Generación Automática de Código a Partir de Máquinas de Estado Finito

Automatic Code Generation from Finite State Machines

Mario Rincón Nigro[1], José Aguilar Castro[1] y Francisco Hidrobo Torres[2]
[1]CEMISID, Facultad de Ingeniería, Universidad de los Andes
Mérida. Venezuela
aguilar@ula.ve
[2]SUMA, Facultad de Ciencias, Universidad de los Andes
Mérida. Venezuela
hidrobo@ula.ve

Artículo recibido en Febrero 16, 2010; aceptado en Octubre 04, 2010

Resumen. Este trabajo presenta una herramienta de generación automática de código fuente en lenguajes orientados a objetos para modelos abstractos expresados en UML. La herramienta permite la generación de código, tanto de la estructura estática como del comportamiento dinámico, presentes en modelos de sistemas de software. En específico, permite generar código fuente en el lenguaje C++, a partir de los diagramas de clases, diagramas de estados, y diagramas de actividad del UML. Dicha herramienta podrá ser integrada a herramientas CASE de modelado, con capacidades apropiadas de exportación de modelos del UML en formato XMI. En el trabajo se presentan detalles sobre el diseño y la implementación de la herramienta, haciendo hincapié en la generación del comportamiento dinámico. Además, se muestran los resultados de su evaluación en casos de estudio.
Palabras clave: Teoría de la Computación, Modelos de Computación, Máquinas de Estado Finito, Generación Automática de Código, UML.

Abstract. In this work, we present a tool for automatic source code generation, in OO languages, from abstract models expressed in UML. The tool allows the code generation, as much of the static structure as the dynamic behavior, present in models of software systems. Specifically, it allows to generate source code in the C++ language, from the classes diagrams, state diagrams, and activities diagrams of the UML. This tool could be integrated to a modeling CASE tool, with appropriate exporting capacities of UML models in format XMI . We present details of tool design and implementation, with special attention in code generation for dynamic behavior. In addition, to evaluate the tool, we present study cases
Keywords: Theory of Computing, Models of Computation, Finite State Machines, Automatic Code Generation, UML.

1 Introducción

El uso de herramientas de generación automática de código, para agilizar el desarrollo de sistemas de software e incrementar su confiabilidad, es un concepto bien conocido en el área de la computación (Bell, 1998; Herrington et al., 2003). Actualmente, gran parte del esfuerzo que se invierte en la investigación en el área de la generación automática de código, está orientada a la generación de código en lenguajes de alto nivel a partir de modelos abstractos de sistemas de software. Por otro lado, el UML (Eriksson et al., 2004; Pilone et al., 2005) se ha convertido en el estándar de la industria para especificar y modelar sistemas de software en general. Este lenguaje permite definir conceptos relacionados a los sistemas de software, establece la notación gráfica para comunicar estos conceptos, y es lo suficientemente formal como para apoyar procesos de automatización dentro del desarrollo de software. Junto a lenguajes como el UML, han surgido iniciativas de desarrollo de software centradas en modelos, como por ejemplo la MDA (*Model Driven Architecture*) del OMG (*Object Management Group*), que se apoyan en las herramientas de generación de código para facilitar la integración entre las distintas fases del desarrollo de software (Herrington et al., 2003; Pinter et al., 2003).
En la actualidad existen herramientas CASE (*Computer Aided Software Engineering*) libres y de código abierto, entre las que destacan ArgoUML (ArgoUML, 2009) y Umbrello (Umbrello UML, 2009), que implementan algunas funcionalidades para

generar código desde modelos UML. Específicamente, generan código para representaciones de estructura estática, esto es, para diagramas de clase. Por otro lado, diversos autores han propuesto patrones de diseño para la implementación de máquinas de estado del UML en lenguajes orientados a objetos (Pinter et al., 2003; Eriksson et al., 2004; Pilone et al., 2005), éstos han servido de base para nuestra propuesta. Ahora bien, la mayoría de las herramientas existentes que permiten la generación de código para representaciones de comportamiento dinámico de modelos del UML no son libres. Se cuenta con muy pocas herramientas libres para la generación de código fuente a partir de modelos del UML, y las capacidades de dichas herramientas resultan modestas en comparación con las que ofrecen las herramientas no libres. Normalmente, ellas se limitan a la generación de código fuente para representaciones de la estructura estática de los sistemas modelados.

En este artículo se presenta una herramienta de "software libre" que permite generar, de manera automática, código fuente en lenguajes orientados a objetos (OO) a partir de modelos del UML. La herramienta usa tanto las representaciones de la estructura estática, como las del comportamiento dinámico. Particularmente, esto último es descrito usando los diagramas de estado. Con dichos diagramas se genera el código fuente que implementa el ciclo de vida de los objetos, y su comportamiento ante eventos externos. La herramienta CASE considerada para exportar los modelos, fue ArgoUML (0.24). La herramienta se prueba en dos aplicaciones que simulan un reloj digital y el juego de Tretis. La elección de ellas es debido a que son aplicaciones bien conocidas, por lo que no es necesario invertir tiempo en el análisis de requisitos. Además, estas aplicaciones han sido presentadas como ejemplo de modelado dinámico con máquinas de estado en (Pinter et al., 2003; Eriksson et al, 2004). El código fuente fue generado para el lenguaje C++.

El artículo está organizado como sigue: en la primera parte hacemos una introducción al problema de Generación de Código para comportamiento Dinámico, en la siguiente sección presentamos nuestra propuesta de generación de código fuente usando Máquinas de Estado Finito. En la sección 3 se detalla la propuesta para generar código automático, en específico, el patrón de diseño. La sección 4 presenta la herramienta, la sección 5 los casos de estudio, y finalmente se presentan las conclusiones.

2 Generación de Código para Comportamiento Dinámico

La *generación automática de código* es el proceso mediante el cual un programa produce, de manera automática, código en un lenguaje, a partir de un esquema expresado en otro lenguaje. Tradicionalmente, se usa para traducir esquemas en lenguajes de alto nivel, más cercanos a la manera de pensar del humano, a lenguajes de más bajo nivel (ensamblador o lenguaje de máquina) orientados a su interpretación por parte de computadores. Se denominan *generadores de código* a las aplicaciones que llevan a cabo dicha tarea. Las aplicaciones de generación de código de uso más extendido son los compiladores. Estos toman un programa escrito en un lenguaje de alto nivel, y lo transforman en código objeto. Los *generadores de código fuente* operan en el nivel inmediatamente superior al de los compiladores, produciendo, valga la redundancia, código fuente (Herrington et al., 2003; Pinter et al., 2003).

La generación automática de código fuente ha sido aplicada exitosamente a diversas actividades, entre las que cabe mencionar: el desarrollo de compiladores, la documentación de sistemas de software, el desarrollo de interfaces gráficas de usuario, el desarrollo de sistemas Web, entre otras. Ejemplos de herramientas de generación automática de código fuente, ampliamente utilizadas para algunas de las actividades mencionadas, son: Yacc, Lex, Doxygen, Javadoc y QtDesigner, por mencionar algunas.

La clase de generación de código de interés para el presente trabajo es la denominada *generación de código fuente basada en modelos*. La generación de código basada en modelos consiste en la producción de código fuente en lenguajes de alto nivel, de manera automática, a partir de modelos gráficos que describen la estructura, el comportamiento, o la arquitectura de los sistemas. Las herramientas de desarrollo de software actuales, muestran la tendencia a facilitar el desarrollo basado en modelos, permitiendo a los desarrolladores trabajar a un nivel de abstracción más alto, y las herramientas de generación de código fuente basadas en modelos facilitan la

transición entre la fase de diseño y la fase de implementación de los sistemas de software.

