alternativa del concepto de calendario.

8.2. Alternativas de implementación

C++ no dispone de un mecanismo nativo de persistencia de objetos, como Java y muchos otros lenguajes recientes. Tampoco existen facilidades de introspección sobre las cuales construir un sistema de persistencia. Estos dos problemas complican mucho la tarea.

Aunque no se puede llegar a la facilidad que dan otros lenguajes, persistir es posible. Una librería que lo permite forma parte de Boost, y se llama Boost.Serialization. Se escriben funciones especiales mediante las cuales se informa a la biblioteca de cuales son los objetos alcanzables desde uno, y ésta sigue todo el grafo alcanzable, teniendo en cuenta ciclos. Se pueden escribir las funciones de soporte a la persistencia fuera de las clases en cuestión, para lograr introducir al sistema clases que no se pueden tocar. Como los eventos de SimPP se basan en Boost.Function e indirectamente en Boost.Bind, se debe lograr la persistencia de estos dos componentes.

Los objetos de Boost.Bind son complicados de persistir porque no exponen suficiente información. El punto de acceso a la librería, la propia función `bind` retorna un tipo indeterminado y opaco. Logra esto gracias al método de deducción de tipos de C++ para las funciones template, de tal manera que los tipos de los argumentos se deducen y el tipo de retorno depende de los anteriores, por lo que no se incomoda al usuario para que tenga que suministrar nada extra. Bajando de nivel y entrando en la implementación es posible acercarse al objeto mismo, e incluso crearlo salteándose la función `bind`. Existe una interfaz de visitación experimental que se puede usar para inspeccionar el contenido del objeto devuelto por `bind`, que es usada por Boost.Signal. Se puede usar esta entrada para persistir los miembros del objeto, aunque hay que cambiar el código fuente de Boost.Bind para que permita también la escritura de los miembros, y no sólo la lectura.

El problema es que el propio objeto función que se está adaptando con Boost.Bind tiene que poder persistirse. En general se usan funciones libres o miembro, porque son más fáciles de escribir y toda la adaptación se hace con Boost.Bind. Los punteros a función no se pueden persistir mediante Boost.Serialization, ya que son completamente opacos. Habría que registrar los punteros a función posibles en una tabla junto a algún identificador, para hacer la conversión al grabar y escribir.

Ahora, un pequeño truco ahorra este trabajo. Además de tipos, valores constantes como la dirección de un objeto estático pueden ser argumentos para templates, en particular, la dirección de funciones. Un puntero a función pasado como argumento de un template no tiene costo de indirección al ser invocado, porque se resuelve todo en tiempo de compilación. Entonces, se puede hacer una clase template objeto que reciba un puntero a función como parámetro template y delegue a éste. Persistir este objeto no es necesario porque es vacío, su comportamiento lo determina totalmente su clase. La persistencia se reduce a reconstruir el objeto, nada más. Desgraciadamente, no es posible capturar un puntero a función como argumento template si no se conoce el tipo, y el mismo no se puede deducir automáticamente. En retrospectiva, éste es el mismo problema que se presenta con la clase member de Boost.MultiIndex, explorado en la sección del caso de estudio. La misma solución que toma Boost.MultiIndex es posible, dando los tipos del puntero a función explícitamente. Evidentemente este camino era problemático, sino Boost.Bind ya tomaría el puntero como una constante para ahorrarse la indirección.

Esta solución es bastante entrometida en los interiores de Boost.Bind, hace varias suposiciones sobre su comportamiento que no están documentadas y aún así no logra satisfacer plenamente. Sin embargo, funciona después de todo. En el peor de los casos, se puede copiar la implementación de Boost.Bind, hacer las modificaciones necesarias y tener una librería similar con soporte para persistencia mediante Boost.Serialization.

El otro escollo es la persistencia de Boost.Function, la base de los eventos. Nuevamente, la interfaz no expone suficientes detalles de implementación como para realizar limpiamente el trabajo. Sin embargo, éste es un componente mucho más simple que Boost.Bind, y solo nos interesa persistir la variedad que no toma ni retorna ningún valor. Con estas restricciones, es fácil implementar un sustituto que exponga lo que sea necesario para su persistencia, siguiendo la guía de [Ale01], mediante una clase base abstracta. Boost.Serialization soporta la persistencia de objetos de clases derivadas a través de referencias del tipo de una clase base, pero con complicaciones. En C++ no es posible crear un objeto dado el nombre de una clase, porque no existe un diccionario donde se almacene la información pertinente. Boost.Serialization mantiene este diccionario, y las clases a ser persistidas polimórficamente tienen que registrarse a si mismas allí. Para el escenario común de derivación alcanza con incluir una macro en el archivo fuente de la clase derivada, dando un nombre a la misma, que si se omite se toma igual al de la propia clase. Sin embargo, si se trata de clases generadas mediante templates, no existe tan fácilmente un nombre para la clase. Se puede inventar algún esquema que combine el nombre de los parámetros del template y registrar la clase cuando es instanciada. No es posible entonces restaurar la clase en una aplicación que no instancie esa variedad del template. Otra solución, que se implementó, fue usar los nombres de las clases que expone C++ como su pobre soporte de reflexión. Esto no es portable porque el estándar no define con exactitud que esperar de los compiladores. Los que probamos, G++ y VC++, difieren ampliamente. Esta solución funciona perfectamente si el programa es exactamente el mismo, lo que se cumple en nuestro caso. Un brillo de luz se aporta de parte de G++, que implementa un esquema estandarizado y soportado por otros vendedores. De cualquier manera, este enfoque, aunque funciona, es demasiado frágil.

8.3. Solución adoptada

Tras la investigación realizada sobre como introducir persistencia en Boost.Bind y Boost.Function ese enfoque debió ser abandonado porque aunque los problemas habían podido ser sorteados hasta el momento, no se tenía la seguridad de haber mitigado suficientemente los riesgos. Nunca quedó la sensación de que no surgirían dificultades sin solución más adelante en el camino, e incluso si fuera así, la dependencia en detalles de las implementaciones de las librerías y los compiladores utilizados no brindaba mayor seguridad, porque estos cambian fuera del control de la librería.

El punto de quiebre para buscar otra solución es aceptar cierta degradación del manejo de eventos por parte del usuario. Los objetos funciones brindan gran poder, y existen herramientas especializadas para manejarlos, pero esta flexibilidad resultó difícil de adaptar al marco de Boost.Serialization. Si se usa otra herramienta más restrictiva es posible subsanar este problema.

La solución que se adoptó consiste en usar clases abstractas de las que deben heredar las clases para los eventos que necesite el usuario. Cada clase del usuario define los métodos necesarios para persistir su parte del objeto, y delega luego a su clase base, como requiere Boost.Serialization. También se tiene que registrar la clase con la librería. Como lo más común es que además de los eventos deba ser persistida cierta información del propio modelo, en el método serialize se enumeran todos los tipos de eventos, para que Boost.Serialization los reconozca al cargar.

Hay que notar que el enfoque asemeja entonces bastante el uso de SimPP al de EOSimulator. En el segundo los eventos se tienen que crear separadamente y hay que darles un nombre. En un punto de la implementación del modelo, los eventos se registran al modelo. Entonces, aunque el esfuerzo de implementación se incrementa para los usuarios de la librería, esto no es excesivo, porque es lo habitual en otras.

En resumen, se logró introducir el requerimiento de persistir el estado de una simulación, el cual apareció tardíamente para el proyecto. Esta característica diferencia SimPP de otras librerías existentes. El objetivo se logró sin impactar en sobremanera el marco conceptual con el que están clasificados los componentes de SimPP, gracias a la definición de interfaces para los componentes, en esta caso mediante conceptos. Si bien al usuario se le imponen ciertas restricciones, el sistema es usable.

Como trabajo futuro queda explorar la posibilidad de clonar en memoria el estado de la simulación, sin realizar la conversión a un torrente de bits. No se pudo abordar para el proyecto porque no se disponía de una herramienta adecuada. Posiblemente Boost.Serialization pueda ser adaptada de manera no invasiva para lograrlo.

9. Referencias

[Pau93] PAUL, Ray J. Activity cycle diagrams and the three-phase method. Los Angeles, California, United States: Proceedings of the 25th conference on Winter simulation, December 1993. p.123-131.

[DO89] DAVIES, Ruth M.; O'KEEFE, Robert M. Simulation modelling with Pascal. Hertfordshire: Prentice Hall, 1989. 302 p. ISBN: 0-13-811571-0

[Bbind05] Boost bind documentation

http://www.boost.org/libs/bind/bind.html Setiembre 2005.

[Bsrzt05] Boost serialization documentation

http://boost.org/libs/serialization/doc/index.html Setiembre 2005.

[Bsgnl05] Boost signal documentation

http://boost.org/doc/html/signals.html Setiembre 2005.

[Bfunc05] Boost function documentation

http://boost.org/doc/html/function.html Setiembre 2005.

[Jsrzt05] Java serialization documentation

http://java.sun.com/docs/books/tutorial/essential/io/serialization.html Setiembre 2005.

[EOS05] EOSimulator Home Page

http://www.fing.edu.uy/inco/cursos/simulacion/paginaEosim/index.htm Julio 2005

[FDFHP93] FOLEY, James D.; DAM, Andries van; FEINER , Steven K.; HUGHES, John F.; PHILLIPS, Richard L. Introduction to Computer Graphics. Addison-Wesley Professional, 1993. 632 p. ISBN: 0-20-160921-5

[Pthr05] POSIX threads

http://www.open-std.org/jtc1/sc22/WG15/ Diciembre 2005

[Rwinthr05] Run-Time Library Reference _beginthread, _beginthreadex

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_crt__beginthread.2c_._beginthreadex.asp Diciembre 2005

[Ale01] ALEXANDRESCU, Andrei. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, February 2001. p. 352. ISBN: 0-201-70431-5

Apéndice E – Librerías externas utilizadas

1. Introducción

Para la construcción de SimPP se utilizaron varias librerías externas. En esta sección se hace una breve descripción de ellas.

Se busca utilizar herramientas mantenidas y probadas de la programación en general para adaptrarlas, a través de las herramientas de nuestra librería, al modelado de sistemas para simulación a eventos discretos. Se busca además poder satisfacer las decisiones de diseño del apéndice D. De hecho, debido a la naturaleza iterativa del proyecto, muchas de estas decisiones se tomaron teniendo en cuenta la variedad de librerías ofrecidas.

Dado que se buscan objetivos de portabilidad para nuestra librería, se requiere que las librerías utilizadas también sean portables.

2. Boost

Boost es un compendio de diferentes librerías para C++ en el estilo de la librería estándar. Un objetivo es establecer "existing practice" y proveer implementaciones de referencia para la futura estandarización de los componentes. Diez librerías de Boost ya se encuentran dentro del primer reporte técnico del comité de estandarización y se incluyen con las últimas versiones de los compiladores más comunes.

2.1. Características

El proyecto nació en 1998, luego de finalizado el trabajo en el primer estándar internacional de C++. Las regularizaciones internacionales mandan un período de estabilización de varios años antes de realizar modificaciones a un estándar. Por eso se propuso Boost como un ámbito para seguir el trabajo en posibles librerías para un próximo estándar.

La comunidad de C++ no poseía anteriormente un punto de referencia como otros lenguajes, debido a que se usa en variados ámbitos y no tiene el apoyo incondicional de ninguna empresa multinacional, como es el caso de Java y C#. Incluso Perl tiene CPAN. Para lograr los objetivos propuestos, y debido a que los fundadores del proyecto provenían del ámbito de estandarización, se instauró un proceso de revisión por pares para asegurar la calidad de las librerías. Antes de ser aceptada, una librería es examinada por otros desarrolladores, y hay un período en el que se reciben los comentarios que cada uno tenga que hacer sobre la propuesta. Existe el rol de review manager, que es quien decide si la librería es aceptada o no.

La licencia de Boost es Open Source. Originalmente cada librería podía tener su propia licencia, mientras que ésta cumpliera ciertos requisitos. El uso comercial cada vez más acentuado de Boost hizo necesario uniformizar la licencia para hacer más fácil la adopción. El proceso de cambiar la licencia no está completo del todo, ya que ciertos autores están desaparecidos y no pudo contactárseles.

2.2. Componentes

Habitan en Boost un variado conjunto de librerías. Algunas de las mismas no pertenecen a ningún dominio en particular, sino que expanden la capacidad del propio lenguaje. Boost contiene, sin ser exhaustivos:

- **any:** contenedor de valores heterogéneos, manteniendo la seguridad de tipos
- **assign:** llenado de contenedores
- **bind:** función de alto orden para la adaptación in situ de functores
- **crc:** librería para el manejo de códigos de redundancia cíclica
- **date_time:** tipos para el manejo de fechas, horas y tiempos
- **filesystem:** operaciones portables sobre el sistema de archivos
- **format:** sustituto de las funciones de la familia printf
- **function:** functor genérico y opaco, para cualquier firma
- **graph:** componentes y algoritmos para el manejo de grafos
- **lambda:** creación de functores anónimos
- **mpl:** framework de meta-programación
- **multi_array:** contenedores multidimensionales
- **multi_index:** contenedores indexados por múltiples claves

- **optional:** contenedor posiblemente vacío de un solo elemento
- **parameter:** creación de funciones con parámetros opcionales por nombre
- **program_options:** datos de configuración a partir de diferentes medios
- **python:** librería de comunicación python-c++
- **random:** generación de números seudo aleatorios
- **regex:** expresiones regulares
- **serialization:** escritura y lectura de estructuras de datos a archivo
- **smart_ptr:** punteros inteligentes, para el manejo de la vida de los objetos
- **spirit:** framework de parser combinators embebidos en c++
- **test:** utilidades para el testeo e instrumentación de código
- **thread:** multi-hilado
- **utility:** diversas utilidades pequeñas, como noncopyable
- **variant:** unión discriminante

2.3. Usos de boost en SimPP

Algunas piezas de Boost juegan papeles fundamentales en nuestra librería. En particular function forma la base de los eventos en el scheduler por omisión de la librería, aunque se puede sustituir por componentes similares, por ejemplo los que se brindan en la librería loki o en libsig++.

bind se usa en varias partes de la implementación, aunque no en la interfaz de la librería. Sin embargo, es la manera más común con que se crean los functores para los eventos en las aplicaciones de usuario de demostración y el caso de estudio. Podría haberse usado Boost.Lambda, también, pero no se vio la necesidad de hacer más que adaptaciones a functores ya existentes, y bind tenía la ventaja de estar en el próximo estándar de C++. Otra alternativa hubiera sido phoenix, que forma parte de spirit. Está planeada la fusión de phoenix y lambda.

Para el enfoque a interacción a procesos se utilizaron las facilidades de multi-threading que brinda el propio sistema operativo. Existen interfaces estandarizadas para acceder a esta funcionalidad en posix, pero no están presentes nativamente bajo windows. Aunque existen varias implementaciones de posix threads para windows como add-ons, incluso de Microsoft, su instalación se hacía problemática y no dejaban de ser interfaces en C. Boost.Thread en cambio presenta una interfaz moderna en C++, en base a functores y con soporte para los idiomas más potentes para el manejo de los objetos de sincronización.

Una característica sobresaliente de SimPP es la ausencia de generadores de números aleatorios. Esto sucede porque se supone la presencia de una librería externa que provea esa funcionalidad. Se tuerce un poco la balanza para el lado de Boost.Random, porque está entre las librerías de pronta estandarización. Brinda una gran variedad de generadores y distribuciones y la interfaz es muy conveniente en combinación con el resto de SimPP. Por ejemplo, que las distribuciones sean functores hace que los feeders de nuestra librería puedan usar distribuciones de Boost.Random sin adaptación necesaria.

Boost.MPL no vio uso en nuestra librería, por una cuestión de tiempo. Aunque quedó fuera del alcance, estaba pensada la realización de un subsistema basado en grafos de eventos, que se situara paralelo a la interacción de procesos. Para que la interfaz fuera amigable al usuario, hubiera sido necesaria mucha plomería por debajo, aunque por suerte hay referencias

adecuadas sobre como lograr este cometido, en [AG04] , a disponibilidad de los integrantes del grupo.

Un requerimiento de SimPP era poder congelar el estado de la simulación para retomarla en otro momento. Aunque no se halló una solución del todo satisfactoria, si se pudo implementar este requerimiento, con compromisos de usabilidad. La base de todo este trabajo la forma la librería Boost.Serialization, que permite convertir bidireccionalmente una estructura arbitraria en una lista de bytes. Como C++ no soporta ningún mecanismo de introspección, es necesario anotar los tipos mediante funciones especiales. Además, puede ser necesario registrar las clases en uso para que el framework pueda crear el objeto del tipo adecuado. Estos requerimientos chocan con el uso de pequeñas utilidades como bind y function, ya que la serialización debe esparcirse a todos los tipos para funcionar. Afortunadamente, igual fue posible acomodar esta funcionalidad en la libraría, aunque se fuerza al usuario a utilizar mecanismos más manuales para agendar eventos y otros usos del scheduler.

Los smart pointers se han convertido en una herramienta fundamental para el uso efectivo de C++, y fueron necesarios para la construcción de SimPP. Por ejemplo, los procesos son objetos que juegan el lugar de la ejecución suspendida de otra thread en nuestro sistema. No puede copiarse realmente una thread, pero es muy inconveniente para el usuario que los objetos procesos no sean copiables, porque no pueden entonces meterse en colecciones. Sin embargo, también es necesario saber cuando no existen más procesos de cierta thread en el sistema, para poder cancelar la thread limpiamente. El uso se presta para usar la técnica de reference counting, pero esta es demasiado engorrosa de implementar a mano, sobretodo considerando que el sistema es multi-threaded. La clase shared_ptr sirve en este caso. Otros punteros de boost vieron casos de uso en otros puntos de la librería o de sus aplicaciones.

Para los tests unitarios se utilizó la libraría Boost.Test. Existen otras alternativas, confeccionadas en base a librerías de otros lenguajes, principalmente jUnit, pero para programas en C++ no son las soluciones más idiomáticas. Por ejemplo, Boost.Test permite la prueba de funciones libres y omite la funcionalidad de construcción y destrucción de un ambiente de pruebas, que en otras herramientas se conoce como setup/teardown, porque la misma se puede obtener con facilidades del lenguaje. SimPP realiza pruebas unitarias para todas sus partes, pero muy especialmente para las funcionalidades de soporte de interacción de procesos, porque es código muy delicado por la interacción entre hilos. Técnicamente Boost.Test no soporta ambientes multihilados, pero no tuvimos inconvenientes, ya que siempre se ejecuta un hilo a la vez.

