Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica

Area Computación

Entorno de programación visual para reglas ECA

Tesis que presenta

Mónica Rivera de la Rosa

para obtener el Grado de

Maestra en Ciencias

en la Especialidad de

Ingeniería Eléctrica

Director de la Tesis

Dr. José Oscar Olmedo Aguirre

México, D.F. **Diciembre 2006**

Agradecimientos

Agradecimiento especial para mi asesor el Dr. José Oscar Olmedo Aguirre, quien gracias a su gran ayuda, paciencia y comprensión, me brindó el apoyo necesario para realizar esta tesis.

A los doctores y el personal administrativo de la Sección de Computación del Departamento de Ingeniería Eléctrica por sus enseñanzas brindadas en el aula de clases. A Sofi, por tener siempre la paciencia necesaria para auxiliarme en todo lo que he necesitado durante mi estancia en el Cinvestav.

A CONACyT por el apoyo económico que me brindó durante el transcurso de la maestría. A servicios escolares y a la biblioteca de Ingeniería Eléctrica por los servicios y ayuda que me brindaron.

A los revisores de este trabajo el Dr. Sergio Chapa y el Dr. Pedro Mejía, por darme el tiempo para la revisión.

Al Centro de Investigación y de Estudios Avanzados del IPN (CINVESTAV-IPN), Unidad Zacatenco, por ofrecerme las instalaciones necesarias y recursos necesarios para realizar este trabajo de investigación.

Resumen

En el proceso de desarrollo de software se han reconocido ampliamente los problemas que se derivan de la falta de integración de la programación con su diseño y documentación. Para ayudar a resolver estos problemas, en esta tesis se propone adoptar a los diagramas UML de secuencia como lenguaje de programación visual que permita diseñar programas autodocumentados.

Con este fin, se construyó el ambiente experimental de programación `Moon` que usa diagramas UML de secuencia para describir y visualizar las interacciones que tienen lugar en sistemas reactivos. El entorno utiliza `ADM`, un lenguaje de reglas evento-condición-acción, como lenguaje y modelo de programación que integra capacidades deductivas al ciclo de reconocimiento-actuación que caracteriza a los sistemas basados en reglas. El ambiente de programación permite construir en forma interactiva las reglas que describen el comportamiento de los componentes del sistema y visualizar los efectos que producen cuando estas se ejecutan. Los diagramas UML de secuencia se traducen en las abstracciones del lenguaje `ADM` que incluye primitivas de comunicación entre componentes por intercambio de documentos XML (como enviar y recibir) y estructuras de control usuales (como secuencial, paralelo y condicional).

Entre las contribuciones de este trabajo se encuentran: (i) la adopción de los diagramas UML de secuencia como lenguaje de programación visual, (ii) la construcción de un entorno de programación que permite editar, compilar, ejecutar y visualizar programas descritos en forma independiente del lenguaje y de la plataforma operativa, y (iii) la documentación de los aspectos dinámicos de un sistema mediante la visualización de sus interacciones (aún cuando los programas no hayan sido producidos en el entorno). `Moon` también puede utilizarse en la preparación de materiales educativos para cursos de programación, comunicaciones e ingeniería de software, entre otros.

Abstract

In the process of software development, it has been widely recognized the problems that derive from the lack of integration of programming with its design and documentation. In order to solve these problems, in this thesis its proposed to adopt UML sequence diagrams as a visual programming language that allows to design self-documented programs.

With this aim, it was constructed the experimental programming environment `Moon` that uses UML sequence diagrams to describe and to visualize the interactions that take place in reactive systems. The environment use `ADM`, a language event-condition-action rule-based, as language and model programming that integrates deductive capabilities to the cycle of recognition-action that characterizes rule-based systems. The programming environment allows to the user to construct interactive rules that describe the behavior of the components of the system and to visualize the effects that it produces when the components are executed. UML sequence diagrams are translated in the abstractions of the `ADM` language that includes communication primitives between components by exchanging XML documents (like send and receive) and the usual structures of control (like sequential, parallel and conditional).

Among the contributions of this work they are: (i) the adoption of UML sequence diagrams as a visual programming language, (ii) the construction of a programming environment that allows to edit, to compile, to execute and to visualize programs that are described independently from the language and the operative platform, and (iii) the documentation of the dynamic aspects of a system by means of the visualization of its interactions (even when the programs have not been developed in the environment). `Moon` also can be used in the preparation of educative materials for courses on programming, computer communications and software engeneering, among others.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Se ha reconocido que el diseño y construcción de programas concurrentes en un ambiente distribuido es una tarea compleja. En este aspecto la programación basada en reglas se ha propuesto para mejorar el entendimiento del comportamiento de programas complejos. Las especificaciones basadas en reglas se usan en sistemas reactivos que monitorean la ocurrencia de eventos los cuales pueden revelar condiciones críticas. En los sistemas basados en reglas, una vez que el evento ha sido detectado, si los parámetros del mismo satisfacen una condición, una acción específica es ejecutada para manejar la situación. El concepto de lenguaje que corresponde a este modelo de interacción es llamado regla ECA (Evento-Condición-Acción).

El uso de reglas ECA en sistemas distribuidos se origina en el hecho de que los sistemas distribuidos son también sistemas reactivos cuyo comportamiento puede conducirse tanto por los mensajes recibidos (eventos) como por los mensajes enviados como respuesta (acciones). Las reglas ECA son la base de ADM, un modelo Activo-Deductivo que usa XML como representación uniforme de datos para el intercambio de mensajes en aplicaciones distribuidas [11]. ADM es un lenguaje de programación que fue diseñado para coordinar agentes de software con un comportamiento autónomo, proactivo, racional y social. Las descripciones basadas en reglas constituyen un poderoso paradigma, cuyas interacciones son difíciles de entender porque estas de manera general no fueron diseñadas para aplicarse en forma estructurada o jerárquica.

Con el propósito de simplificar las interacciones entre reglas, en este trabajo de tesis se utilizan los diagramas de secuencia UML para visualizar la definición de reglas, su selección y ejecución. De esta manera con la definición visual de una regla puede ser más fácil entender su comportamiento mediante un diagrama de secuencia UML que muestre la estructura evento-condición-acción de la regla. Además, el proceso de selección y concretización de reglas presenta mayor coherencia en su descripción como una colección de diagramas de secuencia que pueden ser identificados por la ocurrencia de los eventos

que los inician. Finalmente, la ejecución puede ser más comprensible porque las reglas se agrupan de manera secuencial o paralela.

Aunque la motivación de este trabajo se originó en los sistemas distribuidos, debido a las complejidades inherentes de la infraestructura tecnológica que le da soporte (RPC,Corba,CGI o Servicios Web), en esta tesis se consideran únicamente sistemas que no usan distribución (residen en un computadora).

1.1. Reglas ECA

Con el propósito de que los sistemas administradores de repositorios (o colecciones) de documentos XML brinden mayores prestaciones para la automatización de actividades, recientemente se ha propuesto extenderlos con infraestructura de coordinación que de soporte a la representación y ejecución de reglas evento-condición-acción (ECA). Las reglas ECA constituyen por si mismas un paradigma de programación que ha sido popularizado por la relativa simplicidad conceptual y la eficiencia de implementación que se ha alcanzado en los modernos sistemas de bases de datos activas (*triggers*) como Starburst[23], HiPac[17] y Postgress[22].

Las reglas ECA[12][13][14], como generalmente se les conoce, surgen al extender el modelo de bases de datos relacionales con métodos basados en reglas de producción usados para la representación y extracción de conocimiento y cuyos orígenes se sitúan en el campo de la Inteligencia Artificial. Las reglas ECA[6] incorporan la capacidad de realizar acciones de inserción, eliminación o modificación de registros en un tabla como respuesta a la detección de eventos de este tipo.

La programación basada en reglas para la administración de documentos XML es el resultado de la evolución de diversas áreas de investigación que incluyen también a la programación lógica, a la programación concurrente y a la automatización de actividades y servicios en Internet principalmente a través de servicios Web. Históricamente, esta evolución tiene sus orígenes en los sistemas de bases de datos, los cuales al incorporar reglas deductivas, por un lado, y reglas de producción, por otro, dieron lugar a las bases de datos deductivas y a las bases de datos activas, respectivamente. La conjunción de ambos enfoques ha dado lugar a las bases activas deductivas.

Por otra parte, la incorporación de estos desarrollos en la administración de colecciones de documentos XML ha conducido a investigaciones semejantes ya que el concepto de documento electrónico generaliza los conceptos de registro y tabla de las bases de datos.

A continuación se presentan en orden histórico las áreas de estudio mencionadas así como algunas de sus características más sobresalientes.

Las bases de datos (BD) permiten almacenar la información que usan los sistemas de información[11]. La BDs relacionales se constituyen de tres elementos fundamentales:

modelos de datos, lenguajes de consulta y álgebras de consulta. Los *modelos de datos* describen colecciones homogéneas de registros o tuplas agrupadas en tablas. Los *lenguajes de consulta* permiten describir las condiciones que deben cumplir los datos buscados. Las *álgebras de consulta* son un conjunto de operadores que permiten optimizar consultas.

Las bases de datos deductivas (BDD) extienden las capacidades de las BDs ya que permiten hacer consultas que implican inferencia lógica. Algunos ejemplos de BDD son Datalog[11], LDL, Glue-Nail y Coral entre otras. Las BDD usan la lógica de primer orden como fundamento teórico, lo que permite modelar datos, programas, consultas y restricciones de integridad. Con lo anterior se ofrece un lenguaje común para la programación de aplicaciones. En consecuencia, el usuario emplea únicamente conceptos declarativos relacionados con la aplicación.

Las bases de datos activas (BDA) poseen capacidades de detección y reacción ante eventos. Entre los sistemas de BDA podemos citar Ariel[15], Starburst [23], RPL, Postgres, HiPAC[17] y Ode[19]. Las BDA permiten definir reglas ECA en los sistemas de bases de datos relacionales. Las reglas ECA, también conocidas como reglas de producción, son el componente que hace que los sistemas de bases de datos (o bases de documentos) tengan la característica de ser reactivos.

Dado que XML se ha convertido en el medio predominante para el intercambio de información en la Web, se introdujeron también las bases de documentos XML (BDX). Las BDX, en forma similar a las BD convencionales, también necesitan definir modelos de datos, lenguajes de consulta y álgebras de consulta[8]. En este caso los modelos de datos (DTD, Schema, DOM, etc.) definen la estructura y las reglas para construir documentos válidos. Los lenguajes de consulta (XML-QL, Lorel, Quilt, XQuery[10], etc.) definen expresiones usadas para localizar y extraer datos contenidos en documentos XML. Las álgebras de consulta (álgebra Lore, YAT, álgebra IBM, álgebra ATT, etc.) definen operadores que permiten realizar evaluación eficiente, optimización y planeación de la ejecución de expresiones de consulta.

Las bases deductivas de documentos XML (BDDX) extienden el modelo de datos de XML con conceptos fundamentales de programación lógica (elementos con variables, cláusulas, consultas, etc.). Lo anterior permite hacer inferencias lógicas sobre el contenido de los documentos almacenados en una BDX. Como ejemplos de BDDX podemos mencionar XDD[7] y LogCIN-XML[5].

Las bases activas de documentos XML (BADX) permiten definir reglas ECA sobre repositorios de documentos XML[4]. Las reglas ECA proporcionan funcionalidad para modificar el contenido de la BDX o diseminar información automáticamente como respuesta a cambios percibidos en el contenido de colecciones de documentos XML.

Las bases activas deductivas de documentos XML[9] (BADDX) buscan unificar la reactividad de las BADXs con la capacidad de derivar conclusiones lógicas de las BDDXs a partir del contenido de una colección de documentos XML.

ADM[32] (Active-Deductive Model) es un ejemplo del tipo de bases activas deductivas de documentos XML que se caracteriza por:

- **Reactividad**.

  ADM posee la capacidad de realizar las acciones indicadas por reglas como respuesta a eventos bien definidos.

  El efecto de las acciones se reflejan en la modificación del contenido de la colección (base extensional) de documentos por la inserción de nuevos documentos, la eliminación de algunos ya existentes o el intercambio de documentos mediante servicios de mensajería como correo electrónico o servicios Web. Cuando ocurre un evento en una colección de documentos el administrador de reglas selecciona aquella que sea apropiada de acuerdo al tipo de evento detectado. Si los parámetros del evento verifican la condición de la regla de acuerdo con el contenido de la colección, entonces se ejecutan las acciones indicadas por la regla.

- **Deducción**.

  ADM posee la capacidad para recuperar la información que relaciona a dos o más documentos de acuerdo con un conjunto preciso de condiciones lógicas.

  Este enfoque contrasta con el seguido en la mayoría de los motores de búsqueda tradicionales que agrupan documentos por la similitud de la frecuencia relativa de palabras clave en vez de relacionarlos por las condiciones lógicas que cumple la estructura y el contenido de dichos documentos.

1.2. Programación basada en Reglas ECA

El lenguaje ADM está formado por elementos XML, términos XML, procedimientos lógicos y consultas. Mientras que la parte activa del lenguaje se forma con eventos, acciones y reglas evento-condición y acción. Tales sistemas han encontrado diversas y muy importantes aplicaciones ya que constituyen un marco natural de integración de servicios.

Al igual que la programación orientada a objetos, la programación basada en reglas ECA es un paradigma de programación que facilita el desarrollo coordinado de las interacciones entre componentes. Las interacciones corresponden a invocaciones de los métodos que ofrece como servicios un objeto. Dichas invocaciones no necesariamente están coordinadas por un sincronizador global en la forma de una lista de instrucciones de un programa secuencial centralizado. Por el contrario, las invocaciones pueden realizarse en forma concurrente por una colección distribuida de objetos sin que necesariamente medie entre ellos alguna dependencia lógica, espacial o temporal.

El modelo de concurrencia de ADM se ha diseñado tomando como base al modelo de programación concurrente del lenguaje Ada, conocido como encuentro (*rendezvous*). Este modelo ha sido una de las demostraciones más exitosas de la complementariedad que las nociones de objeto (activo) y evento pueda tener.

El modelo de encuentro de Ada utiliza la construcción `accept` que permite aceptar invocaciones a una entrada (*entry*). La aceptación en una entrada es similar a la invocación usual de un procedimiento excepto que el programa establece las condiciones para aceptar la invocación. En este modelo, una invocación se puede considerar como un tipo especial de evento que puede procesarse solamente cuando el programa se encuentra en alguno de un conjunto de estados bien definidos de aceptación. Solamente en dichos estados de aceptación, el programa puede realizar las acciones asociadas con la invocación siempre que se cumplan algunas condiciones establecidas sobre los valores de los parámetros recibidos.

Puesto que ADM abstrae y generaliza las nociones de objeto y mensaje como tipos de documentos XML, su dominio de aplicaciones no se restringe a la administración de colecciones de documentos sino a la de cualquier aplicación concurrente dirigida por eventos.

Ya en un trabajo previo, algunas de las facilidades que brinda ADM para el desarrollo de aplicaciones distribuidas en la Web quedaron demostradas. En este trabajo se demuestra que también es posible desarrollar programas en entornos de programación convencionales (centralizados y secuenciales).

La generalidad de conceptos de ADM queda demostrada al producir código ejecutable para un lenguaje de programación específico como Java a partir de las reglas ADM. Por tales razones, en este trabajo se ha adoptado el modelo de programación de ADM como suficientemente general y expresivo para representar a un subconjunto significativo de los diagramas de secuencia de UML.

1.3. UML y MDA

En noviembre de 1997, el OMG (Object Management Group) propuso un conjunto de estándares para el diseño y el análisis orientada a objetos. El estándar, conocido actualmente como UML (Unified Modeling Language), incluye un conjunto de diagramas así como un formato para el intercambio de datos entre herramientas CASE. Recientemente, el OMG lanzó una iniciativa denominada MDA (Model Driven Architecture) cuya propuesta fundamental es que los modelos abstractos del sistema bajo consideración sean la base del proceso de desarrollo de software.

Los beneficios que brinda MDA con este enfoque son:

- **Abstracción.** MDA distingue dos niveles de abstracción en la concepción de un

sistema de software. En el nivel de abstracción mayor, se enfatiza el diseño sobre la implementación que es más bien el propósito del segundo nivel. Una vez distinguidos ambos niveles, se sugiere que el desarrollo del sistema se lleve a cabo mediante un proceso de transformación entre las entidades que conforman ambos niveles de abstracción.

- **Portabilidad.** La separación en dos niveles de abstracción simplifica también la tarea de implementar el sistema en diversas plataformas operativas. En MDA la realización de un modelo abstracto consiste en hacerlo corresponder con uno o más modelos concretos mediante un proceso de transformación. Los modelos concretos pueden variar de acuerdo a la infraestructura tecnológica disponible o bien simplemente a consideraciones pragmáticas.

- **Productividad.** La abstracción también promueve la comunicación, la cooperación y el trabajo en equipo entre clientes, diseñadores, analistas y programadores. La utilización de un lenguaje común (UML) y la definición de un modelo preciso de programación permiten integrar las diferentes perspectivas que cada uno de ellos posee. Por otra parte, la productividad se incrementa considerablemente al recurrir a herramientas que asistan en el proceso de interpretación y validación de la especificación abstracta del modelo así como en la generación de alguna implementación concreta en forma automatizada.

- **Interoperabilidad.** La adopción de estándares abiertos y de metodologías de programación basadas en ellos simplifican el diseño y la construcción de sistemas multiplataforma capaces de interactuar de manera eficiente y controlada. Al utilizar a UML como estándar abierto para la descripción abstracta de sistemas, se garantiza también que diversas herramientas puedan incorporarse no solamente en la construcción de modelos concretos para cada sistema operativo y lenguaje de programación sino también en su adaptación a los protocolos de comunicación y de sincronización basados en Internet y en la Web.

- **Facilidad de mantenimiento.** La abstracción que se puede alcanzar con la adopción de MDA en el proceso de desarrollo de software asegura también el desacoplamiento entre el diseño y la codificación. Dicha separación permite aislar las modificaciones de manera que se reduzca el impacto que puedan tener sobre la planificación y la puesta en marcha del sistema.

Los elementos básicos que conforman MDA consisten, como se ha dicho, de modelos de alto nivel de abstracción conocidos como PIM (*Platform Independent Model*), de modelos de bajo nivel de abstracción PSM (*Platform Specific Model*) y de transformaciones entre PIM y PSM así como de PSM a código. Generalmente, las transformaciones

fueron concebidas para ser realizadas por herramientas o entornos de programación que asistan en el proceso de automatización y producción de software.

1.4. Planteamiento del Problema

Desafortunadamente, los ambientes de programación que han adoptado los diagramas UML como la base para su metodología de diseño, en el mejor de los casos se usan para generar esqueletos de programas en leguajes de programación como Java y C++ porque los diagramas UML no pueden ser directamente ejecutados. Aunque reemplazar la representación textual de un modelo de programación por una representación visual constituye ya un avance substancial en proceso la abstracción en el diseño de algoritmos, el desarrollo del programa aún no puede visualizarse directamente en los diagramas UML debido a que su ejecución no está integrada al entorno. Por otra parte, los sistemas de visualización de algoritmos no usan diagramas UML para visualizar el comportamiento del programa.

1.5. Propuesta de solución

En este trabajo de tesis, se ha construido un entorno experimental de programación que resuelve los problemas planteados. Integrando la visualización del programa, la edición y la ejecución, el entorno de programación brinda notables ventajas sobre los trabajos previos:

- Se enfatiza el diseño del algoritmo sobre el código del programa
- Se abstrae la descripción del algoritmo de los lenguajes de programación y las plataformas
- Se visualiza el comportamiento del programa usando estándares ampliamente conocidos y aceptados como los diagramas UML

Para lograr los objetivos anteriores se diseño el entorno de programación compuesto por los siguientes módulos:

- Editor de diagramas UML de clases y de secuencia.
- Compilador de diagramas UML extendidos a programas ADM de reglas activas deductivas.
- Mediador para el intercambio de mensajes con el intérprete de programas ADM.

- Visualizador de la ejecución de procesos dirigida por intercambio mensajes.

Con estos módulos se consigue cubrir las fases de edición, compilación y visualización de los programas generados en el entorno.

El entorno `Moon` está diseñado para ejecutarse en la Web, haciendo uso de la tecnología XML y de su dialecto SVG de visualización. De esta manera el sistema puede estar disponible para poder utilizarlo en línea con las ventajas que esto conlleva como su de facilidad de distribución que no requiere de instalación.

1.6. Contribuciones y perspectiva del trabajo

De acuerdo con el proceso de desarrollo de MDA, el desarrollo de software debe centrarse en el diseño y especificación del modelo de alto nivel de abstracción del sistema. La incorporación de especificaciones precisas y tratables permite identificar y resolver en cierta medida los siguientes problemas:

1. Los puntos de vistas conflictivos que puedan surgir entre clientes, diseñadores y programadores.
2. Las decisiones de diseño que toman los programadores durante la implementación. Las decisiones pueden afectar la forma real en que se conduce una organización ya que no reflejan verdaderamente sus reglas de operación.
3. El escalamiento de un prototipo a su versión final.
4. La falta de un lenguaje de comunicación intuitivo pero bien fundamentado.
5. La ambigüedad en las descripciones expresadas por los clientes.
6. El mantenimiento ágil que debe ofrecer el sistema para apegarse al ritmo demandante que requiere su producción.
7. La visión global que ofrece la diagramación para revisar la satisfacción de los requerimientos establecidos sobre los productos de software.
8. La adopción de modelos formales que permitan automatizar tanto la validación de la especificación como su adecuada realización en la implementación.
9. La validación intuitiva de los resultados esperados de acuerdo a análisis por casos.
10. La visualización del funcionamiento del sistema.

En este trabajo no se consideran los problemas de desarrollo de software sino únicamente se pretende ofrecer un entorno de programación que utiliza los diagramas de secuencia de UML como lenguaje de programación visual.

1.7. Organización de la tesis

Este trabajo está organizado de la siguiente manera. En el Capítulo 2 se discuten algunos trabajos relacionados en las áreas de visualización de programas y herramientas de diagramación UML. En el Capítulo 3 se describe el entorno de programación `Moon` incluyendo su arquitectura global. En el Capítulo 4 se describen las representaciones visual y textual del lenguaje de programación junto con su especificación formal y también se muestran los esquemas de traducción definidos entre las representaciones. En el Capítulo 5, el lenguaje ADM, sus principios básico y como es usado para describir las reglas ECA. En el Capítulo 6 se muestra la implementación del entorno, los diversos módulos de programación que lo conforman y sus interrelaciones. Finalmente en el Capitulo 7 se presentan las conclusiones y algunas sugerencias para el trabajo futuro.

Capítulo 2

Antecedentes

La animación Knowlton [1] fue probablemente la primera representación dinámica de un algoritmo de estructuras de datos producido en 1981. Durante los últimos 15 años se ha desarrollado una amplia variedad de sistemas para visualizar algoritmos utilizando diferentes plataformas para su ejecución así como diferentes formas de interacción con el usuario lo que conduce a escenarios especializados de utilización.

Otro factor también importante a considerar es el tema de la concurrencia, por naturaleza complejo. Así pues lo es también en el aspecto de la visualización, por lo que se han desarrollado algunos sistemas para realizar esta tarea, los cuales son descritos en este capítulo.

También como parte del trabajo relacionado con el entorno desarrollado se tienen las diferentes herramientas CASE utilizadas básicamente para el modelado de diagramas UML.

De esta manera este en la sección 2.1 se presentan algunas de las herramientas de visualización de algoritmos más ampliamente conocidas, en la sección 2.2 se muestran las herramientas de visualización de los programas concurrentes y finalmente se presenta la sección 2.3 que contiene la descripción de algunas de las herramientas CASE (Computer Aided Software Engineering) de mayor aceptación y que algunas en su concepción tienen algún vestigio de la visualización de algoritmos. Otro tema de relevancia que es necesario mencionar en este apartado es el que tiene que ver con las herramientas para elaborar animaciones por lo que se hablara de ellas en la sección

2.1. Herramientas de visualización de algoritmos

Las animaciones interactivas son una herramienta muy valiosa para la enseñanza, divulgación y el aprendizaje electrónico ya que ayudan a mostrar el comportamiento de sistemas complejos, por lo que se han desarrollado múltiples herramientas para cumplir

con este propósito, muchas de ellas son herramientas que se basan en tecnologías WEB para su implementación.

Existen diferentes formas para clasificar los sistemas de visualización, por ejemplo de acuerdo con la interactividad que le permiten al usurario o el que se menciona en esta sección donde se clasifican de acuerdo a manera en que se generan las animaciones. De esta manera podemos decir que los sistemas para visualización de programas se clasifican básicamente en dos categorías: declarativo (manejado por datos) e imperativo (manejada por eventos) por ejemplo JAWAA.

2.1.1. Herramientas declarativas

Dentro de esta clasificación se encuentran las herramientas donde los objetos del programa son mapeados a imágenes y cada vez que hay un cambio en el objeto la imagen se actualiza [25]. Algunos ejemplos de estas herramientas son:

- Jeliot [1] ,s el cual permite al usuario elegir a partir de un programa codificado en Java, el conjunto de variables que quiere visualizar,

- WAVE [1], los algoritmos corren en un servidor remoto y sus estructuras de datos son publicadas en las pizarras del cliente y un manejador se encarga de hacer la animación correspondiente

- Pavane [1], fue diseñado para proveer visualizaciones (en tres dimensiones) de algoritmos concurrentes en modo declarativo. El visualizador declara una transformación entre un conjunto fijo de variables y la imagen final usando reglas de acuerdo al estilo declarativo.

2.1.2. Herramientas imperativas

En esta clasificación la animación responde básicamente a los eventos o ciertos puntos importantes dentro de la ejecución del programa, donde se agregan llamadas a rutinas gráficas , o simplemente pueden escribirse comandos en un archivo de texto el cual es interpretado por un editor gráfico [25], dentro de esta clasificación de encuentran entre otras las siguientes herramientas:

- Samba, esta compuesto por el sistema Tango, el cual es un intérprete que va leyendo comando por comando de un archivo ASCII para ir elaborando la animación.

- JAWAA [1], es un editor visual que también genera como Samba a partir de un script la animación del mismo.

- Balsa, en el que se hacen las animaciones de los algoritmos a través de su script propio para después correr la animación realizada.

2.1.3. Herramientas declarativas e imperativas

Existen también herramientas que utilizan ambos estilos, el imperativo y el declarativo [25], tal es el caso de:

- Leonardo Web [1] que es una herramienta que integra un ambiente de edición y una biblioteca para la creación de animación a partir de un script,
- ALICE [24] es un ambiente de animación interactiva en 3-d para ayudar a los estudiantes a visualizar la ejecución del programa y los estados del mismo.

Algunas de las herramientas mencionadas anteriormente hacen uso de las tecnologías Web para su implementación y distribución, sin embargo ninguna de ellas sigue un estándar al momento de hacer la representación en diagramas de los algoritmos que pretenden visualizar, ni tampoco existe algún estándar en los scripts utilizados por cada uno de ellos.

2.2. Herramientas de visualización de programas concurrentes

La visualización de los programas concurrentes es mas complicada que la visualización de diagramas secuenciales, debido a la presencia de múltiples hilos que participan, compitan por recursos y periódicamente se sincronizan. La mala interpretación del comportamiento de los programas concurrentes puede resultar en interacciones inesperadas y ejecuciones no determinísticas. Algunas herramientas de visualización han sido realizadas sobre dichos aspectos.

En la biblioteca Gthreads [16], se construye un grafo del programa como hilos que son creados y funciones llamadas, donde los vértices del grafo representan entidades del programa y eventos, mientras que los arcos representan el orden temporal entre ellos.

La vista del paso de mensajes es soportada por el sistema Conch [17] donde los procesos aparecen fuera de un anillo y los mensajes son intercambiados entre de ellos atravesando el anillo. De esta manera un mensaje que no es entregado puede ser detectado si permanece dentro del anillo.

El sistema Hence [16] proporciona vistas animadas del grafo del programa obtenidas a través de la ejecución de programas PVM. Nuevamente en estos entornos no se usa ningún estándar como los diagramas UML para describir el proceso de interacción.

2.3. Herramientas CASE

Actualmente existen diferentes herramientas CASE (Computer Aidded Software Engenniering), tanto de uso libre como propietarias que facilitan la edición y creación de Diagramas UML.

En su mayoría estas herramientas generan los esqueletos de código en algún lenguaje (por ejemplo Java), utilizando la información contenida en los diagramas creados, sin embargo no hay manera de trasformar el modelo completamente en un programa ejecutable donde los resultados a su vez puedan ser visualizados en los diagramas.

A continuación se detallan algunas de las características de las herramientas CASE mas relacionadas con el presente trabajo.

2.3.1. ArgoUML

El proyecto ArgoUML [30] fue iniciado por Jason Elliot Robbins y actualmente tiene más de 300 miembros, donde 12 de ellos encabezan el grupo. ArgoUML es un software gráfico que permite diseñar, desarrollar y documentar aplicaciones de software orientado a objetos con UML.

Sus características principales son:

- Está basado en estándares abiertos: XMI, SVG y PGML
- Independiente de plataforma (debido a que esta desarrollado en Java)
- Código abierto, el cual permite ampliarlo y personalizarlo.
- Está basado en tres teorías dentro de la psicología congnitiva: reflexión-en-acción, diseño oportuno y comprensión y solución del problema.

En esta herramienta los diagramas que pueden elaborarse son: Diagramas de clase, de estado, casos de uso, diagramas de actividades, de colaboración y desarrollo y genera esqueletos de código. Es una herramienta de uso libre. En la figura 2.1 se muestra una pantalla de Argo UML.

2.3.2. Rational Rose

Rational Rose es la herramienta CASE desarrollada por los creadores de UML (Booch, Rumbaugh y Jacobson), que cubre todo el ciclo de vida de un proyecto: concepción y formalización del modelo, construcción de los componentes, etc [33].

Soporta el modelado visual de los componentes, arquitectura, objetos, requerimientos técnicos del sistema, procesos de negocios y objetos de negocios, necesarios para la

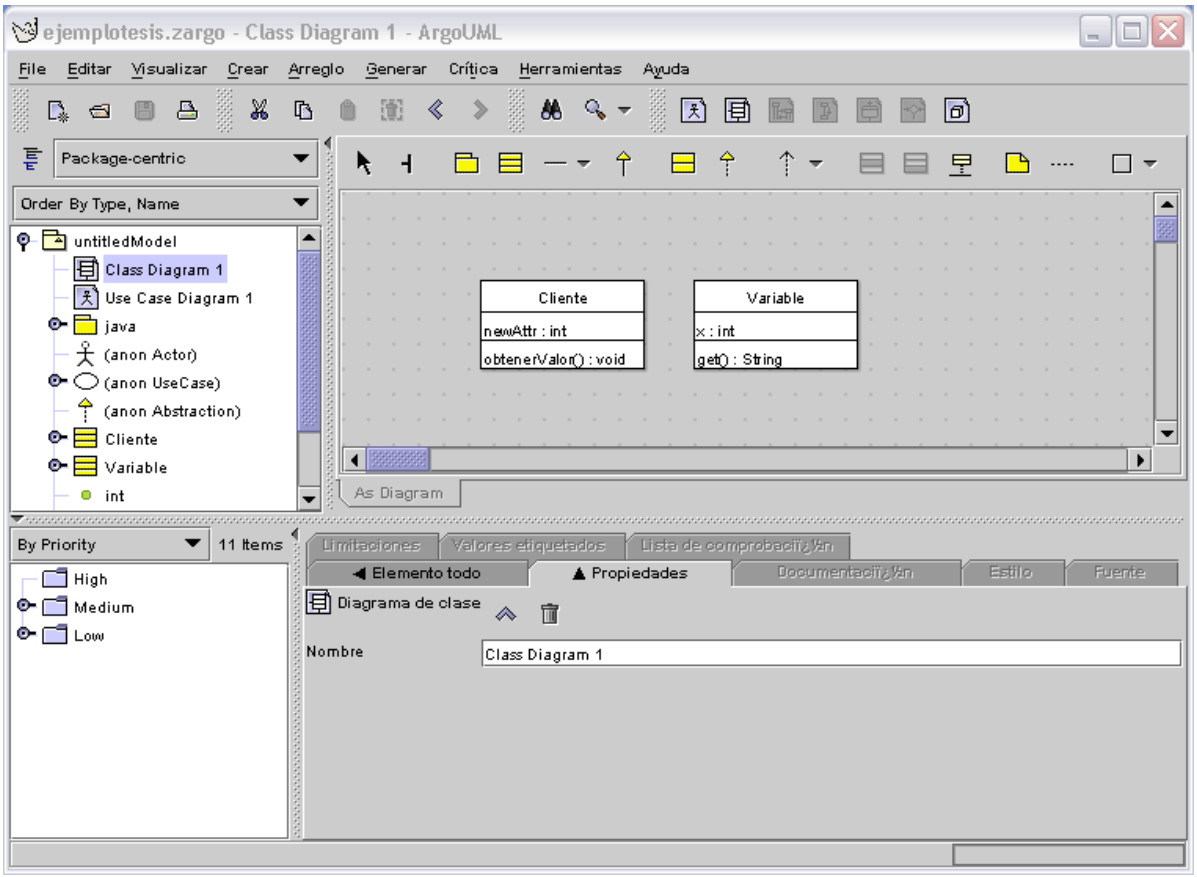


Figura 2.1: ArgoUML

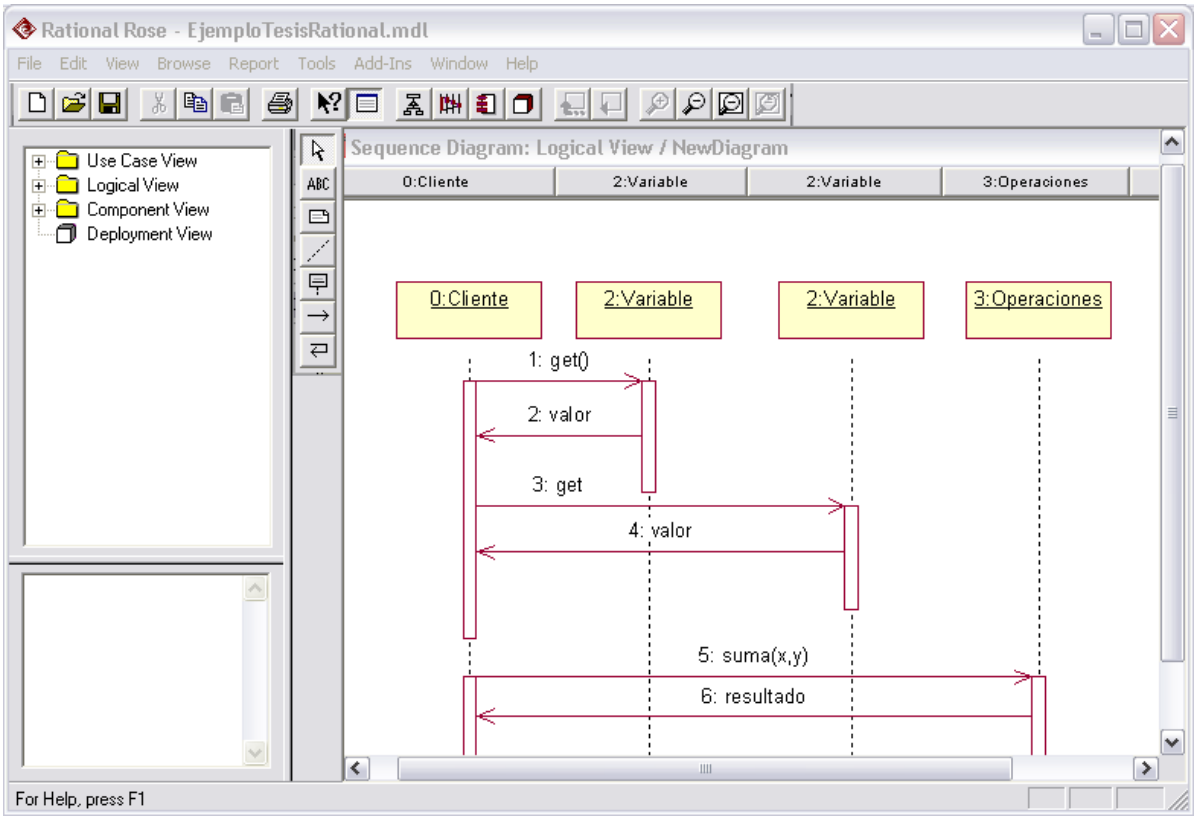


Figura 2.2: Rational Rose

automatización de los procesos de una empresa. Acepta el Unified Modeling Lenguaje (UML 2.0), para modelar componentes de software y sus relaciones.