La generación de código basada en modelos es una tecnología en emergencia. El diseño de lenguajes de modelado que soporten la generación de código posee relativamente una base teórica muy reducida, en comparación al conocimiento teórico que existe para el diseño de lenguajes de programación y el desarrollo de compiladores. Las herramientas que generan código a partir de modelos generan código fuente (en vez de código ejecutable) para que los usuarios puedan modificarlo y compensar así posibles deficiencias y errores.

En (Bell, 1998) se caracterizan tres enfoques para la generación de código basada en modelos: el enfoque estructural, el enfoque de comportamiento y el enfoque de traducción. El orden en el que se presentan los enfoques no es accidental, pues representa la evolución de las herramientas de generación de código en el tiempo. Cada uno de estos enfoques extiende las capacidades del enfoque anterior. El enfoque estructural permite generar el código correspondiente a definiciones de clases y sus relaciones estáticas. El enfoque de comportamiento permite generar código para especificaciones de comportamiento y especificaciones de acción expresadas en un lenguaje de alto nivel. Por último, el enfoque de traducción utiliza modelos de aplicaciones independientes de las arquitecturas, para dar a los usuarios mayor control sobre la traducción a código fuente para arquitecturas de software específicas.

En nuestro caso, el enfoque de comportamiento es el de interés para generar código de la parte dinámica. Específicamente, el enfoque de comportamiento se basa en modelos de máquinas de estado, extendidas con especificaciones de acciones en lenguajes de alto nivel. Un beneficio adicional del enfoque es que posibilita la verificación de los modelos de comportamiento del sistema, antes de que el código sea generado (Herrington et al., 2003; Pinter et al., 2003). En el enfoque de comportamiento los desarrolladores pueden crear un modelo de implementación, a partir de los modelos producidos en las fases de análisis y diseño, mediante un modelado más preciso, añadiendo detalles a la estructura de las clases y a las representaciones de comportamiento. Las herramientas que soportan este enfoque usualmente cuentan con máquinas virtuales que interpretan especificaciones de máquinas de estados (ejemplo de esto es la herramienta descrita en (Pinter et al., 2003)).

Otros trabajos recientes interesantes son los siguientes: en (Zapata et al., 2007) se presenta una metodología para generar automáticamente código para controladores lógicos programables (PLCs), a partir de modelos de automatismo construidos en redes de Petri jerárquicas. Esta metodología permite aprovechar técnicas de la ingeniería de software, como la programación por objetos, y las capacidades de alto nivel embebidas en los controladores lógicos, para resolver problemas complejos de automatización industrial vía la reusabilidad del código. En (Muñeton et. al., 2007) se proponen reglas para la generación de código a partir de metamodelos de diagramas de clases, secuencial y de maquinas de estados de UML. Las reglas están definidas en lógica de primer orden, permitiendo una especificación donde se evitan las ambigüedades y la necesidad de aprender un lenguaje de programación especifico. En (Meszaros et al., 2009) proponen técnicas de modelado visual para definir el comportamiento dinámico de lenguajes. Ellos se basan en la técnica de transformación de modelos, basado en el modelo de transformación reescritura basada en grafos. En (Knap et al, 2002) describen un proyecto que desarrolla un conjunto de herramientas, llamado HUGO, cuyo modelo de verificación de diseño utiliza los diagramas de máquinas de estado y de interacción de UML. El modelo de verificación asegura que un sistema funciona según lo especificado por dichos diagramas. El modelo de verificación detecta errores en los diseños, pero errores de codificación pueden todavía ocurrir. Por otro lado, investigaciones han probado que el meta-modelado es una forma de definir la sintaxis y el comportamiento dinámico en lenguajes de programación. Específicamente, una de esas formas de modelado, denominado DSLs (Domain-Specific Languages) por sus siglas en ingles, es propuesta en (Levendovszky et al., 2009) para automatizar procesos de generación de código de simulación de procesos. Como podemos ver, hay mucho interés en trabajos previos por el uso de modelos formales basados en las máquinas de estado finito o redes de Petri para especificar el comportamiento de un objeto. Algunos han propuestos metamodelos, reglas lógicas, o verifican comportamiento, nosotros en este trabajo proponemos generar código directamente desde UML.

3 Propuesta para Generar Código desde Máquinas de Estado Finito

La información contenida en los diagramas de clase de un modelo del UML brinda la base estática para la implementación en código fuente del modelo. Los diagramas de clase por sí solos no expresan información relevante sobre cómo deben implementarse las operaciones de una clase. Los diagramas de estado definidos en el contexto de una clase resultan de utilidad para representar el comportamiento dinámico de los objetos. Estos permiten describir el comportamiento que muestra una clase ante la recepción de eventos externos, y de esta representación es posible extraer detalles sobre cómo implementar las operaciones asociadas a eventos de llamadas en transiciones.

3.1 Patrón de implementación de diagramas de estado

Existen diversas maneras de implementar en código fuente las representaciones de diagramas del UML. Un diagrama de estado particular del UML puede ser implementado elegantemente en código fuente por un desarrollador, pero la estrategia que dicho desarrollador siga para la implementación puede ser de tal naturaleza que no permita implementar correctamente otros diagramas de estados, o incluso el mismo diagrama de estado con algunas modificaciones. En este sentido, se propone el uso de patrones de diseño para la automatización del proceso de implementación en código fuente desde diagramas UML, particularmente, desde el diagrama de máquinas de estado. En nuestro trabajo, un patrón de diseño es entendido como una solución genérica y repetible para un problema de ocurrencia recurrente en el desarrollo de software. Es importante contar con patrones bien definidos para la implementación de los diagramas del UML, pues estos patrones sirven como guía para el proceso de generación automática de código. El patrón de diseño presentado a continuación se encuentra basado en el patrón descrito en (Pinter et al., 2003). El objetivo de éste es la implementación de máquinas de estado definidas en el contexto de una clase.

Las máquinas de estado permiten representar el comportamiento de un objeto instancia de una clase a lo largo de su ciclo de vida. Muestran el conjunto de estados que puede tener un objeto, y cómo se producen transiciones entre dichos estados en respuesta a ciertos eventos. Expresado de otra manera, las máquinas de estado permiten describir el conjunto de acciones que un objeto debe realizar como respuesta a un evento (por ejemplo, la invocación de un método), dependiendo del estado particular en el que se encuentre.

3.1.1 Descripción del patrón de diseño

El elemento trascendental de nuestra propuesta es el uso de los diagramas de estado de UML para extraer la información del comportamiento dinámico del código a generar. En general, la estrategia de implementación de diagramas de estado se basa en la construcción de un mapeo entre pares estado-evento y un conjunto de acciones a ejecutar. Basado en ello, en nuestra propuesta definimos un patrón de diseño que caracteriza al diagrama de estado de UML como un diagrama de clases. El patrón propuesto permite caracterizar el diagrama de estado de UML como un diagrama de clases que implementa su funcionalidad. La figura 1 muestra el diagrama de clases para el patrón de diseño que implementa los diagramas de estado. Pasamos a describir dicho diagrama de clases. La máquina de estado se representa como una clase activa que compone a la clase *contexto*. Llamamos clase contexto a la clase cuyo comportamiento describe la máquina de estados. Los estados y transiciones que conforman la máquina de estados son representados a su vez como clases. Todas esas clases son descritas en la sección 3.1.2.