Otras librerías fueron utilizadas también, de las que se hará un recuento rápido. multi_index se utilizó para el manejo de las colecciones de objetos del dominio en el caso de estudio (ver Apéndice F): los buses, lineas, zonas, etc. noncopyable es super clase de varias clases de la librería, indicando claramente que no se pueden copiar, y obligando al compilador a no permitirlo.

3. SDL

Para la visualización gráfica era necesario acceder a los recursos del sistema operativo con un nivel de control mayor al que expone C++ directamente. A su vez, no se quería perder portabilidad, ni tener que manejar de manera dispar diferentes entornos. SDL, la Simple DirectMedia Layer, cumple este cometido.

3.1. Características

SDL es una librería multimedia para el lenguaje C. Provee acceso a los recursos en materia de gráficos, sonido, multithreading y entrada de usuario. Corre en una muy variada cantidad de sistemas, que incluyen Linux, Windows, BeOS, MacOS Classic, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX, y QNX.

Mediante SDL un programa puede crear gráficos en una superficie lineal del tamaño que sea necesario. SDL encapsula todo el código necesario para la transformación al formato nativo del sistema operativo. Se soportan efectos de transparencia total y parcial, así como aceleración por hardware si está disponible.

Existe una cola especial mediante la cual SDL comunica a la aplicación de los eventos relevantes para las aplicaciones multimedia, como son los movimientos del ratón u otro tipo de punteros, ocultamiento o aparición de ventanas y pedidos de salida. La interfaz es puramente pull para la aplicación, ya que no se registra un callback, sino que se reciben los mensajes explícitamente mediante funciones exportadas por SDL. Así, es fácil combinar SDL con otro tipo de librerías que quieran manipular el ciclo de funcionamiento del programa, como librerías de comunicación por redes, por ejemplo.

3.2. Usos de SDL en SimPP

SimPP usa SDL para establecer la superficie gráfica sobre la que dibujar. No se usan las primitivas ya existentes, porque dibujar manualmente es más flexible y resultó que la velocidad extra que usar las funcionalidades ya ofrecidas por SDL no fue necesaria. Se hizo cierto esfuerzo por minimizar la cantidad de copias que se realizan entre diferentes formatos gráficos, para que la visualización con SimPP funcionara en máquinas de gama más baja que las que disponemos los integrantes del grupo. El problema radica en que aunque es posible consultar el formato de la superficie lineal más nativa que puede ofrecerse en cierto sistema, con una granularidad de pixeles la adaptación del código lo vuelve demasiado lento. Es posible generar automáticamente código optimizado para diversas configuraciones posibles mediante metaprogramación, y de hecho la infraestructura está presente en el subsistema de rendering usado en SimPP, pero no prever las posibles configuraciones de antemano. Por estas razones igual existe una superficie intermedia con formato fijo, aunque se trató de que se dieran las condiciones para que SDL no creara más.

Hubiera sido posible implementar la interacción a procesos de SimPP sobre las primitivas de concurrencia que brinda SDL, y así limitar la dependencia en Boost. Sin embargo, aunque simple y portable, la interfaz de SDL no deja de ser una librería con una interfaz y estilo propios del lenguaje C, y no C++. Aunque esto queda totalmente oculto al usuario, preferimos manejarnos con la opción que brinda Boost de todas formas.

Al escribir estas líneas nos surgió la curiosidad sobre un detalle de implementación de las primitivas de creación de threads en SDL. Bajo Windows existen dos interfaces para la creación de threads, y solo bajo una es seguro utilizar las funciones de las librerías estándar de C++. Una rápida inspección del fuente de SDL revela que se utiliza la función incorrecta. Es destacable que aunque en falta, SDL tiene la bondad de ser software opensource, y como tal

un simple usuario pudo identificar su problema. Este asunto está siendo tratado en las listas de correo de SDL.

4. Anti Grain Geometry

AGG [AGG05] es una librería en C++ para la generación de imágenes 2D desde datos vectoriales. La filosofía de AGG es obtener la mayor calidad posible, rivalizando otras propuestas, incluso de carácter comercial. Además, AGG presenta una arquitectura única, fuertemente basada en el mecanismo de templates de C++, por lo que podría considerarse que uno genera el renderer que necesita para la aplicación concreta.

4.1. Características

AGG contiene las piezas para generar imágenes 2d de muy alta calidad. Combinándolas de la manera adecuada se pueden obtener los siguientes efectos, entre otros:

- Dibujo de polígonos sin serruchos y con precisión de menos de un pixel
- Gradientes y Sombreado Gouraud
- Líneas punteadas
- Marcadores y flechas
- Imágenes arbitrariamente rotadas
- Cortado de primitivas respecto a rectángulos, tanto vectorial como a bajo nivel
- Máscara por alpha
- Líneas sin serruchos
- Uso de imágenes como patrones

El enfoque de AGG hacia el problema de abarcar un dominio tan extenso es parecido al de STL. Se deben encontrar los conceptos fundamentales tanto para las representaciones de datos como para los algoritmos, y hacer que ínter operen libremente. En este punto se distancia enormemente de lo que hacen casi todas las librerías gráficas. Por lo general, se tiene un punto único de acceso a toda la funcionalidad, por lo que centenares de funciones comparten un mismo ámbito. Esta característica es compartida por OpenGL, DirectX, GDI+, Java2D, DisplayPDF, AdobeSVG. Adobe está en estos momentos haciendo una transición a componentes más separados en toda su línea de productos, también fuertemente basado en el paradigma de STL, lo que no sorprende a la luz de que es donde trabaja actualmente Alex Stepanov, el padre de STL.

Los componentes de AGG se organizan en capas, de manera que uno configura cada capa por separado o incluso se omite una capa si es lo deseado. La primera capa es la fuente de datos vectoriales. Aquí AGG no tiene demasiada incumbencia, ya que es algo totalmente controlado por la aplicación. Aunque AGG evita el uso de STL para poder funcionar en ambientes pobres que no implementan totalmente la librería estándar, es totalmente posible y concebible usar contenedores estándar para esta parte de la aplicación.

La segunda capa es la que convierte las coordenadas de los vértices mediante las reglas de la aplicación. Todos los datos son de punto flotantes (`double`) porque todavía no se está en la etapa en que se baja al nivel de los píxeles. Aquí es cuando se generan los bordes de los objetos, las puntas de las flechas y se separan las líneas puntuadas en cada segmento. Todas las transformaciones son a nivel de objeto.

El proceso de rasterización se hace en dos etapas. Primero las primitivas se convierten en líneas de píxeles horizontales, marcadas por el cubrimiento. Si el rendering fuera inmediato al buffer final, no sería posible considerar el cubrimiento que cada primitiva aporta a cada píxel.

Además, tener dos etapas permite introducir en el medio una variedad de efectos, como gradientes, sombreado interpolado, trasformaciones de imágenes y patrones.

El destino último, desde la perspectiva de AGG, es un buffer en memoria. Lo que la aplicación realice a posteriori con la imagen sintetizada, ya sea mostrarla en pantalla, imprimirla, guardarla, etc., no es de incumbencia para AGG. El formato del buffer tampoco lo fija AGG. Se pueden usar la cantidad de bits que se requieran para componente o incluso que cada componente sea representado por un número de punto flotante a precisión completa. Tampoco está prefijado el espacio de colores con el que se trabaja. Se logra esta flexibilidad sin perder nada de performance porque los componentes que tratan con píxeles están parametrizados con el tipo de éstos.

4.2. Uso de AGG en SimPP

Se evaluaron varias herramientas para la construcción de SimPP, entre ellas OpenGL, Cairo [Cai05], SDL y AGG. OpenGL y SDL tenían la ventaja de ya ser conocidas por el grupo, con lo que el riesgo al utilizarlas se minimizaba. Sin embargo, SDL no daba demasiada flexibilidad, y OpenGL requería de hardware especializado. Como se trata en la sección de decisiones diseño sobre visualización, no es tan importante la fidelidad de los dibujos a la hora de la visualización de una simulación, sino que importa más la claridad y la cantidad de información que puede mostrarse al mismo tiempo sin confundir. Era por eso buena la posibilidad de usar una herramienta que permitiera que la visualización creciera en complejidad.

Cairo es una herramienta para la realización de dibujos de alta definición, al igual que AGG. Una diferencia es que tiene una arquitectura mucho más clásica y rígida. Esto trae la ventaja de que cuenta con varios backends, para los sistemas donde corre, y no es siempre por software como AGG. En particular, cuenta con backends para win32, OpenGL y X, que son las plataformas a las que apunta SimPP. Al momento de realizar este proyecto, Cairo vivió una explosión de su exposición en los medios. Diversos proyectos de alto perfil, como Mozilla y Gnome han estado experimentando con basar el rendering de sus aplicaciones sobre Cairo, para así lograr uniformizar los resultados en distintas plataformas e incrementar la performance, además de eliminar algunas dependencias. Por estas razones Cairo era un gran candidato para las necesidades de este proyecto, y fue investigada antes que AGG. Sin embargo, la documentación de Cairo era muy poca, y se consideró riesgoso intentar aprender una tecnología nueva directamente del fuente y las aplicaciones de muestra.

AGG tampoco contaba con una documentación demasiado extensa, por tratarse sobretodo de un proyecto mayoritariamente de una sola persona. Sin embargo, la arquitectura capturó desde un principio al grupo, y contaba con aplicaciones de muestra por cada característica individual, lo que hacía más fácil encarar el aprendizaje.

Para las propias aplicaciones que vienen con AGG, éste brinda una limitada pero útil interfaz con el sistema operativo subyacente, que permite recibir eventos e interactuar con el sistema de ventanas. Esto toma la forma de una clase de la que el cliente hereda, teniendo la clase base diferentes implementaciones, que son X, Win32 y SDL, entre otras. Aunque no terminamos utilizando esta funcionalidad, el backend de SDL ayudó en gran medida a hacer la integración nuestra con este toolkit, aunque en la versión final casi no existe código extraído de esa fuente.

El primer paso en el uso de AGG fue dibujar varios sprites que representan entidades y moverlos por la pantalla. Así se mitigaba uno de los riesgos más importantes, que era que la performance no fuera adecuada. Las demos funcionaban rápido y eran bastante impresionantes, pero nunca está demás comprobar uno mismo. Cuando quisimos hacer cosas más sofisticadas, sin embargo, volvimos a tener problemas con este punto. Originalmente, el fondo de la visualización era internamente un sprite y se dibujaba mediante AGG. Dadas las proporciones de la imagen, sin embargo, esto resultaba altamente ineficiente. Se procedió

entonces a un dibujado mediante SDL directamente del fondo, lo que en realidad no tiene efectos adversos porque como el fondo nunca está en coordenadas no enteras, el resultado es el mismo que si se hiciera el dibujado con tratamiento sub píxel completo.

Otra cosa a destacar es que nunca llegamos a un uso de verdad fundamental de AGG. Aunque la calidad del resultado es superior al que se puede obtener usando SDL nada más, no se aprovecha ninguna de las características únicas que AGG brinda, como fue pensado en un principio, por razones de alcance del proyecto. No deja de ser, igual, una excelente herramienta, con la que puede obtenerse una gran ventaja a la hora de realizar aplicaciones gráficas.

5. Standard Template Library

La STL es una librería de contenedores, iteradores y algoritmos, abarcando varias de las soluciones más comunes en este dominio. Como su nombre lo indica, está fuertemente basado en la facilidad de templates que ofrece C++. Sin embargo, trasciende esta realidad, y es la primera librería en el enfoque genérico.

5.1. Características

STL debe su popularidad a ser una parte importante de la librería estándar provista por C++, y disponible entonces en todas las implementaciones del lenguaje. Originalmente la librería era más grande de lo que se incluye con C++, pero debió ser reducida su cantidad de componentes para ser aceptada en el estándar del 98, porque era una gran adición que se presentaba bastante tarde en la etapa de estandarización. Los componentes más extrañados por los usuarios que faltan en la versión definitiva son las tablas de dispersión, pero muchos vendedores las incluyen como una extensión. Recientemente este componente ha sido re introducido, con el nombre de unordered_map, unordered_set y unordered_multiset. Además, la librería trae los siguientes objetos: vector, list, deque, set, multiset, map, multimap, queue y priority_queue. Independientemente, STL provee varios algoritmos para manipular los datos de los contenedores, como reverse, for_each, partition, find, etc..

La lista de algoritmos puede ser alargada por el propio usuario, sin ninguna desventaja frente a los algoritmos ya provistos por la librería. Lo mismo sucede con los contenedores. El secreto de esto es que los algoritmos y los contenedores no se conocen entre si. Lo que los une son los iteradores, que son una generalización de los punteros. Aunque aborrecida por muchos programadores, la aritmética de punteros es una herramienta poderosa a la hora de trabajar con arreglos. La idea clave de la STL es que es posible generalizar este concepto, de tal manera de idear una abstracción que cumplan los punteros pero que también pueda ser satisfecha por otros componentes diferentes. Los iteradores encapsulan el moverse por sobre una estructura. Dependiendo de la estructura de la que se trate, este movimiento está acotado en diversos aspectos. Por ejemplo, una lista encadenada simple solo permite ir hacia adelante, y de a un elemento por vez, mientras un arreglo permite moverse hacia adelante y hacia atrás, de a tantos elementos como se quiera. Se establece entonces una clasificación de los iteradores de acuerdo a sus prestaciones, y los algoritmos se escriben contra esta especificación, no una estructura concreta.

Como C++ es utilizado en gran parte de los proyectos que requieren eficiencia sin perder abstracción, la STL se adecua a tal uso. Todas las operaciones están etiquetadas en la documentación con el costo aproximado en notación de órdenes. Incluso los conceptos tienen tales requerimientos. Como ya se mencionó, un iterador de listas encadenadas no permite avanzar de a más de un paso. Lógicamente la operación es plausible, pero aunque en el funcionamiento los resultados son los esperados no es posible razonar sobre la complejidad algorítmica de una operación que haga incrementos sobre un iterador que pueden tomar una cantidad de tiempo no acotada. Las operaciones potencialmente caras están presentes, entonces, pero con interfaces ligera e intencionalmente menos convenientes. Para incrementar un iterador que no necesariamente soporta la operación nativamente existe la función libre std::advance, por ejemplo, que está especializada para el caso en que se puede hacer lo pedido de un solo paso o recurre a una sucesión de incrementos simples si no es el caso.

Otra característica sobresaliente del diseño de la STL es el cuidado en la especificación de los efectos esperables en el caso de situaciones excepcionales. Esto no formaba parte del diseño original de Alex Stepanov, sino que fue introducido en una fecha posterior por David Abraham, uno de los principales promotores de Boost en la actualidad. En un lenguaje imperativo como C++, es imprescindible mantener bajo control el estado de las variables en memoria, para

asegurar la consistencia. Este análisis se realiza comúnmente con las herramientas de pre y poscondiciones e invariantes. En un lenguaje con soporte para excepciones, como C++, todo se complica aún más, porque aparecen en el código arcos de control implícitos generados por la posible elevación de excepciones por parte de las funciones que son invocadas. Como C++ soporta overloading de operadores, aún un trozo de código que parezca sencillo puede acarrear sorpresas, y la ausencia de un recolector de basura hace que no se pueda esconder el código de limpieza bajo el tapete. Esta es la gran diferencia de C++ sobre todos los otros lenguajes que hemos visto, ambas su fortaleza y mayor debilidad. En el contexto de la librería estándar, como los componentes son parametrizables por tipo, juega también un papel el comportamiento de los objetos que se están manipulando genéricamente, por lo que muchas veces el comportamiento se debe especificar en base a estos.

El comportamiento de los componentes de la STL frente a excepciones puede ser uno de los siguientes tres. El componente puede ofrecer la garantía básica, que quiere decir que si se sale de un método debido a una excepción su estado es válido aunque indeterminado. En particular, es seguro destruir el objeto, algo muy importante para la disciplina de destructores de C++. La segunda opción es que el componente prometa que su ejecución siempre es exitosa y nunca se devuelva una excepción. Este es el contrato más fuerte posible. Por último, una opción intermedia es que el componente ofrezca una garantía de atomicidad, llamada la garantía fuerte. Un objeto de estas características o tiene éxito en su invocación o se devuelve una excepción pero el estado no cambia respecto al actual al tiempo de realizarse el llamado.

5.2. Uso de la STL en SimPP

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

—Mark Weiser, "The Computer for the 21st Century"

Pese al tratamiento positivo que se le dio a la STL en los párrafos anteriores. No fue tan evidente encontrar donde los puntos en que fue utilizado en la librería. De ahí la cita de esta parte.

La influencia más grande es en base a prácticas de programación y diseño. SimPP utiliza muy poco polimorfismo dinámico. La palabra clave virtual no aparece ni una vez en todo el código. Los eventos que utiliza el scheduler por omisión, sin embargo se pueden construir a partir de cualquier clase de functor. Esto se logra gracias a la clase function de Boost, que internamente usa polimorfismo dinámico para encapsular el tipo del verdadero functor. Llegado al caso, la implementación de Boost.Function tampoco usa funciones virtuales, sino punteros a funciones, pero el detalle no importa en gran medida. El estilo de usar polimorfismo dinámico solo cuando es necesario y dejárselo disponible al cliente de la aplicación es una lección de la programación dinámica.

El uso de punteros explícitos también es raro en SimPP. Sucede que para poder interoperar correctamente con las clases de STL, los tipos deben cumplir con los conceptos de Copiables, Asignables, etc. Por eso muchas veces un puntero está en desventaja, porque aunque cumple con esos conceptos, la semántica no es rica y se pierde la posibilidad de que tareas rutinarias como la destrucción en los tiempos adecuados sea realizada automáticamente por STL. SimPP se toma el cuidado de que la mayoría de los objetos sean copiables, por ejemplo, los procesos y sprites, aunque internamente se esté manipulando a la misma entidad a través de todas las copias. Así, el código del cliente se simplifica enormemente, especialmente si además usa la STL, que es algo que queremos incentivar a hacer.