Genera también el código de los diagramas editados y soporta ingeniería en reversa. En la figura 2.2 se muestra una pantalla de Rational Rose con una de sus vistas al editar un diagrama de secuencia.

2.3.3. Magic Draw

Es una herramienta CASE, que facilita el análisis y diseño de sistemas Orientados a Objetos, soporta UML 2.0, facilita el análisis y diseño de sistemas orientados a objetos y bases de datos. Permite generar código a partir del modelo UML construido, la ingeniería en reversa. Permite además exportar los diagramas elaborados a SVG y otros formatos de imágenes [42]. Esta herramienta esta desarrollada en java, por lo que puede utilizarse en la mayoría de las plataformas existentes. Soporta además todos los diagramas UML y puede exportar también los archivos generados a Rational Rose.

Es una herramienta ampliamente difundida y para su uso se debe de adquirir una licencia ya que es de uso comercial. En la figura 2.3 se muestra una de sus vistas

2.3.4. Unimod

Es un plugin para Eclipse, como puede observarse en la figura 2.4 Unimod se encuentra dentro de este entorno. Implementa la idea de UML ejecutable. Sugiere un método usando Clases y Diagramas de Estado y entonces çorre.^{es}tos diagramas en un click desde Eclipse.

También permite depurar diagramas visualmente: ya que permite colocar puntos de corte en los diagramas de estado, en las transiciones o acciones, corre el diagrama paso por paso, monitorea variables locales. Además permite depurar remotamente los diagramas. Sin embargo a pesar de hacer la ejecución del diagrama, estos resultados no se muestran directamente en el mismo. Los resultados son desplegados en un panel aparte de Eclipse. Es una herramienta de uso libre.

2.3.5. JGrasp

Es un ambiente de desarrollo, creado específica mente para la mejor comprensión del software y permitir la visualización del mismo [43]. Permite editar, compilar, correr y depurar programas en Java. Contienen además un depurador a nivel código fuente para visualizar la ejecución pasado a paso.

Genera diagramas restringidos de clase a partir de un programa en Java y en un panel por separado muestra la información de las clases (atributos y métodos), con base

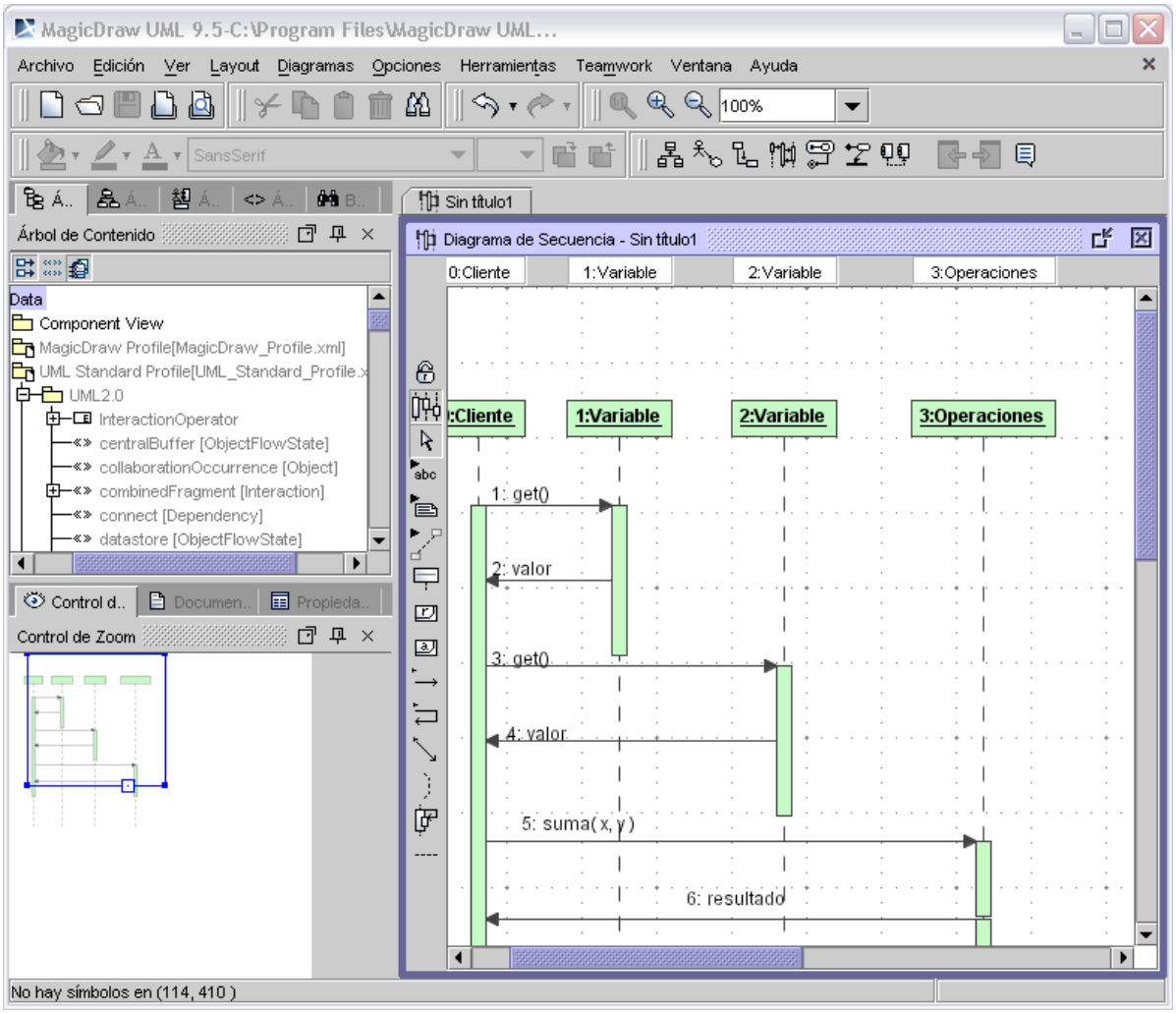


Figura 2.3: MagicDraw

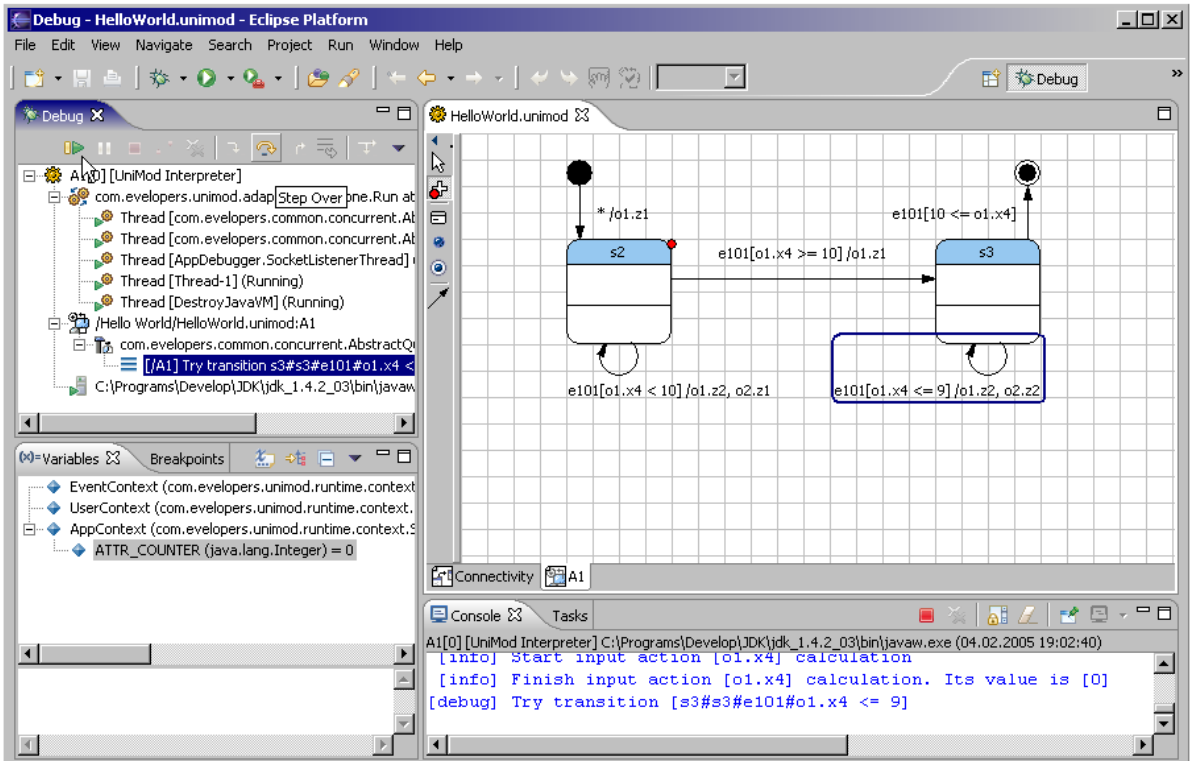


Figura 2.4: Unimod

en dicho diagrama se pueden crear instancias y ejecutar métodos estáticos. A pesar de permitir la ejecución de los programas, esta ejecución se realiza dentro del entorno de programación que tiene integrado, es decir, el diagrama editado no se modifica al correr una aplicación. Es una herramienta de uso libre. En la figura 2.5 se muestra una pantalla de JGrasp.

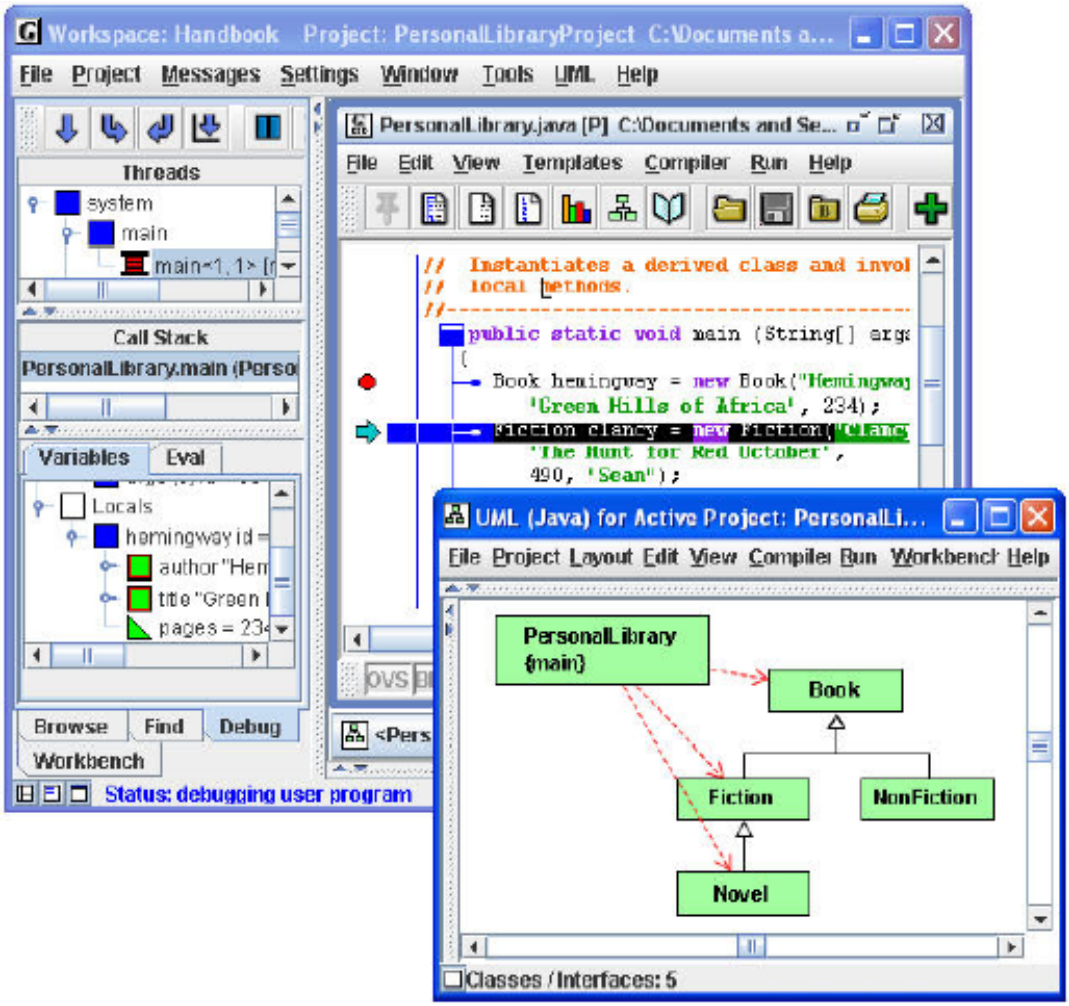


Figura 2.5: JGrasp

2.3.6. ARTiSAN Real-time Studio (RtS)

Es una conjunto de herramientas de modelado enfocados al desarrollo de sistemas técnicos. Utiliza UML 2.0 y SysML que es reconocido como el estándar para extender

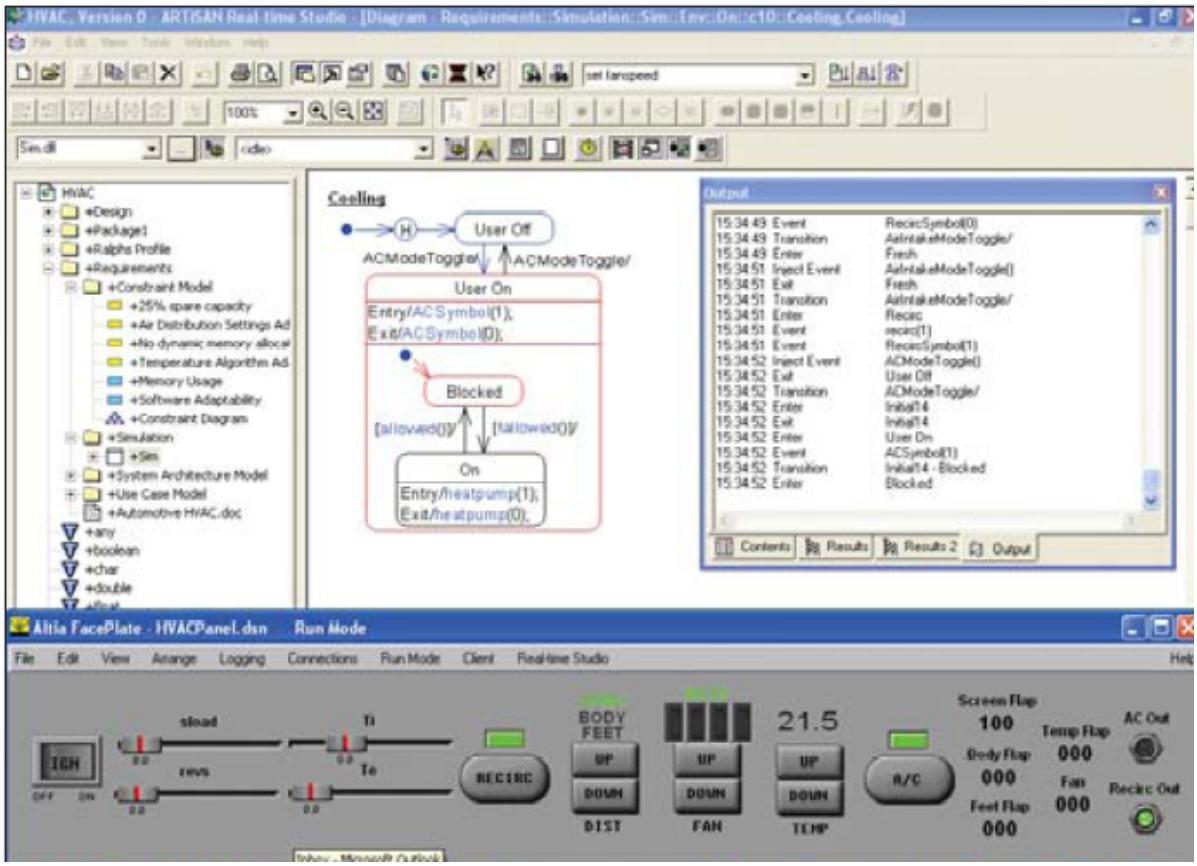


Figura 2.6: ARTiSAN

UML2 en sistemas de ingeniería, RtS es la herramienta escogida para sistemas de larga escala y misión crítica [45].

Artisan cuenta además con un generador de máquina de estados que genera código ejecutable, y las simulaciones pueden mostrarse a través de otra interfaz. Permite también animar diagramas de secuencia. Además de realizar ingeniería en reversa para obtener los diagramas de aplicaciones ya codificadas, y debido a que es una herramienta enfocada a los sistemas de ingeniería cuanta con los módulos para poder intercambiar información con otras aplicaciones científicas como Matlab, con la cual puede comunicarse ampliamente. En la figura 2.5 puede observarse una vista global de estas herramientas

2.4. Herramientas para el diseño de aplicaciones multimedia

En la actualidad existen diferentes herramientas para diseñar aplicaciones que involucran el uso de animaciones de cualquier tipo, estas herramientas proveen de un conjunto de herramientas que agilizan el diseño de entornos multimedia o presentaciones que requieran de alguna animación. Muchas de estas herramientas hacen uso del scripting para implementar las animaciones que se elaboran.

2.4.1. Macromedia Flash

Esta aplicación es una mezcla de un editor de gráficos y un editor de películas, donde se pueden diseñar gráficos vectoriales. Además de esto permite incluir audio, importar otros archivos que no sean vectoriales. También maneja un lenguaje llamado ActionScript para agregar código que permita manipular los diferentes símbolos que se tengan en el escenario y además permite utilizar objetos XML. Debido a que los archivos generados por esta aplicación son muy pequeños, se utilizan en la Web debido a que se descargan en muy poco tiempo y para poder visualizarlas es necesario tener instalado un plugin en el navegador. Una vista del escenario se muestra en la figura 2.7

Una desventaja de esta herramienta es que es de uso propietario, por lo que para utilizarla es necesario adquirir una licencia, otro de sus inconvenientes es que no fue pensada desde un inicio como estándar de la WWW, como es el caso de SVG.

2.5. UML

Desde 1987 se fueron integrando un conjunto de metodologías para llegar a lo que hoy conocemos como UML(Unified Modeling Languaje) [7]. Fueron varios los procesos que se integraron, entre ellos: Process Objectory de Ivar Jacobson, OMT (Object Modelling Technique) de James Rumbaugh, y el Process Objectory de Rational. El proceso unificado es un proceso de desarrollo del software, el cual puede utilizarse para modelar una gran variedad de sistemas de software. El proceso Unificado está basado en componentes, por lo cual los sistemas modelados con UML también están formados por componentes interconectados a través de interfaces bien definidas [27].

Además en el proceso unificado el sistema se describe mediante diferentes vistas para poder tener una imagen completa del sistema a desarrollar. De esta manera se puede hacer una arquitectura que muestre un esquema general con los diferentes factores que pueden influir en ella, tales como la plataforma en la que tiene que funcionar el software, los bloques reutilizables, interfaces gráficas de usuario, etc.

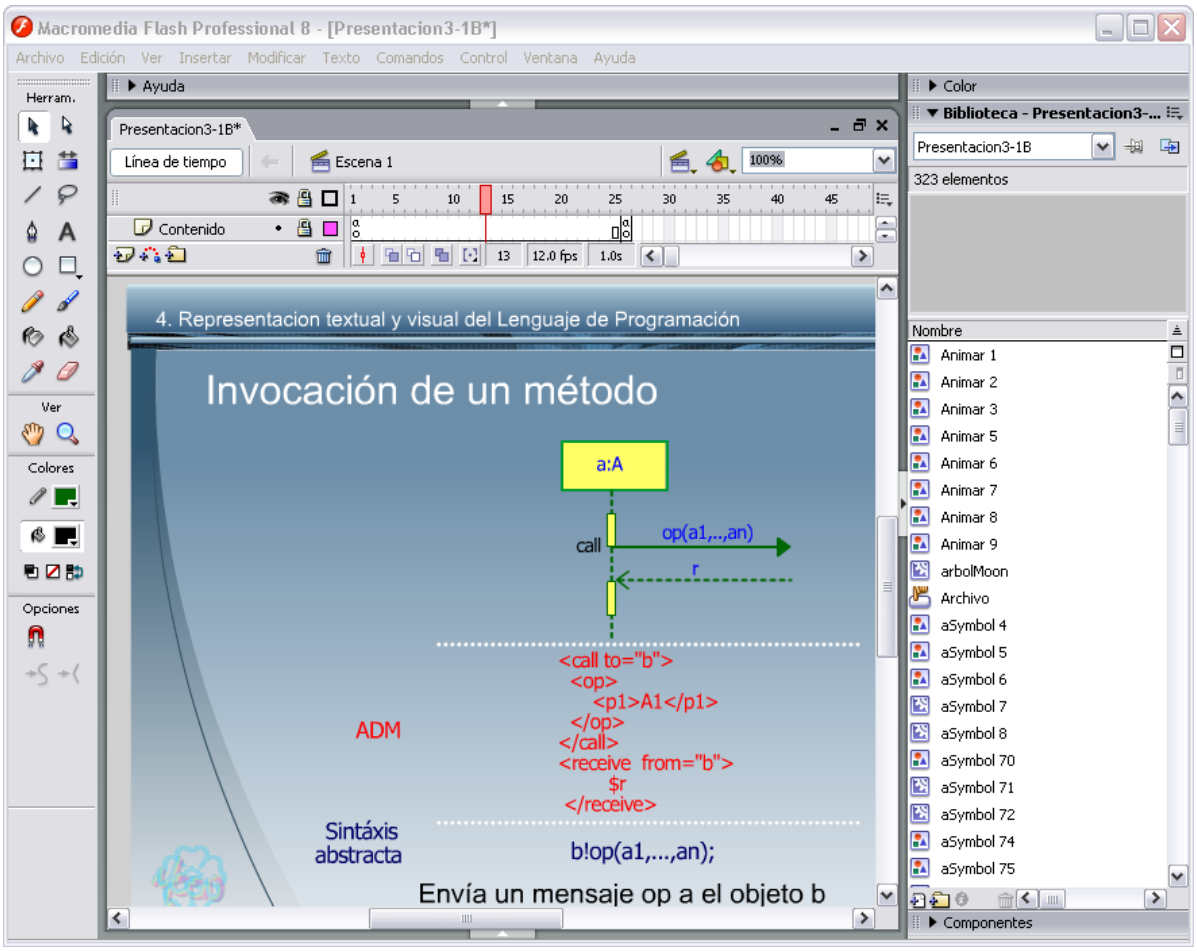


Figura 2.7: Macromedia Flash

En el proceso unificado se dispone de dos tipos diferentes de diagramas: aquellos que dan una vista estática del sistema y aquellos que dan una visión dinámica. Los diagramas estáticos son:

- *Diagrama de clases.* Muestra las clases, interfaces, colaboraciones y sus relaciones.
- *Diagrama de objetos.* Es un diagrama de instancias de las clases mostradas en el diagrama de clases. Muestra las instancias y como se relacionan entre ellas. Se da una visión de casos reales.
- *Diagrama de componentes.* Muestran la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.
- *Diagrama de despliegue.* Muestra los nodos y sus relaciones. Un nodo es un conjunto de componentes. Se utiliza para reducir la complejidad de los diagramas de clases y componentes de un gran sistema. Sirve como resumen e índice.
- *Diagrama de casos de uso.* Muestran los casos de uso, actores y sus relaciones. Muestra quien puede hacer que y relaciones existen entre acciones (casos de uso). Son muy importantes para modelar y organizar el comportamiento del sistema.
- *Diagrama de colaboración.* Es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes.

Los diagramas dinámicos son:

- *Diagrama de secuencia.* Muestra la forma como los objetos de un sistema se comunican entre sí por medio de mensajes. También se le conoce como diagrama de interacción.
- *Diagrama de estados.* muestra los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en sistemas que reaccionen a eventos.
- *Diagrama de actividades.* Es un caso especial del diagrama de estados. Muestra el flujo entre los objetos. Se utilizan para modelar el funcionamiento del sistema y el flujo de control entre objetos.

Desde el punto de vista de diseñadores y programadores el número de diagramas es considerable. En algunos de los casos se puede pensar que son excesivos, por ello UML permite usar solo los requeridos, ya que no todos son necesarios en todos los proyectos.

A continuación se hace una explicación más a detalle del diagrama de secuencia, ya que es parte fundamental de este trabajo.

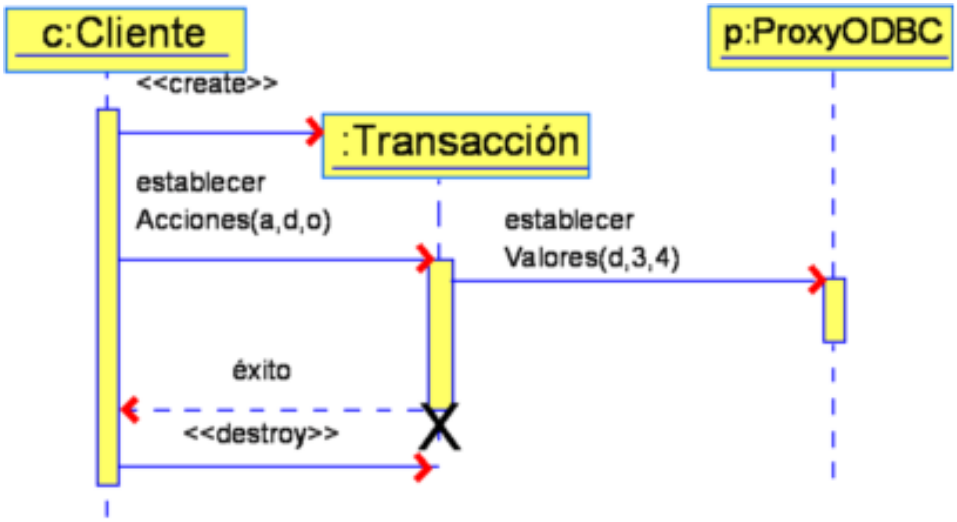


Figura 2.8: Ejemplo de un diagrama de secuencia

2.5.1. Diagrama de Secuencia

Presenta un conjunto de objetos así como de los mensajes enviados y recibidos por ellos. Los objetos suelen ser instancias con nombre. Los diagramas de secuencia [27] se usan para describir la vista dinámica del sistema. En este diagrama, los objetos que participan en la interacción se van colocando horizontalmente en la parte superior y generalmente se coloca a la izquierda el objeto que inicia la interacción y los objetos subordinados a la derecha. Cada objeto se caracteriza por su línea de tiempo que indica el orden en el que tienen lugar las interacciones. Los mensajes se muestran horizontalmente entre los objetos que envían y reciben, ordenados de acuerdo al tiempo. En la figura 2.8, se muestra un ejemplo de un diagrama de secuencia.

En este diagrama los objetos tienen una línea de vida, que es una línea discontinua vertical que representa la existencia de un objeto a lo largo de un período de tiempo y cuando el objeto se destruye se marca al final con una X.

2.6. XML

XML (*Extensible Markup Language*) está basado en SGML (Standard Generalized Markup Language, ISO 8879), que data de 1986 el cual a su vez está basado en el GML creado por IBM en 1969. DOM (Document Object Model), esta asociado a la recomendación del W3C XML, el cual es un modelo de objetos (en forma de API) que permite acceder a las diferentes partes que pueden componer un documento XML o

```
<?xml version="1.0" encoding="UTF-8"?>
<tesis>
   <titulo> Entorno de Programación Visual de reglas ECA</titulo>
   <fecha> Enero, 2005</fecha>
   <autor>
      <nombre>Mónica</nombre>
      <apellidopaterno>Rivera</apellidopaterno>
       <apellidomaterno>de la rosa</apellidomaterno>
   </autor>
</tesis>
```

Figura 2.9: Ejemplo de código en XML

HTML [31].

XML fue desarrollado por un Grupo de Trabajo de XML (originalmente conocido como el comité de revisión editorial de SGML) formado bajo el auspicio del World Wide Web Consortium (W3C) en 1996. XML es un metalenguaje, es decir, es un lenguaje para definir lenguajes. Los elementos que lo componen pueden dar información sobre lo que contienen, no necesariamente sobre su estructura física o presentación, como ocurre en HTML. La diferencia fundamental entre HTML y XML es que HTML es simplemente un lenguaje, mientras que XML como se mencionó anteriormente es un metalenguaje. El lenguaje XML describe una clase de objetos de datos llamados documentos XML y describe parcialmente el comportamiento de programas que pueden procesarlos. Cada documento XML tiene una estructura tanto lógica como física. Físicamente, el documento está compuesto de unidades llamadas entidades. Una entidad puede referirse a otras entidades con el fin de causar su inclusión en el documento. Lógicamente, el documento está compuesto de declaraciones, elementos, comentarios, referencias de caracter e instrucciones de proceso. Todo lo anterior está indicado en el documento mediante marcas explícitas. Las estructuras lógicas y físicas deben anidarse apropiadamente, un ejemplo de ello se puede observar en la figura 2.9.

2.7. SVG

A pesar de que el formato vectorial de flash es ampliamente utilizado para diseño de las páginas Web, no esta especialmente diseñado para la Web, ya que es una aplicación externa a HTML. En 1998 el consorcio WWW formó un grupo de trabajo para desarrollar una representación de gráficos vectoriales como aplicación XML, de esta manera surgió SVG (Scalable Vector Graphics) [29]. SVG es un dialecto de XML, la información acerca de la imagen es almacenada en un texto plano y tiene las ventajas de accesibili-


```
<svg width="140" height="170">
<title>Imagen</title>
<desc>Carita</desc>

<circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
<circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
<circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
<polyline points="35 110, 45 120, 95 120, 105, 110" style="stroke: black; fill: none;" />
<text x="60" y="165" style="font-family: sans-serif; font-size: 14pt; stroke: none; fill: black;">
   Hola
 </text>
</svg>
```

Figura 2.10: Ejemplo del código SVG

dad, portabilidad e interoperabilidad de XML. Algunas de las ventajas que muestra son las siguientes:

- *Escalabilidad:* al ser un formato vectorial los gráficos SVG pueden agrandarse y minimizarse libremente sin perder definición lo cual no ocurre con formatos bitmap en donde un zoom degrada la calidad de la imagen.
- *Inclusion de texto:* Un archivo SVG puede incluir texto de forma tal que el mismo sea buscable y seleccionable desde aplicaciones lo cual no ocurre en otros formatos gráficos.
- *Scripting y animación:* SVG incluye posibilidad de definir animaciones y acciones sobre los gráficos de forma tan poderosa como un archivo flash o más. Una de las ventajas de SVG es que se le puede dar el dinamismo y la animación basándose en la especificación DOM, a través de lenguajes script como Java Script.

Al ser una especificación del consorcio WWW, SVG es un estándar abierto que no es propiedad de ninguna empresa. En la figura 2.10 se muestra un ejemplo del código SVG y la imagen resultante.

Capítulo 3

Entorno de Programación Visual `Moon`

Con el propósito de construir un entorno de programación para una visualización directa se diseñaron 5 módulos principales en el entorno: el editor de diagramas de secuencia, el compilador frontal, el compilador posterior, el mediador en tiempo de ejecución y el visualizador. Tanto el primero como el último de éstos módulos muestran la parte gráfica del entorno y es el lugar donde el usuario interactúa y elabora o visualiza la ejecución del programa según sea el caso.

En este capítulo se da una descripción de la funcionalidad de estos módulos. En la sección 3.1 se muestra la arquitectura general del entorno y la relación entre los módulos que lo conforman. En la sección 3.2 se describen los elementos gráficos que conforman al editor. Por otra parte en la sección 3.3 se da una descripción del compilador frontal y el la sección 3.4 se describen las tareas que realiza el compilador posterior. En la sección 3.5 se detalla la función del mediador, finalmente en la sección 3.6 se describen los elementos gráficos del visualizador.

3.1. Componentes del entorno

Como se ha mencionado,`Moon` consiste de 5 componentes los cuales se describen brevemente a continuación:

1. *Editor de diagramas de secuencia*, diseñado para verificar que el diagrama UML este bien formado, produciendo en tal caso metadatos acerca del modelo. Está basado en la plataforma de los navegadores de Internet y habilitado por los plug-ins de Adobe SVG por medio de scripting.

2. *El compilador frontal*, fue diseñado para traducir los diagramas UML en reglas ADM. Con los metadatos producidos por el editor, el compilador frontal utiliza los esquemas de traducción para transformar la notación gráfica y textual del diagrama de secuencia UML en las construcciones del lenguaje de reglas ADM.

3. *El compilador posterior*, fue diseñado para traducir las reglas ADM en programas Java. Las reglas ADM son transformadas en programas Java ejecutables de acuerdo con el propósito del modelo.

4. *Mediador en tiempo de ejecución*, diseñado con el propósito de coordinar la ejecución del programa con el visualizador de diagramas.