Las clases que representan a la máquina de estados, a los estados y a las transiciones, son todas especializaciones de un grupo de clases generales que definen funciones básicas. En lo sucesivo, a las clases especializadas para máquinas de estado, estados y transiciones, las llamaremos clases *particulares*. El patrón de diseño propuesto implica la sobre-escritura de métodos de las clases base, para proveer la funcionalidad descrita por la máquina de estados. Cada clase particular que represente a una máquina de estados debe sobre-escribir el método *inicializar()*. La clase base para las máquinas de estado es una clase abstracta, es necesario que sus subclases concretas sobre-escriban dicho método, e instancien en él a todas las clases particulares que representen los estados y transiciones, especificando, además, las relaciones de jerarquía que se den entre los estados. El código fuente resultante de la aplicación del patrón descrito es el de las operaciones asociadas a los eventos de

llamadas de las transiciones del diagrama de estado.

3.1.2 Clases base del patrón

La figura 2 muestra el diagrama de clases de la base del patrón de implementación, donde se especifican los atributos, métodos y asociaciones de cada clase. A continuación se describen las clases presentes en dicho diagrama.

- **Clase MáquinaEstado:** es la base para las máquinas de estado particulares definidas en el contexto de una clase. Es una clase activa, esto es, posee un hilo propio de ejecución en el que se procesan los eventos recibidos. Está compuesta por un conjunto de estados y transiciones, según lo indique la máquina de estados que representa. Dicha máquina posee una cola de eventos externos, donde se almacenan los eventos generados externamente y los eventos de tiempo, para su procesamiento. Además de la cola de eventos externos, posee una cola de eventos en la que se almacenan los eventos generados internamente.
- **Clase Estado:** Esta clase abstracta es la base para los estados particulares definidos por la máquina de estados. Los objetos de esta clase son capaces de ejecutar sus acciones asociadas.
- **Clase Transición:** Esta clase representa las transiciones definidas en la máquina de estados. Los objetos de esta clase son capaces de ejecutar la acción efecto asociada a la transición.
- **Clase ColaEventos:** representa la cola de eventos externos de una máquina de estados. La clase puede ser implementada como un envoltorio para una cola, por ejemplo una cola de la biblioteca STL de C++. La clase provee métodos para encolar y desencolar eventos, en orden FIFO, de modo seguro, para su uso por parte de distintos hilos.
- **Clase Evento:** Esta clase representa los eventos que se pueden recibir y procesar en la máquina de estados.
- **Clase ManejadorDisparo:** provee un envoltorio adecuado para una transición compuesta y un estado objetivo. La transición compuesta es la secuencia de transiciones habilitadas que se deben tomar como consecuencia del procesamiento de un evento. Esta secuencia de transiciones y el estado objetivo asociado es calculado por la operación *manejaEvento()* de cada estado particular. Luego es utilizada por la clase MaquinaEstado para ejecutar los efectos del disparo de transición.
- **Clase HiloDestinador:** es el responsable de ubicar los eventos en la cola de eventos de las máquinas de estado. Luego de ser creado, el hilo existe hasta que cumple con la tarea de ubicar el evento en la cola de eventos externos. La creación de estos hilos para ubicar los eventos en la cola tiene como finalidad evitar que el hilo de ejecución de la máquina de estados monopolice la cola de eventos externos. El hilo de ejecución de la máquina de estados puede monopolizar la cola de eventos, pues como consecuencia de acciones de estado es posible levantar eventos de llamada a operaciones asociados a las transiciones de la máquina de estados.
- **Clase Tope:** Esta subclase de la clase Estado representa el estado tope de las máquinas de estado. Toda máquina de estados instancia un objeto de este tipo.

3.1.3 Descripción del Código producido

Como ya se mencionó, el código producido por el patrón descrito implementa las operaciones de clase asociadas a eventos de llamada en las transiciones de la máquina de estados. Una máquina de estados definida en el contexto de una clase, debería permitir el uso de atributos y operaciones definidas por la clase contexto, con el mismo nombre con el que fueron definidas en dicha clase, y sin restricciones de visibilidad. Debido a que la máquina de estados es implementada también como una clase, es necesario definir en la clase contexto un conjunto de operaciones que permitan ejecutar las expresiones asociadas a las acciones y guardias definidos en la máquina de estados. La implementación en C++ de la máquina de estados implica también que ésta debe definirse como una clase *amiga* de la clase contexto, con el fin de que pueda hacer uso de sus características privadas. Adicionalmente, para la clase contexto se deben definir los atributos y operaciones que indiquen los diagramas de clase:

- Atributos para el objeto instancia de la máquina de estados particular, y para los identificadores de eventos de llamada y eventos de tiempo.

- Métodos para cada acción de estado: entrada, hacer y salida; cada guardia de transición, y cada efecto de transición.

Las clases particulares para la máquina de estados pueden ser declaradas de manera anidada en la clase contexto.