El uso explícito de STL en SimPP, además de internamente en la implementación, se da en las colecciones que se ofrecen al usuario. SimPP no ofrece colecciones. Análogamente a lo que se hace con la generación de números aleatorios, se depende de una librería externa, en este

caso la propia del compilador. Esto evita los problemas de inventar una solución nueva, y ofrece gran calidad en las dimensiones de familiaridad y performance a los usuarios. El trabajo necesario pasa por diseñar las interfaces del resto de las clases de SimPP para que funcionen adecuadamente con la STL, como ya se mencionó. Esto lejos de ser una carga es un hilo conductor en la especificación de los componentes.

6. Referencias

[AG04] ABRAHAMS, David; GURTOVOY, Aleksey. C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond. Boston:Addison-Wesley, November 2004. ISBN: 0-321-22725-5

[Bst05] Boost Home Page

http://www.boost.org Mayo 2005

[Cai05] Cairo Home Page

http://cairographics.org/introduction Setiembre 2005

[Daw05] Boost for Visual C++ Developers by Beman Dawes

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/boostvc.asp Diciembre 2005

[Wal05] An Introduction to Boost by Andrew Walker

http://www.codeproject.com/vcpp/stl/boostintro.asp Diciembre 2005

[SDL05] SDL Home Page

http://www.libsdl.org/index.php Agosto 2005

[AGG05] Anti Grain Geometry Home Page

http://antigrain.com Setiembre 2005

[Abr98] ABRAHAMS, David. Exception-Safety in Generic Components. London, UK: Springer-Verlag, Lecture Notes In Computer Science, Selected Papers from the International Seminar on Generic Programming, 1998. v.1766, p.69 - 79. ISBN: 3-540-41090-2

[STL05] STL by SGI

http://www.sgi.com/tech/stl/ Agosto 2005

[Jos99] JOSUTTIS, Nicolai M. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley Professional, August 1999. ISBN: 0201379260

Apéndice F – Caso de Estudio

1. Introducción

El proyecto termina con la utilización de la librería para el modelado de un sistema. Se estudia entonces la utilidad de cada una de las herramientas implementadas y los métodos de uso. Se demuestra además la aplicabilidad de la librería para el uso requerido, dado que el sistema a modelar tiene elementos típicos del área objetivo de la librería.

2. Descripción

Se modelará el sistema de transporte público de la ciudad de Rivera. Este consiste en una serie de líneas cuyos recorridos cubren la ciudad. Dichos recorridos son una secuencia de paradas distribuidas a lo largo de toda la ciudad, pudiendo ser circulares o de ida y vuelta. Como en cualquier otro sistema de este tipo las paradas son visitadas por varias líneas.

Otro componente del sistema son los ómnibus o buses (los dos términos se utilizarán indistintamente). Estos tienen que seguir el recorrido definido por la línea a la que han sido asignados.

Finalmente están los pasajeros. Estos llegan a una parada y esperan llegar a su destino. En la parada esperan hasta que llegue un ómnibus capaz de llevarlos a su destino. Cabe destacar que este ómnibus no necesariamente es el primero que llegue a la parada, esto depende de las preferencias de cada pasajero. Luego de que el pasajero se sube a un ómnibus de su preferencia, este baja en la parada más cercana a su destino. Por una definición más detallada del sistema e hipótesis asumidas al respecto referirse a [Mau05].

3. Elección del caso de estudio

Hubo varias razones para elegir este sistema a modelar:

- La realidad a modelar es bien conocida por los tutores del proyecto. Por lo tanto pueden actuar como clientes a la hora de recabar requerimientos (clientes técnicos, lo cual es mucho mejor) y suministrarnos los datos de entrada al problema, tanto los estadísticos como los determinados para el funcionamiento.
- El sistema en cuestión contiene elementos clásicos a los sistemas usualmente modelados por herramientas similares. Colas de espera, actividades con duración, recursos, estado estacionario, etc. Probar el modelado de estas características es clave en este estudio.
- Se trata de un caso real con datos reales, no un ejercicio teórico. Esto hace que sea posible comparar los resultados con la realidad.

4. Recolección de datos

La recolección de datos se divide en dos tipos: por un lado la descripción del sistema y sus funcionamientos, y por otro lado los datos cuantitativos de frecuencias y tiempos de eventos y actividades. La descripción del sistema se obtuvo del trabajo [Mau05] mencionado anteriormente, refinando y simplificando el modelo tras varias discusiones con los tutores. Los datos cuantitativos se obtuvieron del trabajo referenciado en la descripción, donde también figura el método de obtención.

5. Objetivo y decisiones de modelado

Luego de comprendido el sistema y obtenido los datos necesarios, el siguiente paso es realizar un diseño del modelo a simular. Como todo modelo que trata de estudiar un sistema real, deben hacerse varias simplificaciones. Por un lado no se deben incluir detalles irrelevantes a lo que se quiere observar, y por otro lado se debe eliminar todo aquello que dificulte la observación de los datos buscados.

Hay que tener en cuenta el objetivo de la simulación. Luego de discutirlo largamente con los tutores se llegó a la decisión de hacer una simulación del tipo investigativa. No se pretende jugar con los datos de entrada para analizar la respuesta del sistema ante estas entradas, sino estudiar el comportamiento del sistema durante el transcurso de una simulación con los datos de entrada obtenidos de la realidad. Esto permitirá ver el sistema como "ojo de águila", contemplando todos los elementos a la vez.

Definido este objetivo, hay una lista de decisiones de modelado que fueron tomadas:

- Todos los ómnibus son iguales, a excepción de la línea que recorren. La capacidad de pasajeros de los mismos está definida por la línea, y por los datos de entrada esto es constante pero puede ser cambiada.
- Los ómnibus aceptan pasajeros hasta una proporción de su cantidad de asientos. Se resolvió fijar esta proporción en 1,5. Una vez alcanzado el 1,5 no se aceptan más pasajeros hasta que alguno baje.
- Se ignoran tránsito, accidentes, vehículos lentos o con fallas, desviaciones, todo aquello que no permita a los ómnibus llegar a su destino en el tiempo estipulado.
- El tiempo en que los pasajeros suben y bajan de los ómnibus es despreciable, pues se incluye en el tiempo que demora el ómnibus entre paradas. Este punto y el anterior se deben a que las líneas tienen una agenda que debe ser cumplida estrictamente.
- Los pasajeros no tienen preferencia en cuanto a qué línea tomar, mucho menos qué ómnibus tomar, siempre que los lleve a su destino. Se ignoran así las preferencias de los usuarios por determinada línea que demora menos tiempo o determinado ómnibus en mejores condiciones. Esta asunción no es realista, pero simplifica mucho el modelo en cuanto a que no hay que modelar todas estas condiciones de elección, de las cuales no tenemos ningún dato.
- Dado un ómnibus que recorre dos paradas a la ida y a la vuelta y un pasajero que quiere ir de una de esas paradas a la otra en el orden de la vuelta, el pasajero no se toma ese ómnibus a la ida, aunque termine pasando por el destino. Esperará a que el ómnibus retorne para tomarlo (o tomará otro ómnibus que pase antes y le sirva).
- Pese a que la realidad distribuye a los pasajeros en paradas, los datos de entrada que tenemos agrupan estas paradas en zonas. Es así que, por falta de datos más precisos, se trabaja con las zonas para determinar recorridos de líneas y origen-destino de los pasajeros. Esta es una simplificación para el conjunto entrada-modelado, con una entrada más refinada puede utilizarse exactamente el mismo programa e ignorar esta simplificación.
- Al momento de recabar los datos, las frecuencias de pasajeros de determinado par origen - destino fueron tomadas por paradas. Luego estos datos fueron agrupados en zonas y nos fueron entregados. Por esta razón, si bien ningún pasajero quiere ir de una parada a la misma parada, puede ser que quiera ir de una zona a la misma zona. Como esto no nos determina la dirección ni las líneas que le sirven de entre las que pasan por la zona, se decidió ignorar estos pasajeros. Esto causa una subutilización de recursos, pero no

podemos reducir esa incertidumbre. Este es uno de los problemas que se solucionan si, utilizando el mismo programa, los datos se suministran por paradas y no por zonas.

- El paradigma de modelado a utilizar es de interacción a procesos para los ómnibus y de eventos a dos fases para los alimentadores de ómnibus y pasajeros.

Además se tomaron algunas decisiones respecto a la visualización. Independientemente del funcionamiento del sistema, se debe especificar lo que se desplegará durante la simulación. Dado el objetivo investigativo de la simulación, este es el elemento fundamental de donde se extraen los datos de salida deseados, y mediante el análisis de esta visualización se deberá poder obtener toda la información buscada.

- La visualización desplegará, en una ventana, un plano topográfico de la ciudad de Rivera. No se requerirá poder hacer acercamiento en alguna zona en particular, sino ver todo el sistema a la vez.
- El plano de la ciudad contendrá las calles, las zonas, los ríos y las líneas ferroviarias. Estos dos últimos son a modo de ubicación geográfica.
- Cada ómnibus se representará mediante un ícono que indicará la línea a la cual pertenece, su posición actual y su carga. Se decidió utilizar un pequeño dibujo de un ómnibus. La posición en el mapa indica la posición en el sistema. El color de la carrocería define la línea. El color de las ventanas la carga: gris si todos los pasajeros viajan sentados y rojo si además algunos pasajeros viajan parados.
- Para referenciar las líneas de estos ómnibus se incluye en la ventana una leyenda que indica a que línea pertenece cada color.
- Los ómnibus se moverán constantemente entre zona y zona, en un movimiento lo más uniforme posible. Debido a que no hay datos sobre los recorridos en las calles, se desplazarán de zona en zona en línea recta a velocidad constante.
- Se debe desplegar un reloj del sistema indicando el avance de la simulación.
- Debe ser posible mostrar y ocultar un grafo de las zonas
- Debe ser posible mostrar y ocultar todos los ómnibus de una determinada línea

6. Datos de entrada

El modelo requiere datos sobre la topología de la ciudad donde se sitúa el sistema, los diferentes recorridos que están definidos por las líneas del sistema, e información sobre los arribos de los pasajeros a las paradas sus destinos.

Sobre la topología de la ciudad, que se modela como un grafo de zonas interconectadas, se requiere tanto la posición de los nodos como el costo para transportarse de una zona a otra. Si bien las zonas están todas conectadas entre sí, solo se toman aquellas aristas del grafo que participan de alguna de las rutas definidas. Los costos que se toman son las distancias entre las zonas. Podrían haberse tomado como tiempos, pero como la duración de los trayectos de las líneas son fijos la distancia entre zonas toma un papel preponderante.

En cuanto a las líneas, estas determinan un camino que debe seguir el ómnibus y en cuanto tiempo debe completarlo. Los recorridos pueden ser de ida y vuelta o circulares. Además definen la capacidad de los ómnibus. Aunque esto pueda parecer una decisión arbitraria, no lo es tanto. La capacidad de un ómnibus es una propiedad de cada ómnibus, pero puede verse como que está determinada por la empresa a la cual pertenece, pues estas generalmente tienen una flota relativamente uniforme. Y como las líneas son propias de cada empresa, se puede ver que están relacionadas.

Para los arribos y destinos de los pasajeros se requiere información sobre cada cuanto tiempo los pasajeros llegan a las diferentes zonas y hacia donde se dirigen. Para ello se tiene una matriz de origen – destino que tiene la tasa de arribo de los pasajeros. Estas pueden ser positivas o cero. Las tasas cero no son tomadas en cuenta.

7. Modelado

En base a estas decisiones se elaboró la implementación del modelo, en el mismo entorno de desarrollo que el de la librería SimPP. Dado que el propósito es mostrar el funcionamiento de la librería en un caso aplicado, se buscó utilizar la mayor cantidad de herramientas de la librería posibles.

Se decidió utilizar la metodología de diseño GenVoca con estructura Gestalt para la implementación del caso, debido a que la aplicación se presta para ser dividida en cada uno de sus aspectos. En este diseño se relajaron las condiciones para las clases entidades por considerarlas innecesarias.

7.1. Datatypes de entrada

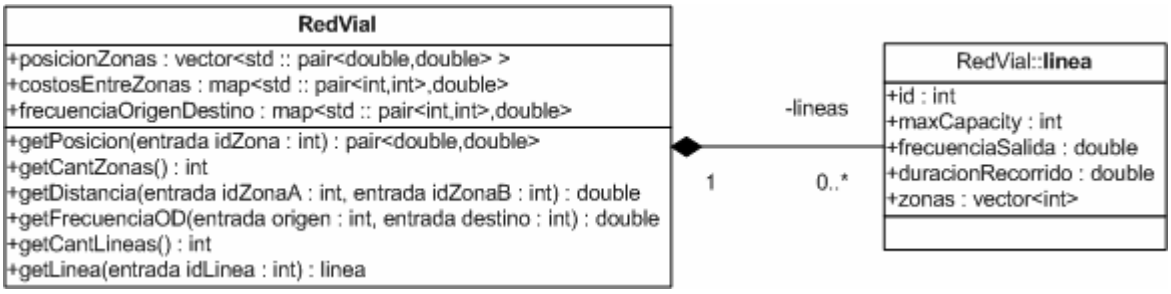


Figura F.1: Estructura de los datos de entrada

Como entrada y parámetro de los constructores de cada capa se creó el datatype `RedVial` con su sub-datatype `linea`. Estos contienen todos los datos de entrada necesarios del modelo.

RedVial::linea

Contiene los datos de una línea en particular

Atributos

- `int id`: identificador de la línea
- `int maxCapacity`: cantidad de asientos de los buses de la línea
- `double frecuenciaSalida`: frecuencia con la cual los buses salen de su terminal
- `double duracionRecorrido`: duración total del recorrido
- `vector<int> zonas`: zonas del recorrido de la línea en orden de visita. Los `int` son índices en el vector `posicionZonas` del `RedVial` contenedor. Un -1 significa un cambio de sentido del bus.

RedVial

Contiene todos los datos de funcionamiento del modelo

Atributos

- `vector<pair<double,double>> posicionZonas`: contiene la posición de cada zona, indexada por el número de zona. Como las zonas comienzan desde el 1, la zona 0 es dummy.

- `map<pair<int,int>,double> costosEntreZonas`: establece el costo de ir de una zona a otra. Los dos `int` del par son los índices de las zonas en el vector `posicionZonas`.
- `map<pair<int,int>,double> frecuenciaOrigenDestino`: establece la frecuencia de aparición de pasajeros que quieren ir de una zona a otra. Los dos `int` del par son los índices de las zonas en el vector `posicionZonas`. No contiene los pares de frecuencia 0.
- `vector<RedVial::linea> lineas`: información de las líneas del sistema.

Operaciones

- `getPosicion(idZona)`: devuelve la posición de una zona. Se requiere que el índice `idZona` sea menor que el tamaño del vector `posicionZonas`.
- `getCantZonas()`: devuelve el tamaño del vector `posicionZonas`, la cantidad de zonas del sistema.
- `getDistancia(idZonaA, idZonaB)`: devuelve la distancia entre dos zonas. Se requiere que los identificadores sean menores que el tamaño del vector `posicionZonas`.
- `getFrecuenciaOD(origen, destino)`: devuelve la frecuencia de aparición de pasajeros que quieran viajar de la zona `origen` a la zona `destino`. Se requiere que los identificadores sean menores que el tamaño del vector `posicionZonas`.
- `getCantLineas()`: devuelve el tamaño de `lineas`, la cantidad de líneas del sistema.
- `getLinea(idLinea)`: devuelve una linea en particular. Se requiere que el identificador sea menor que el tamaño del vector `lineas`.

7.2. Capas Gestalt

A continuación se describen las diferentes capas que se implementaron para el modelo. A excepción de Base, estas capas pueden ser especificadas en cualquier orden. Se describe para cada una lo que requieren, lo que proveen y su estructura. Debido a que Base es obligatoria como fondo del Gestalt y que provee de operaciones terminales, no se especificará en las otras capas el requerimiento de que esta capa exista.

Base

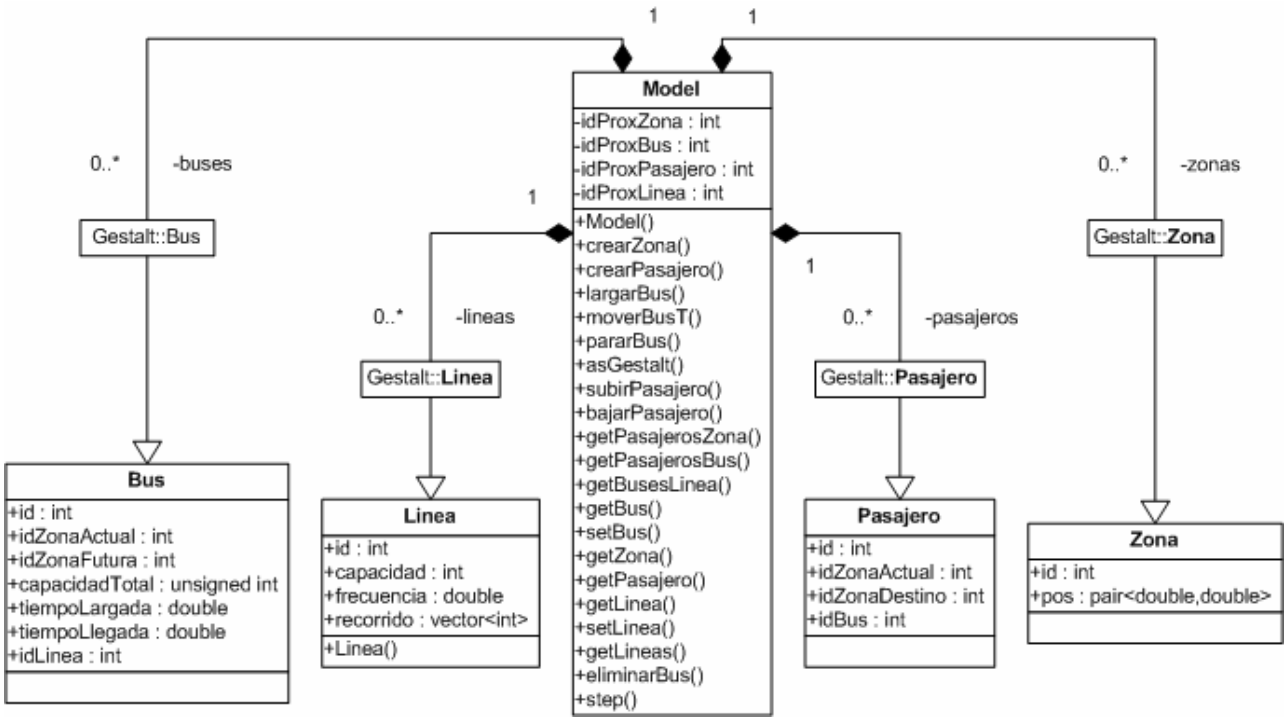


Figura F.2: Diseño de capa Base

Contiene el estado estático de la simulación en un instante determinado. Estos datos determinan el estado de las líneas, zonas, pasajeros y ómnibus en el sistema.