5. *El visualizador del programa*, propuesto para desplegar la ejecución de los diagramas UML. El visualizador recibe los datos de entrada y los transforma en elementos gráficos de SVG para ser mostrados en el navegador.

Las relaciones entre los componentes se muestran en la Figura 3.1, donde se puede observar que, se tienen dos fases principales: la fase de *Edición-Compilación* y la fase de *Visualización*.

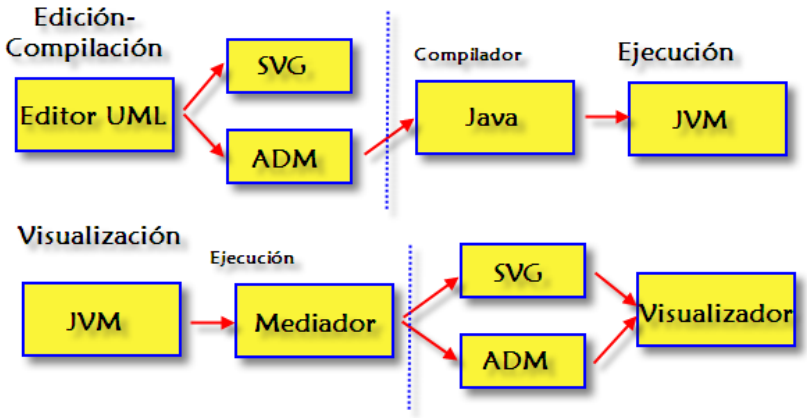


Figura 3.1: Arquitectura del entorno de programación Moon

Como primer paso en la fase de edición se realiza la edición del diagrama de secuencia, el cual se representa internamente con los elementos SVG necesarios y se compila hacia reglas ADM para que posteriormente dichas reglas pasen al *compilador* que se encargará de generar el programa Java que corresponde con el modelo. Una vez hecho lo anterior se pasa a la fase de *visualización* donde el resultado de la ejecución del programa pasa al *mediador* donde se guarda el intercambio de mensajes junto con sus contenidos y

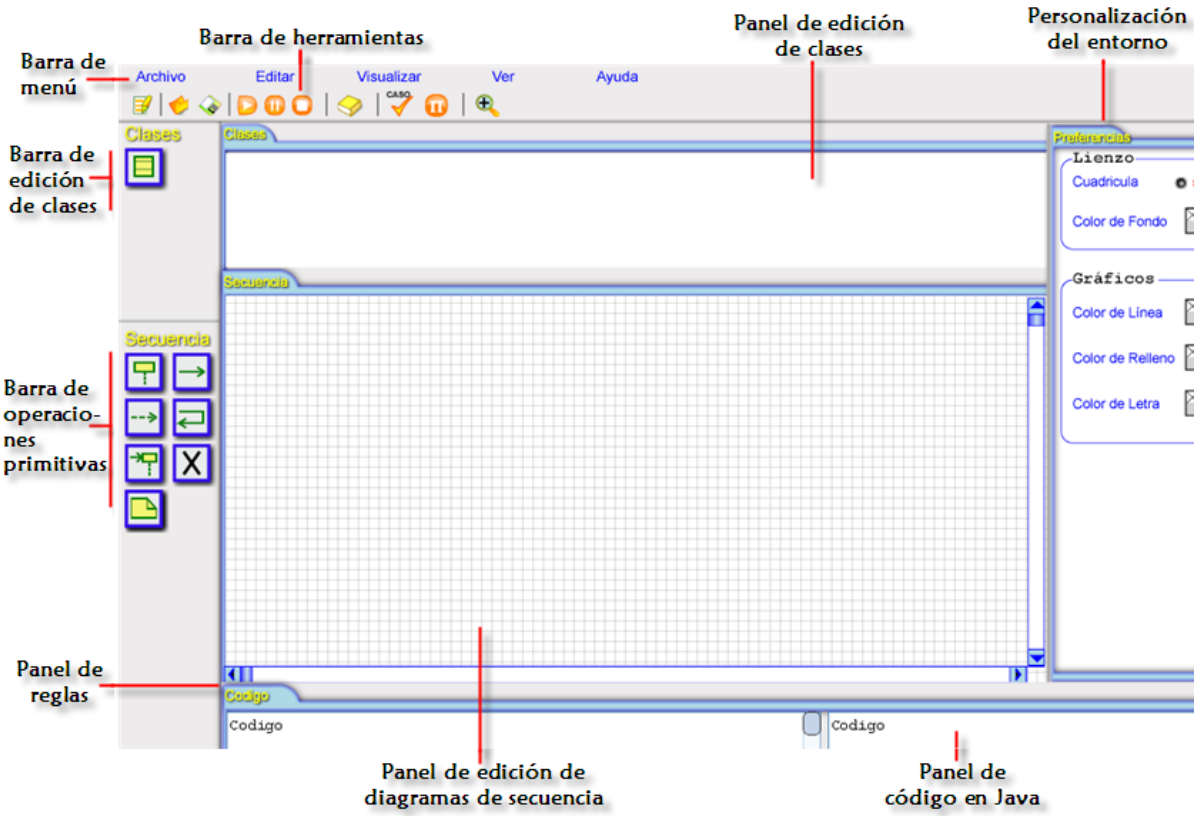


Figura 3.2: Entorno de programación `Moon`

de ahí a su representación en SVG para finalmente obtener la visualización del programa en el diagrama de secuencia.

En la figura 3.2, se observan los componentes principales del entorno Moon y a continuación se describe cada uno de ellos:

1. *Panel de botones de acción*, dentro de este panel se encuentran los botones de operaciones primitivas, con los cuales se puede realizar la edición de los diagramas de secuencia. Como se muestra en la Figura 3.3 cada uno de estos botones tiene asignado el elemento gráfico que corresponde a alguna de las operaciones primitivas que se muestra en el panel de dibujo del diagrama de secuencia. También dentro de este panel se tiene la barra de edición de clases que tiene el botón para agregar las clases que se utilizaran en el diagrama de secuencia en el panel de diagrama de clases.

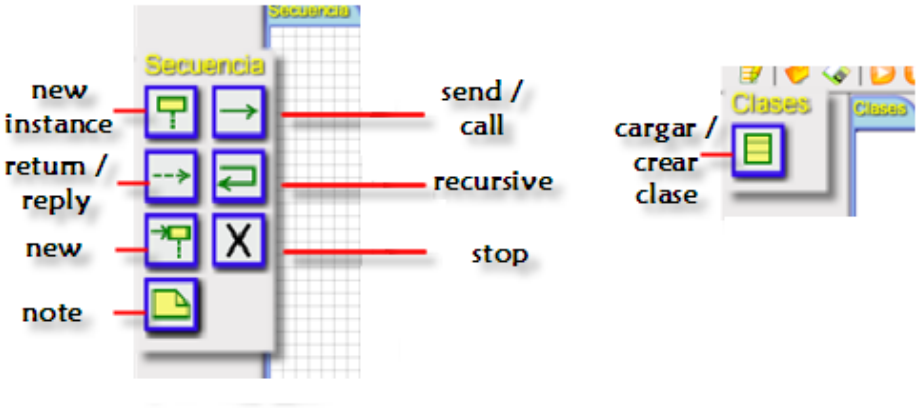


Figura 3.3: Botones de operaciones primitivas

2. *Panel de edición del diagrama de secuencia*, es el área donde se elabora el diagrama de secuencia del programa que se este modelando.

3. *Panel de edición del diagrama de clases* en esta parte se colocan las clases de donde se crearan las instancias para el diagrama de secuencia.

4. *Panel de generación de código ADM*, en este panel se muestran las reglas ADM generadas a partir del diagrama secuencia editado en el panel de edición de diagramas y después de realizar la compilación correspondiente.

5. *Panel de generación de código en Java*, donde se muestra el programa generado a partir del diagrama de secuencia editado y compilado.

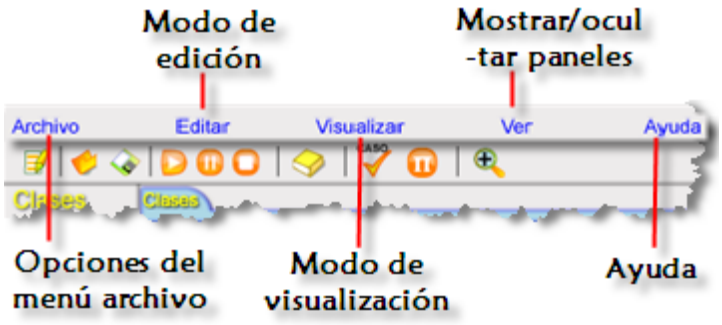


Figura 3.4: Barra de menú

Adicionalmente se tiene dentro del entorno una barra de menú, una barra de herramientas y opciones de preferencias que permiten al usuario personalizar algunos elementos del entorno (cuadrícula y colores), los cuales pueden ser ajustados por el usuario.

La barra de menú, contiene cinco opciones a escoger tal como se observa en la Figura 3.4, las cuales tienen las acciones que pueden realizarse en el entorno.

Los componentes de esta barra de menú son:

- *Archivo*, en esta parte se encuentran las opciones para crear un nuevo diagrama, cargar un diagrama prediseñado, guardar el diagrama actual y la opción para cerrar el navegador y salir del entorno.
- Modo Edición, esta opción si estamos en modo de edición podemos cambiar al modo de visualización para observar la ejecución del programa y si queremos volver al modo de edición basta con presionar nuevamente la misma opción.
- Modo Visualización, con esta opción si estamos en modo de edición podemos cambiar al modo de visualización del diagrama compilado y observar su visualización.
- Ayuda, en esta parte se muestra una pequeña ayuda del uso del editor y también la ventana Acerca de con información general del entorno.
- *Ver* paneles, con esta opción pueden mostrarse u ocultarse los paneles de Código y Preferencias del entorno

Siguiendo con la descripción de los componentes se puede observar la Barra de Herramientas que se muestra en la Figura 3.5, en ella se encuentran de manera accesible algunas de las acciones que se encuentran en el menú Archivo y también se representan acciones nuevas que pueden realizarse dentro del entorno.

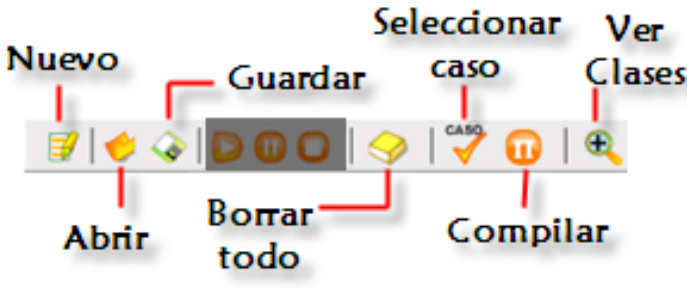


Figura 3.5: Barra de Herramientas

Los botones y las acciones que realiza cada uno de éstos botones de menú son las siguientes:

- *Nuevo*, se encarga de inicializar el ambiente para crear un nuevo diagrama (como la opción Nuevo del menú Archivo).
- *Guardar*, almacena los diagramas editados.
- *Abrir*, abre un diagrama previamente editado.
- *Borrar*, elimina del panel del diagrama de secuencia todos los elementos que se encuentren en el.
- *Caso*, se utiliza para seleccionar del diagrama de secuencia él o los objetos que se utilizaran para generar la Reglas ADM y el código en Java.
- *Compilar*, se generan las Reglas ADM y el código en Java del diagrama de secuencia editado.
- *Vista de Clase*, cambia la vista de diseño de diagramas a la vista de clases, es decir podemos observar las clases participantes con los atributos y los métodos que las conforman.

3.2. Editor de diagramas de secuencia

El editor gráfico, está escrito en JavaScript, a través del cual se transforman los elementos del diagrama de secuencia UML a elementos SVG. Siendo SVG un dialecto de XML, la representación del diagrama se encuentra dentro de las herramientas estándar disponibles (SAX o basadas en el DOM) para almacenarlo de manera estable o transferirlo a través de la red. En particular. Una API del DOM es usada para compilar

los diagramas a las reglas ADM por medio de transformaciones XML. En la Figura 3.2 donde se observa el entorno completo, se muestra el panel de edición y la barra de operaciones primitivas con los elementos que se utilizan al elaborar los diagramas de secuencia, una vista más detallada de estos elementos se puede observar en la Figura 3.3, se encuentran los botones para la creación de los elementos gráficos que representan las siguientes operaciones primitivas:

- *new Instance*, crea un nuevo participante de la clase que se requiera, estos participantes se crean de manera independiente, es decir, su existencia no depende de un participante creado previamente.
- *send/call*, envío de mensajes que requieren respuesta inmediata(*call*) o que no requieren respuesta(*send*).
- *return/reply*, regresar la respuesta al invocador.
- *new*, creación de un nuevo participante.
- *stop*, destrucción de un participante.
- *recursive*, invocación a un método contenido en los métodos del mismo participante.
- *note*, crear una condición o una asignación de variables.

3.3. Compilador frontal

Como se mencionó en la sección 3.1 el *compilador frontal* es el encargado de traducir la representación visual del diagrama de secuencia a su correspondiente representación textual en Reglas ADM. Una vez editado el diagrama de secuencia, y seleccionados los objetos a partir de los cuales se generará el código, con el botón *Caso* de la barra de herramientas se presiona el botón de compilación de la barra de herramientas. El resultado de ésta compilación se muestra en el panel de código del Editor como se muestra en la Figura 3.6.

3.4. Compilador posterior

Además de el código ADM, también se genera el código en Java, como se muestra en la Figura 3.7, una vez que se obtienen las reglas ADM generadas por el *compilador frontal* se pasan posteriormente al *compilador posterior* que se encarga de traducir las reglas a programas Java. Internamente se crean dos códigos java: el primero es el que el

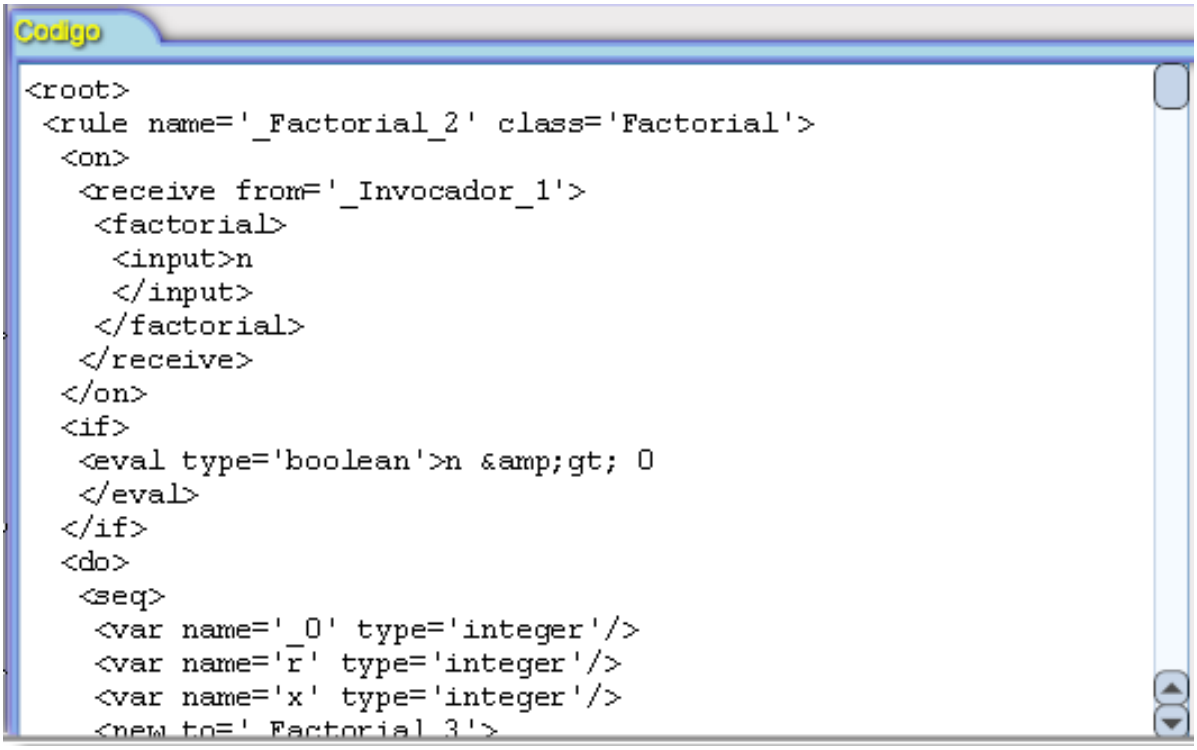


Figura 3.6: Reglas ADM generadas por el compilador

usuario observa en el *panel de código* con las instrucciones en Java normales, el segundo código Java generado no se muestra dentro del entorno, pero es el código necesario para dar paso a la visualización, ya que este código utiliza el paquete de reflexión de Java y que de esta manera el mediador genere los metadatos correspondientes para visualizar la ejecución del programa, una muestra de éste código generado podemos observalor en la Figura 3.8. En ambos casos se crean los archivos que contienen dichos códigos y pueden ser compilador a través del compilador de Java.

3.5. Mediador

Una vez que se ha generado el programa en Java, el mediador es el encargado de hacer, la conversión a los metadatos necesarios que se utilizaran para realizar la representación visual en SVG.

El mediador consiste de una mínima infraestructura de coordinación y comunicación entre la capa del visualizador y la maquina virtual de java que ejecuta el programa. Durante la ejecución del programa el mediador sincroniza ambas capas para intercam-

```
public class Factorial
 {
 public int factorial(int n)
  {
  if (n>0)
   {
   int _0;
   int r;
   int x;
   Factorial _Factorial_3 = new Factorial(null);
   _0 = n-1;
   r = _Factorial_3.factorial(_0);
   x = r*n;
   return x;
   }
  else
  if (n==0)
   {
   return 1;
```

Figura 3.7: Código Java generado por el compilador

```
public String invocaciones(){
  String val1=Mediador.methodExecute(1,"call","get","null");
  String val2=Mediador.methodExecute(2,"call","get","null");
  String res= Mediador.methodExecute(3,"call","suma",val1+","+val2);
  String val3=Mediador.methodExecute(4,"send","set",res);
 }
}
```

Figura 3.8: Código generado por el compilador

biar mensajes que contienen los datos producidos por la ejecución del programa y la interacción del usuario recibida a través del visualizador.

3.6. Visualizador Integrado

El visualizador es el encargado de mostrar de una manera gráfica el programa obtenido a partir del diagrama de secuencia inicial. Para poder hacer esta visualización hay que estar en modo de Visualización utilizando el botón de la barra de Menú una vez hecho el pasamos de la ventana del editor a la ventana del visualizador. La apariencia del visualizador se muestra en la Figura 3.10

En este modo de Visualización la barra de herramientas que tenemos tiene tres

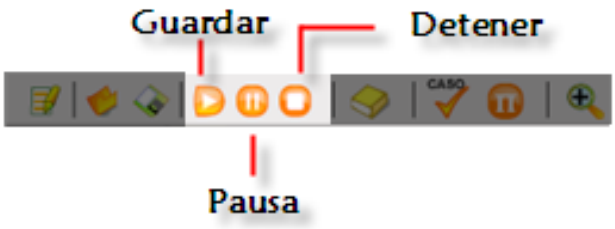


Figura 3.9: Barra de herramientas del Visualizador

elementos que ayudan a controlar la ejecución del diagrama, como se observa en la Figura 3.9.

- El botón *Reproducir* se encarga de iniciar la visualización del diagrama de secuencia editado.
- El botón *Pausa* se encarga de detener la visualización del diagrama con la opción de poderla continuar nuevamente. Para continuar con la reproducción se presiona nuevamente el botón de Reproducir.
- El botón *Detener*, detiene la visualización del diagrama sin que esta se pueda reanudar posteriormente.

3.7. Visualizador

El visualizador también puede mostrarse de manera independiente del entorno, para obtener únicamente la visualización de algún programa generado previamente ya sea mediante la edición de su diagrama de secuencia con el mismo entorno `Moon` o realizar el archivo con las reglas necesarias de manera independiente, siguiendo solamente la sintaxis para que el visualizador lo pueda interpretar y muestre su ejecución.

La pantalla que se muestra para el visualizador independiente es el que se muestra en la Figura 3.11

Los botones que conforman la barra de herramienta son esencialmente los mismos que se utilizan en el visualizador integrado y se muestran en la Figura 3.9

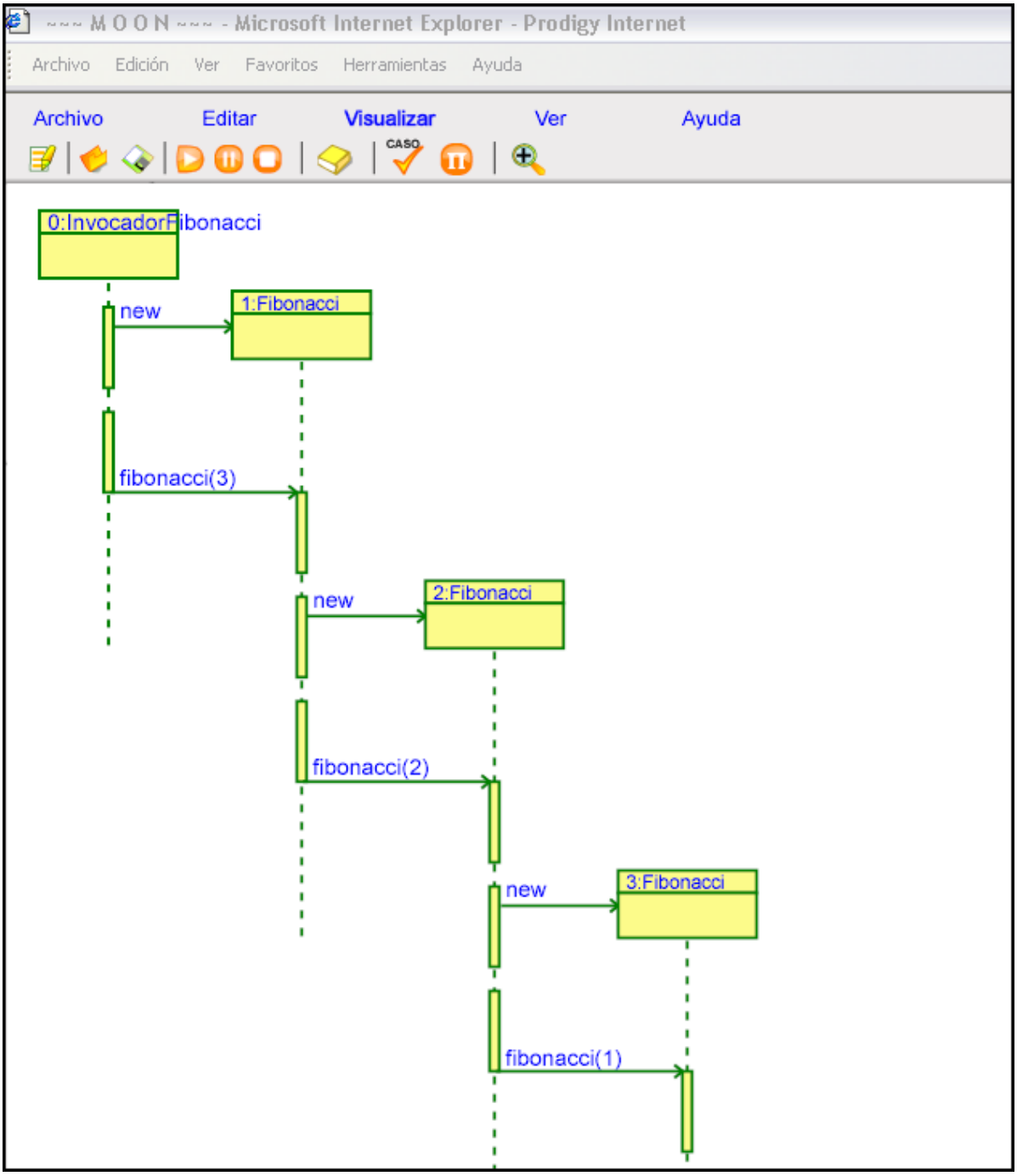


Figura 3.10: Visualizador

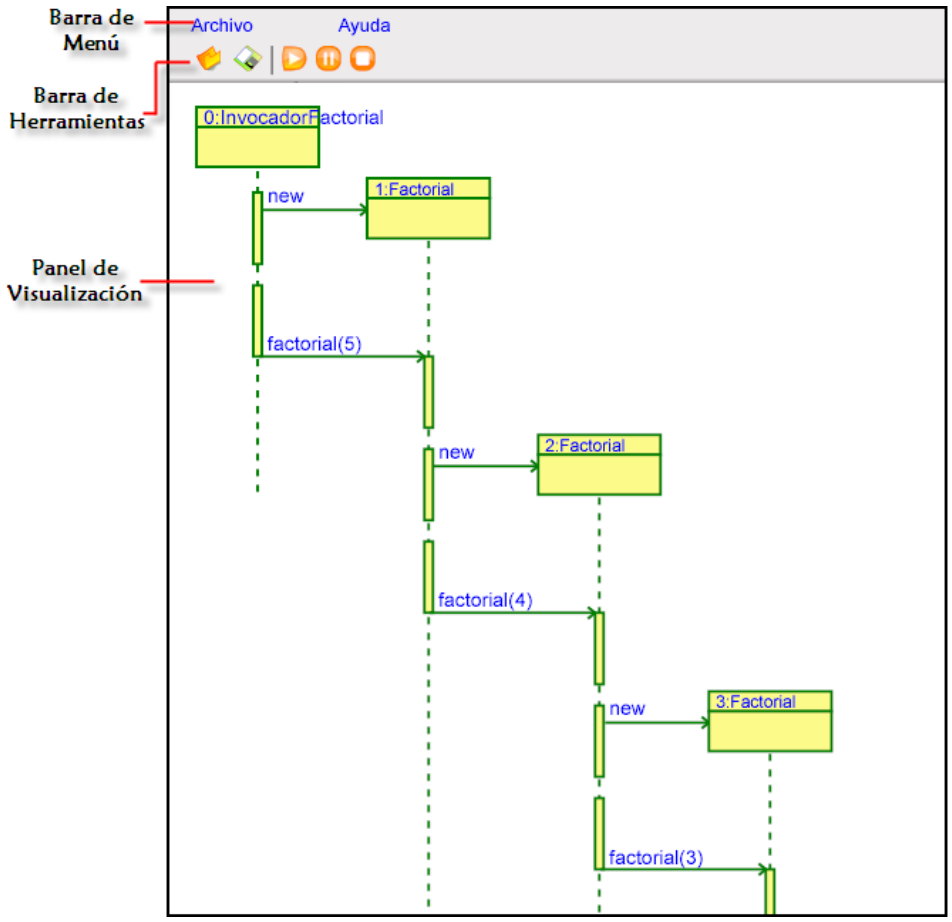


Figura 3.11: Visualizador independiente del entorno Moon

Capítulo 4

Representación visual y textual del modelo de programación

Para simplificar la traducción y la comprensión de las reglas ADM generadas por los diagramas de secuencia, se hace necesaria una representación textual del modelo de programación, para ello se necesita asociar a cada uno de los fragmentos de diagramas que representan a una regla en ADM, a su correspondiente representación textual. Este esquema es presentado en la sección 4.1, posteriormente en la sección 4.2 se muestran tres casos donde se observa la funcionalidad del entorno de programación visual.

4.1. Modelo

Los diagramas de secuencia describen el comportamiento de un sistema desarrollado por la interacción de sus participantes. Estos diagramas contienen objetos (clases) que intercambian mensajes organizados en un instante de la secuencia. Estos están conformados por los siguientes elementos [15]:

1. Clase, denotada por rectángulos dentro de los cuales están el nombre del rol y el nombre de la clase especificando el tipo de objetos que pueden participar con las interacciones y colaboraciones.

2. Líneas de tiempo, denotadas por líneas verticales punteadas, representan la existencia de los roles de clases por un periodo de tiempo, desde que el objeto es creado hasta que es destruido. Estas pueden bifurcarse en dos o más líneas de tiempo paralelas para mostrar la concurrencia o condición de cada línea de tiempo correspondiente a un hilo o a una bifurcación condicional y pueden fusionarse en algún punto subsiguiente.

3. Actividades, denotadas por rectángulos verticales delgados colocados a lo largo de la línea de tiempo, representan el tiempo durante el cual un rol de clase esta ejecutando una acción (operación) o cuando esta activo y tiene el foco de control.

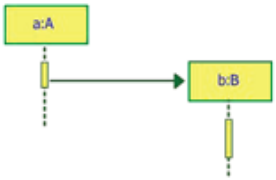
4. Mensajes, denotados por flechas horizontales etiquetadas entre las líneas de tiempo, definen la información contenida en los mensajes que son intercambiados en las interacciones y colaboraciones.

Un modelo de programación concurrente da significado a los elementos gráficos de un diagrama UML de secuencia mediante el concepto de proceso. Un proceso representa hilos de control independientes, cada uno de los cuales ejecuta código secuencial [33]. Un proceso es un objeto activo que además de seguir su propio comportamiento (ejecutar su código), puede proveer servicios a otros objetos aceptando llamadas a los métodos que el objeto activo ofrece. Los procesos pueden ser creados dinámicamente y pueden comunicarse con cada los demás por medio de una variedad de mecanismos: paso de mensajes, llamadas a métodos locales y remotos y puntos de encuentro.

En la representación textual del lenguaje de programación, un proceso es descrito por la sintaxis abstracta que se muestra en la Tabla 4.1 , la cual esta conformada por tres columnas donde se asocia un fragmento de diagrama UML con su correspondiente regla ADM y expresión del cálculo-Π

Tabla 4.1: Elementos textuales y gráficos del lenguaje

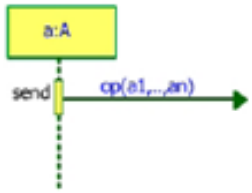
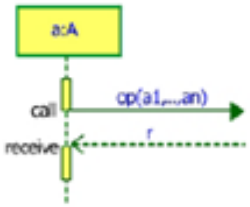
Diagrama UML	ADM	Java

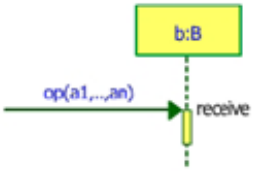
	`<new name="b"`
	`class="B">`
	`<p1>a1</p1>`
	`...`
	`</new>`
		B b = new B(a1,...);
		(crea e inicializa un objeto b de la clase B)

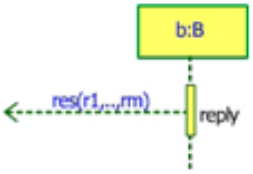
	`<stop/>`	stop (termina la actividad del objeto)

Continua en la siguiente página...

Tabla 4.1 – Continuación

Diagrama UML	ADM	Java
	<send to="b"> <op> <p1>a1</p1> ... </op> </send>	b.op(a1,...,an); (envía el mensaje op al objeto b)
	<seq> <call to="b"> <op> <p1>a1</p1>... </op> </call> <receive from="b"> $r </receive> </seq>	r = b.op(a1,...,an); (envía el mensaje op al objeto b, y espera la respuesta r)

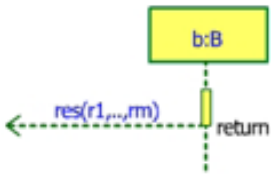
	<receive from="a">
	<op>
	<p1>a1</p1>...
	</op>
	</receive>
		public op(a1,...,an){
		...
		}
		(recibe el mensaje op del objeto b)

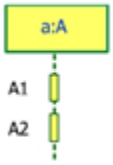
	<reply to="a"> <result> <p1> r1 </p1>... </result> </reply>	No implementada (envía una respuesta de regreso al invocador y continua con la ejecución)

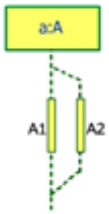
Continua en la siguiente página...

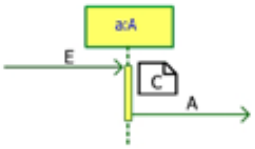
Tabla 4.1 – Continuación

Diagrama UML	ADM	Java

	`<return to="a">`
	`<result>`
	`<p1> r1 </p1>...`
	`</result>`
	`</return>`
		return result(r1,...);
		(envía una respuesta de regreso al invocador antes de que el proceso termine normalmente)

	`<seq>`
	`A1`
	`A2`
	`</seq>`
		A1 ; A2
		(ejecuta A1 y a su terminación ejecuta A2)

	`<par> A1 A2 </par>`
	
	`<alt> A1 A2 </alt>`
		No implementada
		(ejecuta A1 simultáneamente con A2) A1 + A2
		(ejecuta o bien A1 o bien A2)

	`<rule>`
	`<on>`
	`<receive>`
	`E`
	`</receive>`
	`</on>`
	`<if> C </if>`
	`<do> A </do>`
	`</rule>`
		public E(){
		if(C)A
		}
		(después de recibir el mensaje E, verifica si su contenido satisface la condición C y si lo hace, ejecuta la acción A; de lo contrario aborta)

Las reglas ECA generadas desde el editor son compiladas en programas Java. El código es generado de acuerdo al el siguiente modelo de programación que se muestra en

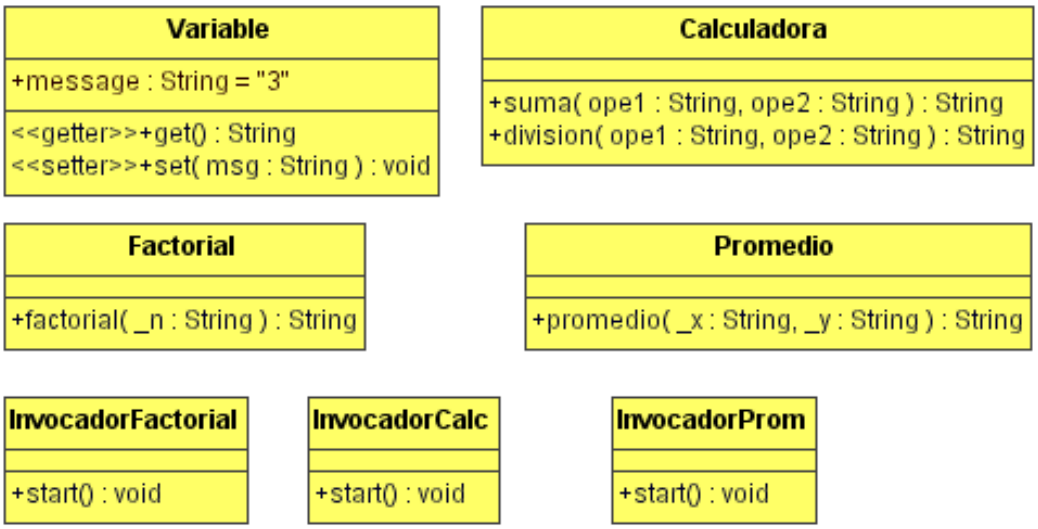


Figura 4.1: Clases de los participantes en los casos de estudio

el siguiente capítulo el cual describe de manera precisa y sin ambigüedades, el significado de cada concepto de programación.