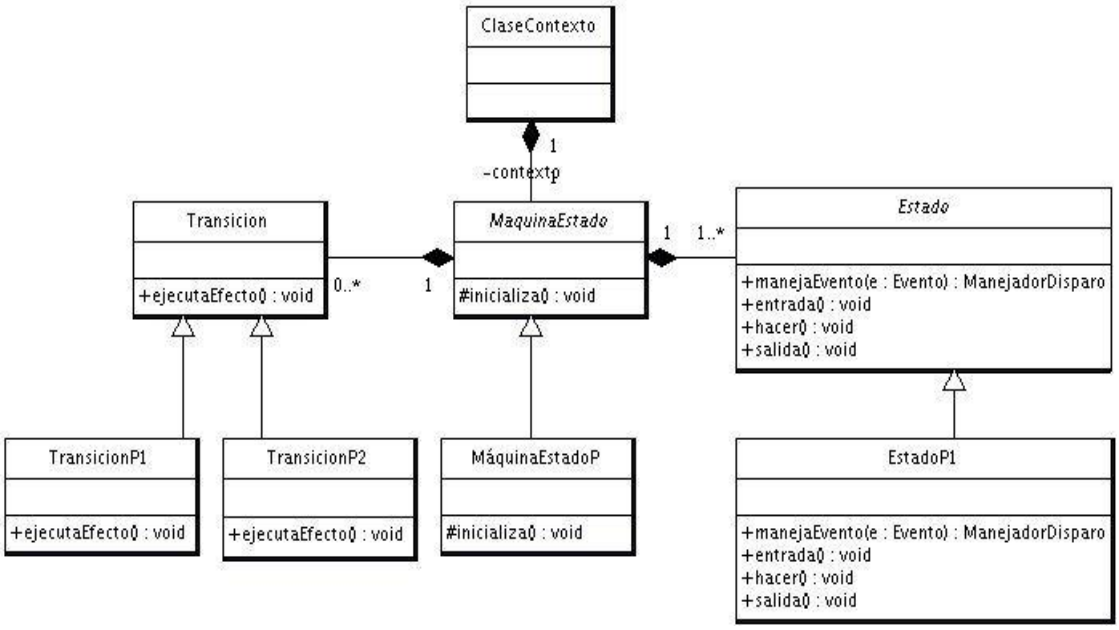

Fig. 1. Diagrama de clase para el patrón de implementación de máquinas de estado

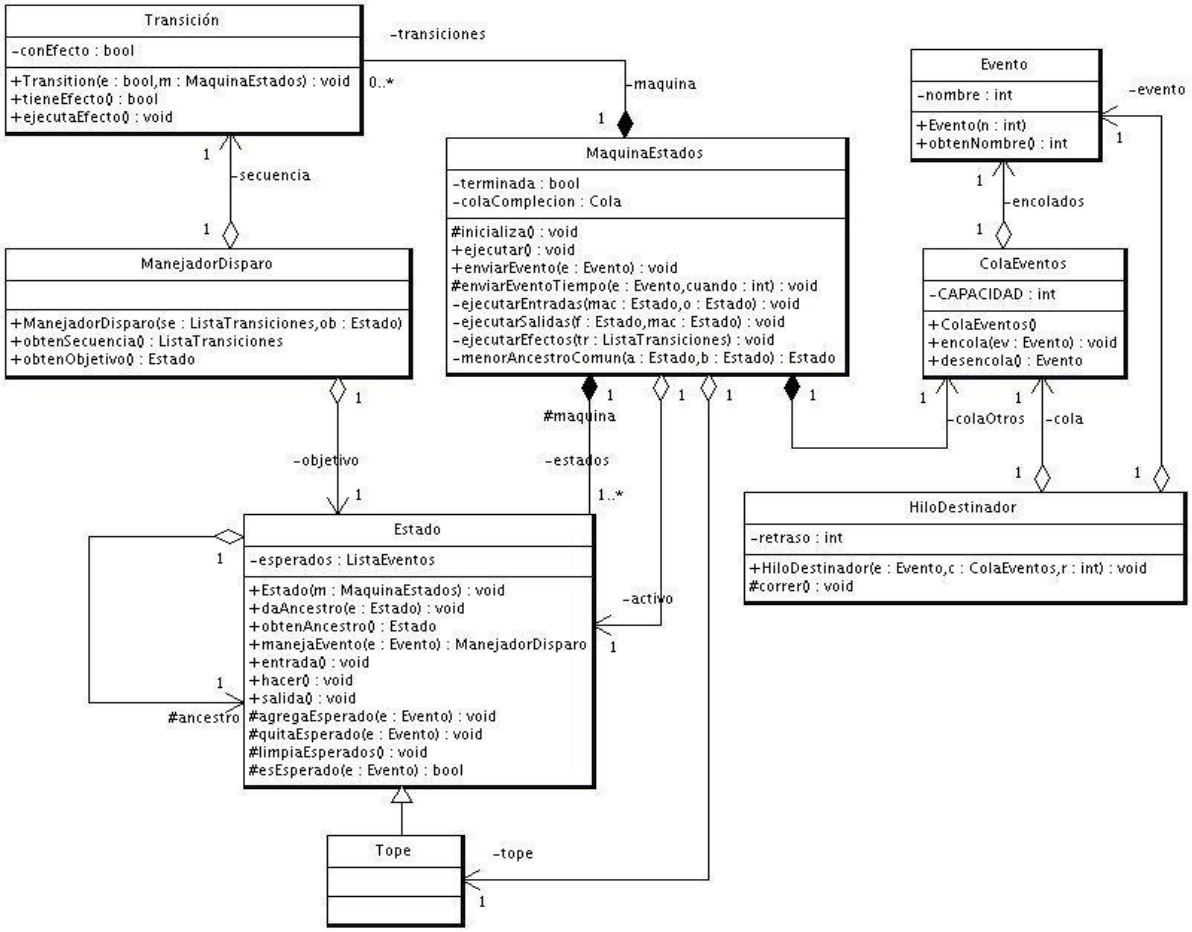

Fig 2. Diagrama de clases base del patrón de implementación

4 Herramienta

4.1 Alcances de la herramienta

La herramienta no está ligada a la generación automática de código para aplicaciones de algún tipo particular, permitiendo generar código para cualquier aplicación orientada a objetos que pueda modelarse mediante los elementos del UML soportados. Esta herramienta puede ser integrada en cualquier aplicación de modelado en el UML, que soporte los elementos del UML 1.4 reconocidos por la herramienta, y que permita exportar los modelos en el formato XMI 1.2. Se da soporte a los elementos presentes en diagramas de clases, diagramas de estados y diagramas de actividad. Además, se soporta la integración de representaciones del UML, reconociendo máquinas de estado definidas en el contexto de una clase. El lenguaje de implementación de la herramienta es el lenguaje de programación Python, y todo el software involucrado en la codificación de la herramienta es software libre.

4.2 Proceso de Generación de Código

La figura 3 representa el conjunto de actividades que realiza la herramienta durante el proceso de generación automática de código fuente.

- El generador de código toma como entrada un archivo en formato XMI, que describe un modelo expresado en el UML (Grose et al., 2002).
- El archivo de entrada es analizado por el *parser* de XMI, y el resultado de esta actividad es un conjunto de objetos (instancias de clases) que representan el modelo UML contenido en el archivo.
- Los objetos se transforman para simplificar su representación. Para esto se aplica, entre otras actividades: resolución de referencias, cálculo de dependencias, cálculo de espacios de nombre, clasificación y ordenamiento.
- Se generan, usando plantillas de texto, las cadenas de caracteres que representan el código fuente de la aplicación.
- Para favorecer su legibilidad y dar uniformidad al Código fuente, se realizan transformaciones al código obtenido (tabulación, eliminación de espacios en blanco, eliminación de líneas en blanco consecutivas, etc.)
- Finalmente, se procede a almacenar, organizados en directorios, los archivos de código fuente generados.

4.3 Implementación

Como se mencionó anteriormente, la codificación de la herramienta se realizó en ***Python***. La herramienta de generación de código hace uso de tres bibliotecas: dos bibliotecas para el análisis de archivos en XML, y una incluida con el motor para el procesamiento de plantillas de texto que permite que scripts en Python interactúen con el motor. Todas estas bibliotecas están escritas en Python. Para la lectura de archivos en XML se usan las bibliotecas *PyXML* y *4SuitXML*. Para el manejo de las plantillas se usa *Cheetah*. Adicionalmente, se usa una biblioteca hebras de QT 4.1.1 para dar soporte al manejo de hebras *POSIX*.

En cuanto al esquema funcional de la herramienta, en la versión actual trabaja en modo comando. Recibe como argumentos el nombre del archivo que contiene al modelo para el que se va a generar código, un identificador para el lenguaje objetivo de la generación de código, y el directorio del sistema de archivos en el que se desea almacenar el código fuente generado. Esto facilita tanto su integración con una herramienta para el modelado en UML, como su uso a modo de herramienta independiente. Debido a que en el lenguaje C++ existen numerosas características no soportadas directamente por el UML, surgió la necesidad de definir un conjunto de elementos del UML para poder expresar en los modelos dichas características. Así, se crearon estereotipos y definiciones de etiquetas. Los estereotipos reconocidos por la herramienta son

- *framework*: Sirve para indicar a la herramienta que las clases y elementos definidos en este paquete ya existen y son utilizados por la aplicación modelada. Por lo tanto, las clases definidas en este paquete o en paquetes anidados, directa o transitivamente, en el paquete, no son generadas en código fuente.
- *utility*: las clases que contengan un estereotipo utility ya existen, al igual que los paquetes framework, por lo que la implementación en código fuente de estas clases tampoco es generada.
- *create*: permite indicar que una operación de clase es un constructor.

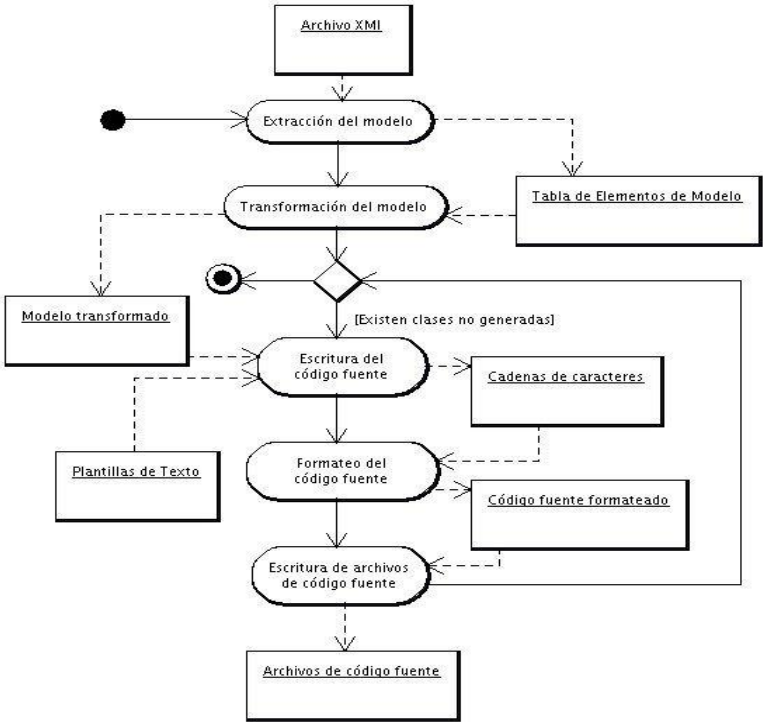

Fig. 3. Actividades de la generación de código

- *destroy*: permite indicar que una operación de clase es un destructor.
- *reference*: permite indicar que un atributo de clase es una referencia.