Requiere

- Esta capa debe ser el fondo del Gestalt.
- Una capa en el Gestalt debe proveer `Model::lastEventTime()`, tiempo actual del sistema.

Provee

- Clases base de los elementos del sistema
- Operaciones para accederlos
- Operaciones para modificarlos manteniendo las invariantes del sistema.
- Operaciones terminales

Base::Bus

Contiene la estructura básica del estado de un bus.

Atributos

- `int id`: identificador del bus en el sistema. Mayor a 0 indica un valor válido.
- `int idZonaActual`: si el bus está en trayecto, es el `id` de la zona de origen. Si está detenido, es el `id` de la zona actual.

- `int idZonaFutura`: si el bus está en trayecto, es el `id` de la zona de destino. Si está detenido, es -1.
- `unsigned int capacidadTotal`: la cantidad de asientos del bus.
- `double tiempoLargada`: el tiempo en el cual salió de su último origen.
- `double tiempoLlegada`: si el bus está en trayecto, es el tiempo en el cual llegará a destino. Si está detenido, es el tiempo actual.
- `int idLinea`: es el `id` de la línea que recorre. En el caso que no haya sido asociado a una línea, es -1.

Base::Linea

Contiene la estructura básica de los contenidos de una línea.

Atributos

- `int id`: identificador de la línea en el sistema. Mayor a 0 indica un valor válido.
- `int capacidad`: la cantidad de asientos de los buses de esa línea.
- `double frecuencia`: la frecuencia a la cual salen los buses de esa línea.
- `vector<int> recorrido`: ids de las zonas a visitar en orden de visita. Los -1 marcan un cambio de dirección.

Operaciones

- `Linea(ind, RedVial)`: construye una línea en base al RedVial y el número de índice que marca su instancia particular

Base::Model

Contiene los componentes básicos del modelo de simulación: los universos de instancias de `Gestalt::Bus`, `Gestalt::Linea`, `Gestalt::Zona` y `Gestalt::Pasajero` y funciones que permiten manejar sus estados.

Atributos

- `set<Gestalt::Zona> zonas`: el conjunto de zonas del sistema, en su forma aumentada por todas las capas.
- `int idProxZona`: contador con el `id` de la próxima zona a incluir en el conjunto.
- `set<Gestalt::Bus> buses`: el conjunto de buses del sistema, en su forma aumentada por todas las capas.
- `int idProxBus`: contador con el `id` del próximo bus a incluir en el conjunto.
- `set<Gestalt::Pasajero> pasajeros`: el conjunto de pasajeros del sistema, en su forma aumentada por todas las capas.
- `int idProxPasajero`: contador con el `id` del próximo pasajero a incluir en el conjunto.
- `set<Gestalt::Linea> lineas`: el conjunto de líneas del sistema, en su forma aumentada por todas las capas.
- `int idProxLinea`: contador con el `id` de la próxima línea a incluir en el conjunto.

Operaciones

- `Model(RedVial)`: Constructor, inicializa los contadores en 1 y genera las zonas y las líneas en base al `RedVial`.
- `crearZona(Gestalt::Zona)`: agrega una zona al sistema y devuelve el `id` asignado. El `id` de la zona introducida se ignora y se le da uno nuevo de acuerdo al contador.
- `crearPasajero(Gestalt::Pasajero)`: agrega un pasajero al sistema y devuelve el `id` asignado. El `id` del pasajero introducido se ignora y se le da uno nuevo de acuerdo al contador.
- `largarBus(Gestalt::Bus)`: agrega un bus al sistema y devuelve el `id` asignado. El `id` del bus introducido se ignora y se le da uno nuevo de acuerdo al contador.
- `moverBusT(idBus, idZona, duracionViaje)`: altera el bus indicado para que quede en el estado de viaje de la zona actual hasta `idZona`. El tiempo del viaje es determinado por `duraciónViaje`. Se requiere que exista un bus cuyo `id` sea `idBus` y se encuentre detenido, y una zona cuyo `id` sea `idZona`.
- `pararBus(idBus)`: detiene el bus en su parada destino. Se requiere que exista un bus cuyo `id` sea `idBus` y se encuentre viajando.
- `asGestalt()`: retorna la referencia al propio modelo, pero como `Gestalt::Model`. Esto permite invocar las funciones de todas las capas.
- `subirPasajero(idBus, idPas)`: intenta subir al pasajero en el bus y devuelve si lo logró. Si no lo logra es porque el bus está lleno. Se requiere que exista un bus cuyo `id` sea `idBus` y se encuentre detenido, y un pasajero cuyo `id` sea `idPas` y no esté en ningún ómnibus.
- `bajarPasajero(idPas)`: baja al pasajero del ómnibus donde se encontraba y lo elimina. Se requiere que exista un pasajero cuyo `id` sea `idPas`, se encuentre en un bus y haya salido de su parada inicial.
- `getPasajerosZona(idZona)`: devuelve el conjunto de pasajeros que se encuentran en la zona indicada. Se requiere que exista una zona cuya `id` sea `idZona`.
- `getPasajerosBus(idBus)`: devuelve el conjunto de pasajeros que se encuentran en el bus indicado. Se requiere que exista un bus cuyo `id` sea `idBus`.
- `getBusesLinea(idLinea)`: devuelve el conjunto de buses que pertenecen a la línea indicada. Se requiere que exista una línea cuyo `id` sea `idLinea`.
- `getBus(idBus)`: devuelve el bus con el identificador `idBus`. Se requiere que exista un bus cuyo `id` sea `idBus`.
- `setBus(Gestalt::Bus)`: coloca el bus introducido en la posición correspondiente a su `id`, reemplazando al existente. Esto permite modificar los elementos del conjunto de buses, que solo se obtienen por copia. Se requiere que haya un bus en la posición destino para sustituir.
- `getZona(idZona)`: devuelve la zona con el identificador `idZona`. Se requiere que exista una zona cuyo `id` sea `idZona`.
- `getPasajero(idPas)`: devuelve el pasajero con el identificador `idPas`. Se requiere que exista un pasajero cuyo `id` sea `idPas`.
- `getLinea(int idLinea)` : devuelve la línea con el identificador `idLinea`. Se requiere que exista una línea cuyo `id` sea `idLinea`.

- `setLinea(Gestalt::Linea)`: coloca la línea introducida en la posición correspondiente a su `id`, reemplazando a la existente. Esto permite modificar los elementos del conjunto de líneas, que solo se obtienen por copia. Se requiere que haya una línea en la posición destino para sustituir.
- `getLineas()`: retorna el conjunto de líneas.
- `eliminarBus(idBus)`: elimina el bus con identificador `idBus` del sistema. Se requiere que exista un bus cuyo `id` sea `idBus`.
- `step()`: utilizada para ejecutar, no tiene acción y sirve como operación terminal.

Base::Pasajero

Contiene la estructura básica del estado de un pasajero.

Atributos

- `int id`: identificador del pasajero en el sistema. Mayor a 0 indica un valor válido.
- `int idZonaActual`: si está en una zona, es la zona donde se encuentra. Si está en un bus, es -1.
- `int idZonaDestino`: la zona adonde quiere ir.
- `int idBus`: si está en una zona, es -1. Si está en un bus, es el bus donde está viajando.

Base::Zona

Contiene la estructura básica del estado de una zona.

Atributos

- `int id`: identificador de la zona en el sistema. Mayor a 0 indica un valor válido.
- `pair<double, double> pos`: la posición en coordenadas cartesianas del centroide de la zona.

Esqueleto de implementación

```
template<class Gestalt_>
struct Base {
public:
    typedef Gestalt_ Gestalt;

    struct Bus {
        // Atributos de Base::Bus
    };

    struct Linea {
        // Atributos y operaciones de Base::Linea
    };

    struct Pasajero {
        // Atributos de Base::Pasajero
    };

    struct Zona {
```

```
        // Atributos de Base::Zona
    };

    class Model {
        typedef typename Gestalt::Bus GBus;
        typedef typename Gestalt::Zona GZona;
        typedef typename Gestalt::Linea GLinea;
        typedef typename Gestalt::Pasajero GPasajero;
        // Atributos y operaciones de Base::Model
    };
};
```

Tiempo

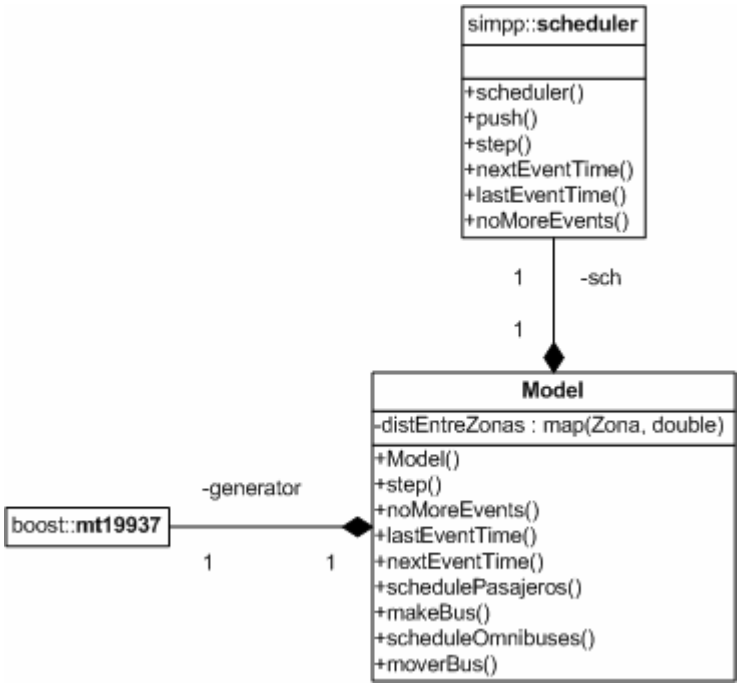


Figura F.3: Diseño de capa Tiempo

Controla el tiempo simulado que demoran las actividades en ejecutarse. Contiene el scheduler del modelo. Cuando un proceso o evento necesita esperar cierto tiempo, lo obtiene de esta capa.

Requiere

- Una capa en el Gestalt debe proveer el functor `BusProcess`. Este functor puede utilizar las operaciones de procesos.

Provee

- Operaciones de scheduler: `Model::noMoreEvents()`, `Model::lastEventTime()` y `Model::nextEventTime()`.
- Suspende procesos `BusProcess` cuando los ómnibus se mueven y los reactiva cuando llegan.
- Feeders de ómnibus y pasajeros.

Tiempo::Model

Contiene el scheduler y las operaciones relacionadas con el tiempo simulado. Utiliza el scheduler básico de SimPP. Para generar valores aleatorios se utiliza Boost.Random

Atributos

- `mt19937 generator`: generador de números aleatorios. Todas las distribuciones del módulo utilizan este generador. Tipo obtenido de Boost.Random
- `scheduler sch`: scheduler para los eventos del sistema. Tipo obtenido de SimPP.
- `map<pair<int,int>,double> distEntreZonas`: mapa de pares de zonas a costo de recorrido. Este costo es bidireccional.

Operaciones

- `Model(RedVial)`: Constructor. Construye el `Model` de la capa inferior, inicializa `generator` con el reloj del sistema, inicializa `distEntreZonas` con los datos de `RedVial` y llama a las funciones `schedulePasajeros(Redvial)` y `scheduleOmnibuses()`, provistas por esta misma clase.
- `step()`: Ejecuta el `step()` del scheduler y continúa la llamada a las capas inferiores.
- `noMoreEvents()`: Devuelve si quedan eventos en el scheduler.
- `lastEventTime()`: Devuelve el tiempo del último evento en el scheduler. Este tiempo, durante la ejecución de operaciones del modelo, corresponde al actual.
- `nextEventTime()`: Devuelve el tiempo del siguiente evento en el scheduler.
- `schedulePasajeros(RedVial)`: Para cada elemento de `frecuenciaOrigenDestino` del `RedVial` se genera un `Gestalt::Pasajero pas`, con `idZonaActual` igual al origen e `idZonaDestino` igual al destino. Luego se pone un feeder en `sch` con `simpp::addFeeder(...)`. Este feeder agenda la función `crearPasajero(pas)` para que se ejecute en intervalos según una variable aleatoria con distribución exponencial. La tasa de esta distribución es el valor de la frecuencia.
- `makeBus(Gestalt::Linea)`: Inicia un proceso con `Gestalt::BusProcess` que recibe una referencia al modelo (como Gestalt) y el `id` de la línea. Activa el proceso.
- `scheduleOmnibuses()`: Para cada una de las líneas del sistema se crea un feeder con `simpp::makeFeeder`. Este feeder agenda en `sch` llamadas a `makeBus(...)` para ejecutarse en intervalos regulares obtenidos de la frecuencia de la línea. Se agenda este feeder en `sch` con tiempo 0.
- `moverBus(idBus, idZona, largoRecorrido, duracionRecorrido)`: Esta operación debe ser llamada por un proceso SimPP. Mueve el bus `idBus` a la zona `idZona`. `largoRecorrido` es el largo total del recorrido de la línea. `duracionRecorrido` es la duración total del recorrido de la línea. En base a estos dos valores y al costo de desplazarse de la zona actual del bus a la zona destino se calcula el tiempo de duración de este tramo. Con este tiempo se invoca `moverBusT(idBus, idZona, tiempo)` sobre `Gestalt::Model`. Luego detiene en `sch` la ejecución del proceso invocador durante el tiempo calculado, utilizando `simpp::hold`. Se requiere, además de que la invocación provenga de un proceso detenible, que `idBus` e `idZona` sean aceptables para los `moverBusT(...)` del sistema.

Esqueleto de implementación

```
template <class Super>
struct Tiempo : Super {
    typedef typename Super::Gestalt Gestalt;

    class Model : public Super::Model {
        typedef typename Gestalt::Bus GBus;
        typedef typename Gestalt::BusProcess GBusProcess;
        typedef typename Gestalt::Zona GZona;
        typedef typename Gestalt::Linea GLinea;
        typedef typename Gestalt::Pasajero GPasajero;
        // Atributos y operaciones de Tiempo::Model
    };
};
```

Tracer

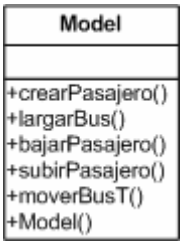


Figura F.4: Diseño de capa Tracer

Intercepta ciertas operaciones que modifican el estado e imprime en la salida estándar un mensaje referente a esta operación y el tiempo simulado cuando se ejecutó. No modifica el comportamiento.

Requiere

- Una capa en el Gestalt debe proveer `Model::lastEventTime()`, tiempo actual del sistema.

Provee

- Tracing de los eventos del modelo

Tracer::Model

Las funciones de esta capa del modelo no modifican el comportamiento del modelo de simulación, sino que interceptan las llamadas para escribir en la salida estándar. No mantiene estado.

Operaciones

- `Model(RedVial)`: Constructor. Construye el `Model` de la capa inferior.
- `crearPasajero(Gestalt::Pasajero)`: Continúa la llamada hacia abajo y obtiene el `id` del nuevo pasajero. Imprime en la salida estándar "**TIME** crearPasajero `id`=**IDPAS** en idZona=**IDZONA**", donde **TIME** es el tiempo actual, **IDPAS** es el `id` adquirido e **IDZONA** es el `id` de la zona origen del pasajero. Se devuelve el `id` adquirido. Hereda los requerimientos de `crearPasajero` del resto de las capas.

- `largarBus(Gestalt::Bus)`: Continúa la llamada hacia abajo y obtiene el `id` del nuevo bus. Imprime en la salida estándar "**TIME** largarBus `id`=**IDBUS** de idZona=**IDZONA**", donde **TIME** es el tiempo actual, **IDBUS** es el `id` adquirido e **IDZONA** es el `id` de la zona origen del bus. Se devuelve el `id` adquirido. Hereda los requerimientos de `largarBus` del resto de las capas.
- `bajarPasajero(idPas)`: Continúa la llamada hacia abajo. Imprime en la salida estándar "**TIME** bajarPasajero `id`=**idPas**", donde **TIME** es el tiempo actual. Hereda los requerimientos de `bajarPasajero` del resto de las capas.
- `subirPasajero(idBus, idPas)`: Continúa la llamada hacia abajo y obtiene si el pasajero pudo subirse al ómnibus. En caso afirmativo imprime en la salida estándar "**TIME** subirPasajero idPas=**idPas** a idBus=**idBus**". En caso contrario imprime en la salida estándar "**TIME** El pasajero idPas=**idPas** no pudo subir a idBus=**idBus**". **TIME** es el tiempo actual. Se devuelve la respuesta recibida. Hereda los requerimientos de `subirPasajero` del resto de las capas.
- `moverBusT(idBus, idZona, duracionViaje)`: Continúa la llamada hacia abajo. Imprime en la salida estándar "**TIME** idBus=**idBus** viajando a zona idZona=**idZona** en duracionViaje", donde **TIME** es el tiempo actual. Hereda los requerimientos de `moverBusT` del resto de las capas.

Esqueleto de implementación

```
template <class Super>
struct Tracer : Super {
    typedef typename Super::Gestalt Gestalt;

    class Model : public Super::Model {
        typedef typename Gestalt::Bus GBus;
        typedef typename Gestalt::Pasajero GPasajero;
        // Operaciones de Tracer::Model
    };
};
```

Display

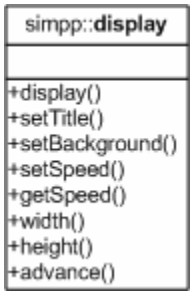


Figura F.5: Diseño de capa Display

Al igual que Tracer, intercepta operaciones que modifican el estado del sistema y refleja estos cambios en una visualización gráfica. Utiliza la llamada a `step()` del ejecutivo para avanzar en el tiempo, obteniendo los lapsos de tiempo entre eventos. Tampoco modifica el comportamiento.

Requiere

- Una capa en el Gestalt debe proveer `Model::lastEventTime()` y `Model::nextEventTime()`, tiempo actual del sistema y próximo evento del sistema respectivamente.

Provee

- Despliegue visual de la simulación
- Operaciones para modificar la visualización, en cuanto a velocidad, visualización del grafo y líneas vistas y ocultas.