4.2. Casos de Estudio

A continuación se describen tres casos de estudio que se diseñaron en el entorno `Moon`, se muestra la parte del diseño y la visualización de los mismos dentro del entorno. Estos tres casos hacen uso de las clases mostradas en la Figura 4.1: *InvocadorCalc*, *Variable*, *Calculadora*, *Factorial*, *InvocadorFactorial*y *Promedio*.

Las clases *InvocadorFactorial*, *InvocadorCalc*,*InvocadorProm* contienen únicamente el método *start()*, que contiene las invocaciones a los diferentes métodos de los participantes según sea el caso. La clase *Variable* contiene dos métodos: el método *get()*, el cual regresa el valor de la variable y el método *set(String msg)* el cual pone un valor a la variable. La clase *Calculadora*, que contiene dos métodos para hacer operaciones aritméticas de *suma(String ope1,String ope2)* y *división(String ope1,String ope2)* de acuerdo a los parámetros que se le pasen. La clase *Promedio* que contiene un método para calcular un promedio entre dos número *promedio(String x, String y)* y finalmente la clase *Factorial* que cuenta con un método: *factorial(String x)* para calcular el factorial de un número.

4.2.1. Invocaciones Simples

En la siguiente sección, describimos dos formas de representación usadas en el entorno junto con los esquemas de traducción asociados a ellos.

El primero de los casos de estudio muestra invocaciones simples entre los diferentes participantes. El diagrama de secuencia consta de 5 participantes: una instancia de la clase *InvocadorCalc*, que se encarga de hacer el envío de mensajes y de recibir la respuesta de los diferentes participantes; 3 instancias de la clase *Variable* y una instancia de la clase *Calculadora*, específicamente en este caso se va a realizar la suma de dos variables y el resultado se va a almacenar en otra variable.

La edición gráfica inicia colocando las clases de todos los participantes en el *panel del diagrama de clases*, mientras que sus instancias son colocadas en el *panel del diagrama de secuencia*, lo cual se muestra en la Figura 4.2.

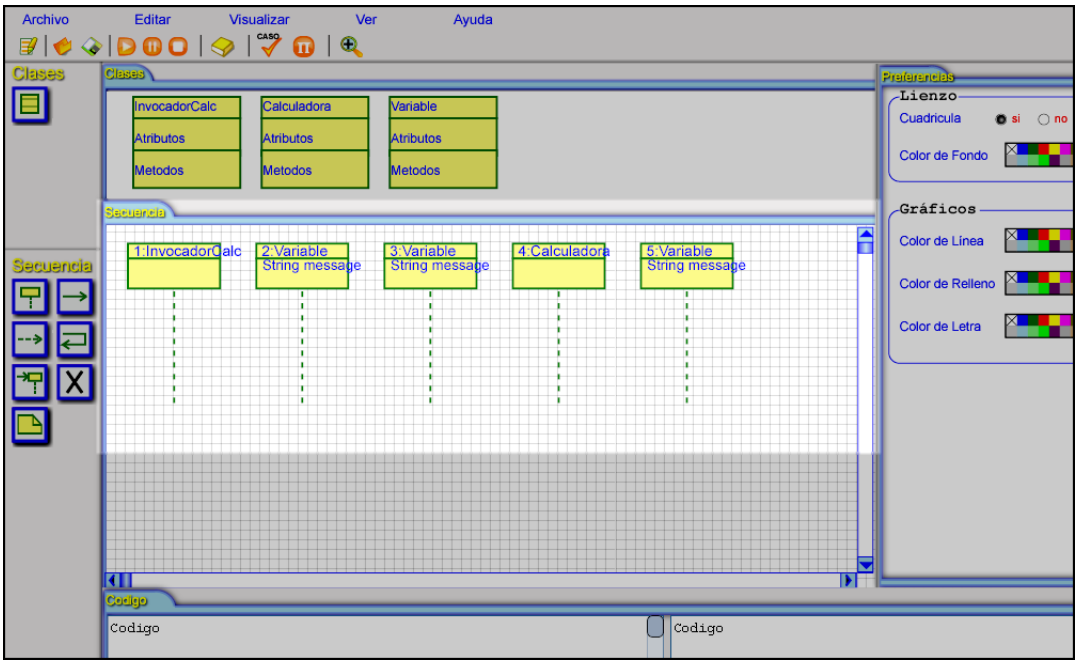


Figura 4.2: Carga de clases y creación de participantes

Una vez colocados los participantes, entonces la interacción y colaboración de los participantes (envío y recepción de mensajes) se edita de acuerdo a los roles y el comportamiento deseado del sistema, haciendo uso de la barra de herramientas y de los comandos disponibles en el entorno `Moon`, de la forma en que se muestra en la 4.3

En este caso primero se envía un mensaje haciendo la invocación al método *get()* de la primer instancia de la clase *Variable* y a continuación esta instancia enviaría un valor

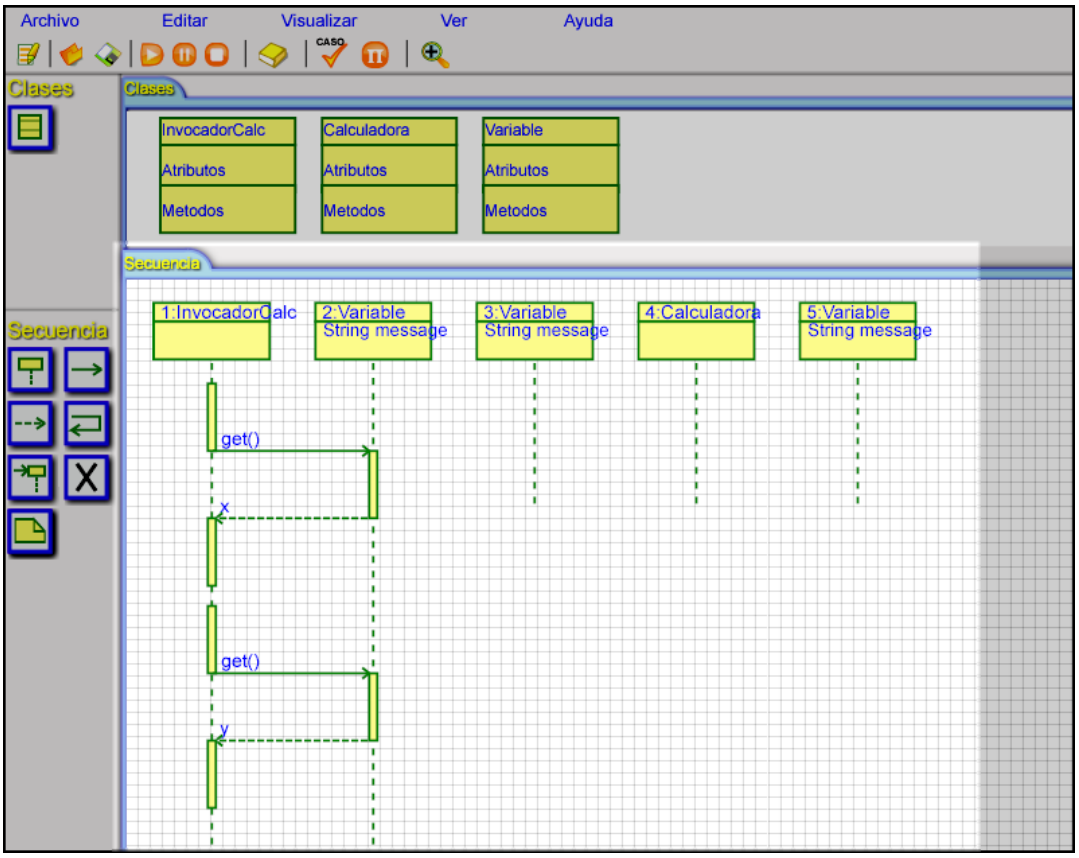


Figura 4.3: Envío de mensajes

de retorno, posteriormente se envía un mensaje *get()* a la siguiente instancia de variable y ésta envía el valor de retorno (los valores de retorno se supone se van almacenando en el cliente).

Una vez realizadas estas acciones se procede a realizar la invocación (desde la instancia de *InvocadorCalc*) del método *suma()* de la instancia de la clase Operaciones y ésta envía el valor de retorno, para finalmente enviar un último mensaje con la invocación del método *set()* desde *InvocadorCalc* hacia una instancia de *Variable* con el resultado de la operación y se procede a finalizar cada uno de los objetos participante. Antes de pasar a la compilación del diagrama, hay que seleccionar el objeto o los objetos sobre los cuales se generará el código, para ello seleccionamos el ícono de la barra de herramientas *Caso* y seleccionamos el objeto, para este caso el código se generará para el objeto de la clase *InvocadorCalc*. En la Figura 4.4 se muestra el diagrama de secuencia completo.

La generación de código inicia después de concluir esta fase de edición: el diagrama UML elaborado es compilado en reglas ADM de acuerdo a los esquemas de traducción

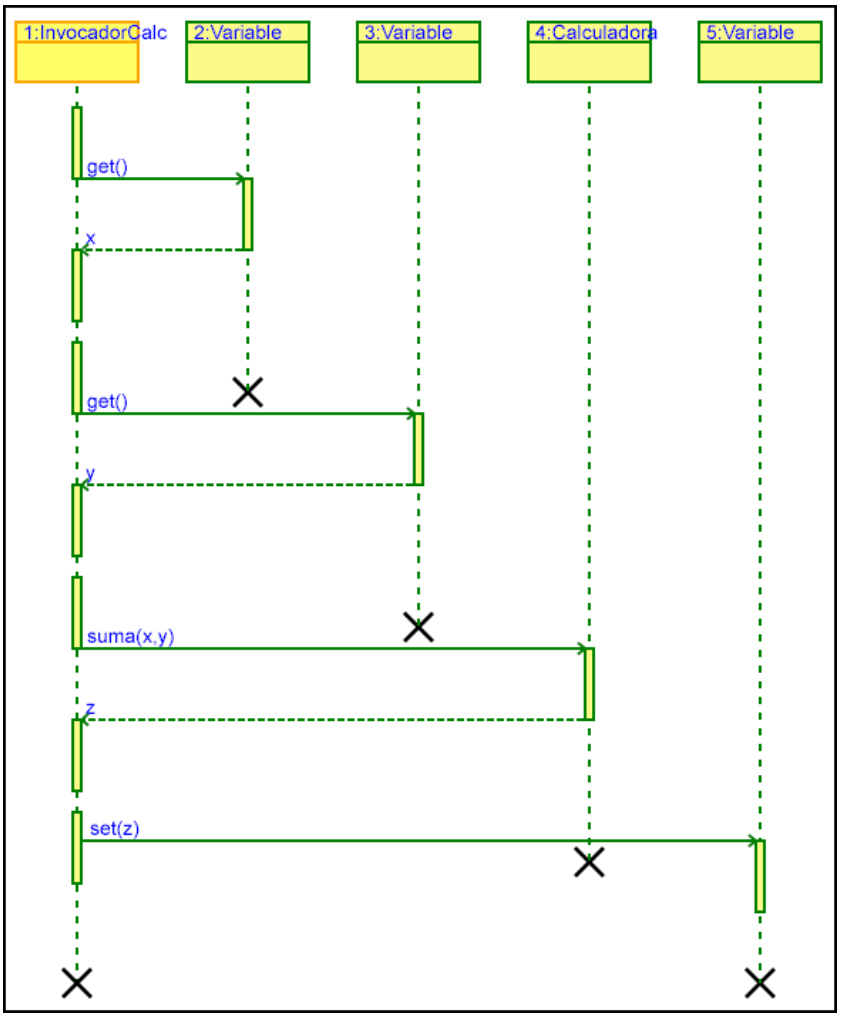


Figura 4.4: Diagrama de secuencia completo del caso invocaciones simples

explicados en la sección 4.1. y éstas reglas a su vez son transformadas al correspondiente código en Lenguaje Java. Las reglas ADM y el código en Java generado se muestran en las Figuras 4.5 y 4.6 respectivamente

Codigo

```
<root>
 <rule name='_InvocadorCalc_1' class='InvocadorCalc'>
  <do>
   <seq>
    <var name='x' type='integer'/>
    <var name='y' type='integer'/>
    <var name='z' type='integer'/>
    <initial to='_InvocadorCalc_1'>
     <InvocadorCalc>
     </InvocadorCalc>
    </initial>
    <initial to='_Variable_2'>
     <Variable>
     </Variable>
    </initial>
    <initial to='_Variable_3'>
     <Variable>
     </Variable>
    </initial>
    <initial to='_Calculadora_4'>
```

Figura 4.5: Reglas ADM del caso invocaciones simples

```
public class InvocadorCalc
 {
 public void start()
  {
   {
   int x;
   int y;
   int z;
   InvocadorCalc _InvocadorCalc_1 = new InvocadorCalc(null);
   Variable _Variable_2 = new Variable(null);
   Variable _Variable_3 = new Variable(null);
   Calculadora _Calculadora_4 = new Calculadora(null);
   Variable _Variable_5 = new Variable(null);
   x = _Variable_2.get("null");
   y = _Variable_3.get("null");
   z = _Calculadora_4.suma(x, y);
   _Variable_5.set(z);
   }
  }
 }
```

Figura 4.6: Código en Java del caso invocaciones simples

En el caso del código Java generado, se puede observar en la Figura 4.6 que se tiene el método *start()*, este método se genera en todos los programas Invocadores que se estan modelando, ya que es el punto de partida para la visualización del programa. Este método contiene las instrucciones acerca de los participantes del diagrama de secuencia y la interacción entre ellos.

La ultima fase es la ejecución del programa la cual es responsable de correr el programa y de coordinar la infraestructura provista por la maquina virtual. Durante la

ejecución del programa, el mediador ayuda a mostrar los resultados en los diagramas UML editados y se elabora la animación correspondiente del diagrama donde se muestra como se van ejecutando las instrucciones del programa en Java generado y los valores reales que se están enviando y recibiendo en el programa y se visualizan en el diagrama. Esta fase de visualización se muestra en la Figura 4.7.

En este caso de estudio y en los casos que se trataran en el resto del capítulo, se generan diferentes archivos durante la edición, compilación y visualización del programa. A continuación se muestra en la tabla 4.2 una lista con el tipo de archivo generados y su uso en el entorno.

Tabla 4.2: Archivos generados por el entorno

Nombre Archivo	Uso
`InvocadorCal.svg`	Archivo SVG que contiene el diagrama creado en el editor.
`InvocadorCal.html`	Archivo html que contiene el archivo InvocadorCal.svg embebido para presentarlo en el navegador.
`InvocadorCal-vis.svg`	Archivo SVG que contiene el diagrama visualizado.
`InvocadorCal-vis.html`	Archivo html que contiene el archivo InvocadorCal-vis.svg embebido para presentarlo en el navegador.
`InvocadorCal.java`	Código Java generado para el diagrama editado.
`InvocadorCal.adm`	Reglas ADM generadas a partir del diagrama editado.
`InvocadorCal.class`	Código Java compilado partir del programa Java.

Continua en la siguiente página...

Tabla 4.2 – Continuación

Nombre Archivo	Uso
`InvocadorCal-evt.xml`	Archivo con la secuencia de eventos que se siguieron para crear el diagrama en el editor. Este archivo es el que se puede abrir para cargar un diagrama creado previamente.
`InvocadorCal-med.xml`	Archivo con la secuencia de eventos generados por el mediador a partir del programa Java en ejecución. Es el archivo que se pasa al visualizador para que este muestre el diagrama.

Si no quiere utilizarse el visualizador integrado para obtener la visualización del programa, puede utilizarse el visualizador independiente, lo único que se requiere es tener el archivo con los eventos necesarios y cargarlos en el visualizador, el cual se encargará de interpretarlos y mostrar el diagrama de secuencia correspondiente. En la figura 4.8 se muestra el archivo con los comandos necesarios para visualizar la Calculadora, en este caso en específico el archivo se generó de manera automática al correr el programa de la calculadora, pero un archivo semejante a este puede crearse en cualquier editor siguiendo el esquema mostrado y al cargarlo en el visualizador obtendremos su representación gráfica.

4.2.2. Invocaciones Complejas

En este caso se tienen dos participantes: una instancia de la clase *InvocadorProm* y una instancia de la clase *Promedio*, con estos participantes se propone hacer un programa para calcular del promedio de dos números, utilizando un objeto intermediario (la instancia de la clase *Promedio*) que se encargará de hacer las invocaciones de las operaciones correspondientes para realizar la acción mencionada, para ello previamente se realizo su respectivo diagrama de secuencia y se compiló para generar la clase *Promedio*, que tiene el método *promedio(String x, String y)* que se invocara desde el objeto de la clase *InvocadorProm*.

Para la edición del diagrama de secuencia correspondiente, como primer paso se crean las dos instancias antes mencionadas y inicia el envío y recepción de mensajes con la invocación por parte de *InvocadorProm*. del método *promedio(String x, String y)* del

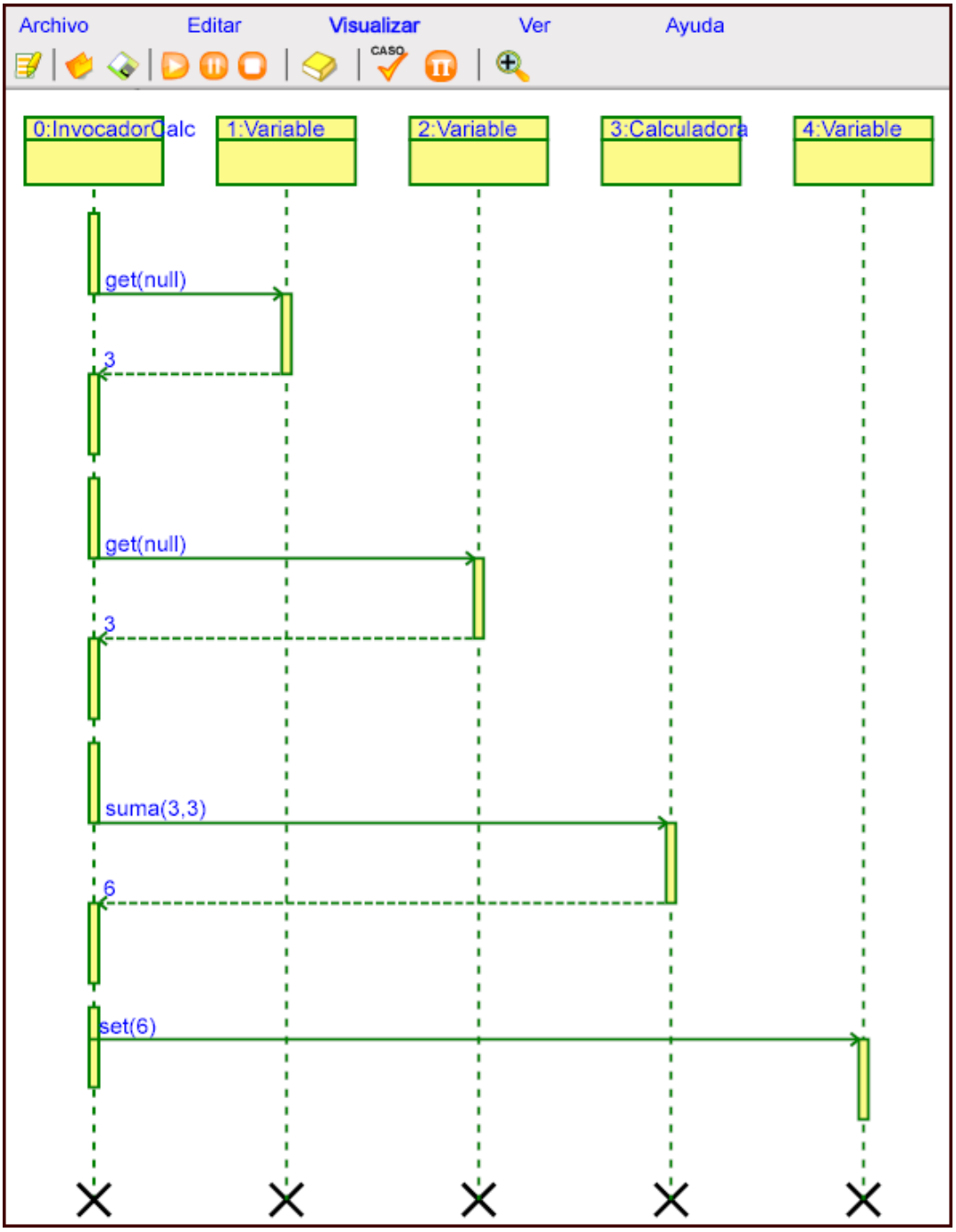


Figura 4.7: Visualización del caso invocaciones simples

```
- <inicio>
    <initial name="0" class="InvocadorCalc" />
    <initial name="1" class="Variable" />
    <initial name="2" class="Variable" />
    <initial name="3" class="Calculadora" />
    <initial name="4" class="Variable" />
    <call from="0" to="1" msg="get(null)" />
    <return from="1" to="0" msg="3" />
    <call from="0" to="2" msg="get(null)" />
    <return from="2" to="0" msg="3" />
    <call from="0" to="3" msg="suma(3,3)" />
    <return from="3" to="0" msg="6" />
    <send from="0" to="4" msg="set(6)" />
  </inicio>
```

Figura 4.8: Instrucciones de entrada para el visualizador

objeto *Promedio* los valores que se le pasan a este método son 15 y 3 como lo muestra la Figura 4.9, que una vez realizadas las operaciones necesarias retornará el valor a la clase *InvocadorProm*. Se finalizan los dos objetos participantes y se procede a seleccionar el objeto para el que se generaran las reglas y el código en este caso *InvocadorProm*.

Seleccionado el objeto se procede a la compilación del diagrama, se generan las reglas ADM y el código en Java de la clase, como se observa en la 4.10

Una vez editado el diagrama se procede a visualizarlo, con lo que se genera el programa en Java correspondiente y en una primera vista se tiene la Figura 4.11 que muestra la parte de la creación de objetos y la invocación de los primeros métodos obteniendo ahora los valores calculados para obtener el promedio entre 15 y 3, el envío y la recepción de mensajes continua en la animación hasta terminar con la ejecución del programa.

4.2.3. Invocaciones Recursivas

Para el caso de las invocaciones recursivas, se realizó un ejemplo para el cálculo del factorial de un número, se tienen dos participantes: una instancia de la clase *Invocador* el cual crea una instancia de la clase *Factorial* para definir el método *factorial(String n)*para un numero *n*.

Primeramente se cargan las clases y se crea la instancia del *Invocador*, una vez creado entonces se hace la creación de la instancia de la clase Factorial como se observa en la

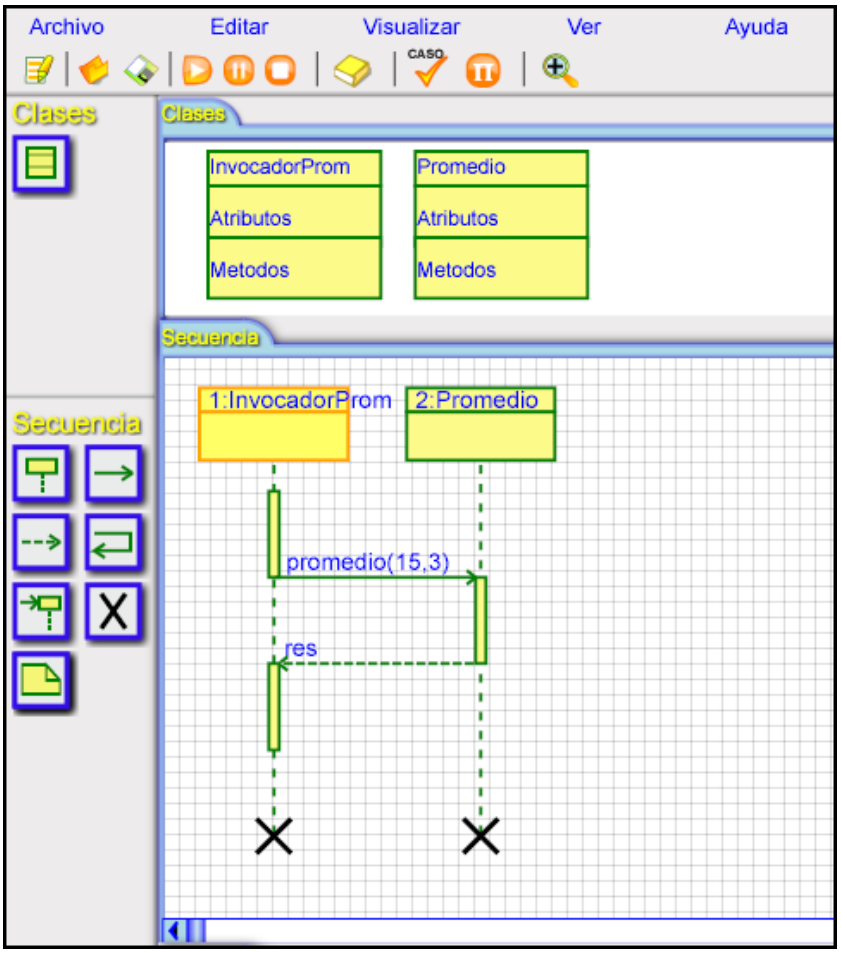


Figura 4.9: Edición del caso de estudio invocaciones complejas

Figura 4.12, para después poder hacer la invocación del método *factorial(String n)*. En este caso como se esta definiendo el comportamiento del método *factorial* hay que definir los casos que se toman en cuenta para calcular el factorial de un número:

$$\begin{aligned} factorial(n) &= 1 && \text{si } n = 0 \\ factorial(n) &= n * factorial(n-1) && \text{si } n > 0 \end{aligned}$$

en el diagrama de secuencia las comparaciones o alguna operación aritmética se representan en una nota donde se coloca la expresión correspondiente, de esta manera se coloca la condición n¿0 en la nota y se realizan las acciones que corresponden a ese caso, es decir se crea un nuevo objeto *Factorial* y se invoca ahora el método *factorial* pero con n-1. El caso para n=0 se define también en el mismo diagrama. Una vez diagramado el

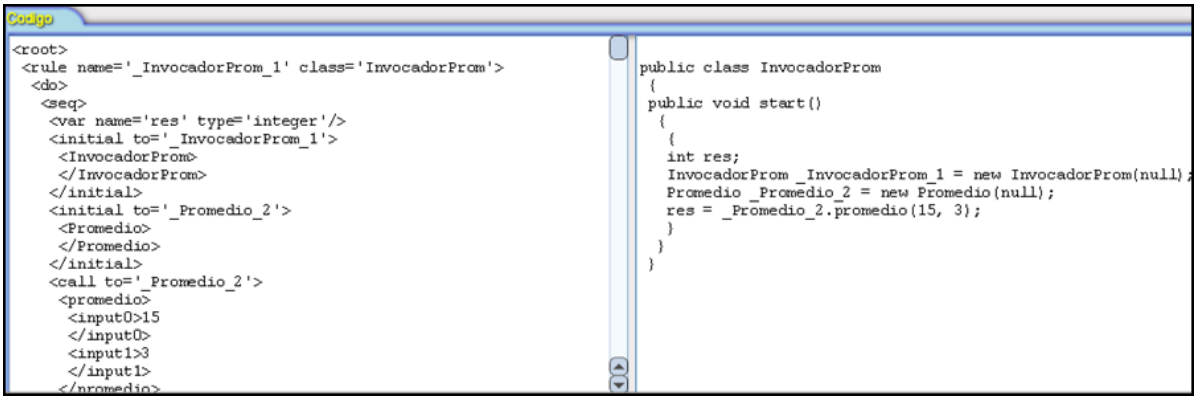


Figura 4.10: Reglas ADM y código Java generado para el caso invocaciones complejas

comportamiento se seleccionan los casos para los que generará el código y se procede a realizar la compilación y se generan las reglas ADM y el código en java como se muestra en la Figura 4.13.

Una vez obtenido el código de la Clase *Factorial*, se crea el diagrama de secuencia para calcular el factorial de un número, para este caso necesitamos un participante de la clase *InvocadorFactorial* que creará un participante de la clase *Factorial* e invocará al método *factorial* con un valor de 2. En la Figura 4.14

También se requiere que se seleccione el objeto para el cual se va a generar el código y una vez seleccionado se compila para crear las reglas ADM y el código en Java de la clase *InvocadorFactorial*. En la Figura 4.15 se muestra el código generado.

Una vez terminada la edición del diagrama de secuencia, se puede visualizar y como en los casos anteriores se inicia la ejecución del programa en Java generado y se comienza a visualizar la creación de los objetos y el envío de los mensajes como lo muestra la Figura 4.16.

En este caso en específico, las instancias de la clase factorial se van creando dinámicamente, ya que para calcular el factorial de un número se necesita crear una nueva instancia para hacer la invocación del método *factorial()*, por lo que se crearan tantas instancias como sea necesario hasta llegar al factorial de 0 y de esta manera iniciar el regreso de cada uno de los valores creados previamente para terminar la ejecución del método e ir retornando los valores calculados hasta llegar a la instancia de *InvocadorFactorial* con el resultado esperado.

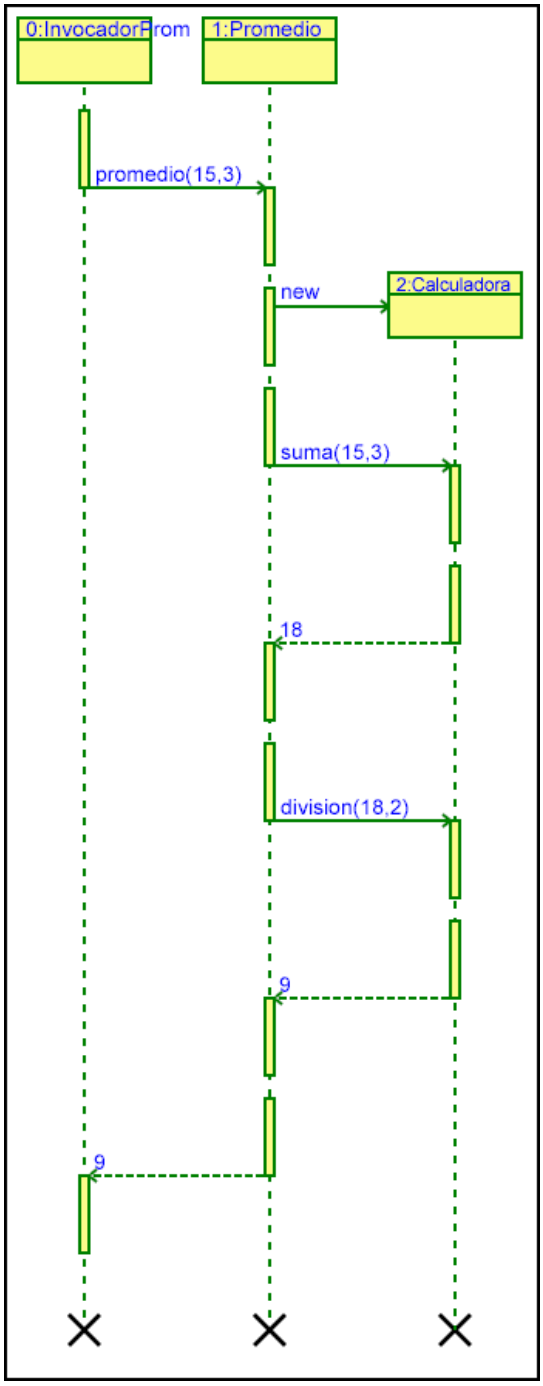


Figura 4.11: Visualización del caso invocaciones complejas

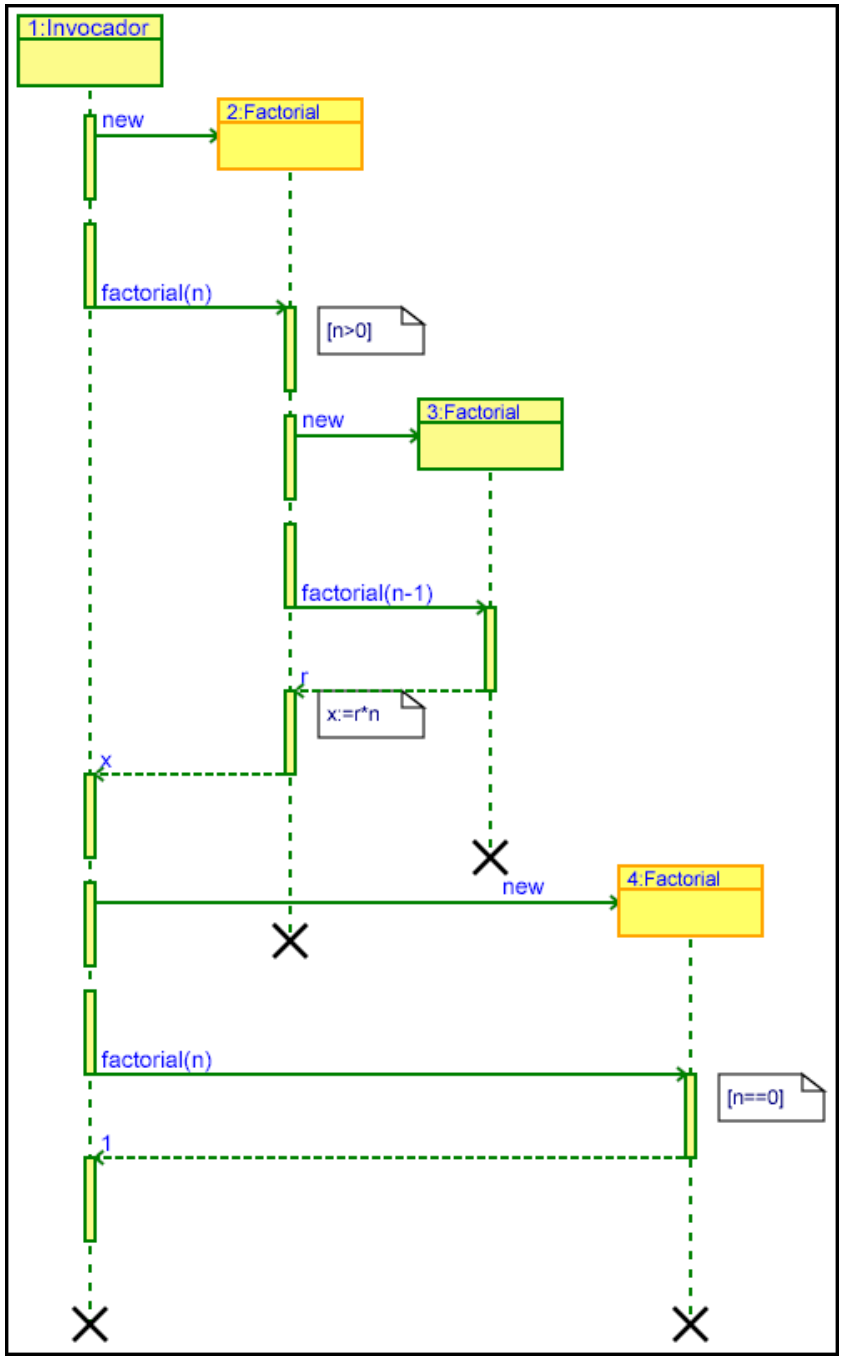


Figura 4.12: Edición del diagrama de secuencia para el Factorial

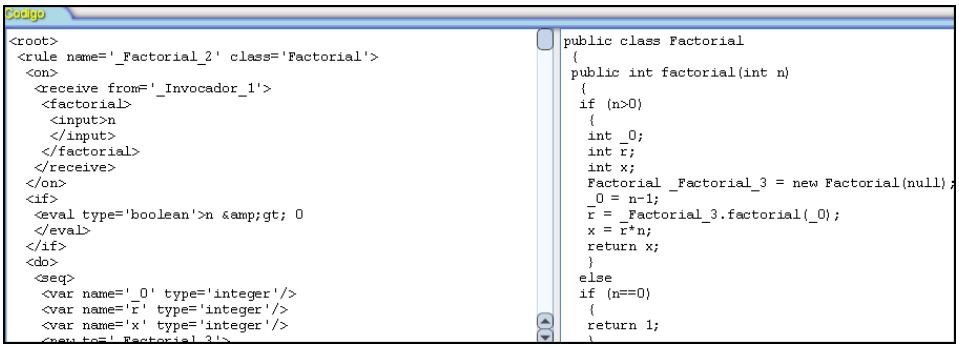


Figura 4.13: Código generado para el Factorial

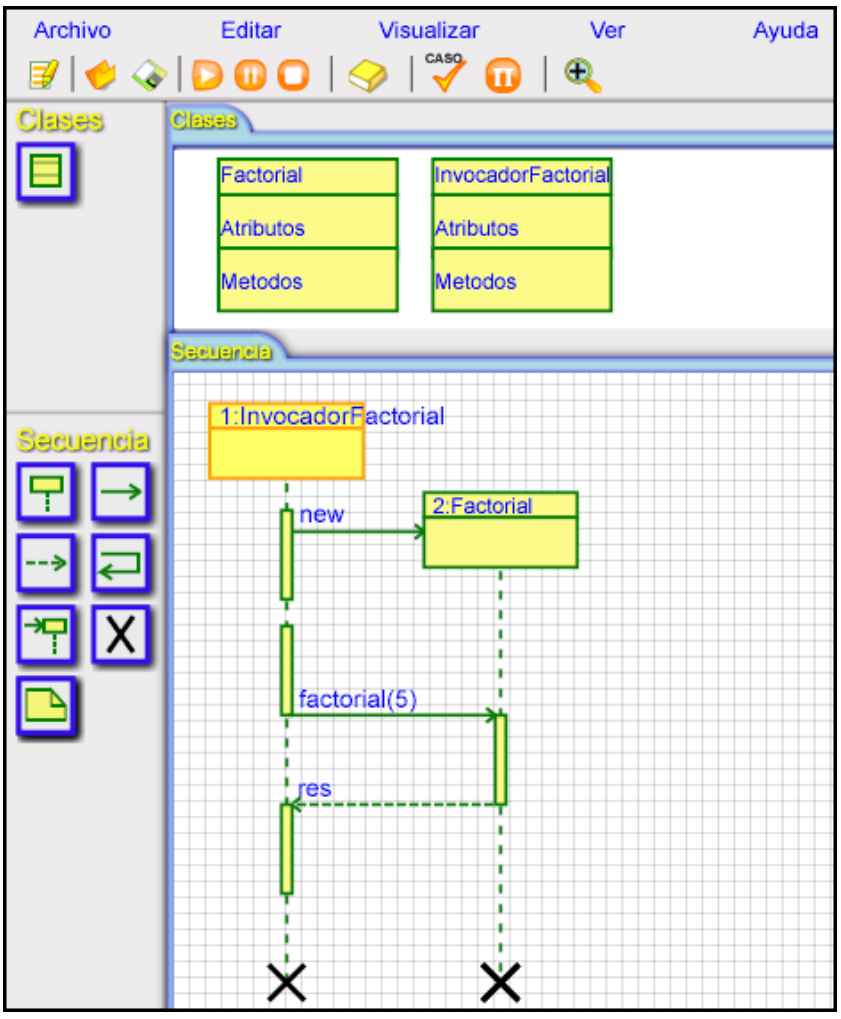


Figura 4.14: Edición del diagrama de secuencia para el caso invocaciones recursivas

```
public class InvocadorFactorial
 {
 public void start()
  {
   {
   int res;
   InvocadorFactorial _InvocadorFactorial_1 = new InvocadorFactori
   Factorial _Factorial_2 = new Factorial(null);
   res = _Factorial_2.factorial(5);
   }
  }
 }
```