Las siguientes definiciones de etiquetas son reconocidas por la herramienta:

- *variables_declaration*: Permite expresar en el lenguaje objetivo de la generación de código, el conjunto de variables que deben ser declaradas dentro de una operación de clase.
- *pointer_status*: Permite expresar si un atributo es de tipo apuntador, apuntador a apuntador, etc.
- *array_status*: Igual que la anterior, pero permite expresar que un atributo de clase es una arreglo y su tamaño.
- *macro_definition*: Permite definir macros en el archivo de definición para una clase.
- *macro_expand*: Permite expandir macros en la definición de una clase.
- *header_include*: Permite incluir archivos de cabecera en el archivo de definición de una clase.
- *using_directive*: Permite declarar que en el archivo de definición de una clase se va a hacer uso de un espacio de nombres.
- *create_initializer*: Permite declarar para los constructores de clase, como deben inicializarse algunos atributos y a que constructores de clase llamar para la inicialización.

5 Caso de Estudio

El código fuente de la herramienta, así como de los casos de estudios presentados en este trabajo, generados a través de ella (al igual que los modelos del UML a partir de los cuales se produjeron), se pueden conseguir en: http://gennaproject.googlecode.com/svn/trunk/.

5.1 Reloj Digital

La primera aplicación simula un reloj digital. El modelado de la aplicación se llevó a cabo de la

siguiente manera: primero fueron construidas las representaciones de estructura estática, y luego sobre éstas fueron definidas las representaciones de comportamiento dinámico. Para las clases en las que se identificó un comportamiento complejo, fueron definidas las máquinas de estado.

5.1.1 Criterios de Evaluación

Para evaluar el grado en el que se corresponden el código fuente generado y el modelo de las aplicaciones, se optó por el uso de algunas métricas simples para sistemas orientados a objetos, descritas en (Booch, 1994). Estas incluyen una comparación del número de clases y el número de características por clase, especificadas en el modelo y presentes en el código fuente. También se presentan tablas con la cantidad de líneas de código "completas" e "incompletas" generadas para la aplicación, con la finalidad de dar una idea intuitiva del esfuerzo que permite ahorrar la herramienta en la implementación en código fuente de los modelos del UML.

Se consideran líneas de código "incompletas" todas aquellas que contengan expresiones escritas por el modelador en el lenguaje de programación objetivo de la generación de código (por ejemplo, las expresiones para guardias, acciones de estado, efectos de transición, definiciones de etiqueta, etc.). Las líneas de código "completas" son aquellas generadas automáticamente por la herramienta a partir de elementos presentes en el modelo que son independientes del lenguaje objetivo de la generación de código. Por ejemplo, son líneas "completas" las que indican la herencia entre clases (lo cual, en modelos del UML se expresa mediante relaciones de generalización). El punto importante de las líneas de código "completas" es que surgen de elementos de los modelos del UML que son independientes de los lenguajes de programación, esos elementos de modelo facilitan la generación de código en distintos lenguajes de programación.

Definir lo que es una línea de código también resulta difícil, pues para ello no existe un consenso general. En nuestro caso, estamos considerando como una "línea de código" en el lenguaje C++, todas las sentencias que terminan en ';' (expresiones aritméticas, de asignación, llamadas a funciones, etc.), bloques de estructuras de control (if(){}, switch(){}, case expr:, break;, while(){}, for(){}, do{}while();), las sentencias iniciales de la declaración de clase (class X {};), especificadores de visibilidad (public:, private:, etc.), declaraciones de miembros de dato, definiciones de funciones miembro, signaturas en declaraciones de funciones miembro, y directivas *include.*

5.1.2 Diseño del Reloj Digital

La aplicación implantada con el apoyo de la herramienta de generación de código es una aplicación gráfica que muestra la hora en formato digital, de la misma manera que un reloj de pulsera. Esta aplicación es presentada como ejemplo de modelado dinámico con máquinas de estado en (Pinter et al., 2003; Eriksson et al, 2004). Al modelo se le agregaron las siguientes funcionalidades: la posibilidad de establecer los segundos que muestra el reloj, la posibilidad de disminuir las horas, minutos y segundos, y un evento de tiempo que evita que el reloj esté inactivo tras establecer algún elemento de la hora.

El reloj digital posee cuatro modalidades: *Mostrar Hora*, en la cual la hora es desplegada y actualizada a cada segundo, y las modalidades *Establecer Hora*, *Establecer Minuto* y *Establecer Segundo*, en las que las horas, minutos y segundos que despliega el reloj pueden ser modificadas. El cambio de modalidad en el reloj ocurre cada vez que se presiona el botón *Mode*, y se da cíclicamente en el orden en el que se mencionan las modalidades. La figura 4, muestra un gráfico de pantalla con la aplicación Reloj Digital. Bajo las tres últimas modalidades, la presión de los botones *Inc* o *Dec* (ver figura 4) incrementa o disminuir la hora (minuto o segundo, según sea el caso), respectivamente. Si el reloj se encuentra en alguna de estas modalidades, y durante un lapso de 10 segundos no ocurre ningún intento por modificar las horas, los minutos o los segundos, según sea el caso, el reloj regresa a la modalidad *Mostrar Hora*.

Fig. 4. Aplicación Reloj Digital

Estructura estática de la aplicación: La figura 5 muestra el diagrama de clases de la aplicación, tres de las cuales (QPushButton, QWidget y QFrame)

son clases de la librería QT, y están contenidas en un paquete del modelo etiquetado con <<framework>>. La función de las otras dos clases (Clock y DigitalDisplay) se describe a continuación:

- Clock: Esta clase representa el motor del reloj, mantiene y actualiza las horas, minutos y segundos que deben mostrarse. La clase es un componente gráfico, para el que se identificó un comportamiento dinámico. Es una subclase de la clase QWidget de la librería QT. Posee tres asociaciones con la clase QPushButton (una por cada botón que debe mostrar), y una asociación con la clase DigitalDisplay. Esta clase presenta el comportamiento descrito en la figura7. Para la clase se definieron cinco operaciones, que se especifican en el modelo de la siguiente manera: i) Mediante máquinas de estado: *inc():void*, *modeButton():void* y *dec():void*. ii) Mediante expresiones en elementos *Method*: *Clock()* y *%Clock()*.
- DigitalDisplay: es también un componente gráfico que permite desplegar la hora gráficamente, en formato militar. La clase define tres operaciones especificadas de la siguiente manera: i) Mediante grafos de actividad: *paintEvent(e: QPaintEvent):void,* ii) Mediante expresiones en elementos *Method*: *DigitalDisplay()* y *show(h: int, m: int, s: int, high: int):void*.

Comportamiento dinámico de la aplicación: El diagrama de la figura 6 muestra la máquina de estados definida en el contexto de la clase Clock, por medio del cual se especificaron las operaciones inc, modeButton y dec.