Display::Bus

Expande la clase `Bus` de Gestalt agregando la representación gráfica de cada bus

Atributos

- `sprite rep`: sprite que representa al bus. Tipo obtenido de SimPP.

Display::Linea

Expande la clase `Linea` de Gestalt para mantener su visibilidad.

Atributos

- `bool visible`: marca si la línea debe ser vista u oculta.

Operaciones

- `Linea(ind, RedVial)`: Constructor. Construye el `Linea` de la capa inferior e inicializa visible en `True`.

Display::Model

Contiene el display de la simulación y algunos objetos dibujables y variables de control. Intercepta las llamadas a funciones y altera la visualización en consecuencia, sin alterar el estado del modelo.

Atributos

- `display dis`: maneja la visualización de la simulación. Tipo obtenido de SimPP.
- `double acumTime`: Tiempo acumulado entre ejecuciones de `dis.advance(...)`.
- `bool conRed`: define si el mapa debe ser dibujado con el grafo de la red.
- `textSprite tiempoLabel`: etiqueta de texto que se imprime en `dis`. Tipo obtenido de SimPP.
- `textSprite clock`: otra etiqueta de texto que se imprime en `dis`, pero varía con el reloj de la simulación. Tipo obtenido de SimPP.

Operaciones

- `Model(RedVial)`: Constructor. Construye el `Model` de la capa inferior. Inicializa `acumTime` en 0 y `conRed` en False. Se setea el display con dimensiones 800x600. Se asocian `tiempoLabel` y `clock` a `dis` en las posiciones adecuadas. El texto de `tiempoLabel` es

"Tiempo de la simulacion:" y el de `clock` es "0". Se setea el título del display con "Sistema de omnibus Rivera" y se carga el archivo "mapita.png" como fondo.

- `step()`: Continúa la llamada hacia abajo. Acumula en `acumTime` el tiempo entre el actual y el del siguiente evento. Cuando se acumula suficiente tiempo, se avanza este tiempo en la visualización, se setea el `clock` con el tiempo actual y se resetea `acumTime`.
- `largarBus(Gestalt::Bus)`: Inicializa el sprite del bus parámetro con un dibujo elegido según la línea del bus, en la posición de la zona origen. Setea la visibilidad de este sprite según sea visible su línea. Luego se continúa la llamada de `largarBus` hacia abajo, pero con este bus modificado, y se retorna el valor de retorno de esta llamada. Hereda los requerimientos de `largarBus` de las otras capas.
- `moverBusT(idBus, idZona, duracionViaje)`: Continúa la llamada hacia abajo. Luego toma el sprite del bus `idBus` y le setea un movimiento hacia la posición de la zona `idZona` que dure `duracionViaje`. Hereda los requerimientos de `moverBusT` de las otras capas.
- `pintarVentanasBus(idBus)`: Obtiene la cantidad de pasajeros del bus `idBus` y utiliza un dibujo acorde a esta cantidad. Si supera su cantidad de asientos, tiene ventanas rojas y grises en caso contrario. El color de la carrocería se mantiene. Se requiere que exista un bus cuyo `id` sea `idBus`.
- `subirPasajero(idBus, idPas)`: Continúa la llamada hacia abajo y obtiene el resultado. Llama a `pintarVentanasBus(idBus)` para actualizar el dibujo. Luego retorna el resultado obtenido. Hereda los requerimientos de `subirPasajero` de las otras capas.
- `bajarPasajero(idPas)`: Continua la llamada hacia abajo y llama a `pintarVentanasBus` con el `id` del bus que tenía antes de bajarse. Hereda los requerimientos de `bajarPasajero` de las otras capas.
- `cambiarVisibilidadLinea(idLinea)`: Setea la visibilidad de la línea `idLinea` con el negado de su valor actual. Obtiene todos los buses de esa línea y setea su visibilidad de acuerdo a este valor. Se requiere que exista una línea cuya id sea `idLinea`.
- `incSpeed(factor)`: Setea la nueva velocidad de la visualización como la anterior multiplicada por `factor`, limitada por cotas máximas y mínimas.
- `cambiarFondo()`: Conmuta entre colocar el fondo con el grafo o sin él. Utiliza `conRed` para llevar control del siguiente fondo a colocar.

Esqueleto de implementación

```
template <class Super>
struct Display : Super {
    typedef typename Super::Gestalt Gestalt;

    struct Linea : public Super::Linea {
        // Atributos y operaciones de Display::Linea
    };

    struct Bus : public Super::Bus {
        // Atributos de Display::Bus
    };

    class Model : public Super::Model {
        typedef typename Gestalt::Bus GBus;
```

```
        typedef typename Gestalt::Zona GZona;
        typedef typename Gestalt::Linea GLinea;
        typedef typename Gestalt::Pasajero GPasajero;
        // Atributos y operaciones de Display::Model
    };
};
```

Control

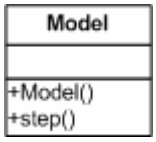


Figura F.6: Diseño de capa Control

Intercepta las llamadas cíclicas a `step()` por parte del ejecutivo para recibir input del teclado del usuario. Este es utilizado para setear la visualización. Puede expandirse para modificar más aspectos de la visualización o factores del propio modelo, pero en su versión actual no modifica el comportamiento. Esta capa depende de SDL, más exactamente de los eventos de SDL, dado que es la librería que se utiliza en el manejo de visualización de SimPP.

Requiere

- Operaciones para modificar la visualización

Provee

- Control sobre la visualización

Control::Model

Este `Model` no interviene en el modelo, pero es el punto donde la capa puede interceptar los `step()`. No contiene estado.

Operaciones

- `Model(RedVial)`: Construye el `Model` de la capa inferior.
- `step()`: Continúa la llamada hacia abajo y captura los eventos del usuario para alterar la visualización o terminar el programa. Utiliza `SDL_Event`.

Esqueleto de implementación

```
template <class Super>
struct Control : Super {
    typedef typename Super::Gestalt Gestalt;

    class Model : public Super::Model {
        // Operaciones de Control::Model
    };
};
```

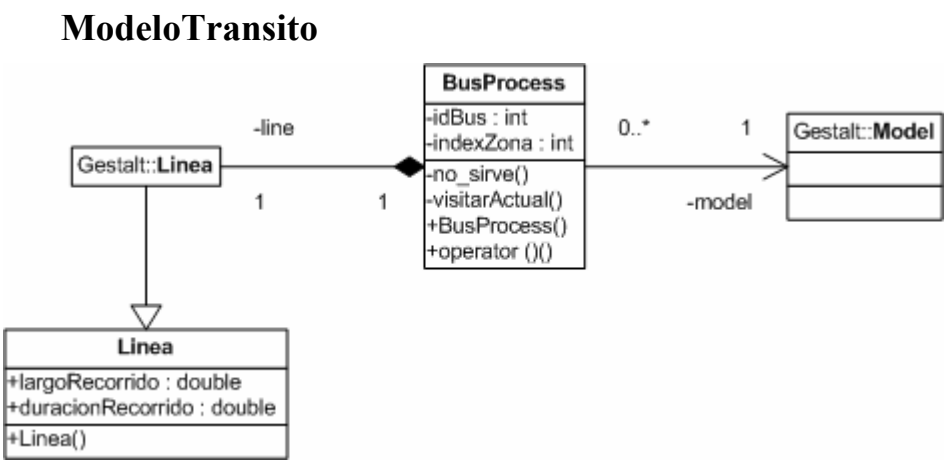


Figura F.7: Diseño de capa ModeloTransito

Determina los comportamientos e interacciones de los elementos del sistema, más allá de como esto afecte el modelo y el tiempo de la simulación. En este caso implementa la clase `BusProcess`, el proceso del bus que a lo largo de su procesamiento va invocando operaciones de las otras capas.

Requiere

- Una capa que brinde la operación `Model::moverBus(idBus, idZona, largoRecorrido, duracionRecorrido)`, que detenga el proceso el tiempo necesario.

Provee

- Mecánicas del modelo
- Proceso de los ómnibus `BusProcess`

ModeloTransito::Linea

Extiende la clase `Linea` agregando el largo y la duración del recorrido. Esto permite a los procesos de los ómnibus moverse por el recorrido sin conocer su tiempo.

Atributos

- `double largoRecorrido`: el largo total del recorrido de la línea
- `double duracionRecorrido`: la duración total del recorrido de la línea

Operaciones

- `Linea(ind, RedVial)`: Constructor. Construye el `Linea` de la capa inferior. Obtiene la duración del recorrido de la línea de `RedVial` y `largoRecorrido` como la suma del costo entre zonas entre cada par origen-destino de la línea.

ModeloTransito::BusProcess

Un functor cuya invocación maneja el ciclo de vida de los ómnibus. Este functor se debe ejecutar como un proceso, pues hace llamadas a funciones de `simpp::process`. Estas llamadas son indirectas, las implementa la capa Tiempo.

Atributos

- `Model model`: Una referencia al modelo (como `Gestalt::Model`), de modo de poder acceder a los atributos de las entidades que maneja e invocar las operaciones que necesita.
- `int idBus`: El `id` del ómnibus cuyo comportamiento se modela.
- `Linea line`: Una copia de la línea del bus. Mantener esta línea local permite asegurarse de que la línea no cambia en medio de la ejecución.
- `int indexZona`: Índice de la zona actual en la lista de zonas de la línea.

Operaciones

- `BusProcess(Gestalt::Model, idLinea)`: Constructor. Crea el functor y le inicializa sus atributos. `model` es el modelo Gestalt recibido, y `line` se obtiene de este modelo con `getLinea(idLinea)`. `indexZona` comienza en 0 y el valor de `idBus` es indefinido. Se requiere que exista en el modelo una línea cuyo id sea `idLinea`.
- `no_sirve(idZonaDestino)`: Devuelve si el bus `idBus` no visita `idZonaDestino` antes de retornar o terminar el recorrido (lo que suceda antes).
- `visitarActual()`: Obtiene todos los pasajeros del bus que deben bajar en la zona actual y los baja. Luego obtiene todos los pasajeros de la zona actual a los que les sirva subirse al bus y los sube (es responsabilidad de `subirPasajero` el controlar el tope de esta subida). Para este segundo paso se utiliza `no_sirve`.
- `operator()()`: Función ejecutada por el proceso. Crea un `Gestalt::Bus` que inicializa con los siguientes valores:
  - `id` es -1, pues tiene que ser inválido hasta que se inserte en el modelo
  - `idZonaActual` es el origen de `line`
  - `idZonaFutura` es -1 (se encuentra detenido)
  - `capacidadTotal` se extrae de `line`
  - `tiempoLargada` y `tiempoLlegada` son 0, pero su valor no importa hasta que comience a moverse.
  - `idLinea` es el `id` de `line`

  Obtiene el `id` del `bus` al invocar `largarBus` en `model`. Luego, para cada destino de `line` (exceptuando los -1) se invoca sobre el modelo:

  - `moverBus(idBus, line.recorrido[indexZona], line.largoRecorrido, line.duracionRecorrido)`
  - `pararBus(idBus)`
  - `visitarActual()`

  Luego de visitadas todas las zonas, se ejecuta `eliminarBus(idBus)` para eliminarlo.

Esqueleto de implementación

```
template <class Super>
struct ModeloTransito : Super {
```

```
    typedef typename Super::Gestalt Gestalt;

    struct Linea : public Super::Linea {
        // Atributos y operaciones de ModeloTransito::Linea
    };

    class BusProcess {
        typedef typename Gestalt::Linea GLinea;
        typedef typename Gestalt::Bus GBus;
        typedef typename Gestalt::Model GModel;
        typedef typename Gestalt::Pasajero GPasajero;
        // Atributos y operaciones de ModeloTransito::BusProcess
    };

};
```

7.3. Todas las capas

La figura F.8 muestra la estructura de un Gestalt con todas las capas. El orden no es relevante, y se requieren como mínimo las capas Base, Tiempo y ModeloTransito. La capa Control requiere la capa Display.

En la implementación, el tipo de un Gestalt con todas las capas se genera mediante la siguiente sentencia:

```
struct Gestalt : Display<Control<Tracer<Tiempo<ModeloTransito<Base<Gestalt> > >
> > > > {};
```

Para eliminar capas, basta con remover los parámetros template no deseados.

Para correr la simulación, se ejecuta el siguiente código.

```
    Gestalt::Model modelo(red);
    simpp::execute(modelo, tiempo);
```

Donde `red` es el `RedVial` con los datos del sistema y `tiempo` es el tiempo simulado de ejecución.

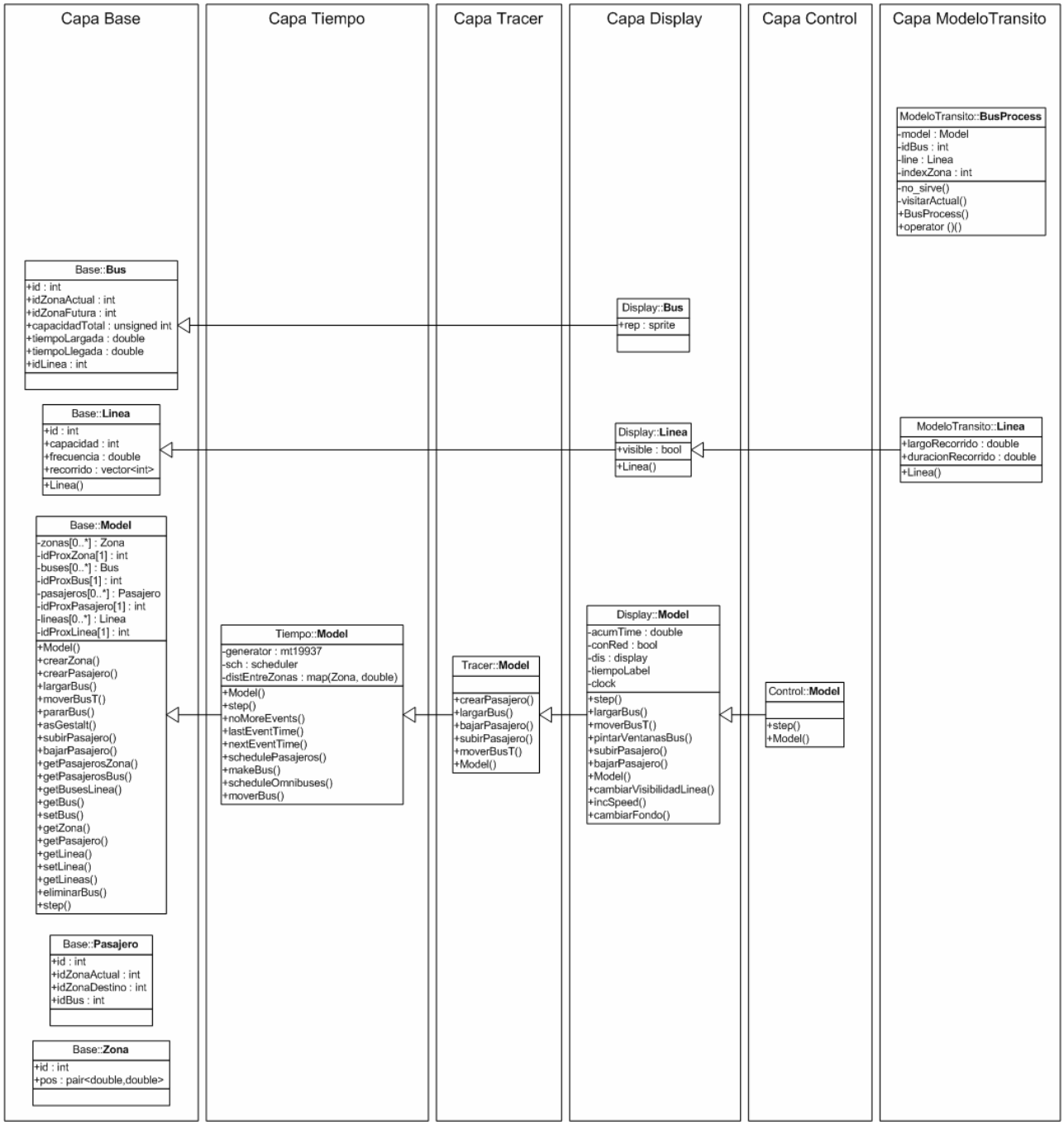


Figura F.8: Diseño general con todas las capas

8. Modo de uso

El ejecutable `Rivera.exe` corre la simulación del sistema durante un tiempo simulado de 5 horas. Durante la simulación puede manipularse la visualización mediante las siguientes teclas:

- 1 al 0, q, w, e: Activan o desactivan la visualización de una línea (1 al 13)
- r: Activa o desactiva la visualización del grafo de zonas
- + (teclado numérico): Acelera la visualización
- - (teclado numérico): Desacelera la visualización

Los datos de entrada son cargados de los siguientes archivos:

- `costos_entre_zonas.txt`: Costo en distancia de desplazarse de zona a zona
- `matrizOD.txt`: Frecuencia de arribos de pasajeros según zonas de origen y destino
- `posicion_zonas.txt`: coordenadas de las zonas en el plano
- `recorridos.txt`: definición de cada línea con capacidad, duración y recorrido

Se proveen los archivos `matrizOD_hora_pico.txt` y `matrizOD_todo_el_dia.txt` para reemplazar el `matrizOD.txt` con las frecuencias en diferentes puntos del día. Originalmente el archivo `matrizOD.txt` es una copia del `matrizOD_hora_pico.txt`.

Las siguientes figuras muestran el despliegue de la simulación durante la ejecución: la visualización gráfica y el tracing.