Figura 4.15: Código en Java para el caso invocaciones recursivas

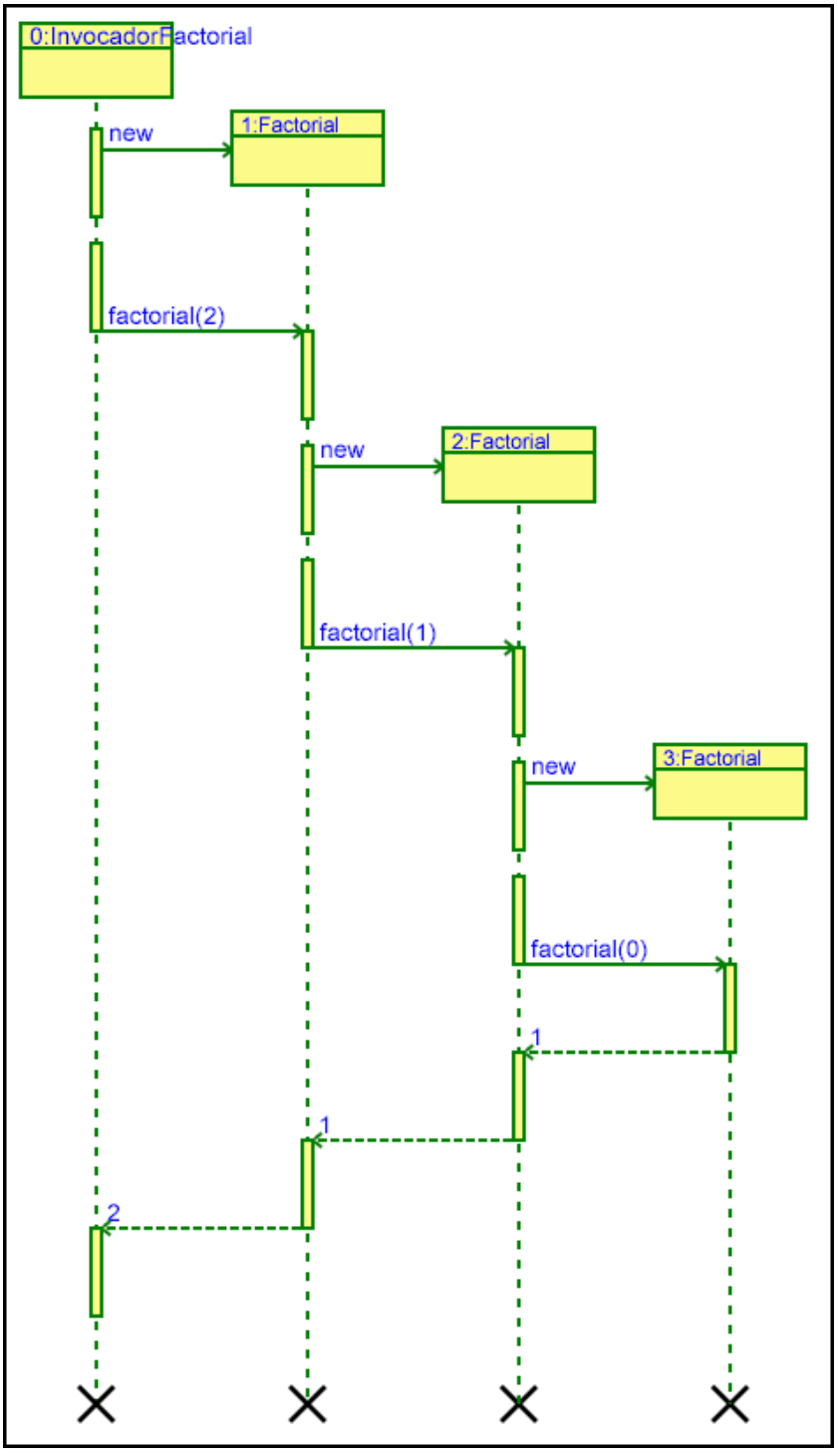


Figura 4.16: Visualización del caso invocaciones recursivas

Capítulo 5

Descripción de ADM

En este capítulo introduciremos formalmente la sintaxis y la semántica operacional de ADM con el propósito de formalizar el modelo de programación de las reglas deductivas y activas. Utilizaremos el estilo de la semántica operacional para describir las reglas de inferencia de cada operación básica de ADM. La descripción del lenguaje consiste en la presentación de dos modelos fundamentales: el modelo de reglas deductivas y el modelo de reglas activas. La descripción comienza con algunas definiciones básicas sobre el dominio sintáctico de los objetos del lenguaje como constantes, variables y términos así como de los procedimientos de substitución y unificación en el contexto de XML. A continuación se describe el modelo de reglas deductivas enunciando tanto el algoritmo del procedimiento de resolución como la correctitud del procedimiento. Aquí se discute brevemente la bien conocida interpretación procedural del principio de resolución. Finalmente, se describe el modelo de reglas activas como sistema de reescritura de multiconjuntos de términos XML.Se presentan y discuten las reglas de inferencia para las operaciones de inserción, eliminación, envío y recepción de (fragmentos de) documentos XML.

5.1. Sintaxis

La sintaxis abstracta de un término XML está dada formalmente por la gramática que aparece en la figura 5.1. Un término XML puede ser una cadena de caracteres, un texto, una variable, un elemento XML o un elemento XML con variables. Una cadena de caracteres es una secuencia de caracteres delimitados por comillas simples o dobles y un texto es una secuencia de caracteres delimitados por elementos XML. Una variable es un identificador que comienza con un signo de pesos. Un término es un elemento XML que puede incluir variables en su lista de atributos o en sus subelementos. La función $vars(T)$ designa al conjunto de todas las variables lógicas que ocurren en el término T.

$$
\begin{array}{rcl}
\mathtt{a}, \mathtt{b}, \ldots & \in & \mathtt{XMLConstant} \\
\mathtt{\$X}, \mathtt{\$Y}, \ldots & \in & \mathtt{XMLVariable} \\
A, B, \ldots, S, T, \ldots & \in & \mathtt{XMLTerm} \\
\sigma, \ldots & \in & \Sigma = \mathtt{XMLVariable} \to \mathtt{XMLTerm} \\
P, \ldots & \in & \mathtt{XMLProgram} = \mathbf{M}\,\mathtt{XMLTerm}
\end{array}
$$

$$
\begin{array}{rcl}
T, A & ::= & \mathtt{a} \\
& | & \mathtt{\$X} \\
& | & \mathtt{<a}\ \mathtt{a}_1\mathtt{="}A_1\mathtt{"} \cdots \mathtt{a}_m\mathtt{="}A_m\mathtt{"/>} \\
& | & \mathtt{<a}\ \mathtt{a}_1\mathtt{="}A_1\mathtt{"} \cdots \mathtt{a}_m\mathtt{="}A_m\mathtt{">}\, T_1 \cdots T_n\, \mathtt{}
\end{array}
$$

Figura 5.1: Sintaxis abstracta de términos XML.

$$
\begin{array}{rcl}
\epsilon\,\{\mathtt{\$X} \mapsto T\} & = & \{\mathtt{\$X} \mapsto T\} \\
(\sigma \cup \{\mathtt{\$Y} \mapsto S\})\,\{\mathtt{\$X} \mapsto T\} & = & \sigma\,\{\mathtt{\$X} \mapsto T\} \cup \{\mathtt{\$Y} \mapsto S\{\mathtt{\$X} \mapsto T\}, \mathtt{\$X} \mapsto T\} \\
& & \quad \text{si } \mathtt{\$Y} \notin vars(T)
\end{array}
$$

Figura 5.2: Composición de substituciones.

El término XML $\mathtt{<a}\ \mathtt{a}_1\mathtt{="}A_1\mathtt{"} \cdots \mathtt{a}_m\mathtt{="}A_m\mathtt{">}\, T_1 \cdots T_n\, \mathtt{}$ está *normalizado* si su lista de pares atributo-valor está ordenada lexicográficamente en orden creciente por el nombre del atributo, es decir, $\mathtt{a}_i \leq \mathtt{a}_j$ si $i \leq j$ para todo $i, j \in \{1, \ldots, m\}$.

Las *sustituciones* $\Sigma = \mathtt{XMLVariable} \to \mathtt{XMLTerm}$ son funciones parciales de variables lógicas a términos XML. El *dominio de la substitución* σ, denotado por $dom(\sigma)$, consiste del conjunto de todas las variables en las que está definida la substitución. Las variables del dominio de una substitución quedan vinculadas a lo más a un término y nunca aparecen en cualquiera de los términos vinculados a las variables en la substitución. La *composición* de sustituciones σ y σ' se escribe $\sigma\sigma'$.

La sustitución nula ϵ es la identidad para la composición de sustituciones $\epsilon\sigma = \sigma\epsilon = \sigma$. Una *substitución concretizada* (*ground substitution*) es aquella que no introduce ninguna variable.

En la figura 5.2 se incluye la definición inductiva de composición de substituciones para el caso en que la segunda substitución consiste únicamente de una vinculación. El primer caso corresponde con la composición de una substitución con la identidad, mientras que en el segundo caso, la variable $\mathtt{\$X}$ se substituye en todos los términos vinculados a variables, agregando la vinculación de $\mathtt{\$X}$ a la substitución.

La *extensión natural* de una sustitución de variables a términos XML, definida en la figura 5.3, se denota por el mismo nombre $\sigma : \mathtt{XMLTerm} \to \mathtt{XMLTerm}$. La extensión natural también se define sobre colecciones (multiconjuntos) de términos usando el mismo nom-

$$
\begin{array}{rcl}
\texttt{a}\,\sigma & = & \texttt{a} \\
\$\texttt{X}\,\sigma & = & \begin{cases} \sigma(\$\texttt{X}) & \text{si } \$\texttt{X} \in dom(\sigma) \\ \$\texttt{X} & \text{si } \$\texttt{X} \notin dom(\sigma) \end{cases} \\
\texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\texttt{"/>}\,\sigma & = & \texttt{<a a}_1\texttt{="}A_1\sigma\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\sigma\texttt{"/>} \\
\boxed{\begin{array}{l}\texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\texttt{">} \\ T_1\cdots T_n \\ \texttt{}\end{array}}\,\sigma & = & \boxed{\begin{array}{l}\texttt{<a a}_1\texttt{="}A_1\sigma\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\sigma\texttt{">} \\ T_1\sigma\cdots T_n\sigma \\ \texttt{}\end{array}}
\end{array}
$$

Figura 5.3: Extensión natural de la substitución de variables a términos XML.

$$
\begin{array}{rcl}
\{\}\,\sigma & = & \{\} \\
\{T_1,\ldots,T_n\}\,\sigma & = & \{T_1\sigma,\ldots,T_n\sigma\} \\
(C_1 \oplus C_2)\,\sigma & = & C_1\sigma \oplus C_2\sigma
\end{array}
$$

Figura 5.4: Extensión natural de la substitución de términos XML a colecciones de términos XML.

bre $\sigma : \mathbf{M}\,\texttt{XMLTerm} \to \mathbf{M}\,\texttt{XMLTerm}$ como se muestra en la figura 5.4. La *instancia* de un término T bajo la sustitución σ se denota por $T\sigma$ y su definición inductiva queda determinada por la estructura del término XML como lo muestra la figura 5.3. Las instancias de términos XML bajo substitutiones concretizadas se llaman *instancias concretizadas*.

El problema de decidir si una ecuación entre términos XML posee soluciones es fundamental. Las soluciones de un conjunto de ecuaciones se obtienen por el procedimiento de *unificación* que transforma al conjunto de ecuaciones en un conjunto de ecuaciones orientadas en donde una variable aparece siempre en el lado izquierdo de a lo más una ecuación. Las ecuaciones orientadas finalmente corresponden con substituciones. La ecuación entre dos términos XML T y S se denota por `<equal>` $T\ S$ `</equal>` pero por simplicidad en lo sucesivo escribiremos simplemente $T = S$.

El *unificador* σ es una sustitución mínima que produce instancias sintácticamente idénticas $T\sigma = S\sigma$ de los términos T y S bajo σ.

El *unificador mas general* (*mgu*) es un unificador que asigna a las variables los términos que estrictamente se requieren para unificar y que no se puede obtener por la composición de otros distintos. El *procedimiento de unificación*, mostrado en la figura 5.5, corresponde esencialmente con aquel introducido por Martelli-Montanari [40] y produce, ya sea, el unificador más general si existe o fallo si no existe unificador alguno.

El unificador más general se calcula por la cerradura reflexiva transitiva de la relación binaria $\triangleright : (\mathbf{P}\,\texttt{XMLTerm}, \Sigma) \to (\mathbf{P}\,\texttt{XMLTerm}, \Sigma)$. Cada par en esta relación describe el progreso en la solución al problema de unificación que comienza con un conjunto que

$$\frac{S = \boxed{\begin{array}{l}\texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\texttt{">} \\ S_1 \cdots S_n \\ \texttt{}\end{array}} \quad T = \boxed{\begin{array}{l}\texttt{<a a}_1\texttt{="}B_1\texttt{"}\cdots\texttt{a}_m\texttt{="}B_m\texttt{">} \\ T_1 \cdots T_n \\ \texttt{}\end{array}}}{(C \cup \{S = T\}, \sigma) \triangleright (C \cup \{A_1 = B_1, \ldots, A_m = B_m, S_1 = T_1, \ldots, S_n = T_n\}, \sigma)}$$

$$\frac{S = \texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_m\texttt{="}A_m\texttt{"/>} \quad T = \texttt{<a a}_1\texttt{="}B_1\texttt{"}\cdots\texttt{a}_m\texttt{="}B_m\texttt{"/>}}{(C \cup \{S = T\}, \sigma) \triangleright (C \cup \{A_1 = B_1, \ldots, A_m = B_m\}, \sigma)}$$

$$(C \cup \{T = T\}, \sigma) \triangleright (C, \sigma)$$

$$(C \cup \{T = \$\texttt{X}\}, \sigma) \triangleright (C \cup \{\$\texttt{X} = T\}, \sigma)$$

$$(C \cup \{\$\texttt{X} = T\}, \sigma) \triangleright (C\{\$\texttt{X} \mapsto T\}, \sigma\{\$\texttt{X} \mapsto T\}) \quad \text{si } \$\texttt{X} \notin vars(T)$$

$$(C \cup \{T = T\}, \sigma) \triangleright failure \quad \text{en otro caso}$$

$$\frac{(\{T = S\}, \epsilon) \triangleright^* (\epsilon, \sigma)}{mgu(T, S) = \sigma}$$

Figura 5.5: Unificación de términos XML.

contiene únicamente al problema inicial $T = S$ y la substitución nula. La relación termina cuando el conjunto de igualdades entre términos queda vacío, siendo la substitución resultante σ el unificador más general. La relación se define por casos de acuerdo a la estructura sintáctica de los términos como se muestra en la figura 5.5.

En el primer caso, se transforma el problema original $T = S$ en m subproblemas obtenidos por la comparación de los atributos correspondientes de acuerdo a su nombre junto con los n subproblemas obtenidos por la comparación de los subtérminos correspondientes de acuerdo a su posición. El segundo caso es similar al primero excepto que los términos del problema original no contienen subtérminos. El tercer caso consiste en la eliminación de dos términos sintácticamente idénticos. El cuarto caso corresponde a la igualdad entre una variable no concretizada y un término que resulta en la orientación de la ecuación que deja a la variable en el lado izquierdo. El quinto caso es similar al anterior excepto que la variable aparece ya en el lado izquierdo. En tal caso, se substituye la variable por su valor en todas las ocurrencias de la variable en el término. Al mismo tiempo, se obtiene una nueva substitución por composición de la substitución anterior σ con $\{\$\mathtt{X} \mapsto T\}$.

En cualquier otro caso, como la comparación de términos con diferente nombre de elemento, diferente número de atributos o diferente número de subtérminos, se produce fallo. La unificación de términos XML provee un mecanismo uniforme tanto para el paso de parámetros como para la selección y construcción de la información contenida en el documento XML.

5.2. Modelo de Reglas Deductivas

El modelo de reglas deductivas de `ADM` coincide con aquel de la programación lógica el cual se basa en los principios de unificación y de resolución SLD [39]. La razón de haber adoptado este modelo es la bien establecida fundamentación teórica que posee así como el amplio conocimiento que se tiene de él. Los enfoques de implementación generalmente coinciden en la bien conocida interpretación procedural sugerida por Kowalski del principio de resolución SLD. Recíprocamente, la lectura declarativa de un procedimiento lógico establece la validez de la llamada del encabezado del procedimiento si todas las condiciones que forman su cuerpo son válidas.

Un *paso de resolución SLD* se define por la relación binaria $\Rightarrow$: $(\mathtt{XMLTerm}, \Sigma) \rightarrow (\mathtt{XMLTerm}, \Sigma)$ que transforma la consulta $(\mathtt{<if>}\, C_1 C_2 \cdots C_n \,\mathtt{</if>}, \sigma)$ en la consulta: $(\mathtt{<if>}\, B_1\sigma' \cdots B_m\sigma' C_2\sigma' \cdots C_n\sigma' \,\mathtt{</if>}, \sigma\sigma')$, remplazando la llamada C_1 al procedimiento $\mathtt{<a\ a}_1\mathtt{="}A_1\mathtt{"} \cdots \mathtt{a}_k\mathtt{="}A_k\mathtt{"/>}$ por la instancia bajo σ' de su cuerpo $B_1\sigma' \cdots B_m\sigma'$, siempre que exista el unificador mas general σ' de C_1 y $\mathtt{<a\ a}_1\mathtt{="}A_1\mathtt{"} \cdots \mathtt{a}_k\mathtt{="}A_k\mathtt{"/>}$.

La *respuesta calculada* de una consulta se obtiene a partir de cero, uno o más pasos de resolución SLD comenzando desde la consulta inicial. La ejecución del programa ter-

$$\frac{\texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_k\texttt{="}A_k\texttt{">}\, B_1\cdots B_m\, \texttt{} \in \texttt{XMLProgram} \quad \exists\sigma'.\ \sigma' = mgu(head(C_1), \texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_k\texttt{="}A_k\texttt{">})}{(\texttt{<if>}\, C_1C_2\cdots C_n\, \texttt{</if>}, \sigma) \Rightarrow (\texttt{<if>}\, B_1\sigma'\cdots B_m\sigma' C_2\sigma'\cdots C_n\sigma'\, \texttt{</if>}, \sigma\sigma')}$$

$$\frac{\texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_k\texttt{="}A_k\texttt{"> } \in \texttt{XMLProgram} \quad \exists\sigma'.\ \sigma' = mgu(head(C_1), \texttt{<a a}_1\texttt{="}A_1\texttt{"}\cdots\texttt{a}_k\texttt{="}A_k\texttt{">})}{(\texttt{<if>}\, C_1C_2\cdots C_n\, \texttt{</if>}, \sigma) \Rightarrow (\texttt{<if>}\, C_2\sigma'\cdots C_n\sigma'\, \texttt{</if>}, \sigma\sigma')}$$

Figura 5.6: Relación de inferencia SLD.

$$\frac{(\texttt{<if>}\, C_1\cdots C_n\, \texttt{</if>}, \epsilon) \Rightarrow^* (\texttt{<if> </if>}, \sigma)}{P \models_\sigma \texttt{<if>}\, C_1\cdots C_n\, \texttt{</if>}}$$

Figura 5.7: Correctitud de la relación de resolución SLD.

mina cuando ya no hay mas invocaciones de procedimientos que resolver en la consulta. La respuesta calculada se obtiene de la composición de las sustituciones usadas en cada paso del procedimiento de resolución. La figura 5.7 establece que la respuesta calculada es efectivamente la respuesta correcta, es decir, la sustitución σ es la solución de la consulta. Un resultado bien conocido de la programación lógica [39] dice que una respuesta calculada σ obtenida por el cálculo de resolución SLD es un modelo para ambos el programa P y la consulta $\texttt{<if>}\, C_1\cdots C_n\, \texttt{</if>}$. Por lo tanto, las respuestas calculadas son aquellas que satisfacen las condiciones lógicas del programa.

5.3. Modelo de Reglas Activas

La definición del comportamiento reactivo y la preservación de las restricciones de integridad se describen mediante *reglas activas*, también conocidas como *reglas evento-condición-acciíon* (*ECA*). El detector de eventos percibe y notifica al sistema cuando una colección de documentos XML se ha modificado debido a que un nuevo documento fue insertado, uno ya existente fue eliminado o algún otro fue recibido por mensajería. Formalmente, la colección de documentos XML se representa por un multiconjunto de términos XML cuyo tipo se denota por $\mathbf{M}\,\texttt{XMLTerm}$. La representación de una colección de documentos XML como multiconjunto obedece al hecho de que la multiplicidad de los documentos es importante aún cuando las relaciones espaciales relativas que puedan guardar los documentos entre si no lo sean. En lo sucesivo designaremos como *programa* a la colección $P \in \mathbf{M}\texttt{XMLTerm}$ de documentos formada por las bases extensionales e intensionales de documentos XML. Esta designación se justifica por el hecho de que

la colección entera de documentos incluye a los documentos XML ordinarios (bases de datos extensionales) y a las reglas deductivas y activas (bases de datos intensionales).

5.3.1. Ambito de una regla

Aunque el modelo subyacente de repositorio compartido de documentos es un principio fundamental de diseño en ADM, la partición de la colección global de documentos simplifica el modelo de programación porque permite identificar colecciones individuales. Por esta razón, la colección global de documentos se encuentra particionada en localidades referidas por un nombre simbólico. Cada colección puede a su vez particionarse de nuevo para dar lugar así a una estructura jerárquica similar a la estructura de árbol de un directorio de archivos. Esta organización permite localizar con facilidad a cualquier documento dentro de la colección global.

Una *localidad* define una partición jerárquica en la colección de documentos la cual puede designarse por un nombre simbólico. La notación P/L designa a la colección de documentos ubicados en la localidad L del programa o colección P, en tanto que la notación $P/L/D$ designa a un documento D ubicado en la localidad L de P. El ámbito de una regla activa se extiende a la localidad más pequeña que contiene a la regla, es decir, que solamente la colección de documentos de la localidad participan en la selección de la regla. La importancia de la noción de ámbito radica en que restringe los documentos considerados durante el proceso de consulta usado en el modelo deductivo. De esta forma, la designación de documentos por localidades restringe en forma controlada la selección y activación de reglas activas. En lo sucesivo, por colección nos referimos o bien a la colección global P o a la subcolección P/L de P en la localidad L.

5.3.2. Selección

Al procedimiento que permite determinar las reglas que pueden ejecutarse como respuesta a la ocurrencia de un evento externo se le llama *selección*. Una vez que el administrador de eventos ha indicado la ocurrencia del evento A_e (originado por la realización de alguna acción), la estructura del documento asociado con el evento determina la selección de la regla R cuando el elemento evento E de R posee la misma estructura que A_e. El procedimiento de unificación puede utilizarse para determinar la igualdad de las estructuras y al mismo tiempo recuperar la información σ asociada con el evento A_e de tal manera que $E\sigma = A_e\sigma$. La estructura e información que se obtienen a partir del contenido del documento XML participante en el evento permite seleccionar todas las reglas que atienden exactamente al mismo tipo de evento. Este grado de indeterminismo es aceptable cuando se busca garantizar la imparcialidad en la selección de las reglas que rigen el comportamiento de los sistemas distribuidos. De las reglas seleccionadas, se

$$R = \boxed{\begin{array}{l}\texttt{<rule>}\\ \quad\texttt{<on>}\, E\, \texttt{</on>}\\ \quad\texttt{<if>}\, C\, \texttt{</if>}\\ \quad\texttt{<do>}\, A\, \texttt{</do>}\\ \texttt{</rule>}\end{array}}$$

$$\frac{P \cup \{A_e\} \models_\sigma \texttt{<if>}\, E\; C\, \texttt{</if>}}{P, A_e \mid\!\models_\sigma R}$$

$$\frac{\neg\exists\sigma \in \Sigma.\; P \cup \{A_e\} \models_\sigma \texttt{<if>}\, E\; C\, \texttt{</if>}}{P, A_e \not\mid\!\models R}$$

Figura 5.8: Selección de la regla R por la ocurrencia del evento A_e.

escogen solamente aquellas cuya condición es satisfecha por la información recuperada del evento. Utilizando el procedimiento de resolución SLD, si la condición $C\sigma$ se satisface (por refutación), se ejecutan entonces las acciones $A\sigma$ de la regla, que pueden incluir operaciones de inserción, eliminación, envío y recepción de documentos de la colección.

La figura 5.8 formaliza la selección de una regla mediante la relación de *selección* $\mid\!\models$ de una regla en una colección de documentos. La relación de selección $P, A_e \mid\!\models_\sigma R$ se cumple siempre que ambos el evento E y la condición C de la regla R sean consecuencia lógica de la colección P cuando ocurre (extendida temporalmente) el evento A_e. En tal caso, existe una respuesta calculada σ que es la solución de la consulta extendida `<if>` E C `</if>` en la colección temporalmente extendida de documentos $P \oplus \{A_e\}$. Como se indica en la consulta, la estructura del evento A_e debe coincidir con la estructura dada por E.

La segunda regla de inferencia establece que la relación $P, A_e \not\mid\!\models R$ de selección no se cumple en la colección de documentos cuando no existe una solución a la consulta formada por el evento y la condición de la regla. La relación de selección de una regla es un concepto fundamental ya que determina cuando una regla es elegible para ejecutarse. Esta relación la usaremos en todas las reglas de inferencia de las operaciones sobre documentos que se describen en adelante.

5.3.3. Ejecución

La semántica operacional de ADM se define mediante las nociones de configuración y de transición. Una *configuración no terminal* $(P, \sigma) \in \mathbf{M}\,\texttt{XMLTerm} \times \Sigma$ consiste de un programa P (base intensional y extensional de documentos XML) y de una substitución Σ conocida también como el *estado de la configuración*.na *configuración ter-*

$$R = \boxed{\begin{array}{l}\texttt{<rule>}\\ \quad\texttt{<on>}\, E \,\texttt{</on>}\\ \quad\texttt{<if>}\, C \,\texttt{</if>}\\ \quad\texttt{<do>}\, A \,\texttt{</do>}\\ \texttt{</rule>}\end{array}}$$

$$\frac{\exists R \in P.\ P, A_e \mid\!\models_{\sigma'} R}{(P \oplus \{A_e\}, \sigma) \longrightarrow (P \oplus \{A\}, \sigma\sigma')}$$

$$\frac{\forall R \in P.\ P, A_e \not\mid\!\models R}{(P \oplus \{A_e\}, \sigma) \longrightarrow P\sigma}$$

Figura 5.9: Reglas de inferencia para la ejecución de reglas.

minal P consiste simplemente un programa $P \in \mathbf{M}\,\texttt{XMLTerm}$. La relación de *transición* entre configuraciones $\longrightarrow$: $(\mathbf{M}\,\texttt{XMLTerm} \times \Sigma) \rightarrow (\mathbf{M}\,\texttt{XMLTerm} \times \Sigma \cup \mathbf{M}\,\texttt{XMLTerm})$ establece la evolución del comportamiento del sistema por las interacciones de sus participantes. La relación de transición es *terminante* cuando cada secuencia de transiciones $(P_0, \sigma_0) \longrightarrow (P_1, \sigma_1) \longrightarrow \cdots \longrightarrow (P_{n-1}, \sigma_{n-1}) \longrightarrow P_n$ es finita y resulta en una configuración terminante. Por el contrario, la transición es *divergente* cuando es no terminante.

En la figura 5.9 aparecen las reglas de inferencia para la ejecución de reglas activas la cual se define como la cerradura reflexiva transitiva de la relación de transición. Asumiendo que la regla R en P se compone del evento E, la condición C y la acción A, la primera regla de inferencia establece que la ocurrencia del evento A_e causa la transición de la colección P extendida con A_e $(P \oplus \{A_e\})$ en el estado σ a la colección $P\sigma'$ después de ejecutar R en el estado $\sigma\sigma'$. La ejecución de R consiste en insertar en la colección de documentos la instancia de la acción A de R a realizar $(P\sigma' \oplus \{A\sigma'\})$. La respuesta calculada σ' se obtiene al demostrar que R es seleccionada por A_e en P.

La ocurrencia explícita de la instancia $P\sigma'$ de la colección de documentos en la relación de transición establece la substitución de las variables ya resueltas lo que conduce a la propagación de los fragmentos de documentos XML y su composición para formar nuevos documentos. Mediante este mecanismo de propagación y composición, el contenido de los documentos XML se construye a medida que las variables se van concretizando como resultado de la ejecución de las reglas activas.

Puesto que pueden ser varias las reglas seleccionadas para la creación de instancias, la selección final de la instancia se determina por la aplicación de las reglas de inferencia de interacción entre reglas activas dadas más adelante (figura 5.3.4).

La segunda regla de inferencia establece la condición de terminación de la transición cuando ninguna regla puede activarse por la ocurrencia de A_e en P. En este caso, la

$$\frac{(P_1,\sigma)\longrightarrow(P_1',\sigma') \quad docset_P(P_1\sigma)\cap docset_P(P_2\sigma)\neq\emptyset}{(P_1\oplus P_2,\sigma)\longrightarrow(P_1'\sigma'\oplus P_2,\sigma\sigma')} \qquad \frac{(P_2,\sigma)\longrightarrow(P_2',\sigma') \quad docset_P(P_1\sigma)\cap docset_P(P_2\sigma)\neq\emptyset}{(P_1\oplus P_2,\sigma)\longrightarrow(P_1\oplus P_2'\sigma',\sigma\sigma')}$$

$$\frac{(P_1,\sigma)\longrightarrow(P_1',\sigma_1) \quad (P_2,\sigma)\longrightarrow(P_2',\sigma_2) \quad docset_P(P_1\sigma)\cap docset_P(P_2\sigma)=\emptyset}{(P_1\oplus P_2,\sigma)\longrightarrow(P_1'\sigma_1\oplus P_2'\sigma_2,\sigma\sigma_1\sigma_2)}$$

$$\frac{(P_1,\sigma)\longrightarrow P_1\sigma \quad (P_2,\sigma)\longrightarrow P_2\sigma}{(P_1\oplus P_2,\sigma)\longrightarrow P_1\sigma\oplus P_2\sigma}$$

$$doc(\texttt{<send to="}L\texttt{">}\, D\, \texttt{</send>}) = D$$
$$doc(\texttt{<receive from="}L\texttt{">}\, D\, \texttt{</from>}) = D$$

$$docset_P(R) = \{doc(A_e) \mid P, A_e \mid\!\models R\}$$

$$docset_P(\biguplus R) = \biguplus docset_P(R)$$

Figura 5.10: Reglas de inferencia para la interacción de programas.

secuencia de transiciones siempre resulta en una configuración terminante, dejando como resultado a la instancia $P\sigma$ de la colección que se obtiene al substituir todas las variables por sus valores en aquellos documentos donde estas aparecen.