5.1.3 Evaluación de resultados

La herramienta genera las definiciones de clases y características, conforme a la representación de estructura estática expresada en el modelo, tomando en cuenta todos los elementos presentes: clases, atributos, operaciones, relaciones de herencia, asociaciones, estereotipos y valores de etiqueta. El código fuente correspondiente a las representaciones de comportamiento dinámico del modelo también fue generado, conforme a la funcionalidad que expresan. El código fuente que implementa las operaciones especificadas en los diagramas de estado y diagramas de actividad fue producido tomando en cuenta acciones de estado, efectos de transición, eventos asociados a transiciones, etc. El código fuente generado es compilable y está ajustado a la funcionalidad expresada en el modelo.

Las tablas 1 y 2 muestran el número de características (atributos, asociaciones y operaciones) para cada clase en el modelo del Reloj Digital y en el código fuente generado. La diferencia entre ambas tablas se da en la clase Clock, la cual define una máquina de estados. El código fuente generado define diez atributos y quince operaciones más que las especificadas explícitamente en el modelo, además de que requiere de más asociaciones. Estas características son las requeridas por el patrón de implementación de máquinas de estado. Las veintiséis clases que se definen dentro de la clase *Clock* son las subclases de *MaquinaEstado*, *Estado* y *Transicion*, presentadas en la sección 3; estas subclases se corresponden con la máquina de estados, los dos estados iniciales, el estado final, los cuatros estados simples, el estado compuesto, y las diecisiete transiciones que muestra el diagrama de la figura 6.

Los datos presentados en la tabla 3 muestran la cantidad de líneas de código “completas” e “incompletas”. La tabla 3 indica que el 88% de las líneas de código generadas para la aplicación fueron líneas de código producidas en base a información expresada en el modelo, y que es independiente del lenguaje de implementación objetivo. Los datos que se ofrecen sólo permiten dar una idea intuitiva del esfuerzo que debe llevar a cabo un programador para terminar la aplicación, trabajando sobre el código fuente generado, si el modelo de entrada no tuviera ninguna expresión en el lenguaje de programación objetivo C++. Un programador debería agregar expresiones en 83 líneas de código, y crear una función principal en C++, para que la aplicación del Reloj Digital pudiera ser compilada y ejecutada con todas sus funcionalidades. Las expresiones que serían necesario agregar son todas expresiones sencillas del lenguaje C++ (ejemplos serían las expresiones en C++ de los guardias de transición, y de las acciones de los estados, mostradas en la figura 6). Por otro lado, la declaración de la función principal tampoco requiere mucho esfuerzo (sólo requiere la declaración de una variable de aplicación QT y de una variable de tipo Clock).

Tabla 1. Número de atributos, asociaciones y operaciones, para cada clase en el modelo del Reloj Digital

Clase	Atrib.	Asoc.	Oper.
Clock	3	4	5
DigitalDisplay	6	0	3

Tabla 2. Número de atributos, asociaciones y operaciones, para cada clase en el modelo del Reloj Digital en el código fuente generado con la herramienta del Reloj Digital

Clase	Atrib.	Asoc.	Oper.
Clock	13	26	20
DigitalDisplay	6	0	3

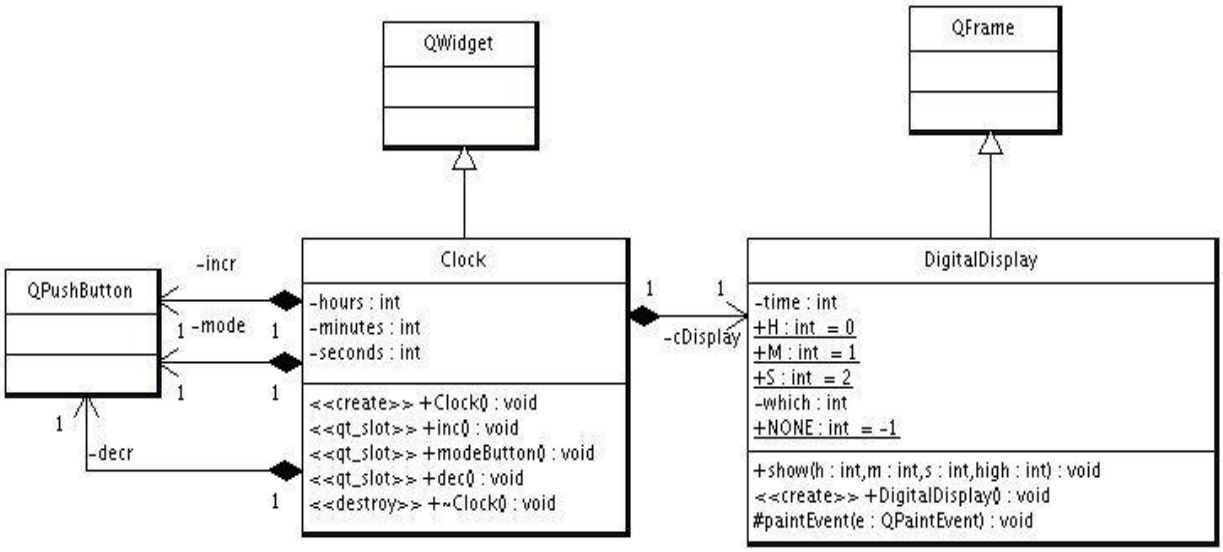

Fig. 5. Diagrama de clases del modelo del Reloj Digital

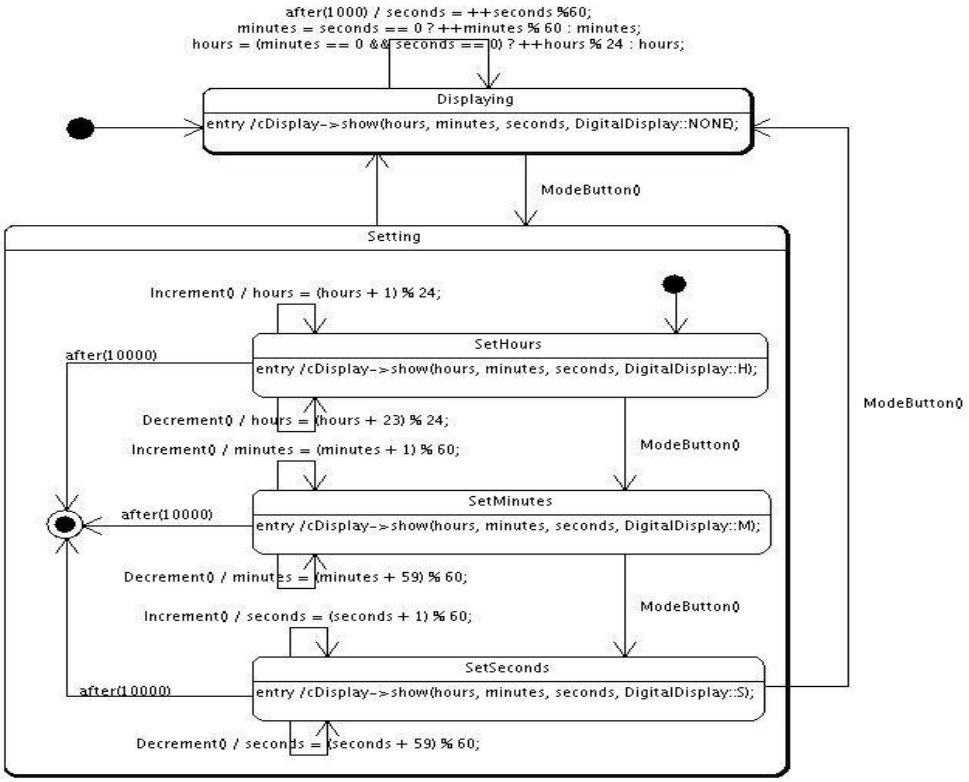

Fig. 6. Diagrama de máquina de estado definida en el contexto de la clase Clock

Tabla 3. Líneas de código generadas para el modelo del Reloj Digital

Clase	Líneas de código		%Líneas		Total
	Completas	Incompletas	Completas	Incompletas	
Clock	557	54	91	9	611
DigitalDisplay	58	29	67	33	87
Total	615	83	88	12	698