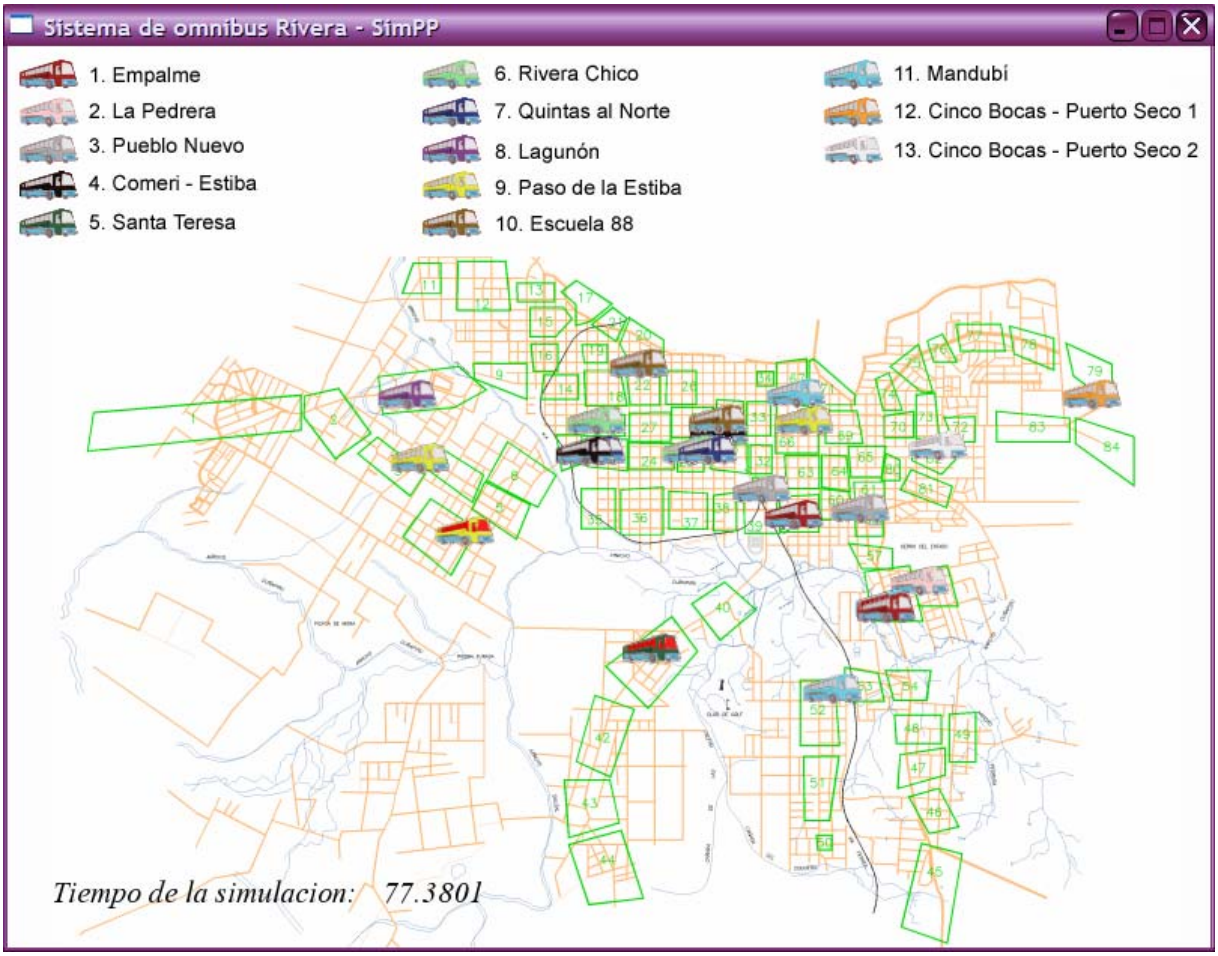


Figura F.9: Screenshot de la visualización gráfica

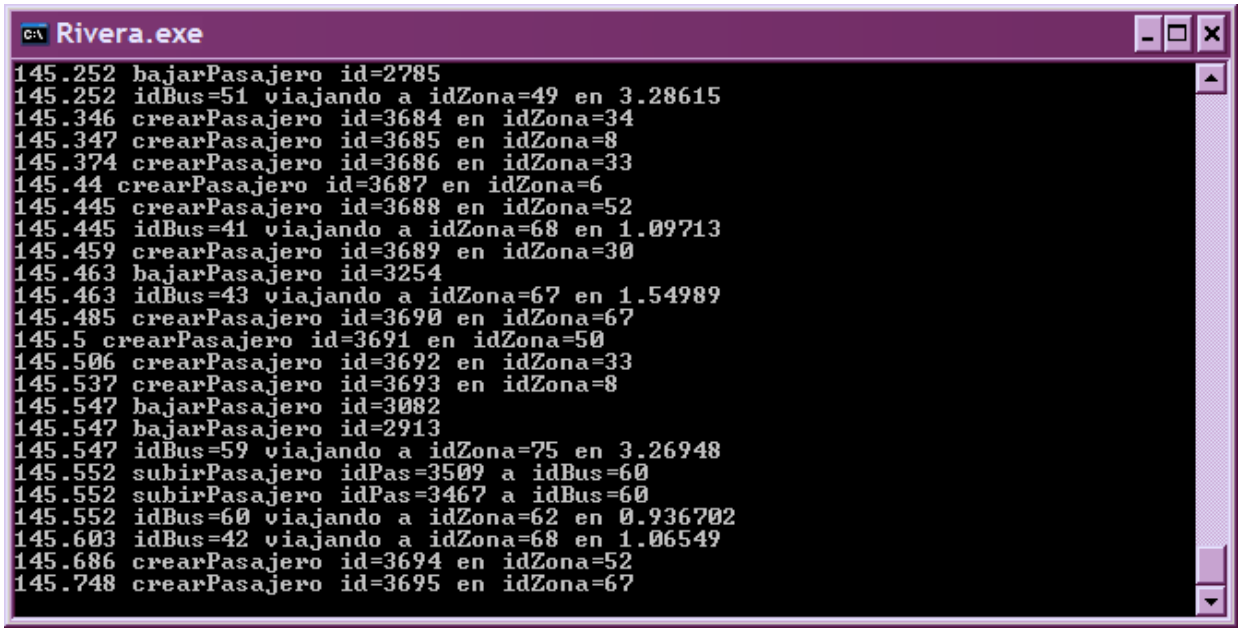


Figura F.10: Screenshot del tracing

9. Verificación y validación del modelo

Dado que la aplicación desarrollada tiene un fuerte componente de salida al usuario, la verificación y validación de la misma se hace sencilla, pues todo comportamiento erróneo se hace fácilmente visible. Ante cualquier problema es posible redundar en la salida respectiva para ayudar a la detección.

Una primera etapa de la verificación y validación se dio con los tutores del proyecto. Antonio Mauttone, que ya había estado estudiando el sistema y nos dio los datos necesarios, validó el funcionamiento del sistema utilizando la aplicación en varias corridas, analizando los recorridos de los ómnibus de cada línea, observando las frecuencias de salida y los patrones de saturación de pasajeros en ómnibus en las zonas y recorridos correspondientes

La segunda etapa de la validación se dio con el personal del Departamento de Tránsito y Transporte de la ciudad, y la misma también fue positiva. Esto muestra la representatividad del modelo ante alguien que desconoce sobre decisiones de diseño y los principios de simulación, y lo compara contra una realidad que conoce bien.

10. Particularidades de implementación

Además de la propia clase que representa a todo el modelo, que es refinada sucesivamente en las distintas capas GenVoca de que consiste la solución, lo mismo les sucede a las clases de las entidades, `Bus`, `Pasajero`, etc. El modelo es universo de instancias de las otras clases o home, pero como las clases se componen a partir de sus roles, el tipo de las mismas no es el que define la capa que contiene las colecciones. Gracias a que el tipo final puede alimentar a su vez a todo el stack de capas, es posible hacer que las colecciones tengan el tipo adecuado, y no se pierda ningún refinamiento posterior. Sin embargo, esto trajo problemas sutiles con la implementación de las colecciones usadas que debieron ser salteados.

Las colecciones se implementaron utilizando la librería Boost.MultiIndex. Las mismas permiten configurar el tipo de los elementos y los órdenes por los cuales se los tiene ordenados separadamente. Así, es fácil agregar criterios sin tener que agregar colecciones nuevas y mantener el código viejo. Este trozo de fuente proviene de la implementación del caso de estudio:

```
typedef multi_index_container<
    GBus,
    indexed_by<
        ordered_unique<member<Bus,int,&Bus::id> >,
        ordered_non_unique<member<Bus,int,&Bus::idLinea> >
    >
> Bus_set;
```

`GBus` es el tipo de bus con todos los refinamientos, el final. Luego se indican dos criterios para ordenarlo, por su identificador, que es una clave primaria, y por el identificar de la línea. Como hay varios coches por línea, el índice no es único. La interfaz para crear colecciones con las necesidades de la aplicación es muy pulida. Lo único criticable es que los argumentos para member son redundantes; `&Bus::idLinea`, el puntero a miembro, es para miembros de tipo entero de la clase `Bus`. Desgraciadamente, no se puede simplificar más aún esto en C++.

El problema que se encontró fue que así como está escrito el código no compila. Se siguió la pista hasta el código fuente de la clase `member`. `member` es un *key extractor*, un concepto que forma parte de la librería Boost.MultiIndex. Sus instancias reciben un elemento de la colección y son capaces de extraer el dato por el cual se ordena. `member` soporta extraer el miembro con el cual es instanciada tanto de una referencia al objeto en cuestión como de un puntero o puntero a puntero, etc. a un objeto. El problema es determinar que es un puntero. En C++ existe un tipo primitivo que es un puntero, pero también se pueden definir operadores `->` y `*`(unario) para los objetos, con lo que pueden ser objetos a su vez. Entonces member tiene dos maneras de extraer un miembro, o el objeto es del tipo final esperado, o se intenta removiendo un nivel de indirección, y así hasta que no se pueda, como en el siguiente extracto:

```
template<class Class ,typename Type ,Type Class::*PtrToMember>
struct member
{
    template<typename ChainedPtr>
    Type& operator()(const ChainedPtr& x) const
    {
        return operator()(*x);
    }
```

```
    const Type& operator()(const Class& x) const
    {
        return x.*PtrToMember;
    }
}
```

La concordancia de patrones en C++ funciona de tal manera que la segunda versión del `operator()` gana cuando se llega al objeto final, y la otra es tomada en todo otro caso. Pero volviendo al ejemplo del caso de estudio, el puntero a función es para una clase derivada de la del elemento. En este caso, `member` entra por la primera implementación de `operator()` y la compilación falla dentro del método.

El problema de fondo es que no se pueden poner restricciones de tipo a los parámetros de los templates. Un puntero es un objeto que tiene implementaciones para `->` y `*`, pero no hay manera de saber a priori si esto sucede, y de requerir que lo haga. En general la consecuencia de esto son los errores de compilación gigantes, a los que el programador en C++ ya está habituado. Alternativamente, puede pasar como en este caso, que por razones del orden entre las versiones de una misma función que prueba el compilador no se encuentra la adecuada. La solución existe dentro del lenguaje, bajo el nombre de SFINAE, *Sustitution Failure is not an Error*. Sin embargo, los compiladores más viejos no soportan estas características y es por eso que no está implementada esa solución en la librería.

El caso de estudio no utiliza `member`, sino que se derivó una alternativa con el nombre de `miembro`, que no persigue los punteros como los hace `member`. Hubiera sido posible aplicar el arreglo directamente a la librería Boost.MultiIndex, pero se juzgó que complicaba demasiado el seguimiento de versiones. El autor de Boost.MultiIndex aseguró que el problema estaría solucionado para la siguiente versión. [PBMi05]

11. Conclusiones

A través de este caso de estudio se probó la capacidad de SimPP para:

- Combinación de paradigmas
- Visualización en tiempo real, con interacción del usuario
- Ejecutivo de dos fases
- Eventos y calendario básico
- Soporte de interacción de procesos
- Feeders

Muchas de estas herramientas fueron corregidas y mejoradas durante la elaboración del caso, debido a que se pudo experimentar directamente con estas facilidades. La mayoría de las funcionalidades de la visualización y el funcionamiento de los feeders se desarrollaron en esta etapa.

Otra de las ideas desarrolladas es la estructura Gestalt para la estructuración del modelo. La adaptabilidad y corrección de cada módulo se simplificó enormemente, al punto de crear y adaptar al Gestalt la capa Display (la última implementada) en sólo media hora.

La validación fue exitosa y el modelo representó fielmente la realidad, salvando las simplificaciones efectuadas. Esto demuestra también la capacidad de SimPP de no solo crear modelos que funcionan, sino que también son correctos.

12. Referencias

[ISO98] ANSI/ISO C++ Committee. Programming Languages — C++. ISO/IEC 14882:1998(E). American National Standards Institute, New York, 1998.

[PBMi05] Problemas con Boost multindex, [Boost-users] [MultiIndex] Key extraction from a base member

http://lists.boost.org/boost-users/2005/10/14657.php Diciembre 2005

[Mau05] MAUTTONE, Antonio. Optimización de recorridos y frecuencias en sistemas de transporte público urbano colectivo. Montevideo: InCO, PEDECIBA Informática, Tesis de Maestría, 2005. p.177. ISSN: 0797-6410

Apéndice G - Gestión de Proyecto

1. Introducción

En esta sección se describirá la metodología de Ingeniería de Software utilizada para desarrollar SimPP.

El modelo de proceso aplicado por el equipo fue una metodología ágil, iterativa e incremental y basada en testeo. Esta metodología es la más apropiada debido al alcance del proyecto, cantidad y características de los integrantes.

Se definió en forma temprana un conjunto de herramientas para conformar el ambiente de desarrollo, así como ciertas políticas de testeo para la implementación de componentes de la librería.

En cuanto a la planificación, no se realizó una planificación detallada de todo el proyecto antes de comenzar el mismo, sino que se definieron sobre la marcha los hitos importantes.

2. Modelo de Proceso

A la hora de escoger un modelo de proceso se debe tener en cuenta los objetivos y alcance del producto que se desea construir, los recursos materiales y humanos, y el tiempo en el que se desea terminar el producto.

En nuestro caso el producto a construir es una librería de simulación a eventos discretos. La definición de este tipo de software es extremadamente variada, abarcando desde sistemas que brindan una funcionalidad limitada hasta "aplicaciones para el soporte de decisiones". El producto que se desea construir es una librería capaz de soportar los requerimientos definidos anteriormente en el informe ejecutivo, lo que nos define un software de pequeño porte. Pero en nuestro caso el tamaño no es el problema sino la complejidad que se quiera alcanzar. Por tratarse de un proyecto de grado, uno de los objetivos es construir aplicaciones con base en las ideas estudiadas previamente y esto será el factor determinante del esfuerzo requerido.

En cuanto a los recursos que se poseen, el equipo de proyecto está integrado por tres estudiantes. Todos poseen una computadora propia con banda ancha y dichos equipos tienen características similares. Por tanto es fácil definir un ambiente de desarrollo común. Otro aspecto importante es que el equipo de proyecto se conoce desde el inicio de la carrera, y ha trabajado en conjunto durante mucho tiempo. Por ello ya tiene una estructura definida y una buena comunicación. Además se desea terminar el proyecto en el plazo de un año.

Por todos estos motivos, el uso de un modelo de proceso ágil es adecuado. Las tareas que demandan más esfuerzo son la investigación y la aplicación de las técnicas estudiadas.

Nuestro modelo de proceso es iterativo e incremental en este sentido. El desarrollo de la librería se llevó a cabo implementando nuevos componentes y refinando los anteriores en cada iteración. El hecho de que la librería sea un conjunto de herramientas disconexas ayuda a la implementación de este tipo de modelos. Las iteraciones son de dos semanas, y se definen los objetivos de la misma en la reunión con los tutores. Dichos objetivos pueden ser de implementación o investigación, con un peso mayoritario de la segunda en la primera mitad del año.

El modelo utilizado también está basado en pruebas. Conjuntamente con el desarrollo de los componentes se definían pruebas para dichos componentes. De este modo se define la interfaz de estos y la funcionalidad que brindan. Un componente está listo si pasa todas las pruebas que se le han definido. A su vez, el desarrollo de pruebas particulares inspira a agregar más funcionalidades a las herramientas, funcionalidades testeadas luego mediante estas pruebas.

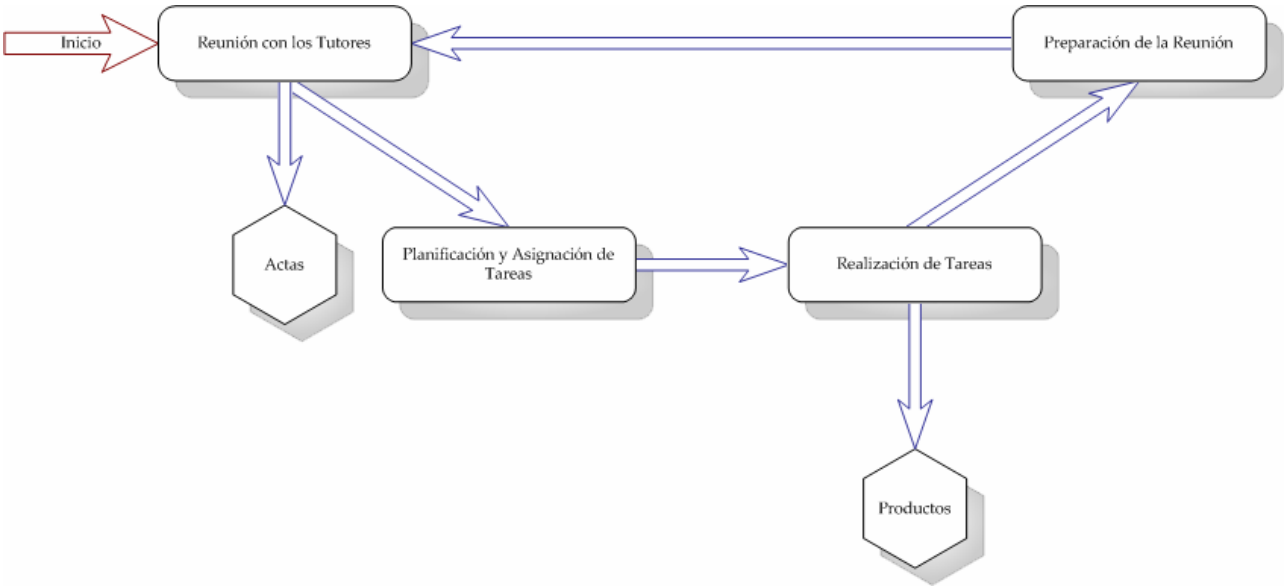
Otra característica del modelo es que es una metodología ágil. Estas metodologías son intensivas en desarrollo y comunicación con el cliente y con poca documentación de gestión [XP05]. La documentación de gestión es casi innecesaria por el tamaño del equipo y la comunicación que ha desarrollado, además de que los integrantes tienen la suficiente disponibilidad horaria como para reunirse frecuentemente. La intensidad del desarrollo está dada más que nada en la aplicación de técnicas, lo que requiere mucho tiempo de diseño. En cuanto a los clientes, que son los tutores del proyecto, definieron a grandes rasgos los requerimientos de la librería y luego dieron libertad al equipo para implementarlos, modificarlos hasta cierto punto y agregar nuevos. En este entorno la aplicación de una metodología ágil es lo más adecuado.