5.3.4. Interacción entre Reglas Activas

Los programas o colecciones de documentos pueden interactuar entre ellos dependiendo de los documentos que compartan. En la figura 5.3.4 se muestran las reglas de inferencia que describen el comportamiento global de dos programas que interactúan. La función *docset* de la regla R en el programa P determina el multiconjunto de todos los documentos que hacen a R seleccionable para su ejecución. El propósito de la función *docset* es determinar aquellos documentos (eventos) que pueden hacer seleccionables a más de una regla, lo que señalaría a dicho multiconjunto como una región crítica de documentos compartidos por reglas. Las reglas de inferencia de interacción entre programas deben, por lo tanto, garantizar el acceso exclusivo de una regla a la vez de entre varias seleccionadas.

Las primeras dos reglas se aplican cuando ambos programas comparten un multiconjunto no vacío de documentos. En tal caso cualquiera de los dos programas, pero no ambos simultáneamente, puede desarrollar su comportamiento. La tercera regla se aplica

$$\frac{(P \oplus \{P_1\}, \sigma) \longrightarrow P\sigma \oplus \{P_1\sigma\}}{(P \oplus \{\texttt{<seq>}\, P_1\ P_2\, \texttt{</seq>}\}, \sigma) \longrightarrow (P\sigma \oplus \{P_1\sigma\} \oplus \{P_2\}, \sigma)}$$

$$\frac{(P \oplus \{P_1\}, \sigma) \longrightarrow (P \oplus \{P_1'\}, \sigma')}{(P \oplus \{\texttt{<seq>}\, P_1\ P_2\, \texttt{</seq>}\}, \sigma) \longrightarrow (P \oplus \{\texttt{<seq>}\, P_1'\ P_2\, \texttt{</seq>}\}, \sigma\sigma')}$$

$$(P \oplus \{\texttt{<par>}\, P_1\ P_2\, \texttt{</par>}\}, \sigma) \longrightarrow (P \oplus \{P_1\} \oplus \{P_2\}, \sigma)$$

Figura 5.11: Reglas de inferencia para la composición secuencial y paralela de programas.

solamente si los programas no comparten alguna colección de documentos. En tal caso, si ambos programas exhiben algún progreso por separado, entonces los programas no interfieren entre si y podrán desarrollar su comportamiento independientemente el uno del otro. Finalmente, la última regla describe la condición de terminación. En este caso, si dos programas terminan independientemente, también lo hacen uniendo las colecciones de documentos que constituyen ambos programas.

5.3.5. Secuencialidad y Concurrencia

La ejecución de las reglas puede dar lugar a comportamientos inaceptables debido a que el orden en la ejecución de las reglas puede causar interferencia entre ellas impidiendo la selección de unas o promoviendo la ejecución reiterada de otras. Para reducir el no determinismo en el orden en el que las reglas se ejecutan, se introduce la composición secuencial y concurrente de programas para restringir el orden de su ejecución.

La figura 5.11 muestra las reglas de inferencia para la composición secuencial y concurrente de programas. La composición secuencial y concurrente de programas se introduce sintácticamente mediante las marcas `<seq>` y `<par>`, respectivamente. La primera regla de inferencia describe la condición de terminación de dos programas en composición secuencial. Si el programa P_1 termina en el estado σ, entonces la composición secuencial de los programas P_1 y P_2 se comporta como P_2 partiendo del mismo estado σ. En este caso, puesto que P_1 es terminante no existe interacción entre los subprogramas P y $\{P_1\}$ en el estado σ, el programa $P \oplus \{P_1\} \oplus \{P_2\}$ la interacción tendrá lugar solamente entre $P \oplus \{P_1\}$ y $\{P_2\}$.

La segunda regla de inferencia describe la condición de progreso de dos programas bajo la composición secuencial. Si en el estado σ el programa P_1 se reduce al programa P_1' con estado σ', entonces la composición secuencial de P_1 y P_2 reduce a la composición secuencial de P_1' y P_2 con estado $\sigma\sigma'$.

La tercera regla de inferencia describe la composición concurrente de dos programas P_1 y P_2, la cual consiste simplemente en el desarrollo de sus comportamientos individ-

uales. Aunque la composición concurrente no establece un orden en la ejecución de los subprogramas, las reglas de interacción de programas (sección 5.3.4) se cumplen igualmente.

La capacidad de describir secuencialidad y concurrencia en las acciones que desarrollan los programas tiene consecuencia inmediata en los modelos de comunicación disponibles. En ADM existen dos modelos principales de sincronización y de comunicación entre programas conocidos como modelo de espacio compartido de documentos y modelo de intercambio de mensajes. En este trabajo solamente consideramos el modelo de intercambio de mensajes.

5.3.6. Comunicación

El *modelo de intercambio de mensajes* introduce un modelo de comunicación asíncrona que extiende el modelo de programación de ADM como fue presentado en [32]. Este modelo de comunicación permite suspender el progreso en la ejecución de un programa mientras no se detecte un evento de recepción de un documento con la estructura y el contenido indicados por el evento y la condición, respectivamente. Las reglas de inferencia que se presentan en la figura 5.12 describen la recepción y el envío de documentos XML.

La primera regla de inferencia describe el efecto que se produce al detectar un evento de recepción en la colección P/L en donde existe una regla R que responde al evento. Como resultado, la colección P/L se modifica insertando el documento recibido D_e y ejecutando las acciones $A\sigma'$ de R.

La segunda regla describe el efecto de la acción localizada en la colección P/M de enviar el documento D_e a la colección P/L. El propósito de la segunda regla de inferencia es describir la semántica de la operación de envío, interpretada en términos de su operación complementaria, como la recepción del documento en el destino del envío. Como puede apreciarse, la segunda regla de inferencia no hace alusión a la infraestructura tecnológica del protocolo de transporte usado en la comunicación. Este es el tema que se trata a continuación en donde se discute la implementación de ADM.

$$R = \boxed{\begin{array}{l}\texttt{<rule>}\\ \quad\texttt{<on>}\,\texttt{<receive from="}M\texttt{">}\,D\,\texttt{</receive>}\,\texttt{</on>}\\ \quad\texttt{<if>}\,C\,\texttt{</if>}\\ \quad\texttt{<do>}\,A\,\texttt{</do>}\\ \texttt{</rule>}\end{array}}$$

$$\frac{\exists R \in P/L.\ \exists \sigma' \in \Sigma.\ P/L, D_e \mid\!\models_{\sigma'} R}{\begin{array}{c}(P/L \oplus \{\texttt{<receive from="}M\texttt{">}\,D_e\,\texttt{</receive>}\}, \sigma)\\ \longrightarrow (P/L \oplus \{A\,\sigma', D_e\}, \sigma\sigma')\end{array}}$$

$$\begin{array}{c}(\{P/M/\texttt{<send to="}L\texttt{">}\,D_e\,\texttt{</send>}\}, \sigma)\\ \longrightarrow (\{P/L/\texttt{<receive from="}M\texttt{">}\,D_e\,\texttt{</receive>}\}, \sigma)\end{array}$$

Figura 5.12: Reglas de inferencia para la recepción y el envío de documentos.

Capítulo 6

Implementación

La implementación del entorno de programación visual está conformado por tres módulos principales: el editor, el mediador y el visualizador.Estos módulos corresponden con aquellos descritos en el modelo del entorno `MOON` presentado en el Capítulo 3. El visualizador puede utilizarse también en forma independiente del entorno integrado de programación. A esta versión del visualizador la designaremos como el visualizador independiente para distinguirla de su contraparte integrada en `MOON`. Con excepción del visualizador independiente, los módulos de `MOON` se sitúan en el lado del cliente, generalmente, el visualizador de Internet (*browser*). Para la implementación de cada módulo se utilizaron diferentes lenguajes de programación de acuerdo a su propósito:

- El editor, escrito en JavaScript, utiliza SVG para representar tanto a la interfaz gráfica de usuario como a los diagramas UML.
- El mediador, escrito como Applet de Java, gestiona los permisos de acceso requeridos para acceder leer, escribir, compilar y ejecutar programas en Java.
- El visualizador, escrito en JavaScript, utiliza SVG para visualizar la salida producida por los programas controlados por el mediador.

La implementación de estos módulos se encuentra en diferentes archivos y su interacción entre ellos se muestra en la sección 6.1. En la sección 6.2 se describe el módulo del editor, la sección 6.3 muestra la descripción de la clase del Mediador y finalmente en la sección 6.4 se describen los elementos de la implementación del Visualizador.

6.1. Interacción entre los módulos

Como se menciono se tienen tres módulos en general que interactúan entre sí para darle la funcionalidad al entorno de programación `MOON`. Primeramente para poder cargar

el entorno, hay que hacer uso del archivo HTML en el cual se encuentra embebido el archivo SVG que contienen toda el diseño visual del entorno y las diferentes funciones que lo implementan, también en ese archivo se encuentran embebido el applet que forma parte del Mediador y el applet que forma parte del editor para almacenar los diagramas. Una vez que se encuentran cargados estos dos módulos e iniciada la sesión inicia la interacción entre los diferentes módulos, ya que las diversas funciones que se requieren se encuentran distribuidas en los archivos que los conforman. En la Figura 6.1 se muestra un esquema general de los tres bloques y los archivos que conforman a cada uno de ellos.

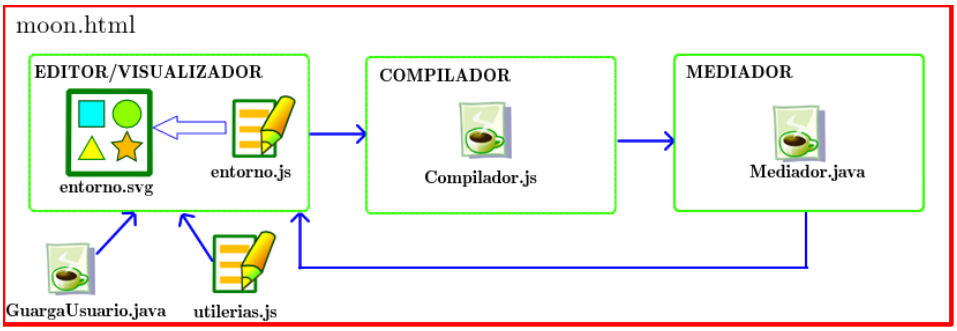


Figura 6.1: Componentes del entorno de programación `Moon`

Se puede observar que todos los módulos se encuentran dentro del archivo *MOON.html*, ya que como se mencionó los archivos que conforman cada módulo se encuentran embebidos dentro del mismo. De esta manera la comunicación desde el editor hacia el mediador se hace utilizando como puente las funciones del archivo HTML *utilerias.js*. De la misma forma cuando se quiere hacer la compilación del archivo SVG, se utiliza también las funciones en Java Script del archivo HTML principal. Esta comunicación se puede hacer gracias al Modelo de Objeto de Documento (DOM), que permite la interrelación de diversos componentes dentro de un archivo HTML.

A través del objeto *Document* (que en la jerarquía del W3C DOM es el objeto padre del documento HTML) podemos hacer las invocaciones de las funciones que contiene, prácticamente desde cualquier parte de los módulos con los que cuenta el entono. Además de permitirnos también la comunicación con los applets cargados en el entorno.

6.2. Módulo del editor

En lo que respecta al módulo del editor este se encuentra dividido principalmente en tres archivos:

- entorno.svg

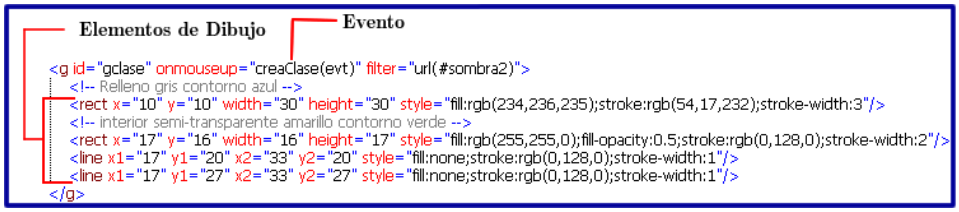


Figura 6.2: Fragmento del archivo entorno.svg

- entorno.js
- varias.js

Con estos tres archivos se realiza la creación del entorno en todo lo que se refiere a la parte gráfica, ya que la mayoría de las funciones se encargan ya sea de elaborar una parte de alguno de los diagramas que se están elaborando o se encargan de realizar alguna de las funciones que se encuentran en el menú principal o la barra de herramientas.

Aunque existen otros archivos, éstos son los más importantes, ya que en ellos esta programada la funcionalidad del entorno, los otros archivos contienen funciones de utilería y elementos estéticos del entorno.

6.2.1. Estructura del archivo entorno.svg

El archivo *entorno.svg* contiene el documento SVG con todos los elementos tales como: gradientes, filtros, estilos, figuras, grupos, etc.; que se utilizan para realizar la pantalla principal del entorno de programación.

Dentro de el se encuentran definidos cuatro lienzos. El primero de ellos: svgLienzoClases, es el lienzo en el cual se agregan los elementos que representan las clases en SVG (líneas, rectángulos, texto y eventos). El segundo es: svgLienzo, en el cual se agregan también los elementos que representan el diagrama de secuencia (líneas, rectángulos, texto y eventos). El tercer lienzo: svgCodigo, es donde se agregan los elementos que representan a la caja de texto donde se almacena el código ADM, y el último lienzo svgLienzoClasesCompletas, contiene los elementos necesarios para representar a las clases pero con toda la información que contienen, es decir para mostrar los atributos con su tipo y visibilidad; y los métodos con su visibilidad y argumentos que reciben.

Ya que los elementos SVG contenidos en el archivo así lo permiten, además de contener únicamente especificaciones de diseño, se tiene también la definición de los eventos a los que responden dichos elementos, tal como se muestra en la Figura 6.2

6.2.2. Estructura del archivo entorno.js

Como se mencionó en el apartado anterior, los elementos SVG pueden responder a diferentes eventos, por lo que dentro del archivo entorno.js se encuentran todas las funciones de Java Script asociadas a dichos elementos con lo que se logra dar la interactividad necesaria al entorno de programación visual. Allí se encuentran las funciones para poder dibujar las clases, los objetos, los mensajes entre objetos, etc. Podría decirse que dentro de este archivo se encuentra el código capaz de generar los diagramas de secuencia y su visualización.

Las variables globales y una breve descripción acerca del valor que almacena se muestran a continuación en la Tabla 6.1

Tabla 6.1: Variables globales

Variable	Descripción
command	Almacena el comando que se esta ejecutando en ese momento
countAnim	Variable que contabiliza el número de animaciones que se van realizando
countClick	Variable para saber el número de veces que se ha presionado el botón del mouse
EDIT	Constante que indica el modo de Edición
Fill	Constante del color de relleno de los rectángulos
heightFocus, widthFocus	Alto y ancho predeterminados del foco de control
inScrollHor, inScrollVer	Arreglos para manejar el tamaño de las barras de desplazamiento horizontales y verticales
J	Contador del número de mensajes
lengthGreater	Almacena la longitud mayor de la línea de tiempo
MARGINY	Constante que contiene la distancia en Y entre los focos de control

Continua en la siguiente página...

Tabla 6.1 – Continuación

Variable	Descripción
methodsClass	Arreglo que almacena el nombre de las métodos de las clases que se van cargando
mode	Variable que indica si se esta en modo de Edición o visualización
nameClasses	Arreglo que almacena el nombre de las clases que se van cargando
nodeSource	Variable para almacenar el nodo origen (donde se dio click primero)
numClass	Almacena el número de clases que se han colocado en el diagrama
numObject	Almacena el número de objetos dibujados en el diagrama de secuencia
parametersClass	Arreglo que almacena el nombre de los parámetros de las clases que se van cargando
selctedIcon	Almacena el icono seleccionado
send, sendRecursive, return, finalize, par, newObject	Constantes boleanas que indican cual es el comando en curso
sSend	Almacena el código de los diferentes objetos
stroke	Constante del color de línea de los rectángulos
stroke_width	Constante del grosor de línea de los rectángulos
svgdoc	Contiene la raíz del documento SVG
svgDocumentAct	Contiene la raíz del documento svg actual

Continua en la siguiente página...

Tabla 6.1 – Continuación

Variable	Descripción
toolTipData	Arreglo que contiene las leyendas de la ayuda visual de los focos de control
VISUALIZE	Constante que indica el modo de Visualización
X	Valor constante para la posición en x de los objetos
xClass	Valor constante para la posición en x de las clases
xSource, ySource, xEnd, yEnd	Variables para almacenar las coordenadas del objeto origen y del objeto destino
Y	Valor constante para la posición en y de los objetos
yClass	Valor constante para la posición en y de las clases

Algunas de estas variables también son utilizadas dentro del archivo *varias.js* que se describirá en la siguiente sección.

Las funciones que se encuentran en este módulo se describen a continuación en el cuadro 6.2, éstas son solo algunas de las funciones más representativas, las cuales sirven para hacer la representación de los elementos del entorno gráfico, tales como íconos, menús, lienzos, ventanas emergentes y demás objetos que conforman dicho entorno, También se utilizan para la creación de los elementos que representan de manera visual a cada una de las partes que conforman a las clases representadas o a los diagramas de secuencia. Además se encuentran las funciones encargadas de responder a la interacción del usuario con los elementos del diagrama que esta editando y del entorno en general, como por ejemplo el manejo de las barras de desplazamiento, las ventanas emergentes, etc.

La mayoría de estas funciones se utilizan también para hacer la visualización de los diagramas creados.

También se encuentran las funciones encargadas de hacer la representación en XML de los elementos que conforman los diagramas para posteriormente poder almacenarlos o cargarlos según sea el caso.

Tabla 6.2: Descripción de la funciones del archivo entorno.js.

Entrada-Salida	Acción
`initialize(evt)` Recibe la constante evt con el evento inicial	Se encarga establecer las condiciones iniciales en el entorno para comenzar a utilizarlos
`loadClass (nameClass,id)` Recibe el nombre de la clase y el número de clase	Se dibuja la clase en el panel de dibujo y también se carga toda la infomación de la misma(metodos y atributos)
`testInvoke (opc)` Recibe la opción del programa	Hace la invocación del programa en java correspondiente
`visualize (strTrace)` Recibe la cadena con los comandos en ADM del programa en java que se este ejecutando en ese momento	Se encarga de hacer la traducción de las operaciones en ADM a la representación interna para poder hacer la visualización de dichas operaciones
`terminateObjects ()`	Empareja todas las líneas de vida de los objetos (en modo visualización) para posteriormente finalizarlos
`returnMessage (evt,mensaje)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
Recibe la variable con el evento ocurrido, y el mensaje a retornar	Obtiene el nodo destino para poder dibujar la flecha con el mensaje de retorno, en la posición correcta respecto al nodo origen. El mensaje se agrega a la rama de mensajes y también de genera el código correspondiente en ADM. Actualiza el contador de mensajes
`sendMessage (evt,mensaje)` Recibe la variable con el evento ocurrido, y el mensaje a enviar	Obtiene el nodo destino para poder dibujar la flecha con el mensaje a enviar, en la posición correcta respecto al nodo origen. El mensaje se agrega a la rama de mensajes y también de genera el código correspondiente en ADM. Actualiza el contador de mensajes
`changeNameClass (nameClass,id)` Recibe el nuevo nombre de la clase y su identificador	Renombra la clase de acuerdo al parámetro especificado
`changeNameObject (ind,id)` Recibe el índice de la clase y el identificador del objeto	Cambia el nombre del objeto para ligarlo con el nombre de la clase a la que pertenece
`showToolTipMsg (evt)` Recibe el objeto donde ocurrió el evento	Muestra la ayuda visual en el foco de control dependiendo del comando seleccionado
`hideToolTipMsg (evt)` Recibe el objeto donde ocurrió el evento	Oculta la ayuda visual del foco del control
`changeFilter (idIcono, mouse)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
Recibe el identificador del icono seleccionado y el evento del mouse que ocurrió	Cambia el filtro de los iconos de la barra de herramientas de acuerdo al evento ocurrido
`changeLenghtLine (linea,inc)` Recibe el objeto tipo línea y su nueva longitud Se verifica si se esta en modo de edición o de visualización.	Cambia el atributo y2 de la línea para darle una nueva longitud
`animateChangeLengthLine (linea,inc)` Recibe el objeto tipo línea y su nueva longitud	Hace la animación en el cambio de longitud de la línea
`changeColor (evt,color)` Recibe el objeto donde ocurrió el evento clic, y el nuevo color	Cambia el color del contorno de un objeto
`createClase ()`	Se encarga de hacer las invocaciones necesarias para crear una clase de manera gráfica
`createG (translate,id)` Recibe una traslación y un identificador del grupo	Se encarga de crear un nuevo grupo g, que se enlazará al lienzo SVG de acuerdo al id recibido
`createGroupMessages (id)` Recibe el identificador para el grupo de mensje.	Se encarga de crear el grupo donde se enlazará algún mensaje creado
`createRect (valx, valy,valwidth,valheight)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
Recibe las coordenadas en x,y, y la dimensión de un rectángulo Retorna el objeto rectángulo SVG creado	Crea el elemento SVG que corresponde a un rectángulo de acuerdo a los argumentos recibidos. También verifica si se esta en modo de edición o de visualización
`animateRect (rectNode,valwidth)` Recibe el nodo del rectángulo y el ancho del rectángulo animar	Realiza la animación de un rectángulo de acuerdo al ancho del mismo, es decir inicia en el ancho 0 hasta llegar a la longitud que se pasó en el argumento
`animateAlpha (attrib)` Recibe el nombre de un atributo Retorna la animación creada	Establece los atributos necesarios para hacer una animación de transparencia de un atributo determinado
`createRectObject (valx, valy, valwidth, valheight)` Recibe las coordenadas en x,y, y la dimensión de un rectángulo Retorna el grupo donde se ligaron los dos rectángulos	Crea los dos rectángulos para representar a un objeto en el diagrama de secuencia
`createRectClass (valx,valy,valwidth, valheight, tipo, id)` Recibe las coordenadas en x,y, y la dimensión de un rectángulo, también el identificador de la clase y que parte de la clase es (nombre, atributos o métodos) Retorna el grupo con todos los elementos creados	Crea los elementos necesarios para realizar los símbolos que aparecen a los lados de la clase para poder visualizar su contenido, además de ir creando cada uno de los rectángulos que conforman la clase
`showMore (evt,vis)` Recibe el objeto donde ocurrió el evento mouseover, y visibilidad del objeto	Oculta o muestra el objeto que indica si se quieren ver todos los elementos de la clase
`showCombo (tipo,id)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
Recibe el tipo de elemento y el id de la clase	Dependiendo de a que ícono de más se le haya dado clic en la clase muestra una lista desplegable con los atributos o métodos o pide el nombre de la clase
`createLine (valx1, valy1, valx2, valy2,valdash)`	
Recibe las coordenadas iniciales y finales para trazar una línea y el tipo de línea	Se encarga de crear el elemento SVG correspondiente a una línea y también verifica el modo
`animateLinea (lineNode, valx1,valy1, valx2,valy2)`	
Recibe la línea, los valores iniciales y finales de la línea a animar	Se encarga de hacer la animación de la línea de manera creciente, es decir, desde el valor inicial hasta alcanzar el valor final
`createArrow (ox,oy, dx, dy,stroke_dasharray sentido)`	
Recibe las coordenadas iniciales y finales de la flecha, el tipo de línea y la dirección de la flecha (izquierda o derecha) Retorna el grupo SVG que contiene la flecha	Crea un los elementos SVG necesarios para representar una flecha en el sentido indicado
`createDestroyLine (valx1,valy1,valx2, valy2)`	
Recibe las coordenadas iniciales y finales de la línea Retorna la línea SVG creada Crea la línea SVG que formará parte del símbolo que representa la destrucción de un objeto.	Verifica también si se esta en modo de visualización
`animateDestroyLine (lineNode, valx1, valy1, valx2, valy2)`	
Recibe el elemento SVG de la línea a animar y las coordenadas íniciales y finales de la misma	Realiza la animación de la línea, de acuerdo a las coordenadas recibidas e incrementa el contador de la animación
`createText (info, x, y, tam, color, id, tipo)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
Recibe la información del texto, su posición x,y, el tamaño de la letra, color de letra, un identificador de texto, y el tipo(mensaje o un texto en cualquier otra parte del diagrama) Regresa el nodo de texto SVG Crea el elemento SVG estableciendo las propiedades de acuerdo a los argumentos recibidos, además de definir la funcionalidad del mismo dependiendo del tipo de texto que se vaya a crear (mensaje u otro tipo texto).	Verifica además si se esta en modo de visualización
`animateText (textNode)` Recibe el nodo de Texto	Se encarga establecer los atributos necesarios del nodo de texto para crear la animación que afecta la transparencia del texto, desde completamente transparente hasta llegar a un color sólido
`createTextClass (info,x,y,tam, color,id)` Recibe el texto, la posición x,y del mismo, el tamaño de letra, color y su identificador.Regresa el nodo de texto creado	Crea el elemento SVG necesario para crear el texto de la clase estableciendo sus propiedades de acuerdo a los atributos recibidos
`createNodeMessage (ox,oy)` Recibe la esquina superior izquierda de la posición de un foco de control Regresa el nodo que contiene el foco de control	Crea los elementos SVG necesarios para crea un foco de control y establece sus propiedades para poder mostrar la ayuda visual que indica las operaciones que se pueden realizar
`createControlFocus ()`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
	Se encarga de encontrar la posición en donde se dibujará un nuevo foco de control y lo muestra.
`changeLengthFocus (focoControl,incy)` Recibe el nodo del foco de control y el nuevo tamaño.	Redimensiona la altura del foco de control
`typeSend ()` Localiza cual es el botón de el panel de operaciones que se presionó para habilitar la bandera correspondiente	
`clickLine (evt)` Recibe el objeto donde ocurrió el evento clic	Se encarga de hacer el conteo de las veces que se ha presionado el botón izquierdo del mouse sobre una línea de tiempo, para determinar cual será la acción a realizar dependiendo del numero de click y tipo de envío que se haya establecido
`destroyObject (obj)` Recibe la línea de vida del objeto que se va destruir	Obtiene la posición inicial y final de donde se dibujará el elemento que representa la destrucción de un objeto y lo muestra. Además crea el código correspondiente y cambia el filtro del icono que corresponde a dicha operación y deshabilita la bandera de la operación
`createNewObject (evt, nombre,mensaje)`	

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
	Se crea el nodo del mensaje (flecha y mensaje) new para la creación del nuevo objeto(línea de vida y rectángulo que lo representa), calcula la posición a partir de la cual se de creará un nuevo objeto a partir de un objeto ya creado con el comando new, le agrega además los atributos con la funcionalidad necesaria y se muestra en la pantalla. Crea también el código ADM correspondiente y cambia el filtro del icono seleccionado
`modifyText (evt)` Recibe el objeto de texto en donde ocurrió el evento click	Obtiene el nodo de texto a modificar y envía un cuadro de diálogo para pedir el nuevo texto con el cual se reemplaza el texto del nodo
`messageToXML (nsend,msg)` Recibe el nombre del nodo desde donde se envia el mensaje y el mensaje	Se encarga de traducir el mensaje a ADM
`createObj (evt,nombre)` Recibe el objeto donde ocurrió el evento clic y el nombre del objeto	Calcula la posición en donde se dibujaran los objetos iniciales, los crea con su línea de tiempo y establece los eventos a los que responderá con la funcionalidad necesaria y los muestra en pantalla y contabiliza el objeto creado en la variable correspondiente
`refreshCanvas (xml)` Recibe el código xml de un archivo	Se encarga de borrar el lienzo de diagramas de secuencia y transforma el código XML a SVG para visualizarlo

Continua en la siguiente página...

Tabla 6.2 – *Continuación*

Entrada-Salida	Acción
`eraseCanvas (nomLienzo)` Recibe el nombre del lienzo a borrar	Elimina todos los elementos SVG que se encuentre en el lienzo SVG que se pasó como argumento
`eraseCode()`	Elimina el código ADM de la caja de texto
`loadXMLToSVG (strXML)` Recibe la cadena que contiene código XML	Transforma la cadena recibida a un objeto XML para poder así hacer la correspondencia a SVG
`XmlTosvg (element,xmlDocument)` Recibe el primer hijo del nodo raíz de un documento XML y el nodo raíz	Transforma los nodos del árbol documento XML a nodos SVG para poder visualizarlos
`scrollCanvasVer (param)` Recibe el sentido de la desplazamiento (arriba o abajo)	Se encarga de hacer el desplazamiento del lienzo de diagrama de secuencia de manera vertical
`scrollCanvasHor (param, index,lienzo)` Recibe el sentido de la desplazamiento (izquierda o derecha), el índice de la barra, y el nombre del lienzo	Se encarga de hacer el desplazamiento de manera horizontal del lienzo correspondiente, verificando los límites del mismo

6.2.3. Estructura del archivo general.js

En este archivo se encuentran las funciones que involucran al menú del entorno gráfico que se muestra ne la figura 6.3 y también funciones para la inicialización de los lienzos

que se describieron en la sección anterior. Estas funciones hacen uso de algunas de las variables globales que aparecen en el cuadro 6.1

Tabla 6.3: Descripción de la funciones del archivo general.js.

Entrada-Salida	Acción
`choiceMenu (op)` Recibe la opción del menú seleccionada	Selecciona la función correspondiente a la opción de la barra de menú seleccionada
`new()`	Se encarga de inicializar los lienzos(clases, secuencia y código) para poder elaborar un nuevo diagrama.
`save()`	Muestra el cuadro de diálogo correspondiente donde se introducirá el nombre del archivo en el cual se almacenará el diagrama de secuencia elaborado como documento SVG
`saveClass ()`	Muestra el cuadro de diálogo correspondiente donde se introducirá el nombre del archivo en el cual se almacenará el diagrama de clases elaborado como documento SVG
`showCompleteClassDiagram ()`	Da una vista detallada del diagrama de clases generado, para poder visualizar todos sus atributos y métodos
`initializeCanvasSeq (creaLienzo)`	

Continua en la siguiente página...

Tabla 6.3 – Continuación

Entrada-Salida	Acción
Recibe la opción para crear o no el lienzo	Elimina los elementos dibujados en el lienzo del diagrama de secuencia y en caso ser necesario crea nuevamente el lienzo yt establece los valor iniciales para iniciar un nuevo diagrama de secuencia.
`initializeCanvasClass ()`	Borra el lienzo de las Clases abreviadas y el lienzo de las clases completas, además de establecer los valores iniciales para crear un nuevo diagrama de clases.
`initialValuesClass ()`	Da los valores iniciales a las variables para dibujar un nuevo diagrama de clases
`initialValuesSeq ()`	Da los valores iniciales a las variables para dibujar un nuevo diagrama de secuencia
`changeMode (mod)` Recibe el modo seleccionado	De acuerdo al modo seleccionado cambia a modo de edición o a modo de visualización, ocultando o mostrando la cuadrícula, redimensionando los lienzos, dando valores iniciales, almacenando el diagrama editado, etc.
`initializeSquare (valor)` Recibe el valor de oculto o visible	Oculta o muestra la cuadricula del lienzo de diagrama de secuencia
`saveEdition ()`	Almacena en una cadena el documento SVG

Continua en la siguiente página...

Tabla 6.3 – Continuación

Entrada-Salida	Acción
`loadEdition ()`	Carga el diagrama guardado previamente en el lienzo en modo de edición
`resizeCanvas (x,y, width, height, viewBox)` Recibe las coordenadas de la esquina superior izquierda del lienzo, la nueva dimensión y las coordenadas del viewbox a visualizar	Establece las propiedades del lienzo SVG del diagrama de clases para poder cambiar su dimensión.
`resizeScrolls (x,y, escala,x2, y2, escala2)` Recibe las coordenadas de la esquina superior izquierda, la escala para el scroll vertical y los mismos datos pero para el scroll horizontal	Cambia el atributo de transformación para cambiar la escala y posición de las barras de scroll.
`initializeMessages()`	Inicializa los nodos en donde se almacenan los mensajes del diagrama de secuencia.
`showToolTip (evt,mensaje)` Recibe el objeto donde ocurrió el evento mouseover y el mensaje a desplegar	Despliega las ayudas visuales de los iconos de la barra de herramientas.
`hideToolTip ()`	Oculta las ayudas visuales de los iconos de la barra de herramientas.
`outEventMenu (tipo,evento)` Recibe el id del objeto y el evento del mouse ocurrido	Cambia el filtro de el icono seleccionado y oculta el tooltip
`showAbout ()`	

Continua en la siguiente página...

Tabla 6.3 – Continuación

Entrada-Salida	Acción
	Muestra la ventana de acerca de ... de la aplicación.

La referencia cruzada de las variables globales con las funciones descritas en las dos secciones anteriores se muestra en el cuadro 6.4.

Tabla 6.4: Referencia cruzada de las variables globales.

Variable	Funciones
command	Visualiza, enviaMensaje, regresaMensaje, muestraToolTipMsg
countAnim	animacambiaLongitudLinea, animaRect, animaLinea, animaLineaDestruir
countClick	clickLinea
EDIT	cambiaLongitudLinea, creaLine, creaRect, creaText, creaLineDestruir, cambiaModo
Fill	creaRect
heightFocus, widthFocus	regresaMensaje, enviaMensaje
inScrollHor, inScrollVer	
j	regresaMensaje, enviaMensaje
lengthGreater	cambiaLongitudLinea
MARGINY	muestraToolTipMsg

Continua en la siguiente página...

Tabla 6.4 – Continuación

Variable	Funciones
methodsClass	loadClass
mode	cambiaLongitudLinea, creaLine, creaRect, creaText, creaLineDestruir, cambiaModo
nameClasses	loadClass, cambiaNombreObjeto
nodeSource	regresaMensaje, enviaMensaje, clickLinea
numClases	creaClase
numObjetos	creaObj,creaNuevoObj
parametersClass	loadClass
selctedIcon	cambiaFiltro
send, sendRecursive, return, finalize, par, newObject	tipoEnvio, clickLinea, destruyeObjeto
sSend	regresaMensaje, enviaMensaje, destruyeObjeto
stroke	creaRect
stroke_width	creaRect
svgdoc	lista, promptSVG

Continua en la siguiente página...