5.2 Juego de Tetris

5.2.1 Descripción del problema

El juego de Tetris consiste en un conjunto de bloques de formas distintas, a los que se llama tetrominoes, que caen y se acumulan en el fondo de en un pozo, por efecto de una gravedad simulada. El objetivo del juego es evitar que los tetrominoes lleguen el tope del pozo durante la mayor cantidad de tiempo posible. Cuando los tetrominoes forman una línea completa desde uno de los muros del pozo, hasta el muro opuesto, dicha línea desaparece y todos los bloques sobre esa línea caen efecto de la gravedad simulada. El jugador puede mover los bloques hacia la derecha del pozo, hacia la izquierda, o puede rotar los bloques, mediante el teclado. Puede también mover los bloques hacia abajo de dos maneras: puede iniciar una "caída suave", esto es, el bloque se mueve hacia abajo un solo espacio, o puede llevar a cabo una "caída dura", o sea, el bloque cae

inmediatamente hasta el fondo del pozo o hasta que colisione con otros bloques en el fondo del pozo. Cada vez que un bloque llega al fondo del pozo, otro bloque es seleccionado de manera aleatoria por el juego, y comienza su caída desde el tope. La velocidad de caída de los bloques aumenta a lo largo del juego, cada vez que el jugador alcanza un cierto número de líneas completas de bloques (por ejemplo: cada vez que el jugador completa diez líneas). El puntaje del jugador aumenta cada vez que un nuevo tetromino es generado, y cada vez que el jugador completa una línea de bloques. El juego termina cuando la pila de bloques que comienza en el fondo, alcanza el tope.

5.2.2 Modelo de la aplicación

Estructura estática: Las clases definidas y sus operaciones son.

- Block: representa un bloque o tetromino del juego tetris. El bloque puede tener siete formas distintas. La forma y la ubicación del bloque se representa en un arreglo que contiene las coordenadas del pozo, en las que se ubica el bloque. La clase provee operaciones para calcular las nuevas coordenadas del bloque luego de los movimientos. Las operaciones fueron especificadas mediante grafos de actividad.
- Playground: representa el pozo en el que caen los bloques. La clase provee operaciones para acumular los bloques cuando llegan al fondo, para limpiar las líneas completadas, y para generar nuevos bloques aleatoriamente. Las operaciones fueron especificadas mediante grafos de actividad.
- Engine: es un motor de juego, esta clase se encarga de hacer las llamadas a los métodos correspondientes conforme el jugador genera eventos, y se encarga de crear los eventos de tiempo que simulan la gravedad. El diagrama de la máquina de estados de la figura 7 describe el comportamiento de Engine.
- NextWidget: es un componente de interfaz gráfica que renderiza el próximo bloque que va a caer. Las operaciones fueron especificadas mediante grafos de actividad.
- PlaygroundWidget: es el componente de interfaz gráfica que renderiza el pozo en el que caen los bloques. Las operaciones fueron especificadas mediante grafos de actividad.
- TetrisWindow: componente de interfaz gráfico para la ventana del juego. Incluye botones para pausar el juego, comenzar un nuevo juego y terminar la aplicación. Las operaciones fueron especificadas mediante grafos de actividad.

Comportamiento dinámico: La figura 7 muestra la máquina de estados definida en el contexto de la clase Engine. Dicha máquina de estado presenta características no presentes en la máquina de estados del Reloj Digital, que son: transiciones compuestas y pseudoestados de decisión.

5.2.3 Evaluación de resultados

Se generaron todas las definiciones de clases, sus características y relaciones conforme a lo expresado en el modelo. El código fuente correspondiente a las representaciones de comportamiento dinámico también está ajustado a la funcionalidad expresada en ellas. Las operaciones definidas en la máquina de estado funcionan conforme a lo expresado en el modelo, el código fuente generado para la máquina de estado (que incluye eventos de tiempo, puntos de decisión, transiciones compuestas, etc.) mostrada en el diagrama de la figura 7 es correcto. La tabla 4 muestra el conteo de características entre el modelo y el código fuente para la aplicación del Tetris de la clase Engine (para el resto de clases no hubo diferencias). El mismo requiere de más características que las definidas en el modelo.

Tabla 4. Número de atributos, asociaciones y operaciones, para la clase Engine del Modelo y del Código generado

Engine	Atrib.	Asoc.	Oper.
Modelo	8	2	17
Código Generado	19	38	33

La tabla 5 muestra el conteo de líneas de código completas e incompletas, para cada una de las clases generadas por la herramienta. La interpretación es similar a la de los datos de la tabla 3. En este caso existen 339 líneas de código incompletas, lo cual representa el 20% del total de las líneas de código generadas.

La generación del código correspondiente a la base estructural que proveen los diagramas de clases libra al programador de la tarea (en ocasiones tediosa) de declarar las clases y sus atributos en el lenguaje de implementación, y provee los fundamentos para la integración en el modelo de las representaciones dinámicas. Los diagramas de estado resultan un medio poderoso para el

modelado del comportamiento de los objetos. La implementación manual en código fuente de la funcionalidad descrita por medio de las máquinas de estado resulta una tarea más difícil que el modelado de las mismas. En este sentido, la herramienta de generación de código resulta un medio idóneo para acelerar el proceso de desarrollo de las dos aplicaciones. Sobre el uso de diagramas de actividades para especificar las acciones que deben llevar a cabo las operaciones, no es un requisito para la generación de código con la herramienta, y utilizarlos queda como decisión de los desarrolladores.

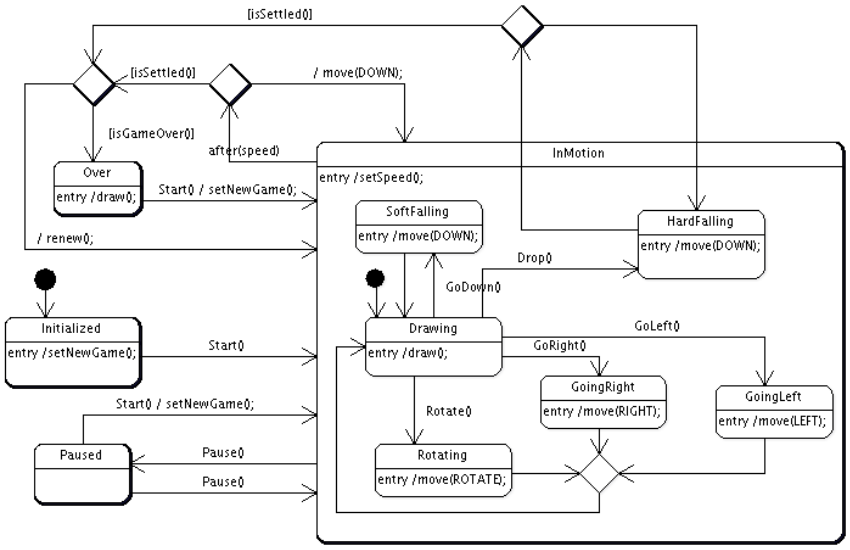

Fig. 7. Diagrama de máquina de estado definida en el contexto de la clase Engine

Tabla 5. Líneas de código generadas

Clase	Líneas de código		%Líneas		Total
	Completas	Incompletas	Completas	Incompletas	
Block	122	47	72	28	169
Playground	154	57	73	27	211
Engine	884	100	90	10	984
PlaygroundWidget	59	40	60	40	99
NextWidget	43	28	61	39	71
TetrisWindow	58	37	61	39	95
Total	1351	339	80	20	1690