2.1. Descripción

Nuestro modelo de proceso es iterativo e incremental como fue mencionado arriba. Tiene una duración de aproximadamente 9 meses. Cada iteración dura dos semanas y posee 4 etapas básicas:

- Reunión con los Tutores: En la reunión con los tutores el equipo muestra el avance que se ha logrado en la iteración anterior. Los tutores examinan dichos resultados, proponen ajustes y objetivos para la próxima iteración. El equipo manifiesta su opinión sobre dichos objetivos y se llega a un acuerdo sobre los objetivos que serán tomados para la próxima iteración y cuales serán pospuestos. Finalmente uno de los miembros del equipo confecciona un acta de la reunión.
- Planificación y Asignación de tareas: En esta etapa se planifica la iteración y se asignan a cada miembro las tareas que debe cumplir para completar los objetivos de la iteración. Esta etapa se realiza luego de la Reunión con los Tutores, pero puede ser refinada a lo largo de la iteración.
- Realización de Tareas: En esta etapa los integrantes realizan las tareas que tienen asignadas para la iteración. Las tareas pueden ser de 3 tipos: investigación, implementación o documentación. La investigación consiste en recabar y estudiar información sobre uno o varios temas. La implementación se refiere tanto al desarrollo de componentes de la librería, lo cual incluye la confección de una prueba del mismo, o el desarrollo de un caso de estudio. La documentación consiste en redactar un documento que contenga: resumen de la investigación realizada, documentación de un componente desarrollado, presentación de avances del proyecto.
- Preparación de la Reunión: Es la última fase de la iteración. Aquí se preparan todos los artefactos a ser mostrados a los tutores. Si no se tienen artefactos listos al final de la iteración, se deberá dar un informe del avance en forma oral. Dicha información será recabada en el acta de la reunión.

Tanto en la Reunión con los Tutores como en la Realización de Tareas se generan artefactos. Todos los artefactos generados son almacenados en un repositorio accesible por todos los miembros del equipo. Se distinguen 2 tipos diferentes de artefactos: Actas y Productos. Las actas son los únicos documentos de gestión que se manejan. Allí se registran todos los temas discutidos en la Reunión con los Tutores. En cambio, los productos son aquellos artefactos generados por el equipo a lo largo del proyecto y cuyo propósito no es la gestión del proyecto. O sea, los productos finales e intermedios del proyecto. Estos pueden ser informes de investigaciones realizadas, informes sobre prototipos construidos, documentación de la librería, presentaciones del proyecto, y componentes de software.



2.2. Ambiente de desarrollo

Definir un ambiente de desarrollo es vital para todo proyecto. En nuestro caso necesitamos un editor, utilidades para el testeo de componentes y un sistema para el control de versiones. Debido a que todos los integrantes poseen computadoras de porte similar, es fácil encontrar herramientas comunes.

Como el proyecto será implementado en C++ y la facultad posee convenio con Microsoft(R) se decidió utilizar como IDE al Microsoft Visual Studio .Net 2003(R). Para testeo de componentes se utilizó Boost Test, una de las librerías que forma parte de Boost. Esta librería provee de varios componentes para escribir y organizar pruebas unitarias. La forma de utilización es descrita en el apéndice E de Librerías Externas.

Como se explico en el punto anterior, el equipo necesita un repositorio donde almacenar todos los artefactos producidos. Para ello se solicitó un sistema de control de versiones (CVS) a los administradores del sistema del InCo. Para utilizar este repositorio de forma más amigable se utilizo Tortoise CVS [TCVS05].

3. Cronograma

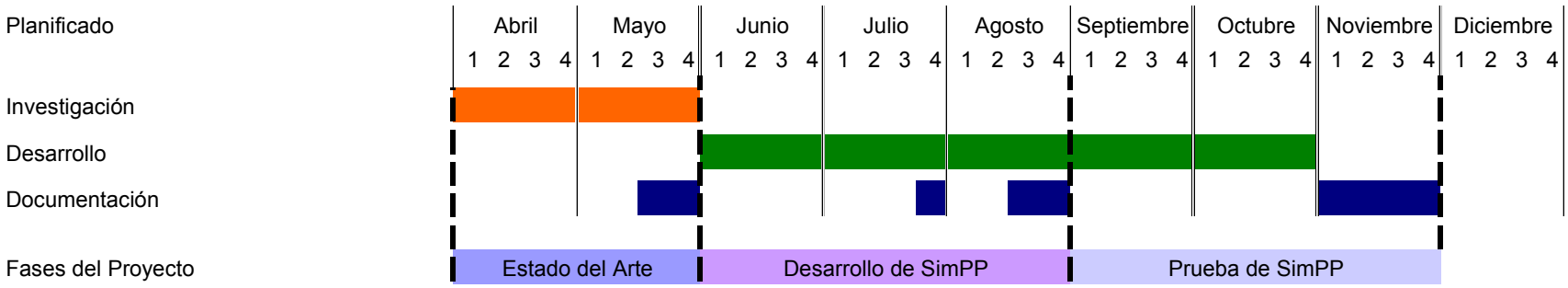
Nuestro proyecto consiste en la implementación de una librería de simulación a eventos discretos. Para poder realizarlo adecuadamente debemos realizar una investigación sobre el estado del arte en este tema, diseñar e implementar nuestra librería, probarla con un caso real, y preparar el informe final del proyecto.

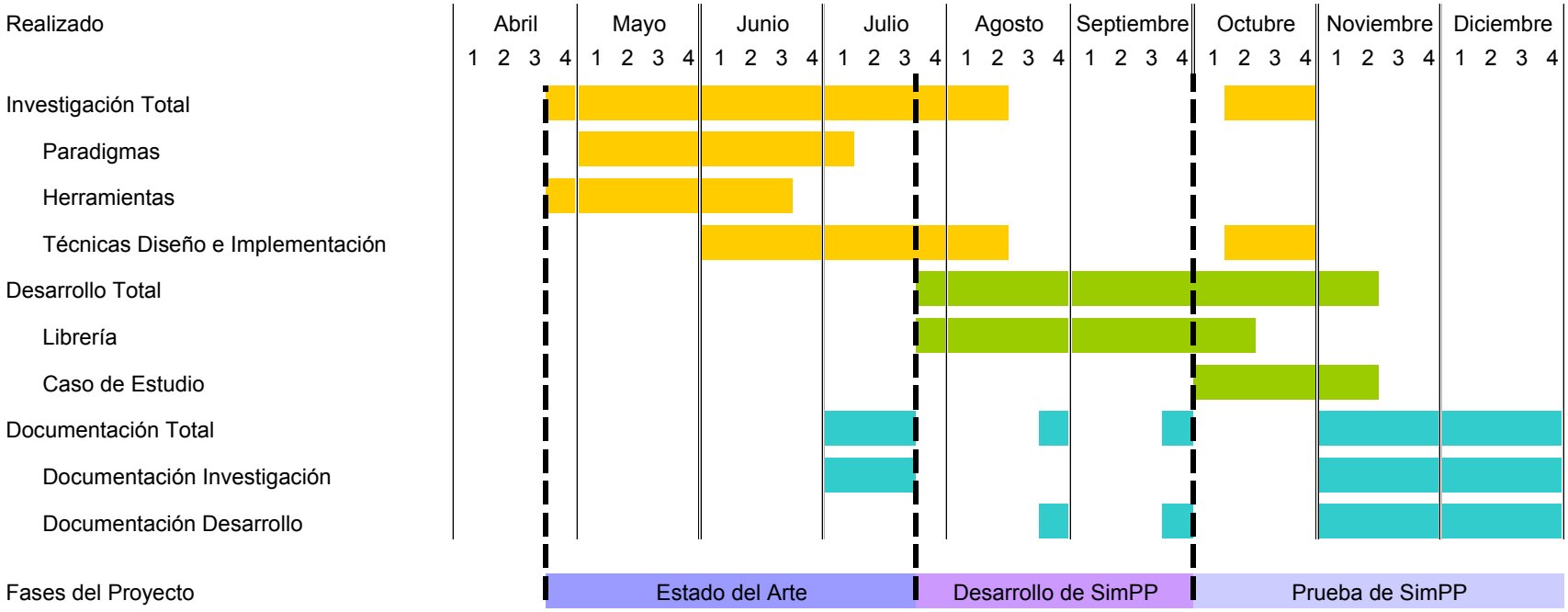
En cuanto la duración del proyecto, este fue pensado para tener una duración de 8 meses, de abril a diciembre. Se dividió al proyecto en tres etapas:

- **Estudio del Estado del Arte:** En esta etapa se estudia el estado del arte de las simulaciones a eventos discretos. Dicho estudio consiste en: investigar varios paradigmas de simulación a eventos discretos, investigar diversas herramientas de simulación a eventos discretos, estudiar algunas técnicas de diseño e implementación para el desarrollo de librerías en C++. Se planificó que esta etapa se extendería desde abril a mayo. Al final se tendría un informe de resumen de la investigación realizada.
- **Diseño e Implementación de la Librería:** Aquí se comienza a diseñar e implementar la librería. Se construirán versiones preliminares de los componentes y se efectuarán pruebas con casos de estudio sencillos y tests unitarios. Se planificó que esta fase duraría de junio a agosto. Al final de la fase se confeccionará un informe sobre los prototipos implementados.
- **Prueba de la Librería:** Esta es la etapa final donde se probará la librería con un caso de estudio real: el sistema de transporte público de la ciudad de Rivera. Además se redactará el informe final del proyecto y la documentación de la librería. Esta fase durará desde setiembre hasta noviembre. Al terminar esta fase se habrá terminado el proyecto.

Esta primera planificación sufrió cambios durante el proyecto. Hay que recordar que las planificaciones en este tipo de desarrollos casi nunca son fielmente seguidas, y tienden a extenderse más de lo planeado. Nuestro caso no fue una excepción, pero hubo razones que impidieron al equipo una dilatación del proyecto a lo largo del tiempo. La razón principal es la partida al exterior de uno de los integrantes en enero del 2006. Esta restricción hizo que el proyecto no pudiese ser extendido más allá del mes de diciembre. El equipo tuvo que acelerar el desarrollo del proyecto y se dispuso solamente de un mes extra para terminar el proyecto. Sin embargo el proyecto pudo ser llevado a cabo en tiempo y forma.

En cuanto a la planificación planeada y la registrada, podemos ver que la Investigación fue un área que fue subestimada en un principio. Esta área tuvo una gran dedicación por parte del equipo y fue de mucha importancia a lo largo de todo el proyecto. En cuanto al Desarrollo, éste fue sobreestimado en la primera planificación. En realidad requirió bastante menos esfuerzo del estimado, gracias a la cuidadosa elección de herramientas realizada por el equipo. Finalmente el área de Documentación tuvo la planificación más ajustada. Esto se debió a que las instancias de documentación fueron bien definidas en la planificación general, y su esfuerzo fue bastante bien ajustado.


Planificado
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Noviembre
Diciembre
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Investigación
Desarrollo
Documentación
Fases del Proyecto
Estado del Arte
Desarrollo de SimPP
Prueba de SimPP


Realizado
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Noviembre
Diciembre
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Investigación Total
Paradigmas
Herramientas
Técnicas Diseño e Implementación
Desarrollo Total
Librería
Caso de Estudio
Documentación Total
Documentación Investigación
Documentación Desarrollo
Fases del Proyecto
Estado del Arte
Desarrollo de SimPP
Prueba de SimPP

4. Actas

Como se explicó en la descripción del modelo de proceso seguido por el equipo, las Actas son el único documento para la Gestión del Proyecto. A continuación se presentan todas las actas de las reuniones que fueron realizadas por el equipo.

4.1. Proyecto de grado SimPP - Acta de la reunión del 07/06/2005

Asistentes

Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia.

Investigación durante las semanas anteriores

Sebastián se dedicó a leer sobre Devs y expuso el modelo propuesto, sobre los autómatas que definen modelos atómicos y como se acoplan para modelar grandes realidades, todo en lenguaje de autómatas. Bruno leyó un poco de segunda y tercera fase pero no encontró mucha información. Habló un poco de lo que encontró sobre programación funcional y simulación (en Erlang) y algunos patrones de arquitectura de frameworks, como GenVoca. Fernando analizó la biblioteca C++SIM y entregó un informe escrito.

Planificación para las próximas 2 semanas

Sebastián: continuar con la investigación en Devs (Cell Devs, Parallel Devs, etc.)

Bruno: encontrar si hay algo de fases y arquitecturas, leer papers de ACM Portal

Fernando: estudiar simulación paralela en simposios y papers.

Proyección al futuro

Seguir analizando sistemas existentes, sobre todo en la parte de modelado y visualización más que en la parte estadística. Sentar las bases, paradigmas y patrones para comenzar a programar a partir del segundo semestre.

Próxima Reunión

Martes 21 de Junio, 18:00

4.2. Proyecto de grado SimPP - Acta de la reunión del 21/06/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia.

Investigación durante las semanas anteriores

Sebastián avanzó en su investigación de Devs y expuso el modelo de simulación utilizado, con los componentes del simulador abstracto y su mecanismo de mensajes. Bruno leyó artículos de tercera fase en ACM y encontró varias discusiones acerca de diferentes posturas, pero nada sustancioso. Buscando en bibliotecas de simulación acerca del tema encontró una biblioteca en C++ bastante elemental llamada SIM orientada a eventos y comentó sus características. Fernando leyó acerca de paralelización en simulaciones en un simposio de simulación de la

IEEE y comentó los enfoques utilizados para la división de tareas y sincronización de los simuladores distribuidos.

Planificación para las próximas 2 semanas

Redondear la investigación en cada una de las áreas y tener reuniones para analizar qué alternativa tomaremos para cada aspecto de nuestra biblioteca, eligiendo de entre lo estudiado y nuestra propia visión de funcionamiento de una biblioteca. Consultar con Luis Sierra acerca de estudios teóricos en el área.

Proyección al futuro

Terminar en estas semanas la recopilación del estado del arte y características, a la vista del material adquirido y la falta de más material interesante. Establecer las bases y tener para Julio una idea básica para comenzar a trabajar en la aplicación.

Próxima Reunión

Martes 5 de Julio, 18:00

4.3. Proyecto de grado SimPP - Acta de la reunión del 02/07/2005

Asistentes

Sebastián Alaggia, Bruno Martínez.

Objetivos de la reunión

Definir el calendario del proyecto a corto plazo y cierre de fase de investigación sobre el estado del arte.

Calendario

En las próximas semanas el equipo deberá completar la documentación sobre el estado del arte, comenzar con el diseño del producto final, definir un entorno de desarrollo e implementar prototipos.

- Del 2 al 15 de Julio: En esta fase, cada miembro deberá informar en forma anticipada los temas sobre los cuales entregará informes, para evitar redundancias. Dichos informes deberán contener una introducción, un desarrollo, conclusiones y las referencias correspondientes (ver formularios de tesis enviados por María el 2005-05-24).
- Del 15 de Julio al 15 de Agosto: Se definirá y establecerá el ambiente de desarrollo, y se comenzará el diseño y la prototipación.

Macro Diseño

Se apunta a diversos paradigmas de modelado interoperando en simultáneo. No solo se quiere que un proyecto de simulación pueda ser encarado de varias maneras con nuestra librería, sino que también se puedan juntar ideas de distintos paradigmas en el mismo proyecto.

La arquitectura inicial consiste de la definición de una interfaz de núcleo, de la que dependerán todos los otros subsistemas. En este núcleo entraría básicamente el manejo del tiempo global, y de la lista de eventos pendientes. Los eventos son completamente opacos. Dentro de un evento puede estar escondido un modelo Devs o todo un scheduler secundario. La interfaz de

este componente núcleo es básicamente el agendamiento de nuevos eventos, y la consulta y avance del tiempo.

Hay varios subsistemas que construyen sobre este núcleo. Estos incluyen simulación basada en procesos, en eventos con 2 y con 3 fases y Devs. El desarrollo de los adapters correspondientes corre por cuenta de la librería.

Definición del ambiente de desarrollo

Se trabajará principalmente con Visual C++ y su IDE. Cada miembro tendrá también disponible el G++ para chequear compatibilidad una vez que se tiene cierta confianza en el código.

Se instalará algún mecanismo de control de versiones. Este puede estar localizado en facultad, lo que probablemente significaría usar CVS. Si no fuera el caso, dos miembros tienen conexión permanente así que se podría ver de instalar algo en alguna de esas máquinas. En ese caso podría ser útil investigar otras herramientas, como por ejemplo Darcs, que permiten trabajo más fluido.

Se distribuirán las librerías de base necesarias (Boost y SDL por ahora) y se procurará su instalación.

Se instaurará la infraestructura para realizar tests unitarios sobre el código, mediante la librería Boost test.

Próxima Reunión

Martes 5 de Julio, 18:00

4.4. Proyecto de grado SimPP - Acta de la reunión del 14/08/2005

Asistentes

Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Trabajo realizado

Se decidió empezar con una versión reducida de la librería para minimizar discusiones de diseño en cuanto al nivel de generalidad deseado. La idea es trabajar con un enfoque iterativo e incremental, más específicamente, test-driven. Luego de llegar a una primera versión se impactarán los requerimientos deseados sobre el estado actual de la librería, primero escribiendo los tests que se quieren hacer pasar y luego quizá modificando la librería para que lo hagan. Lo ideal es que el núcleo no sufra mayores modificaciones, pero para algunos requerimientos será inevitable.

Se discutieron las modificaciones que se prevén serán necesarias para la incorporación de los requerimientos de implementación de tercera fase, en dos formas posibles, interacción de procesos, serialización y salida gráfica.

Trabajo de los miembros a futuro cercano

Se apunta a un mayor dominio de las técnicas esenciales de la programación moderna en C++, por lo que se continuará con el estudio del libro "Modern C++ Design" de Andrei Alexandrescu. Así mismo, se estudiará la librería Boost.Test, indispensable para la metodología planteada.

Ahora que el uso de CVS está establecido, se desea mover todo el trabajo pasado al mismo, de manera de solo depender de esa herramienta para la distribución del trabajo. Esta actividad no es de gran prioridad por el momento, sin embargo.