Tabla 6.4 – Continuación

Variable	Funciones
svgDocument	cambiaNombreClase, cambiaNombreObjeto, ocultaToolTipMsg, cambiaFiltro, creaRect, animaAlpha, creaRectObjeto, creaLine, animaLinea, creaFlecha, creaLineDestruir, animaLineaDestruir, creaText, creaTextClase, creaFocoControl, destruyeObjeto, redefineTexto, creaObj, borraLienzo, XmlTosvg
svgDocumentAct	inicializa, lista, promptSVG, creaG, creaGrupoMensajes, creaRectClase, creaFlecha, cargaXMLToSVG, redimensionaScrolls, redimensionaLienzo
toolTipData	muestraToolTipMsg
VISUALIZE	cambiaLongitudLinea, creaLine, creaRect, creaText, creaLineDestruir, cambiaModo
X	creaClase, creaNuevoObjeto, creaObj
xClase	loadClass
xSource, ySource, xEnd, yEnd	clickLinea
Y	creaClase, creaNuevoObjeto, creaObj
yClass	loadClass

6.2.4. Estructura de la clase GuardaUsuario

Esta clase es un applet que se carga al momento de ejecutar la aplicación, aunque no se encuentra directamente dentro de los archivos que conforman al editor, se encarga de almacenar el diagrama editado. Esta clase se encuentra embebida dentro del archivo *MOON.html*, pero través del W3C DOM se puede acceder a sus métodos desde algunas de las funciones de Java Script que forman parte del editor.

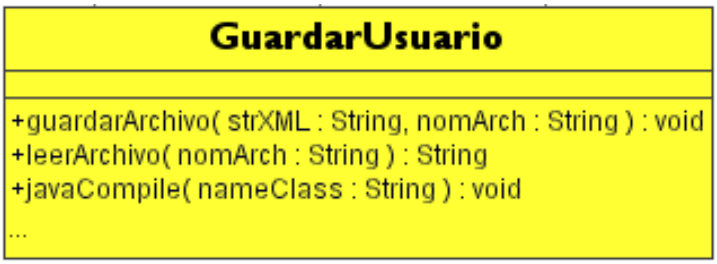


Figura 6.3: Clase GuardarUsuario

Otra de las funciones que realiza esta clase es la de leer el archivo XML, previamente almacenado para pasar el resultado de esa lectura a la función de Java Script que la haya invocado y poder mostrar el diagrama en la pantalla. En la figura 6.3 se muestra el diagrama de esta clase.

Esta clase almacena tanto las clases editadas, así como los diagramas de secuencia editados.

6.3. Módulo del Compilador

Este módulo es el encargado de traducir el diagrama de secuencia del programa representado a las reglas ECA y a partir de ellas al código Java que le corresponde. Se generan realmente dos programas en Java: uno con el código Java simple, es decir, el código en Java que se puede correr en el compilador de Java si ningún problema; el otro es el código en Java que se necesita para poder hacer hacer la visualización del mismo, esto debido a que se necesitan de algunas instrucciones especiales para que se puedan generar los comandos para la visualización.

6.3.1. Estructura del archivo compilador.js

Tabla 6.5: Descripción de la funciones del archivo compilador.js.

Entrada-Salida	Acción
`invokeCompiler(file)` Recibe el nombre del archivo que se va a compilar. Como salida da las reglas ECA y los programas en Java	Se encarga de invocar a los métodos necesarios para realizar la compilación de las reglas ECA al programa Java
`strToDOM2(strDiagram)` Recibe la cadena donde están almacenados los comandos del diagrama	Esta función convierte la cadena en una estructura DOM
`visitAttribute(element,attribute)` Recibe un elemento y uno de sus atributos	Selecciona la acción de buscar un objeto preconstruido (elemento ïnitial) o construye un nuevo objeto (elemento ñew).
`Pair()`	Constructor de pares de nombre y de clase de un objeto
`RuleContext()`	Constructor del contexto (conjunto de objetos con sus códigos, es decir, reglas)
`SeqRule(name,type)` Recibe el nombre y el tipo de la regla	Constructor de reglas
`ActionSequence()`	Define la secuencia de acciones a realizar representado como un arreglo
`ClassContext()`	Define las asociaciones de nombres de clase y definiciones de clase para facilitar la búsqueda por el nombre de clase.

Continua en la siguiente página...

Tabla 6.5 – Continuación

Entrada-Salida	Acción
`AltRule(name,type) ()` Recibe el nombre y clase (tipo) de la regla	Constructor de reglas que se aplican de manera excluyente. Las reglas elementales se combinan como alternativas siempre que compartan el mismo evento. Las condiciones deben ser mutuamente excluyentes, aunque esto no lo verifica el compilador.
`Action(name,source,target)` Recibe el nombre, el origen y el destino de una acción	Constructor de acciones elementales. Las acciones elementales pueden ser: enviar, recibir, asignar un valor a una variables, etc.
`doActionInit(element)` Recibe la información de la acción de inicialización	Realiza la acción de inicializar un objeto preconstruido
`doActionNew(element)` Recibe información para la creación del objeto	Realiza la acción de crear e inicializar un objeto nuevo
`doActionAssignment(rule,expression)` Recibe la regla en la que se creará una nueva variable local para asignarle el valor de la expresión	Realiza la acción de asignar el valor de una expresión a una variable
`doActionCall(element)` Recibe información sobre la invocación	Realiza la acción de invocar un método identificado por su nombre y por el nombre de su clase usando una lista de parámetros. Esta invocación corresponde con una comunicación síncrono de envío de parámetros y espera de resultados.

Continua en la siguiente página...

Tabla 6.5 – Continuación

Entrada-Salida	Acción
`doActionSend(element)` Recibe información para realizar una invocación sin esperar respuesta	Realiza la acción de enviar un mensaje a un método identificado por su nombre y por el nombre de su clase usando una lista de parámetros. Esta invocación corresponde con una comunicación asíncrono de envío de parámetros sin espera de resultados.
`doActionReturn(element)` Recibe información sobre el valor que se va a regresar al invocador	Realiza la acción de regresar el valor calculado por un método al invocador y dar por terminada la ejecución del método.
`doActionReply(element)` Recibe información sobre el mensaje con el que se va a responder al invocador	Raliza la acción de regresar el valor calculado por un método al invocador y continuar con la ejecución del método.
`doActionReceive(element` Recibe información sobre la estructura de la invocación	Realiza la acción de recibir la solicitud de ejecución de un método y de recibir los parámetros
`doActionAnnotate(element)` Reciba la expresión que se usa en la anotación	Establece condiciones o restricciones lógicas sobre el contenido de los parámetros de las invocaciones y de las variables en asignamientos
`generateAltRule()`	Genera reglas considerando múltiples casos cada uno de ellos definido por una regla elemental (secuencial)

Continua en la siguiente página...

Tabla 6.5 – Continuación

Entrada-Salida	Acción
`doActionDefine(element)` Recibe información sobre un caso particular	Indica que regla se usará como parte de una definición por casos

Tabla 6.6: Descripción de las funciones que generan las reglas en el lenguaje ADM

Entrada-Salida	Acción
`getSeqCode()`	Genera el código para una secuencia de instrucciones
`getSeqRule(rule)` Recibe la regla	Genera el código para una regla secuencial elemental
`getAltCode()`	Genera el código de la instrucción alternativa
`getAltRule(rule)` Recibe la regla	Genera el código de las reglas que se activan con el mismo evento combinadas bajo una sola regla
`getEvent(event)` Recibe el evento	Genera el código para el evento de la regla dado por el evento observado por el objeto receptor en la forma de invocación a un método

Continua en la siguiente página...

Tabla 6.6 – Continuación

Entrada-Salida	Acción
`getCondition(condition)` Recibe la condición	Genera el código para la condición de la regla a partir de las anotaciones hechas sobre el contenido de los parámetros en una invocación o de asignamientos
`getAction(action)` Recibe la acción	Genera el código para la acción de la regla al invocar los métodos especializados para cada tipo de acción
`getLocalVars(action)` Recibe la acción	Genera el código para la declaración de las variables locales que se usan en la regla junto con las variables auxiliares que se generan para simplificar expresiones que involucran invocaciones anidadas en una secuencia de invocaciones y asignamientos
`getBasicAction(action), getReceiveAction(action), getCallAction(action), getSendAction(action), getInitAction(action), getReturnAction(action),getReplyAction(action), getNewAction(action)` y `getAssignAction(action)` Reciben la acción	Generan el código para la operación sugerida

Tabla 6.7: Descripción de las funciones que generan el código en Java

Entrada-Salida	Acción
`getJAltCode(moon)`	

Continua en la siguiente página...

Tabla 6.7 – Continuación

Entrada-Salida	Acción
Recibe la opción de generar código para el mediador	Genera código Java para la clase
`getJAltRule(rule,moon)` Recibe la regla y la opción de generar código para el mediador	Genera código Java para el método que agrupa a todas las alterativas
`getJFormalPars(list,moon)` Recibe la lista de parámetros formales y la opción de generar código para el mediador	Genera código para los parámetros formales del método
`getJEvent(event,moon)` Recibe el evento y la opción de generar código para el mediador	Genera código Java para el encabezado del método
`getJCondition(condition,moon)` Recibe la condición y la opción de generar código para el mediador	Genera c'odigo Java para cada condición usando `if-else`
`getJAction(action,moon)` Recibe la acción y la opción de generar código para el mediador	Delega la generación de código a las acciones especializadas
`getJLocalVars(action,moon)` Recibe la acción y la opción de generar código para el mediador	Genera código Java para las variables locales auxiliares
`getJBasicAction(action,moon)`,`getJReceiveAction(action,moon)`, `getJCallAction(action,moon)`, `getJSendAction(action,moon)`, `getJReturnAction(action,moon)`,`getJReplyAction(action,moon)`, `getJInitAction(action,moon)`,`getJNewAction(action,moon)`, `getJAssignAction(action,moon)` Recibe la acción y la opción de generar código para el mediador	Obtiene el código en Java para la operación sugerida

6.4. Módulo del mediador

En este módulo se realiza la ejecución del programa en Java generado por el compilador. A partir de la ejecución del programa se construye el archivo de comandos necesario para enviarlo hacia el visualizador para que este las represente en su correspondiente código en SVG y se pueda visualizar la animación del programa en Java ejecutado. La Figura 6.4 muestra la descripción de la clase mediador y en la tabla 6.8se describen los métodos que ésta clase contiene.

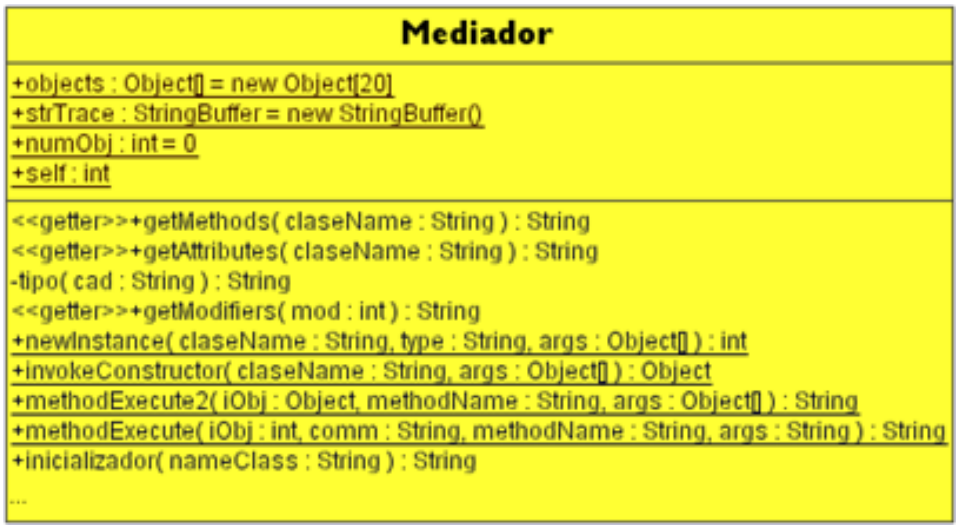


Figura 6.4: Clase Mediador

Este módulo está compuesto por la clase llamada Mediador y también por la función *visualize*() descrita en la sección anterior, para poder tomar del código de los comandos generados para cada uno de los elementos y así poder visualizarlos utilizando las funciones que generan a cada uno de los elementos correspondientes.

6.4.1. Estructura de la clase Mediador

En esta clase se hace uso de las clases del paquete java.lang.reflect, para poder obtener los atributos y métodos de las clases utilizadas, así como también para ejecutar los métodos del programa generado por el compilador y crear nuevas instancias de las clases.

Uno de los métodos más importantes de esta clase es el método methodExecute, ya que a partir al momento de ejecutar un método del programa se crea la sentencia que

contiene el método invocado, quien lo invoco, los valores pasados y si retorna algún valor, con esta información y a través de la función visualize() se puede obtener la animación del diagrama con los valores reales del programa ejecutado.

Tabla 6.8: Descripción de los métodos de la clase Mediador

Entrada-Salida	Acción
`public String getMethods(String claseName)`	
Recibe el nombre de una clase	Genera código Java para la clase
`public String getAttributes(String claseName)`	
Recibe el nombre de la clase	Obtiene a través de las clases de reflexión los métodos de la clase y los regresa en una cadena
`String getModifiers(int mod)`	
Recibe el entero que representa a los modificadores que tiene un método o un atributo	Regresa en una cadena los modificadores de un método o atributo de una clase determinada
`public static int newInstance (String claseName,String type,Object[] args)`	
Recibe el nombre de la clase, el tipo para la creación de la nueva instancia y los argumentos del constructor	Crea una nueva instancia de una clase, dependiendo del tipo de instancia que se crea se genera el comando correspondiente para su visualización posterior.
`public static Object invokeConstructor (String claseName,Object args[])`	
Recibe el nombre de la clase y los argumentos del constructor	Crea una nueva instancia de la clase y la retorna
`public static String methodExecute (int iObj,String comm,String methodName,String args)`	
Recibe el identificador del objeto, el comando, el nombre del metodo y sus argumentgos	Invoca el método indicado del objeto referenciado por el identificador, además de que generara la secuencia de comandos necesaria para la visualización.

Continua en la siguiente página...

Tabla 6.8 – Continuación

Entrada-Salida	Acción
`public String inicializador(String nameClass)`	
Recibe el nombre de la clase	Este método es el que se invoca al iniciar la ejecución de los programas generados. Se encarga de invocar al método start de la clase que se le pasa.
`public void javaCompile(String nameClass)`	
Recibe el nombre de la clase	Invoca al compilador de Java para que éste compile la clase requerida.

6.5. Módulo del visualizador

En esta parte es donde se realiza la parte de la visualización del programa generado. Los archivos, funciones y variables utilizadas en este módulo son las mismas que las que se utilizan para el módulo de edición con algunos cambios. El cambio más significativo es la animación del diagrama de secuencia, ya que cuando se esta editando no se observa ninguna animación, pero cuando se esta visualizando el programa generado cada una de las acciones realizadas (creación de objetos, envío de mensajes, destrucción de objetos y retorno de mensajes) se muestra de forma animada. Esto se puede realizar ya que se tiene la variable mode que es la que se encarga de almacenar el modo en que se encuentra el entorno de programación, y puede tomar el valor de EDIT o VISUALIZE, con esta información cada una de las funciones puede realizar las acciones correspondientes dependiendo del modo en que se encuentre. De esta manera podemos darnos cuenta que la mayor parte del código del editor puede ser reutilizado en el visualizador agregando las instrucciones para lograr la animación de los elementos SVG.

Dentro del archivo entorno.js, se encuentran descrito en la sección 6.2.2 se encuentran las funciones que son de uso exclusivo del editor.

Tabla 6.9: Variables del modulo de visualización

Variable	Descripción
countAnim	Variable que contabiliza el número de animaciones que se van realizando en el modo de visualización.

Tabla 6.10: Funciones del visualizador

Entrada-Salida	Acción
`animateChangeLengthLine (linea,inc)`	
Recibe el objeto tipo línea y su nueva longitud	Hace la animación en el cambio de longitud de la línea.
`animateRect (rectNode, valwidth)`	
Recibe el nodo del rectángulo y el ancho del rectángulo animar	Realiza la animación de un rectángulo de acuerdo al ancho del mismo, es decir inicia en el ancho 0 hasta llegar a la longitud que se pasó en el argumento.
`animateAlpha (attrib)`	
Recibe el nombre de un atributo	Establece los atributos necesarios para hacer una animación de transparencia de un atributo determinado.
`animateLinea (lineNode, valx1,valy1, valx2,valy2)`	
Recibe la línea, los valores iniciales y finales de la línea a animar	Se encarga de hacer la animación de la línea de manera creciente, es decir, desde el valor inicial hasta alcanzar el valor final.
`animateDestroyLine (lineNode, valx1, valy1, valx2, valy2)`	
Recibe el elemento SVG de la línea a animar y las coordenadas íniciales y finales de la misma	Realiza la animación de la línea, de acuerdo a las coordenadas recibidas e incrementa el contador de la animación

Continua en la siguiente página...

Tabla 6.10 – Continuación

Entrada-Salida	Acción
`animateText (textNode)` Recibe el nodo de Texto	Se encarga establecer los atributos necesarios del nodo de texto para crear la animación que afecta la transparencia del texto, desde completamente transparente hasta llegar a un color sólido

```
animateNode2.setAttribute("id", "anima"+contAnim);
animateNode2.setAttribute("attributeName", attrib);
animateNode2.setAttribute("attributeType","XML" );
animateNode2.setAttribute("begin", inicio);
animateNode2.setAttribute("dur","500ms" );
animateNode2.setAttribute("from","0");
animateNode2.setAttribute("to", "1");
animateNode2.setAttribute("fill", "freeze");
```

Figura 6.5: Fragmento de código para la animación

Las funciones mostradas en el cuadro 6.10son la encargadas de realizar la animación de los elementos SVG involucrados en el diagrama de secuencia visualizado, básicamente son funciones para animación del texto, de formas o redimensionamiento. También aquí se encuentran las funciones para poder controlar la visualización, es decir los controles para pausar, detener o ejecutar la animación.

Para realizar las animaciones se utilizan los atributos de animación que provee SVG, en ellos debemos especificar que atributo hay que animar, cuando inicia la animación, su duración, los valores iniciales y finales que tomará el atributo animado, etc., un ejemplo de este código se muestra en la Figura 6.5. De esta manera podemos colocar el código exclusivamente en los elementos que se desean animar.

Capítulo 7

Conclusiones

En esta tesis se ha abordado el problema de simplificar el proceso de desarrollo de software al conjuntar las etapas de diseño y documentación en la programación. La solución consiste en construir un entorno de programación visual que utilize diagramas de secuencia UML para diseñar programas y documentar su ejecución mediante su visualización. La solución utiliza el lenguaje y modelo de programación `ADM` para representar los elementos fundamentales de los diagramas de secuencia de UML. Esta representación posee un modelo de programación bien definido para dar interpretación formal e inambigua a los diagramas de secuencia. Aunque es posible escribir un programa que de interprtación a las construcciones del lenguaje `ADM`, en este trabajo se ha adoptado el enfoque de transformaciones entre modelos que se está convirtiendo en el enfoque dominante en la práctica desde la aparición de MDA (Model Driven Architecture). En cualquier caso, el entorno de programación utiliza los diagramas de secuencia para visualizar ya sea su interpretación o su ejecución. La adopción del enfoque de transformaciones conduce generalmente a elegir a lenguajes ampliamente aceptados como Java como destino del proceso de transformación.

El modelo abstracto de `ADM` consigue no solo neutralidad en la especificación en relación al sistema operativo o lenguaje de programación sino que también ofrece un enorme poder expresivo. La expresividad de `ADM` proviene de la conjunción de los modelo de programación deductivo e interactivo. El modelo interactivo, propio de los sistemas reactivos como los sistemas operativos, extiende la funcionalidad y el comportamiento del servidor de aplicaciones interpretando un conjunto de reglas activas que modelan el propósito del sistema. El modelo deductivos, propio de los sistemas basados en conocimiento como los sistemas expertos, incorpora técnicas de razonamiento automatizado para extraer información que se consecuencia lógica de la información contenida en las colecciones de documentos.

En el modelo de programación de `ADM` todo es un documento XML. El modelo in-

teractivo se encarga de administrar las interacciones que tienen lugar una colección de documentos como inserción, eliminación, modificación, envío o recepción. Dichas interacciones se describen en el lenguaje de reglas activas de ADM. Por otra parte, el modelo deductivo se encarga de representar y recuperar información de la colección mediante consultas. Aunque existen diferencias fundamentales entre ambos modelos de programación, en ADM se sugiere que el modelo de regla activa combina los aspectos interactivo y deductivo de las aplicaciones bajo una sola notación consistente y uniforme.

De esta manera, el entorno experimental Moon utiliza UML como lenguaje visual de programación y ADM como modelo de programación. Moon adopta los principios de MDA estableciendo reglas de transformación entre ADM y Java, aunque es posible escribir transformaciones a otros lenguajes como Pascal, C, C++ o C#, entre otros. Al mismo tiempo, define un lenguaje neutral basado en XML para recuperar los eventos más importantes de la ejecución del programa para su posterior visualización. El acoplamiento estrecho entre los componentes de Moon facilita el proceso de edición, compilación, ejecución y visualización con lo cual se ofrecen evidencias de que es posible conjuntar las etapas de diseño y documentación en la programación en forma independiente del lenguaje y de la plataforma operativa.

7.1. Contribuciones

Al comparar Moon con el trabajo relacionado se puede afirmar que este trabajo contribuye en:

1. La adopción de los diagramas UML de secuencia como lenguaje de programación visual.

2. La construcción de un entorno de programación que permite editar, compilar, ejecutar y visualizar programas descritos en forma independiente del lenguaje y de la plataforma operativa.

3. La documentación de los aspectos dinámicos de un sistema mediante la visualización de sus interacciones (aún cuando los programas no hayan sido producidos en el entorno).

A continuación discutiremos brevemente las contribuciones.

La adopción de los diagramas de secuencia de UML como lenguaje de programación visual parece ser una propuesta original no declarada explícitamente como característica de las herramientas conocidas de diagramación para UML. La posibilidad de usar UML como lenguaje de programación no ha sido explorado seguramente porque los diagramas

fueron concebidos para diseñar y documentar un sistema de software y no para programarlo. La incapacidad de usar UML como lenguaje visual de programación se debe a que no cuenta con un modelo de programación definido sobre una semántica formal bien establecida para cada uno de sus tipos de diagramas. Por tal razón, el primer paso a seguir fue adoptar un modelo de programación cuyas abstracciones posean una interpretación precisa. No obstante que `ADM` es un lenguaje experimental sobre el cual queda mucho trabajo por hacer, posee características ideales para la programación de aplicaciones en Internet que se caracterizan por el interacambio de información de documentos XML, la coordinación entre componentes heterogeneos y la capacidad de extraer información que se derive lógicamente del contenido de las colecciones de documentos XML disponibles, ofreciendo además características como orientación a aspectos, movilidad, persistencia y seguridad.

Dotado del modelo de programación de `ADM`, a los diagramas de secuencia de UML se les puede atribuir un significado inambiguo suceptible de automatización. La atribución de significado de cada elemento de un diagrama de secuencia toma la forma de una plantilla en `ADM` que deben llenarse en forma recursiva con la información detallada proveniente del diagrama. Una vez concluido el proceso de traducción, el programa queda listo para su interpretación (por la máquina abstracta de `ADM`) o para su traducción a otro lenguaje (como C, C++ o Java) de manera que preserve el significado deseado para el programa. Si bien es cierto que existen varias herramientas que produce código con mayor o menor grado de sofisticación y de calidad, los trabajos relacionados no utilizan los diagramas para retroalimentar al diseñador o programador visualizando la ejecución de lo que el espera obtener de dichos diagramas. Por tal motivo, los diagramas de secuencia de UML se han modificado para convertirlos en un lenguaje de programación que es computacionalmente completo ya que posee los esquemas de composición secuencial y recursiva sobre cualquier conjunto de operaciones básicas (cero, sucesor, igualdad con cero, etc.). Los ejemplos que se han mostrado en esta tesis demuestran el estilo de programación que resulta de la estructura de los diagramas de secuencia: organizados por objetos, por casos, secuenciales y con pocas decisiones.

Aunque la programación visual con diagramas de secuencia ofrece muchas ventajas, en el presente trabajo se han tomado decisiones de diseño que pueden dificultar la programación para aplicaciones que requieren de la evaluación de muchas condiciones, porque cada una de ellas se representa por un diagrama. Las dificultades son inherentes a los diagramas de secuencia ya que solamente muestran la secuencia de acciones sin tener encuenta las decisiones involucradas: incluir las decisiones llevaría a un crecimiento exponencial en el número de diagramas requeridos de acuerdo al número de decisiones involucradas. Por ejemplo, la función factorial se define sobre dos casos (base y recursivo) y cada caso requiere de un diagrama. Si sobre cada uno de estos casos se definiera a su vez otros dos casos, entonces el número de diagramas requeridos para definir el

comportamiento de la función podría aumentar a cuatro. Otro problema sobre el que se adoptó una decisión simple de diseño es el de restringir los tipos de datos usados en los mensajes al tipo entero únicamente. A diferencia del anterior, esta decisión de diseño no representa una dificultad insuperable ya que se puede incorporar un sistema de verificación de tipos para ofrecer un sistema estricto de tipos. Sin embargo, por limitaciones de tiempo para este proyecto el sistema de tipos no fue incorporado. Siendo Moon un entorno experimental, este trabajo no ha perseguido tampoco representar en `ADM` cada aspecto de los diagramas de secuencia de UML sino solamente aquellos que son los más característicos.

En relación al entorno integrado de programación visual la contribución radica en demostrar que un lenguaje visual erigido sobre un estándar condce a un nivel de mayor de abstracción e independencia del lenguaje y de la plataforma operativa. El entorno demuestra que es posible conceptualizar un algoritmo usando la notación gráfica de los diagramas de secuencia. Esta característica hace a Moon un entorno único por su enfoque ya que generalmente los trabajos relacionados buscan generar código en algún lenguaje sin esperar visualizar su ejecución en el mismo diagrama. La falta de integración en las etapas de diseño, programación y visualizacón conduce a que los programadores se enfoquen al código producido por los diagramas y no sobre los diagramas mismos, dando lugar a los bien conocidos problemas de actualización o de consistencia entre diagrama y código, proliferación de versiones no vigentes, confusiones en el equipo de programación, etc. El entorno de programación, si bien bastante incompleto, permite ilustrar que es posible mejorar el proceso de software unificando las etapas de diseño, codificación, programación, depuración (mantenimiento) y documentación en una sola etapa, bajo la forma de un lenguaje visual de muy alto nivel.

Finalmente, en relación a visualizar la ejecución de programas no desarrollados en el entorno también merece calificarse como una contribución en el área de visualización de software. Los diferentes enfoques de visualización adoptados por los trabajos relacionados generalmente definen elementos básicos y de composición propios del lenguaje visual por lo que su entendimiento queda limitado al reducido número de usuarios del sistema y del lenguaje. Al contar UML con una gran aceptación y poseer una interpretación bastante intuitiva, la comprensión de la visualización queda virtualmente asegurada. La amplia disponibilidad de información en la forma de herramientas, libros, reportes y artículos técnicos permiten conocer a fondo el lenguaje y poner en práctica los conocimientos adquiridos abordando los muchos problemas y soluciones publicados. Tal diseminación de información asegura un mayor número de usuarios, lo que puede contribuir a una creciente aceptación de UML como lenguaje de programación y a los diagramas de secuencia como paradigma de visualización de software. Por otra parte, un gran número de aplicaciones se pueden beneficiar de esta posibilidad al integrar un servicio de visualización de las actividades de comunicación no solo de las que se han desarrollado,

pasadas y presentes, sino también de aquellas actividades que quedan pendientes. Por ejemplo, es más fácil conocer el estado de una transacción en forma de diagramas de secuencia, que buscar en un listado de mensajes, sobre todo cuando se trata de dar seguimiento a una operación que involucra a varios participantes.

7.2. Trabajo Futuro

Este trabajo abre varias posibilidades de trabajo futuro, sobre las cuales comentaremos brevemente. Al utilizar solo un tipo de diagrama de UML, el siguiente paso es el de incorporar los otros tipos de diagramas en el entorno. Sin embargo, queda abierta la pregunta de cuáles son los diagramas de UML que son candidatos a convertirse en lenguajes visuales de programación. La respuesta a la pregunta puede incluir a los diagramas de actividades y de interacción pero sería objetable que incluyera a los diagramas de casos de uso, por ejemplo. En forma similar, para cada diagrama se puede preguntar cuáles son los elementos que se pueden interpretar con precisión. Ambas cuestiones cubren decisiones de diseño importantes que son sin duda temas de discusión.

Aún con el subconjunto de UML que se ha considerado queda bastante trabajo por hacer para convertir `ADM` en un lenguaje de programación viable. En relación al problema de la complejidad en el número de diagramas requeridos para definir completamente un algoritmo, se pueden incorporar elementos de los diagramas de actividades de UML que incluyen decisiones con el fin de agrupar bajo una decisión a varios diagramas como casos. Esta solución representaría una desviación substancial del estándar de UML la cual sería inaceptable por la comunidad de Ingniería de Software pero del mayor interés para la comunidad de los lenguajes visuales de programación.

Debe reconocerse que el entorno de progamación Moon es reducido en capacidades cuando se le compara con sus alternativas comerciales. Entendiendo que Moon es un proyecto de investigación, la mayoría de estas limitaciones no constituyen problema alguno. Si la idea de usar UML como lenguaje de programación es aceptada y si los usuarios (diseñadores, programadores, etc.) encuentran ventajoso la integración de las etapas de desarrollo de software, entonces se pueden planear versiones más sólidas y ambiciosas. En un entorno para producción se buscaría la convergencia de las facilidades que caracterizan a las etapas involucradas. Por ejemplo, el entorno debe incluir un analizador sintáctico y semántico que permita detectar errores durante la edición gráfica, tanto como sea posible, y señalarlos sobre el elemento gráfico correspondiente. Como herramienta de documentación, el entorno debe integrar capacidades de navegación de hipertexto que vincule al programa con otras versiones, que vincule a los métodos usados con los módulos que los definen, que permita recuperar el documento de requerimientos posiblemente representado como diagrama de casos de uso, etc.

Por último pero no de menor importancia, Moon se puede utilizar en la preparación

de cursos de programación, entre otros, por sus facilidades de autodocumentación. Contando con la funcionalidad de un sistema de hipermedia, como en parte ya se encuentra actualmente, Moon puede ilustrar conceptos con programas preconstruidos, verificar el comportamiento y comparar las respuestas que produce el programa. Moon también puede contribuir a descubrir y a ayudar a entender los patrones de interacción comunes en la programación como los esquemas recursivos, cliente-servidor, intermediario, delegado, etc.

Adoptar a los diagramas de UML como lenguaje visual de programación puede conducir a estudios y desarrollos que sean del mayor interés desde el punto de vista tanto de la investigación como de la aplicación.

Apéndice A

Introducción a ADM

A.1. Introducción al lenguaje ADM

El propósito de este apéndice es introducir el lenguaje ADM, mostrando los estilos de programación lógica y dirigida por eventos que se pueden alcanzar. Para tal efecto se desarrolló una aplicación que permite demostrar las capacidades activas y deductivas de ADM. La aplicación corresponde con el conocido problema del parentesco familiar que consiste en definir reglas que permitan deducir cuál es el parentesco que relaciona a dos individuos.

El lenguaje ADM está formado por elementos XML, términos XML, procedimientos lógicos y consultas. Mientras que la parte activa del lenguaje se forma con eventos, acciones y reglas evento-condición y acción. A continuación se especifica los componentes deductivo y activo de ADM.

A.2. Reglas deductivas

El enfoque que se adopta en este trabajo para proporcionar las capacidades deductivas de ADM consiste en extender a los elementos XML con variables lógicas y procedimientos lógicos.

A.2.1. Elementos XML

Los elementos XML consisten del nombre del elemento y de una lista posiblemente vacía de atributos (pares nombre y valor). Los atributos están ordenados lexicográficamente con base en su nombre. La sintaxis de XML establece una distinción entre los tipos texto y cadenas de caracteres. Una constante de tipo texto es una secuencia de

caracteres delimitados por elementos, mientras que una constante de tipo cadena de caracteres es una secuencia de caracteres delimitados por comillas. La diferencia se muestra en la figura A.1, en donde se puede apreciar que el elemento persona está compuesto a su vez por los elementos `nombre` y `edad`, cuyos valores respectivos son `Moni` y `24`, ambos de tipo texto. El elemento `nombre` posee además una lista de atributos aunque dicha lista consiste únicamente del atributo `sexo` que tiene el valor `femenino` y que es de tipo cadena de caracteres.

```
<persona>
    <nombre sexo="femenino">Moni</nombre>
    <edad>24</edad>
</persona>
```

Figura A.1: Elemento XML

La estructura de las listas de atributos y de elementos es muy similar a la estructura de los términos en lógica formal, lo que permite descomponer un documento en sus piezas básicas de información usando los métodos usuales de programación en lógica. En la versión actual de `ADM`, no se dispone de facilidades para denotar elementos y atributos por expresiones XPath. De contar con dichas facilidades se conseguiría simplificar considerablemente el tamaño de las expresiones usadas para extraer la información de elementos y atributos. Sin embargo, desde un punto de vista formal, se puede alcanzar una expresividad comparable a la de XPath con las construcciones de `ADM` que aquí se presentan.

A.2.2. Términos XML

Desde el punto de vista de la programación lógica, los elementos XML corresponden a términos sin variables. `ADM` extiende los elementos XML con la noción de variable lógica para introducir términos XML. La importancia de las variables lógicas radica en que permiten definir las condiciones que debe cumplir la información deducida por el sistema. En `ADM`, las variables lógicas son nombres simbólicos que comienzan con `$` (signo de pesos). Las variables anónimas están permitidas y se designan por `$_` (signo de pesos seguido por una subraya).

En la figura A.2 se escribe el elemento `persona` como un término con variables. Las variables `$S`, `$N` de tipo texto y `$E` de tipo elemento se introducen para establecer condiciones lógicas que debe cumplir el contenido de los documentos con quienes se comparen. Por ejemplo, si los términos XML presentados en las figuras A.1 y A.2 se consideran sintácticamente iguales, entonces las variables `$S` y `$N` deben ser iguales a los

```
<persona>
    <nombre sexo="$S">$N</nombre>
    $E
</persona>
```

Figura A.2: Término XML

valores `femenino` y `Moni`, respectivamente, mientras que la variable `$E` debe ser igual al elemento `<edad>24</edad>`.