6 Conclusiones

La experiencia de utilizar la herramienta para el desarrollo de aplicaciones, dejó en claro cuáles de los elementos del UML considerados en el presente trabajo son los más útiles para el proceso de generación de código: los elementos de diagramas de clases y las máquinas de estados. Los diagramas de estado resultan un medio poderoso para el modelado del comportamiento de los objetos. La implementación manual en código fuente de la funcionalidad descrita por medio de las máquinas de estado, resulta una tarea más difícil que el modelado de las mismas. En este sentido, la herramienta de generación de código resulta de utilidad para acelerar el proceso de desarrollo. En general, el código fuente generado por la herramienta es correcto y compilable, en la medida en que los modelos que le sirven de entrada estén definidos con precisión. Aún cuando los modelos que sirven de entrada a la herramienta no posean el grado de detalle requerido para que el código fuente generado sea compilable, éste puede ser completado manualmente. Como último comentario, debemos mencionar que el UML no permite expresar de manera independiente a los lenguajes de programación, las expresiones que especifican las acciones a realizar en operaciones, acciones, las condiciones de guardias, etc. El uso de dichas expresiones en los modelos los hace dependientes de los lenguajes de programación objetivo. Si se deseara generar código fuente para los mismos modelos en otros lenguajes de programación, dichas expresiones tendrían que ser sustituidas por expresiones en el nuevo lenguaje objetivo.

Como trabajos futuros se deben hacer implementaciones de la herramienta que generen código en otros lenguajes de programación. Además, se deben realizar una mayor cantidad de pruebas de generación de código sobre otros dominios de uso frecuente. La herramienta de generación de código aún no soporta el chequeo sintáctico y semántico automático de los modelos que le sirven de entrada, por lo que un nuevo componente que realice dicha tarea esta en desarrollo, basado en la herramienta de verificación de modelos SPIN [29]. Tampoco en la presente versión se considero la generación de código fuente para elementos que expresan procesamiento concurrente en los diagramas de estados. La herramienta será extendida para incluir la implementación de los elementos de concurrencia que pueden modelarse en dichos diagramas.

Referencias

1. **ArgoUML Project Home.** (s.f.). Retrieved from http://argouml.tigris.org/
2. **Bell, R. (1998).** Code Generation from Object Models. *Embedded Systems Programming*. 11 (3) 74 – 88
3. **Booch, G., Maksimchuk R., Engle M., Young B., Conallen J., & Houston K. (2007).** *Object-Oriented Analysis and Design* with applications (3rd ed.). Upper Saddle River, NJ : Addison-Wesley
4. **BOUML User Manual,** (s.f.). Retrieved from http://bouml.free.fr/doc/index.html
5. **Eriksson, H., Penker, M., Lyons, B., Fado, D. (2004)** *UML 2 Toolkit*. Indianapolis, Ind. : Wiley Publishing.
6. **Grose, T., Doney, G. C., Brodsky, S.A. (2002).** *Mastering XMI: Java programming with XMI, XML and UML*. New York: John Wiley.
7. **Herrington, J. (2003).** *Code Generation in Action*. Greenwich: Manning Publications.
8. **Knapp A. Merz S. (2002).** Model Checking and Code Generation for UML State Machines and Collaborations, *5th Workshop Tools for System Design and Verification*, Augsburg, Alemania, 59-64.
9. **Levendovszky T, Meszaros T (2009).** Tooling the Dynamic Behavior Models of Graphical DSLs*, Human-Computer Interaction. Novel Interaction Methods and Techniques*, *Lecture Notes in Computer Science*, 5611, 830-839.
10. **Meszaros T., Levendovszky T., Mezei G. (2009).** Code Generation with the Model Transformation of Visual Behavior Models. *Electronic Communications of the EASST,* 21, 110-119
11. **Muñeton A., Zapata C.M., Arango F. (2007).** Reglas para la Generación Automática de Código definidas sobre Metamodelos Simplificados de los Diagramas de Clases de Secuencias y Máquinas de Estado de UML2.0, *Dyna,* 74 (153), 267-283.
12. **Pilone, D., Pitman, N. (2005).** *UML 2.0 in a Nutshell*. Sebastopol, CA: O'Reilly Media.
13. **Pinter, G., Majzik, I. (2003).** Program Code Generation Based on UML Statechart Models. *Periodica Polytechnica-Electrical Engineering*. 47 (3-4), 187-204.
14. **Umbrello UML Modeller** (s.f.). Retrieved from http://uml.sourceforge.net/index.php
15. **Zapata G., Branch J. Quintero L.F. (2007).** Metodología para el Modelado y Generación de Código de Control de Sistemas Secuenciales mediante Redes de Petri Jerárquicas, *Revista Avances en Sistemas e Informática*, 4 (1), 59-65

Mario Rincón Nigro

Recibió el título de Ingeniero de Sistemas de la Universidad de Los Andes, Mérida, Venezuela en 2007. Actualmente cursa estudios doctorales en la University of Houston, Houston TX, USA, y pertenece al Computer Graphics & Interactive Media Lab en la misma institución. Sus intereses incluyen computación de altas prestaciones en GPUs y algoritmos de renderización.

José Aguilar Castro

Obtuvo una Maestría en Informática en 1991 en la Universidad Paul Sabatier-Toulouse-France, y el Doctorado en Ciencias Computacionales en 1995 en la Universidad Rene Descartes-Paris-France. Además, realizó un Postdoctorado en la Universidad de Houston entre 1999 y 2000. Él es Profesor del Departamento de Computación de la Universidad de los Andes e investigador del Centro de Microcomputación y Sistemas Distribuidos (CEMISID) de la Universidad de los Andes. Dr. Aguilar ha sido profesor/investigador visitante en varias universidades y laboratorios (Universite Pierre et Marie Curie Paris-France, Laboratoire d'Automatique et Analyses de Systemes Toulouse-France, University of Houston-USA, Universidad Complutense Madrid-España, Institute National de Recherche en Informatique Niza-Francia, entre otros).Sus áreas de interés son los Sistemas Paralelos y Distribuidos, Computación Inteligente, Optimización Combinatoria, Reconocimiento de Patrones, y Automatización Industrial. Ha publicado más de 250 artículos en revistas, libros y actas de congresos y ha participado en diversos proyectos de investigación financiados, entre otros, por Petróleos de Venezuela S.A. (PDVSA), FONACIT, CNRS. Es miembro del Sistema Nacional de Promoción del Investigador (PPI) Nivel IV.

Francisco Hidrobo Torres

Se graduó de Ingeniero de Sistemas en la Universidad de Los Andes (ULA) en 1993; luego obtuvo el grado de MSc. en ComputaciÃ³n en la misma Universidad. En 2003 obtiene el Diploma de Estudios Avanzados (DEA) y en 2004 el grado de Doctor en Informática, ambos en la Universidad Politécnica de Catalunya (España). Trabaja en las áreas de computación inteligente y computación de alto rendimiento, en las cuales ha publicado varios trabajos en conferencias internacionales y revistas especializadas. Asesor en diversos proyectos de investigación y desarrollo llevados en conjunto entre la Universidad de los Andes y Petróleos de Venezuela S.A. (PDVSA). Profesor Titular de la Facultad de Ciencias de la Universidad de Los Andes. Miembro del Sistema Nacional de Promoción del Investigador (PPI) Nivel I y del Programa de Estímulo al Investigador (PEI) de la ULA desde 1997.