Próxima Reunión

Martes 31 de Agosto, 12:00

4.5. Proyecto de grado SimPP - Acta de la reunión del 31/08/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia.

Trabajo durante las semanas anteriores

Se instalaron todas las herramientas de desarrollo y CVS en los equipos de todos los integrantes.

El equipo se dedicó a la elaboración de un prototipo que permitiera simular modelos en un bajo nivel de lenguaje, pero que brindara las funcionalidades varias. Hubo varias reuniones de diseño, implementación y testeo. Al final se obtuvo el prototipo deseado. Durante la reunión se expuso el prototipo y se entregó un informe con los principales puntos de este prototipo: descripción, arquitectura, uso y posible expansión.

Planificación para las próximas 2 semanas

Continuar con el desarrollo del prototipo según los puntos detallados en el informe. Investigar sobre políticas de concurrencia. Aumentar el nivel del lenguaje de simulación utilizado. Empezar a trabajar con los datos de la aplicación (caso de estudio de Rivera).

Proyección al futuro

En poco tiempo se piensa terminar con el código de la librería y sus funcionalidades para comenzar a trabajar en la aplicación propuesta. Se busca verificar la herramienta exhaustivamente. Continuaremos en el estudio de técnicas modernas de implementación a fin de tener más herramientas de diseño en código.

Próxima Reunión

Martes 13 de Septiembre, 12:00

4.6. Proyecto de grado SimPP - Acta de la reunión del 13/09/2005

Asistentes

María Urquhart, Antonio Mauttone, Héctor Cancela, Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Discusión

Se aprovechó la presencia de Héctor para resolver algunas dudas en cuanto a resolución de conflictos de activación de procesos al trabajar en este paradigma. En este tema no se tomó ninguna alternativa en particular como preferido, así que decidimos seguir con la política

actual. También se le preguntó acerca de la serialización/clonación de las corridas, ante lo cual la respuesta fue de 'útil' a ambas opciones.

Se expusieron los avances en las áreas de trabajo de cada uno durante la semana. Fernando se ocupó de investigar más a fondo la visualización gráfica. Sebastián creó unos módulos básicos de histogramas, pero aún sin reportar salida. Bruno investigó las alternativas de serialización y expuso la problemática de la serialización de funciones y threads.

Próxima Reunión

Martes 27 de septiembre, 12:30 horas

4.7. Proyecto de grado SimPP - Acta de la reunión del 27/09/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Discusión

Se realizó la presentación del caso de estudio sobre el sistema de transporte de Rivera. Se simulará los diferentes coches que recorren la ciudad y los pasajeros que desean ir de un sitio a otro. Existen dos posibilidades, agrupando las diferentes paradas en zonas o tratándolas individualmente. Se cuentan con datos preprocesados sobre la primera opción, que será la primera en ser abordada.

Se mostraron los progresos obtenidos con el subsistema de salida gráfica. Habrá que decidir en el futuro si se usa para el caso de estudio esta infraestructura o si por el contrario se escribe código a medida para poder representar mejor la simulación de tránsito.

Se atendió a la cercanía del mes de diciembre y las posibilidades de terminar para esas fechas con el proyecto. Un integrante tiene la necesidad de quedar libre para enero, además de que es la fecha inicialmente estipulada. Se prevé una intensificación de los esfuerzos, así como la realización de un nuevo cronograma ajustado, que tenga en consideración las influencias de las otras ocupaciones de los integrantes del grupo en el alcance de las metas.

El viernes siguiente a la reunión se realizará una presentación sobre el progreso del trabajo, en la sala de postgrado del InCo. Se acordó la reunión del grupo para esa misma tarde de martes para la confección de las transparencias y de la tarde siguiente para el ensayo y puesta a punto de la ponencia.

María Urquhart pidió se le mostraran las actas sobre las reuniones anteriores que el grupo realiza como parte de la documentación del proceso. Se acordó la reunión siguiente para la entrega de las mismas.

Próxima Reunión

Martes 11 de octubre, 12:30 horas

4.8. Proyecto de grado SimPP - Acta de la reunión del 11/10/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Discusión

El viernes 30 de setiembre tuvo lugar la presentación de avance del proyecto SimPP. Dicha presentación consistió en un resumen del estudio del estado del arte efectuado por el grupo en los meses iniciales del proyecto, una breve explicación del diseño de la biblioteca de simulaciones que se ha implementado hasta el momento y una muestra de la misma. La presentación fue buena, se logró presentar adecuadamente el trabajo realizado por el equipo hasta el momento.

En la reunión del martes 11, el equipo presentó una primera versión de la simulación del caso de estudio presentado en la reunión anterior. El enfoque utilizado para implementar el sistema es un enfoque híbrido de interacción de procesos y eventos. Los ómnibus son procesos que recorren paradas definidas por el recorrido que tienen asignado. En cada parada suben o bajan pasajeros. Los feeders de los ómnibus y pasajeros son programados como eventos fijos.

El equipo implementó el modelo en su forma agregada, con zonas en vez de paradas debido a que los datos de este modelo se encuentran más fácilmente disponibles.

La simulación construida hasta el momento es una primera aproximación al problema, por tanto no consideró las restricciones de capacidad de los ómnibus o los datos brindados por los tutores. Pero se logró implementar una versión con un modelo reducido y con el mismo formato de los datos del modelo.

De todos modos se está tratando de implementar el modelo de forma más modular utilizando el patrón Model-View-Controller y GenVoca, pero todavía no se tienen muchos avances al respecto. Además se solicitaron datos geográficos sobre el sistema de transporte público de Rivera para la salida gráfica de la simulación.

Se mostraron las actas de las reuniones anteriores, solicitadas por María Urquhart.

Finalmente se remarcó la importancia de establecer un cronograma que permita terminar el proyecto en diciembre. Dicho cronograma aún no ha sido confeccionado.

Próxima Reunión

Martes 25 de octubre, 12:30 horas

4.9. Proyecto de grado SimPP - Acta de la reunión del 25/10/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Discusión

Para esta reunión se elaboró un primer prototipo del caso de estudio, modelado utilizando SimPP y una arquitectura GenVoca con capas de modelo, display, tracer, controlador de tiempo y controlador de procesos. Se demostraron las capacidades de modelado y visualización básicas de la herramienta. Utilizando este modelo se validaron varios puntos del sistema y se discutió acerca de los siguientes puntos:

- Despreciar el tiempo de subida de pasajeros: se decidió despreciarlo, pues es absorbido por el tiempo de viaje.
- Tiempo de recorrido de ómnibus estocástico: se decidió utilizar tiempo fijo pues en la realidad se ciñen al horario. Se ignoran posibles accidentes, atascos de tráfico, etc.

- Tiempo de llegada de pasajeros estocástico: se decidió tomar una distribución exponencial con las tasas sugeridas.
- Visualización de los ómnibus: se decidió utilizar ómnibus que cambien de color a medida de que van más llenos
- Capacidad de los ómnibus: si bien no es realista, se decidió no tomar ninguna tolerancia a sobrepasar la capacidad de los ómnibus con pasajeros parados. En todo caso, se deberá aumentar el número absoluto de aceptación.
- Visualización de las líneas: se debe permitir mostrar o no determinadas líneas, a modo de facilitar la comprensión.

Acerca de la visualización, se discutió acerca de como representar los eventos de duración cero. Se puede parar el reloj y mover, o simplemente teleportar. No se tomó ninguna decisión acerca de esto.

Se discutió acerca de la posibilidad de viajar a Rivera a recolectar más datos sobre la realidad, aunque parece poco probable o útil. El objetivo del caso de estudio es principalmente probar y demostrar las capacidades de la herramienta, no el estudio de la realidad.

Trabajo para las próximas semanas

Se comenzará a trabajar sobre la estructura y el contenido del informe de proyecto. Se espera tener una estructura organizada para la próxima reunión.

Próxima Reunión

Martes 8 de noviembre, 12:30 horas

4.10. Proyecto de grado SimPP - Acta de la reunión del 09/11/2005

Asistentes

Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia

Discusión

De aquí en más las reuniones se cambian para los miércoles, debido a la disponibilidad horaria de Sebastián Alaggia.

Para esta reunión se elaboró un primer temario de la documentación del proyecto. Esta consistirá en 2 documentos: Manual del Usuario Programador de SimPP e Informe Ejecutivo del Proyecto. En el primero se documentará tanto las funcionalidades de la librería como la implementación del la librería para que pueda seguir siendo desarrollada. El informe ejecutivo consta de un resumen de todo el proyecto, y luego varios apéndices donde se desarrollan en profundidad los diversos temas investigados a lo largo del proyecto, las decisiones de diseño tomadas por el equipo y el proceso seguido por el mismo. La idea de ofrecer dos documentos separados es para que quien use la librería no tenga que andar buscando en el informe los detalles de uso e implementación de esta. La estructura del informe que se propuso fue aceptada, aunque algunos apéndices fueron cambiados de lugar para dar un mejor orden a dicho informe.

Luego se habló sobre preparar una demo para llevar a Rivera el 28 de noviembre. Dicha demo implica algunas modificaciones al modelo de caso de estudio implementado y agregar algunas modificaciones a la librería. Dichas modificaciones son

- Cambiar la imagen de fondo de la demo, para que se pueda ver la red en el sentido correcto y un mapa de Rivera.
- La velocidad de los ómnibus no debe ser constante, sino que debe calcularse de forma tal que dé para cumplir con la duración del recorrido.
- Se debe agregar un factor de carga a la capacidad del ómnibus. Esto implica que los ómnibus tiene una capacidad constante (cantidad de gente sentada) y un factor que permite subir pasajeros por encima de esa capacidad, para permitir a algunos pasajeros viajar parados. Un valor de 1.5 parece razonable para este factor.
- Los ómnibus tienen distintos colores de acuerdo con su línea. Además se debe cambiar a otro color (o ícono) si el ómnibus ha sobrepasado su capacidad (lleva pasajeros parados).

En principio ningún integrante del grupo viajará a Rivera para mostrar la demo, está será entregada a los tutores.

Finalmente, se logró implementar serialización para simulaciones en SimPP. Aunque la interfaz lograda no es tan pura como la implementada para los modelos anteriores, ya que requiere heredar de ciertas clases y se pierde la funcionalidad de Boost bind. La serialización solo está implementada para eventos y con un scheduller especial. La prueba de esta funcionalidad consistió en tomar un modelo implementado anteriormente, correrlo por una cantidad de tiempo, serializarlo, correrlo nuevamente y por último correr la copia serializada. Dicha prueba resultó exitosa, se llegó a los mismos estados del modelo.

Para la próxima reunión debe estar lista la demo para Rivera, y se debería comenzar a escribir la documentación.

Próxima Reunión

Miércoles 23 de noviembre, 12:30 horas

4.11. Proyecto de grado SimPP - Acta de la reunión del 23/11/2005

Asistentes

Antonio Mauttone, Fernando Pardo, Sebastián Alaggia

Discusión

La primera parte de la reunión se dedicó a hacer un relevamiento de los requerimientos faltantes en el modelo de Rivera. Se expuso el modelo actual y una breve demostración de su funcionamiento. Tras discutir los distintos aspectos se decidió que los elementos faltantes eran:

- Poder ocultar y mostrar buses de una línea
- Poder ocultar y mostrar el grafo de zonas
- Mostrar una leyenda con la referencia color-línea en los buses.
- Mostrar un reloj de la simulación

Luego se discutió el posible tribunal a reunir para la evaluación del proyecto. Dentro de las opciones se manejaron Juan José Cabezas, Daniel Calegari y Héctor Cancela. Sobre las fechas, se pensó sobre finales de diciembre. Se le comunicarán estas opciones a María Urquhart para que encuentre el tribunal disponible más conveniente.

Próxima Reunión

Miércoles 7 de diciembre, 14:00 horas

4.12. Proyecto de grado SimPP - Acta de la reunión del 07/12/2005

Asistentes

María Urquhart, Antonio Mauttone, Bruno Martínez, Fernando Pardo, Sebastián Alaggia.

Discusión

En esta reunión se mostró la versión final del ejecutable del caso de estudio, el cual será mostrado a los clientes finales de la ciudad de Rivera la semana siguiente. Se discutió la escalabilidad de la aplicación, dándose explicaciones de parte del grupo sobre cuales son los puntos que más importan en la performance de la aplicación, en particular como al hacerse la actualización de la salida gráfica en cada tercera fase podría traer que la simulación corriera lentamente.

La última fecha posible para defender la tesis de grado quedó establecida para el 28 de diciembre, por disponibilidad del profesor Daniel Calegari. También estaría confirmada la presencia de Héctor Cancela en el tribunal, pero no así la de Juan José Cabezas. Se iniciaron gestiones para tratar de que sea Alfredo Viola el tercer integrante de la mesa.

Se hicieron averiguaciones con los tutores sobre si era pertinente o no documentar el proyecto desde un punto de vista anecdótico, sobre todo la motivación inicial y los antecedentes como pasantía del integrante Sebastián Alaggia. Quedó resuelta la inclusión de esto.

Como los tutores estarán unos días en la ciudad de Rivera, quedó para su regreso la entrega de la documentación completa producida hasta el momento, a fin de acelerar su evaluación sin complicar con versiones a los tutores.

Posiblemente se trate ésta de la última reunión por el proyecto dada la cercanía de la entrega, pero se espera continuar el intercambio con los tutores por otros medios para asegurar la mayor calidad posible en la recta final.

5. Referencias

[BCNN00] BANKS, Jerry; CARSON, John S. II; NELSON, Barry L.; NICOL, David M. Discrete-event system simulation. Upper Saddle River, NJ: Prentice Hall, 2000. p.594. ISBN: 0130221021

[PFL98] PFLEEGER, Shari L. Software engineering : theory and practice. Upper Saddle River, NJ: Prentice Hall, 1998. p.576 ISBN: 0-13-624842-X

[Btst05] Boost test

http://www.boost.org/libs/test/doc/index.html Agosto 2005

[TCVS05]

Tortoise CVS Home Page

http://www.tortoisecvs.org/ Agosto 2005

[XP05] Extreme Programming: A gentle introduction.

http://www.extremeprogramming.org/ Noviembre 2005

Apéndice H - Glosario

[Actividad]

Término utilizado por los paradigmas clásicos de simulación que define la ocupación de una entidad en una tarea determinada. Se inicia mediante un evento condicionado (la liberación de un recurso necesario) y termina en un evento fijo (el tiempo de duración de la actividad).

[Backend]

Unidad de procesamiento de datos, cuya entrada proviene de un front-end. Refiere a motores de bases de datos, compiladores de tokens, filesystems, etc.

[Balking]

Cuando una entidad no entra en una cola por el largo de la misma o condiciones similares. La entidad se elimina del sistema.

[Calendario]

El conjunto de los eventos fijos con tiempos mayores que el actual de la simulación.

[Concepto]

Nombre que se le da en programación genérica al conjunto de requerimientos sintácticos y semánticos sobre un tipo.

[Ejecutivo]

El mecanismo por el cual se corre una simulación.

[Entidad]

Un objeto activo del dominio del modelo que tiene propiedades propias.

[Evento]

La ocurrencia de un cambio de estado en un sistema. En simulación a eventos discretos se distinguen los eventos fijos, que ya se sabe cuando sucederán y marcan la finalización de una actividad, y los eventos condicionados, que dependen de que se satisfagan algunas condiciones.

[Función libre]

Una función que no pertenece a ninguna clase, sino que habita directamente el espacio de nombres global.

[Functor]

Los objetos de C++ que pueden ser invocados como una función, ya sea porque sobrecargan el operator() o porque son punteros a función.

[Framework]

Organización de una librería orientada a objetos en la cual se provee una estructura para el programa. El cliente de la librería agrega clases que heredan de las de la librería.

[Frontend]

Interfaz de entrada con procesamiento previo. Parte de un programa que obtiene datos, los procesa y los envía al backend.

[GNOME]

Interfaz de usuario desarrollada por el movimiento GNU para sistemas tipo Unix. Es la interfaz de usuario oficial del proyecto GNU. La sigla GNOME significa GNU Network Object Model Environment.

[GNU]

Proyecto de la Fundación por el Software Libre que desarrolla un entorno completo de software, incluyendo desde el sistema operativo hasta las aplicaciones de alto nivel.

[Histograma]

Un conjunto de datos indexados.

[Metaprogramación]

La escritura de programas que escriben programas a su vez.

[Mozilla]

Navegador de código abierto de la Fundación Mozilla, que forma el centro de los navegadores FireFox y Netscape.

[Píxel]

Abreviación en inglés de elemento de imagen. Es la unidad más chica de área que puede tomar un color.

[Proceso]

La descripción lineal de las acciones de una entidad.

[Ojo de águila]

Capacidad de poder ver de lejos y con detalle.

[Overloading]

Cuando dos elementos tienen el mismo nombre en un programa. El compilador las diferencia por el contexto, por ejemplo, por la cantidad de argumentos que recibe cada una de dos funciones.

[Overriding]

La acción de volver a definir una función en una subclase, sustituyendo la función de la clase base.

[Polimorfismo]

Cuando un mismo código puede aplicarse a objetos de diferentes tipos. En C++ se usa en los contextos de funciones virtuales o polimorfismo dinámico y de despacho mediante funciones sobrecargadas a partir de templates, o polimorfismo estático.

[Rasterización]

El procedimiento de convertir una descripción de una imagen en una matriz de píxeles.

[Reificar]

Término utilizado en computación e inteligencia artificial para describir la acción de crear un modelo de datos para un concepto abstracto. Esto permite a la computadora procesar la abstracción como si fuera cualquier dato.

[Recurso]

Un ítem presente en la simulación que no tiene entidad y es pasiva, pero restringe el progreso de las entidades.

[Reneging]

Cuando una entidad que estaba en una cola sale de la misma antes de ser atendido.

[Renderer]

El componente responsable de rasterizar y mostrar en pantalla una escena.

[Smart Pointer]

Objeto de C++ que se comporta sintácticamente como un puntero, pero diseñado para ayudar a controlar el tiempo de vida del objeto apuntado.

[Sobrecarga]

Traducción de overloading.

[Sprite]

Imagen bidimensional independiente que es posible mover libremente por la pantalla.

[Stream]

Secuencia de elementos posiblemente infinita que es accedida bajo demanda.

[Swopping]

Cuando una entidad se pasa de una cola a otra, debido a que le conviene el cambio por alguna razón.

[Template]

Construcción específica de C++ que permite parametrizar una clase o función con respecto a otros tipos o constantes.