A.2.3. Procedimientos lógicos

Los procedimientos lógicos representan los axiomas que describen las características del sistema, estableciendo las restricciones lógicas que debe cumplir la información contenida en la colección de documentos. Para teorías de primer orden que usan el lenguaje de cláusulas de Horn, los axiomas se pueden interpretar como procedimientos. Así, el cuerpo del procedimiento describe la secuencia de acciones cuya ejecución conduce a inferir aquella información que satisface las restricciones establecidas por el procedimiento y por los axiomas del sistema. Sintácticamente, un procedimiento lógico consiste de una secuencia de invocaciones a otros procedimientos lógicos. En `ADM` se pueden clasificar en: procedimientos primitivos, procedimientos básicos y procedimientos genéricos definidos por el programador. Con relación al alcance de los nombres, se puede decir que, en general, el nombre de un procedimiento tiene ámbito global mientras que el nombre de una variable tiene ámbito local al cuerpo del procedimiento.

Entre las características más sobresalientes del modelo de programación de `ADM` se encuentra su capacidad de recuperar información contenida en colecciones de documentos, y de realizar inferencia a partir de dicha información . El mecanismo que se usa para este fin se le conoce como búsqueda con retroceso (*backtracking*) que ha sido ampliamente discutido en la literatura relacionada con el lenguaje Prolog [9]. Como se discutirá más adelante, en `ADM` es posible generar el contenido de nuevos documentos a partir de la información extraída de la colección usando métodos de programación lógica y de satisfacción de restricciones.

A continuación se describen con más detalle los diferentes tipos de procedimientos que se pueden definir en `ADM`.

A.2.4. Procedimientos primitivos

Los procedimientos primitivos, predefinidos por ADM, amplían considerablemente el poder expresivo del modelo de programación lógica. Sin embargo, el aumento en la expresividad trae consigo una pérdida en la consistencia y uniformidad del modelo declarativo al introducir construcciones de naturaleza imperativa como las operaciones de entrada y salida, la evaluación de expresiones aritméticas y su asignamiento a variables lógicas.

- Unificación y comparación de términos.
- Conjuntos de soluciones.
- Meta-predicados.
- Modificación de programas.
- Aritmética.
- Lectura/Escritura de términos.

En ADM los procedimientos primitivos se invocan siguiendo el estilo de lenguajes imperativos como SmallTalk, en donde los parámetros formales no se identifican por su posición en la lista sino por un nombre que identifica a cada uno.

Por ejemplo, para asignar el resultado de la evaluación de una expresión aritmética a una variable, el procedimiento primitivo `assign` se usa de la siguiente forma:

```
<assign to="$Y" type="integer">$X+1</assign>
```

en donde, los nombres de los atributos `to` y `type` identifican el papel que desempeñan los parámetros en la invocación del procedimiento primitivo `assign`. Este ejemplo muestra como se asigna a la variable `$Y` la suma de uno al valor vinculado a la variable `$X`. El atributo `type` declara que la expresión `$X+1` es de tipo entero y que el valor que resulta de su evaluación se asigna a la variable `$Y` (en forma similar al procedimiento `is/2` de lenguaje Prolog estandard). En general, cada procedimiento primitivo establece el tipo de cada parámetro formal, de modo que éstos sean de compatibles con los tipos de los parámetros usados en la invocación.

El concepto de variable lógica es fundamental para establecer relaciones entre los contenidos de dos o más documentos e incluir restricciones sobre ellos. En ADM se pueden asignar a las variables lógicas no solo elementos simples sino también estructurados. Por ejemplo, el procedimiento primitivo `equal` usa el procedimiento de unificación de términos para decidir sobre su igualdad sintáctica. Así, la invocación:

```
<insert>
    <padre-de>
        <padre>Jose</padre> <hijo>Ivan</hijo>
    </padre-de>
</insert>
```

Figura A.3: Inserción de un documento a la colección (instancia de la relación `padre-de`).

```
<equal this="$T" to="&lt;hijo&gt;Ivan&lt;/hijo&gt;"/>
```

vincula la variable `$T` con el término `<hijo>Ivan</hijo>`. La expresión:

```
&lt;hijo&gt;Ivan&lt;/hijo&gt;
```

es una forma alternativa de escribir al elemento `<hijo>Ivan</hijo>` en donde las referencias `<` y `>` evitan usar los caracteres especiales `<` y `>`. `ADM` interpreta correctamente este fragmento de texto como un elemento cuando aparece en el contexto apropiado. Mientras este mecanismo permite pasar elementos XML en el paso de parámetros, su uso debe limitarse ya que la escritura de múltiples elementos anidados conduce a errores de sintaxis difíciles de identificar.

Entre los procedimientos primitivos más importantes se encuentran `insert` y `delete` definidos para modificar el contenido de las colecciones de documentos. El procedimiento `insert` introduce un documento a la colección sin considerar sus posibles repeticiones, mientras que el procedimiento `delete` lo elimina siempre que exista uno en la colección que concuerde con su estructura y contenido. Por ejemplo, para modificar el conocimiento que se tiene de las relaciones de parentesco de modo que refleje el hecho de que Jose tiene un hijo llamado Ivan, se usa el procedimiento `insert` como se indica en la figura A.3.

El documento insertado en la colección describe una instancia de la relación `padre-de`, en la que `Jose` y `Ivan` juegan los papeles de `padre` e `hijo`, respectivamente. Al ejecutar la acción `insert`, la colección se compone de los documentos que se enlistan en la figura A.3. Al final de esta lista se encuentra el documento recién insertado. Las bases extensionales de documentos pueden obtenerse de `ADM` por la aplicación reiterada del procedimiento primitivo `insert` o de otras aplicaciones que producen documentos XML. En en modelo declarativo de `ADM`, los documentos XML se consideran como una forma elemental de procedimiento lógico (conocidos como procedimientos básicos) y constituyen la fuente primaria de la información disponible.

```
<padre-de>
    <padre>Benito</padre> <hijo>Juan</hijo>
</padre-de>

<padre-de>
    <padre>Benito</padre> <hijo>Jose</hijo>
</padre-de>

<padre-de>
    <padre>Juan</padre> <hijo>David</hijo>
</padre-de>

<padre-de>
    <padre>Juan</padre> <hijo>Antonio</hijo>
</padre-de>

<padre-de>
    <padre>Jose</padre> <hijo>Ivan</hijo>
</padre-de>
```

Figura A.4: Colección de cinco documentos que forman la relación `padre-de`.

A.2.5. Procedimientos básicos

En `ADM`, un *procedimiento básico* es, o bien un documento XML, o bien una regla elemental. La regla, generalmente anónima, establece la validez de un documento genérico XML. El propósito de los procedimientos básicos es representar el conocimiento basado en los hechos conocidos. A una colección de documentos que consiste de documentos XML generados por editores u otras aplicaciones se le llama *base extensional de documentos*. La figura A.4 muestra algunos de los documentos de la base extensional de documentos que contiene a la relación `padre-de`.

Los documentos pueden contener no solo elementos XML sino también términos XML, es decir, los documentos pueden incluir variables lógicas. Por ejemplo, si a los documentos de la figura A.4 agregamos el documento que aparece en la figura A.5, la consulta a la relación `padre-de` en la que Adan desempeña el papel de padre es verdadera. Por ejemplo, en la consulta mostrada en la figura A.6, la primera condición es verdadera por la existencia del documento de la figura A.5, mientras que la segunda condición es verdadera por la existencia del primer y segundo documentos que aparecen

```
<padre-de>
    <padre>Adan</padre> <hijo>$Hijo</hijo>
</padre-de>
```

Figura A.5: Documento genérico en la relación `padre-de`.

```
<if>
    <padre-de>
        <padre>Adan</padre> <hijo>Benito</hijo>
    </padre-de>
    <padre-de>
        <padre>Benito</padre> <hijo>$H</hijo>
    </padre-de>
</if>
```

Figura A.6: Ejemplo de consulta en ADM.

en la figura A.13. La ventaja de introducir variables en documentos XML es que permiten describir conjuntos de documentos indicando el contenido y la estructura que todos ellos poseen, en lugar de recurrir a la enumeración exhaustiva. Las variables lógicas también permiten describir las condiciones que los documentos de la colección deben cumplir aunque para ello se requiere de una forma más elaborada de procedimiento lógico conocido como procedimiento genérico.

A.2.6. Procedimientos genéricos

Los procedimientos genéricos son reglas definidas por el programador que permiten introducir restricciones lógicas más complejas que aquellas que se pueden conseguir con los procedimientos básicos. Las restricciones consisten de una conjunción de predicados agrupados en la parte `if` de la regla. Desde el punto de vista de la programación lógica, los predicados se pueden representar por procedimientos lógicos. Bajo esta interpretación, la conjunción de predicados consiste de una secuencia de invocaciones a procedimientos lógicos (primitivos, básicos o genéricos) en la cual la información inferida puede pasar de unos a otros mediante variables lógicas compartidas.

Un procedimiento genérico consiste de un encabezado y de un cuerpo. El encabezado del procedimiento, incluyendo los parámetros formales, se describe en la parte `on`, mientas que el cuerpo del procedimiento se describe en la parte `if`. La acción de aceptación

```
<rule>
    <on>
        <padre-de>
            <padre>$P</padre> <hijo>$H</hijo>
        </padre-de>
    </on>
    <do> <accept/> </do>
</rule>
```

Figura A.7: Documento que contiene una regla definida sobre la relación `padre-de`.

o rechazo se indica en la parted `do`. De esta manera, los procedimientos genéricos se consideran como reglas de un tipo especial que se caracteriza porque las únicas acciones posibles son las de aceptar o rechazar la invocación del procedimiento.

La forma más elemental de procedimiento genérico es la que aparece en la figura A.7. Este procedimiento establece que para toda consulta que involucre a un documento cuya estructura corresponda con la dada en la parte `on`, se deberá realizar la acción indicada en la parte `do`, es decir, aceptar como válida la invocación. Esta forma elemental del procedimiento genérico es equivalente a la del documento genérico que se analizó en la sección previa. Por ejemplo, si la variable `$P` que aparece en el documento de la figura A.7 se remplaza por el texto `Adan`, entonces esta regla produce el mismo efecto que el producido por el documento de la figura A.5.

La regla usa las variables lógicas `$P` y `$H` para recuperar el contenido de los documentos de la relación `padre-de` siempre que aparezcan sus nombres en la colección y desempeñen, respectivamente, los papeles de `padre` e `hijo` en la relación. Puesto que existen cuatro documentos como instancias de esta relación, la regla se ejecutará cuatro veces, tomando las variables lógicas, en cada ocasión, los valores correspondientes:

```
$P=Benito  $H=Juan
$P=Benito  $H=Jose
$P=Juan    $H=David
$P=Juan    $H=Antonio
$P=Jose    $H=Ivan
```

El conjunto de procedimientos que comparten tanto el mismo nombre de encabezado como el mismo número y tipo de parámetros constituye la definición del procedimiento. Cada uno de los procedimientos de una definición representan alternativas no necesariamente excluyentes. La selección de una alternativa depende de las condiciones que imponga sobre los valores de los parámetros en la invocación. El orden que se sigue para

seleccionar una alternativa coincide con el orden textual con el que aparecen las reglas que definen al procedimiento en el programa.

Generalmente los procedimientos se escriben siguiendo el estilo bien conocido en la programación conocido como genera y prueba (*generate and test*). Como su lo nombre sugiere, este estilo consiste en encontrar todas soluciones posibles consultando la colección de documentos para después seleccionar aquellas que cumplan con las restricciones indicadas en la prueba. Este estilo de programación se fundamenta en una técnica conocida como *retroceso* (*backtrack*) la cual utiliza las definiciones alternativas de un procedimiento para generar todas las soluciones posibles. Las soluciones se obtienen por la combinación sistemática y exhaustiva de los valores que toman las variables en sus dominios respectivos. Aunque esta técnica es de gran utilidad se le considera ineficiente en espacios de soluciones grandes y potencialmente no terminante. A pesar de estos inconvenientes, `ADM` al igual que la mayoría de las implementaciones del lenguaje Prolog la han adoptado por su simplicidad conceptual y facilidad de implementación.

No obstante, las capacidades deductivas de `ADM` no se limitan a la búsqueda y recuperación de información en documentos aislados. Mediante una *consulta* se puede además relacionar la información extraida de diferentes documentos con la ayuda de variables lógicas. Una variable lógica permite identificar un valor (texto o elemento XML) que puede ocurrir una o más veces en uno o más documentos de la colección. Por ejemplo, para saber si dos personas son hermanos se define la relación `hermano-de` que se muestra en la figura A.8.

Como puede observarse, este procedimiento utiliza las variables lógicas `$X` y `$Y` para designar a los nombres de las personas sobre las que se desea saber si cumplen la relación `hermano-de`. Esta consulta vincula los contenidos de dos documentos (como instancias de la relación `padre-de`) mediante la variable `$Padre` cuya solución (`$Padre=Benito`) cumple las condiciones de ser padre tanto de `Juan` como de `Jose`. El procedimiento primitivo `notequal` restringe los valores aceptables para `$X` y `$Y` a valores distintos. De acuerdo a la interpretación declarativa de un procedimiento, las variables (como `$Padre`) se interpretan como variables locales cuantificadas existencialmente, por lo que su ámbito se reduce al cuerpo del procedimiento en donde ocurren. La parte `do` de la regla establece que la consulta se acepta por que existe al menos una solución para ella.

La *aceptación* (`<accept/>`) tiene efecto sobre el curso de las invocaciones ya que permite su curso normal, es decir, invocando al procedimiento que le sigue en la consulta (si hay alguno). En contraste, la acción de *rechazo* (`<reject/>`) intenta producir otra solución siempre que exista alguna definición alternativa; en otro caso, el rechazo revierte el curso normal, invocando al procedimiento que le precede en la consulta (si hay alguno), descartando con ello las soluciones parciales encontradas. En caso de que se agoten todas las definiciones alternativas, se rechaza la consulta sin producir solución alguna.

La figura A.9 muestra el procedimiento `primo-de` cuya definición consiste de invoca-

```
<rule name="hermano-de">
    <on>
        <hermano-de>
            <subject>$X</subject> <object>$Y</object>
        </hermano-de>
    </on>
    <if>
        <notequal>
            <subject>$X</subject> <object>$Y</object>
        </notequal>
        <padre-de>
            <padre>$Padre</padre> <hijo>$X</hijo>
        </padre-de>
        <padre-de>
            <padre>$Padre</padre> <hijo>$Y</hijo>
        </padre-de>
    </if>
    <do> <accept/> </do>
</rule>
```

Figura A.8: Relación `hermano-de`.

```
<rule name="primo-de">
    <on>
        <primo-de>
            <subject>$X</subject> <object>$Y</object>
        </primo-de>
    </on>
    <if>
        <padre-de>
            <padre>$Padre1</padre> <hijo>$X</hijo>
        </padre-de>
        <padre-de>
            <padre>$Padre2</padre> <hijo>$Y</hijo>
        </padre-de>
        <hermano-de>
            <subject>$Padre1</subject> <object>$Padre2</object>
        </hermano-de>
    </if>
    <do> <accept/> </do>
</rule>
```

Figura A.9: Relación `primo-de`.

ciones a procedimientos básicos (`padre-de`) y a procedimientos genéricos (`hermano-de`). Para el problema de las relaciones de parentesco, a partir de la relación básica `padre-de` se definieron, entre otras, las relaciones `hijo-de`, `tio-de` y `abuelo-de`. Sin embargo, estas definiciones no fueron incluidas. A la colección de documentos que consiste únicamente de procedimientos lógicos se le llama *base intensional de documentos*.

El modelo deductivo asume que el programa describe un conocimiento estático en el que las bases extensionales e intensionales de documentos no cambian. Esta suposición es razonable porque generalmente las consultas se realizan por iniciativa de un usuario o agente de software, dejando el problema de la consistencia de la consulta fuera del modelo deductivo. En `ADM`, una colección se modifica por la interacción que tiene lugar entre sus documentos, al producir nuevos documentos, destruir otros ya existentes o modificar su contenido.

A.3. Reglas activas

En comparación con las reglas deductivas, las reglas activas remedian la falta de interactividad y de extensibilidad del servidor de aplicaciones. Las reglas activas definen el comportamiento reactivo que modifica el contenido de las bases de documentos como respuesta a los eventos de inserción, eliminación y recepción de documentos. La importancia de las reglas activas radica en que le ofrecen al programador mecanismos para definir e incorporar su propia infraestructura de control dentro del servidor de aplicaciones. Para introducir el modelo activo de programación en `ADM`, en este trabajo se ha desarrollado e incluido un administrador de reglas activas el cual es responsable de interactuar con otros componentes activos, otros clientes u otros servidores.

A.3.1. Estructura de las reglas activas

Una *regla activa* establece las acciones que se pueden realizar como respuesta a un evento cuya estructura y contenido de información satisfacen la condición dada. En consecuencia, una regla activa consiste de un *evento*, una *condición* y una *acción*, cuya especificación aparece bajo los elementos `on`, `if` y `do`, respectivamente. Existe una relación causal que une a eventos y a condiciones: cada acción produce un evento que da evidencia de la acción realizada y, recíprocamente, cada evento observable se origina en el desarrollo de alguna acción. La relación entre eventos y acciones asegura la continua ejecución de un sistema siempre que existan las reglas apropiadas. Por tal razón, la especificación de eventos debe revelar la información necesaria tomando en cuenta el contexto en el que se desarrolló la acción.

Aunque ya se ha mencionado que los procedimientos lógicos constituyen un tipo especial de reglas es necesario destacar las diferencias principales entre los dos:

- primero, el evento no se restringe a la invocación (*existencia*) de un procedimiento
- segundo, la acción no se restringe a la aceptación o al rechazo del procedimiento

No obstante que el modelo declarativo supone una base estática de conocimientos, la interpretación sugerida por Kowalski de que las cláusulas de Horn se pueden representar como procedimientos introduce un elemento de dinamismo en esta suposición. Para acomodar este dinamismo, en `ADM` se sugiere reformular esta interpretación para representar a las cláusulas de Horn como reglas activas. Dada la cláusula A `if` $B_1, B_2, \ldots, B_n$, la lectura declarativa dice que A es verdadera si B_1, B_2, ..., B_n son todas verdaderas, mientras que en `ADM`, la cláusula se interpreta como regla activa diciendo que la demostración de A es aceptada siempre que las demostraciones de B_1, B_2, ..., B_n sean todas aceptadas; en otro caso, la demostración de A es rechazada. En esta interpretación, la demostración se refiere a la actividad de realizar una prueba cuya aceptación (conclusión exitosa) depende de la realización y aceptación de otras actividades de demostración. Así, mientras en el modelo declarativo una cláusula es una conclusión lógica, en el modelo activo una cláusula es un plan que coordina las actividades que permitan derivar dicha conclusión. Esta interpretación permite conjuntar, bajo una misma notación, el modelo deductivo (estático) de los procedimientos lógicos con el modelo activo (dinámico) de las reglas activas.

Cabe mencionar que el evento de invocación `call` existe en `ADM` y se define como la composición secuencial del envío del nombre de la operación junto con los valores de los parámetros seguida de la recepción de los resultados producidos por el procedimiento. La introducción del evento `call` es particuarmente útil cuando se requiere actualizar los resultados de una consulta en la presencia de alteraciones a la colección de documentos.

Especificación de eventos

En `ADM`, un evento se define como un cambio observable en la estructura o el contenido de la colección de documentos. El origen de la alteración puede clasificarse de acuerdo al tipo de modificación que induce en: inserción, eliminación, modificación, envío o recepción de documentos. En cualquier caso, el evento se genera después de que la acción correspondiente haya terminado exitosamente. La estructura e información asociada con una acción se incluyen en el evento para que sea utilizada en la condición y en la acción. De esta manera, si se considera que un mensaje debe inducir un comportamiento determinado, entonces es necesario hacer explícita dicha información en su contenido. Las semejanzas y diferencias entre reglas deductivas y activas se puede apreciar comparando entre si los eventos de las reglas mostradas en las figuras A.8 y A.10. En la primera no aparece en la parte `on` el nombre de evento alguno, mientras que en la segunda aparece el evento de recepción `receive` asociado con la acción de envío `send`.

Especificación de la condición

La condición es la parte de la regla que especifica una consulta sobre los datos en la colección y, si la condición se satisface, la acción de la regla puede ejecutarse. Las condiciones son expresiones lógicas que se construyen sobre las constantes y los conectivos lógicos usuales, a los cuales se incorporan los operadores relacionales definidos sobre diferentes tipos de datos y aplicados al contenido de información tanto de la colección como de los eventos. Para distinguir entre ambos, designaremos como *contexto* a la información que contiene la colección. Así el contexto establece el marco de evaluación de una condición.

Entre las restricciones más importantes se encuentran aquellas que se designan bajo el nombre de *restricciones de integridad* que aseguran la consistencia de la información. Las restricciones de integridad pertenecen a un grupo de propiedades de *correctitud* conocidas como de *seguridad* y se refieren a aquellas propiedades que se cumplen en todos los estados del sistema. Debido a las diferencias fundamentales entre eventos y condiciones, es necesario reconocer que la evaluación de una condición puede dar diferentes resultados a lo largo de la formación de un evento compuesto. Por lo tanto, cuando se definen condiciones sobre eventos, la verificación de restricciones de integridad debe indicar además el momento en el que se debe realizar la evaluación de la condición.

Especificación de la acción

La acción especifica las operaciones que se realizan cuando se activa una regla y su condición se satisface. Las acciones varían desde operaciones de inserción, borrado y actualización a secuencias arbitrarias de comandos de recuperación o modificación o también pueden especificar aceptación o rechazo. Las acciones se definen sobre las operaciones básicas que exponen los componentes de un sistema y sobre los cuales se pueden agrupar en estructuras de control clásicas como la composición secuencial, paralela, alternativa, condicional e iterativa.

Al igual que en las condiciones, el contexto determina el marco en el que se realizan las acciones, pero a diferencia de ellas, las acciones modifican el contexto. De acuerdo al modelo de programación, las acciones pueden observarse por su invocación o por el efecto que producen ya que generan cambios en la estructura y la información del sistema. En general, los eventos relacionados con invocaciones son más flexibles y expresivos que aquellos relacionados con modificaciones. Por ejemplo, para asegurar el cumplimiento de las restricciones de integridad de una colección, es preferible determinar si la inserción de un documento preserva la consistencia antes de insertarlo que restaurar la colección a un estado consistente después de haberlo insertado.

A.3.2. Procesamiento de reglas

La semántica del procesamiento de reglas determina el orden de ejecución de las reglas seleccionadas, también determina como interactúan las reglas con las operaciones y transacciones arbitrarias de la colección. Se denomina *resolución de conflictos* al procedimiento de selección de la regla a ejecutarse cuando hay más de una regla elegible. Al igual que otros administradores de bases de datos activas, en `ADM` la selección se toma en base a relaciones de precedencia entre eventos `before`, `after` o `instead` que indican que una acción determinada debe realizarse antes, después o en lugar de la acción que generó el evento de referencia.

De las diversas alternativas existentes para la ejecución de reglas, `ADM` usa una variación del ciclo *reconocimiento-actuación* adoptado por la mayoría de los sistemas basados en reglas en Inteligencia Artificial. El ciclo inicia ejecutando la acción de una regla. A continuación se determinan qué reglas se hacen elegibles por los eventos generados y se prueban las condiciones correspondientes para obtener las instancias de las reglas que se pueden seleccionar. El *conjunto en conflicto* consiste de todas las instancias de las reglas que son elegibles para su ejecución. Este ciclo se repite hasta que las condiciones de regla ya no producen más instancias. Durante este proceso se usa una *estrategia de resolución de conflictos* para escoger una regla y ejecutar la acción de dicha regla para cada instancia producida por la condición, luego se determina si el ciclo sigue o se detiene. `ADM` ejecuta una regla por saturación, es decir, la acción de la regla se aplica a todas las instancias como se demuestra en el siguiente ejemplo.

A.4. Ejemplo de Interacción y Deducción

En el problema de la relación de parentesco, la regla que se muestra en la figura A.10 se selecciona cuando se recibe un nuevo documento como notificación de que un nuevo miembro de familia ha nacido. La regla envía como respuesta invitaciones a todos sus primos. Se supone para este ejemplo que el contenido de la colección contiene a los documentos que se muestran en la figura A.4. Sin embargo, en ella no aparecen los documentos que forman la relación directorio, la cual asocia una dirección de correo para cada persona registrada en ella.

La figura A.11 muestra el documento recibido por el módulo activo que anuncia que José tiene un nuevo hijo llamado Jorge. El módulo activo de `ADM` notifica que ha ocurrido un evento de recepción de un mensaje el cual aparece en la figura A.11 El módulo de recepción de mensajes envía el contenido al administrador de reglas de `ADM` para seleccionar la regla apropiada. El administrador mantiene una lista de reglas indexadas por el evento y el tipo de operación. De esta manera, la regla `invitacion` está indexada por el evento `receive` y por la operación `padre-de`. El administrador entonces extrae el

```
<rule name="invitacion">
    <on>
        <receive from="$C">
            <padre-de>
                <padre>$X</padre> <hijo>$Y</hijo>
            </padre-de>
        </receive>
    </on>
    <if>
        <primo-de>
            <subject>$Y</subject> <object>$Z</object>
        </primo-de>
        <padre-de>
            <padre>$P</padre> <hijo>$Z</hijo>
        </padre-de>
        <directory>
            <name>$P</name> <email>$M</email>
        </directory>
    </if>
    <do>
        <seq>
            <send to="$M">
                <invitacion>
                    <anfitrion>$Y</anfitrion> <invitado>$Z</invitado>
                </invitacion>
            </send>
            <insert>
                <invitado>$Z</invitado>
            </insert>
        </seq>
    </do>
</rule>
```

Figura A.10: Regla activa para enviar invitaciones.

```
<padre-de>
    <padre>Jose</padre> <hijo>Jorge</hijo>
</padre-de>
```

Figura A.11: Documento recibido que activa a la regla `invitacion`.

```
<invitacion>
    <anfitrion>Jorge</anfitrion> <invitado>David</invitado>
</invitacion>

<invitacion>
    <anfitrion>Jorge</anfitrion> <invitado>Antonio</invitado>
</invitacion>
```

Figura A.12: Documentos enviados por la regla `invitación`.

contenido del mensaje y con él formula la consulta indicada en la condición de la regla. Dado el nombre del hijo `$Y`, la condición determina si existe un primo `$Z` de `$Y` cuyo padre `$P` tiene registrado una dirección de correo electrónico `$M` en el directorio. Si la condición tiene al menos una solución, entonces la acción de la regla se aplica a cada una de las soluciones encontradas. En otro caso, la regla no se aplica. Si la regla se aplica, las acciones consisten en enviar un mensaje de invitación a nombre del primo pero a la dirección de correo del padre, registrando además el nombre del primo invitado.

La regla `invitacion` se ejecuta por saturación sobre todos los primos de Jorge (David y Antonio) lo que resulta en el envío de los mensajes que aparecen en la figura A.12 y a insertar en la colección los documentos que se muestran en la figura A.13

```
<invitado>David</invitado>

<invitado>Antonio</invitado>
```

Figura A.13: Documentos insertados en la colección.

A.5. Resúmen

El lenguaje ADM reúne características de reactividad y deducción en un modelo uniforme de programación. En este capítulo se mostraron algunas de las características más sobresalientes del lenguaje. Las reglas deductivas consisten de elementos XML, términos XML, procedimientos lógicos y consultas. Los términos XML son elementos XML que se extienden para crear elementos con variables lógicas. Un procedimiento lógico consiste de una consulta que se ejecuta bajo el nombre de una operación que incluye parámetros que permiten recuperar información que cumple las restricciones establecidas. Las consultas permiten recuperar información que es consecuencia lógica del contenido de la colección de documentos.

Por otro lado, las reglas activas del lenguaje proporcionan mayor flexibilidad al programador para ejercer cierto control en el servidor de aplicaciones. Los eventos que se emplean en ADM son de inserción, eliminación y recepción (fragmentos) de documentos XML. Los eventos son detectados por el administrador de reglas activas el cual selecciona la regla apropiada, si la hay, para realizar las acciones correspondientes en respuesta al evento observado. El problema de definir las relaciones de parentesco ha servido de ejemplo para explorar en este capítulo las características más sobresalientes del lenguaje.

Bibliografía

[1] B.A. Colombo, C. Demetrescu, I. Finocchi, and L. Laura, *A Java-based Sys-tem for Building Animated Presentations over the Web*,Journal paper accepted for publication in Elsevier Science of Computer Programming (SCP), special issue on "Practice and Experience with Java in Education". An extended abstract ap-pears in the Proceedings of the AICCSA'03 Workshop on Practice and Expe-rience with Java Programming in Education, Tunis, pp. 901-946, July 2003.

[2] R. Baker, M. Boilen, M. Goodrich, R. Tamassia, and B. Stibel, *Testers and Visualizers for Teaching Data Structures*In Proc. 1999 ACM SIGCSE Symp., ACM, pp.261-265, 1999.

[3] W. Dann, S. Cooper, R.Pausch, *Making the connection: programming with animated small world*,in Proceedings of the Conference Integrating Technology into Computer Science Education, pp. 41-44, 2000.

[4] J. Bailey, A. Poulovassilis, P.Wood,*Analysis and optimisation for event-condition-action rules on XML*, 2001.

[5] K. Escobar,*Atribución de significado a documentos XML con logcin-xml*, Master's thesis, Centro de Investigación y Estudios Avanzados del Instituto Politénico Nacional-Unidad Zacatenco, D.F., Mexico, April 2004.

[6] E.Ñ. Hanson, J. Widom,*An overview of Production Rules in Database Systems*, The knowledge Engineering Review, 8(2) pp.121-143, June 1993.

[7] V. Wuwanges, C. Anutariya, K. Akama, E. nantajewarawat,*XMl Declarative Description:A Languaje for the Semantic Web*, IEEE Intelligent System, pp. 54-65, 2001.

[8] M. Chinwala, R. Malhotra, J. A. Miller,*WC Progress towards Standards for XML database.*

[9] H. Boley, S. Tabet, G. Wagner*Design Rationale of RuleML: A Markup Languaje for Semantic Web Rules*, In International Semantic Web Working Simposium, 2001.

[10] M. Fernández J. Simeón, P. Wadler,*SML Query Languajes Experience and Examples*.

[11] A. Silberschatz,*Fundamentos de Bases de Datos*, Mc. Graw Hill, 2002.

[12] N. Paton, O. Díaz,*Active Database System*, ACM Computer Surveys, 31(1), pp. 63-103, 1999.

[13] N. Paton, O. Díaz,*Active Rules in Database Systems*, Springer, 1999.

[14] J. Widom and S. Ceri,*Active Database System*, Ed. Morgan Kaufmann Publishers, 1996.

[15] E. Hanson,*The Ariel Project in Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[16] K. Kulkani, N. Mattos and R. Cochrane,*Active Database Features in SQL3, in Active Database Systems* , N. Paton, Ed. Springer Verlag, Berlin, 1998.

[17] U. Dayal, A. Buchmann and S. Chakravarthy,*The HiPAC Project in Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[18] S. Ceri, P. Fraternali, et al, *Active Rule Management in Chimera, in Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[19] N. Gehani and H. Jagadish,*Active Database facilities in Ode in Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[20] S. Gatxiu and R. Dittrich, *SAMOS in Active Database Systems*, N. Paton, Ed. Springer Verlag, Berlin, 1998.

[21] C. Collet, *NAOS in Active Database Systems*, N. Paton, Ed. Springer Verlag, Berlin, 1998.

[22] S. Potamianos and M. Stonebraker, *The Postgress Rule System in Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[23] J. Widom, *The Starburst Rule Systemin Active Database Systems Triggers and Rules for Advanced Database Processing*, J. Widom, Ed. Morgan Kaufmann Publishers, 1996.

[24] T.L,Naps, G. Rößling, *Evaluating the Educational Impact of Visualization*, Paving the Way Towards Excellence in Computing Education, Volume 35, Number 4, pp.124-136, ACM Press, New York, 2003.

[25] R. Fleischer and L. Kucera *Algorithm animation for teaching*, In Software Visualization, State-of-the-Art Survey, Springer LNCS 2269 pp.113-128, 2002.

[26] S. R. Hansen, N.H. Narayanan, and M. Hegarty,*Designing educationally effective algorithm visualizations*, lnl. of Visual Languages and Computing, pp.291-317, 2002.

[27] G. Booch, J Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman Inc., 2000.

[28] C. Lilley, D. Jackso. *Scalable Vector Graphics (SVG) XML Graphics for the Web*, http://www.w3.org/Graphics/SVG, W3C, 2004.

[29] J.D. Eisenberg, *SVG Essentials*, O Reilly, pp.13-20, 2002.

[30] *ArgoUML Quick Guide*, http://argouml.tigris.org/documentation/defaulthtml/quick-guide/index.html, Tigris org, 2005.

[31] L. Quinn. *Extensible Markup Language (XML)*, http://www.w3.org/XML/, W3C, 2004.

[32] O. Olmedo, K. Escobar, G. Alor,G. Morales, *ADM: An Active Deductive XML Database System*, Springer, LNAI 2972,pp. 139-148, 2004.

[33] T. Quatrani, *Visual modeling with Rational Rose 2000 and UML*, Pearson, pp.10-15, 1999

[34] S. Si Alhir *UML in a nutshell. A desktop quick reference.*, O'Reilly, 1998.

[35] A. Begeulin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam*Graphical development tools for network based concurrent supercomputing*, In Procee-dings of Supercomputing'91, pp.435-444, 1991.

[36] B. Topol, J. Stasko and V. Sunderam*Integrating visualization support into distributed computing systems*,In Proceedings of the 15-th International Confe-rence on Distributed Computing Systems, pp.19-26, 1995.

[37] P. Crescenzi, C. Demetrescu ,I. Finocchi, R. Petreschi,*Leonardo: a software visualization system*, In Proceedings WAE'97,pp. 146-155, 1997.

[38] O.Olmedo, M. Rivera.*Visual Programming Environment for ECA Rules*, Advances in Artificial Intelligence and Computer Science Research on Computing Science 14, pp.253-264, 2005.

[39] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[40] A. Martelli, U.Montanari. An efficient unification algorithm,ACM Transactions on Programming Language and Sytems, 4(2),pp.258-282, 1982.

[41] Rational Rose XDE Developer, http://www-306.ibm.com/ software/awdtools/developer/rosexde/features/IBM, 2005.

[42] *Magic Draw*, http://www.magicdraw.com, 2005.

[43] J.H. Cross, *JGrasp*, http://www.jgrasp.org/, 2006.

[44] *Unimod*, http://unimod.sourceforge.net, 2006.

[45] *Artisan*, http://www.artisansw.com, 2005.