6º WORKSHOP IBEROAMERICANO
de Ingeniería de Requisitos y Ambientes Software

Ideas
2003

30 de abril
al 2 de mayo
de 2003

Asunción - Paraguay

Editores:
Mario Piattini
Luca Cernuzzi
Francisco Ruíz

Memorias

6º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software

Asunción, Paraguay
del 30 de Abril al 2 de Mayo del 2003

Memorias

Editores
Mario Piattini
Luca Cernuzzi
Francisco Ruíz

Organización
Proyecto WEST/CYTED
Universidad Católica "Nuestra Señora de la Asunción"
Universidad Autónoma de Asunción

ISBN: 84-96023-05-2

ISBN: 84-96023-05-2

Incluye ponencias en español, portugués e inglés.
A la cabeza de la portada: 6° Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software.

1. Ingeniería de Requisitos. 2. Tecnología de Procesos. 3. Orientación a Objetos. 4. Métodos Formales en Ingeniería del Software. 5. Ambientes y Herramientas de Desarrollo. 6. Calidad y Estimación de Software. 7. Desarrollo Basado en Componentes. 8. Ingeniería del Software para Sistemas Concurrentes y Distribuidos. 9. Bases de Datos. 10. Almacenes de Datos y Minería de Datos. 11. Bibliotecas Digitales. 12. Ingeniería Web.

Esta actividad fue patrocinada por: Proyecto VII.18. WEST (*Web-based Software Technology, http://www.dsic.upv.es/~west/*), CYTED (*Programa Iberoamericano de Ciencia Y Tecnología para El Desarrollo, http://www.cyted.org/Nueva.asp*), La Universidad Católica "Nuestra Señora de la Asunción", Sede Regional de Asunción, y la Universidad Autónoma de Asunición. Abril del año 2003.

PRÓLOGO

Este volumen recoge los trabajos presentados en el 6° Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software (IDEAS´2003) celebrado en Asunción del 28 de abril al 2 de mayo de 2003.

Sin duda, IDEAS se ha consolidado como el foro de encuentro y discusión para el intercambio de experiencias y conocimientos entre los principales investigadores de la Ingeniería del Software en el ámbito Iberoamericano.

En esta edición se han recibido un total de 47 trabajos de gran calidad científica, lo que avala, por un lado, la creciente madurez del área y, por otro, el prestigio del Workshop, fruto del buen hacer de las ediciones anteriores. Cada trabajo ha sido evaluado por, al menos, dos miembros del comité de programa, no habiendo prácticamente evaluaciones divergentes. Con el fin de mantener la tradición de no desarrollar sesiones paralelas así como el nivel alcanzado, se decidió aceptar como comunicaciones largas un total de 23, y como comunicaciones cortas otras 12.

Además de las sesiones técnicas, esta edición ha apostado por las conferencias invitadas, que se han agrupado en tres sesiones, la primera dedicada a la tecnología Web que incluye las conferencias: "Modelado Conceptual en la Web (Oscar Pastor)", "La Tecnología .NET en el desarrollo de Software para la Web (Miguel Katrib)", y "Evaluación y Calidad de Sitios WEB" (Luis Olsina); la segunda a la Ingeniería del Software, con las conferencias: "Ingeniería de Requisitos" (Jaelson Castro), "Tecnología J2EE para el Desarrollo de Aplicaciones Distribuidas" (Raúl Monge), y "Procesos y Arquitecturas de Software" (Ernesto Pimentel); y la tercera a tendencias en ambientes Web, con las conferencias invitadas de Ramez Elmasri sobre los temas: "Base de Datos, Comercio Electrónico, Web, y XML: ¿Cómo están relacionados?" y "Extracción de Ontologías y Modelado Conceptual para Información en la Web".

Resulta imprescindible aprovechar estas líneas para agradecer los conferenciantes invitados y a la comunidad científica su apoyo a este Workshop, enviándonos sus trabajos. Asimismo, nuestra gratitud a todos los miembros del Comité de Programa por el esfuerzo realizado. Igualmente debemos reconocer el patrocinio del proyecto WEST-CYTED, de la Universidad Católica "Nuestra Señora de la Asunción", Sede Regional de Asunción y la Universidad Autónoma de Asunción. Por último, queremos agradecer a todos los miembros del Comité Organizador su entrega y dedicación que ha hecho posible la celebración de este Workshop que esperamos resulte del interés de todos los que en él participan.

Juan de Dios Garbett
Presidente del Comité Organizador–UAA

Luca Cernuzzi
Presidente del Comité Organizador–UCA

Mario Piattini
Presidente del Comité de Programa

Comité de Programa

Presidencia:

Mario Piattini *(U. Castilla-La Mancha, España)*

Miembros:

José Bogarín *(U. Católica de Asunción)*
Pere Botella *(U. Politécnica de Catalunya)*
Nieves Brisaboa *(U. de La Coruña, España)*
Antonio Brogi *(U. de Pisa, Italia)*
Rodrigo Cardoso *(U. de los Andes, Colombia)*
Jaelson Castro *(U. de Pernambuco, Brasil)*
Carmen Costilla *(U. Politécnica de Madrid, España)*
Luca Cernuzzi *(U. Católica de Asunción, Paraguay)*
Joao Falcao e Cunha *(U. do Porto, Portugal)*
Alexander Gelbukh *(IPN, México)*
Luis Joyanes *(U. Pontificia de Salamanca, España)*
Carlos Heuser *(UFRGS, Brasil)*
Julio C. Leite *(PUC-Rio, Brasil)*
Hanna Oktaba *(UNAM, México)*
Miguel Katrib *(U. de La Habana, Cuba)*
Esperanza Marcos *(U. Rey Juan Carlos, España)*
Mauricio Marín *(U. de Magallanes, Chile)*
Alfredo Matteo *(U. Central de Venezuela)*
Raul Monge *(UTFSM, Valparaiso, Chile)*
Luis Olsina *(U. La Pampa, Argentina)*
Oscar Pastor *(U. Politécnica de Valencia, España)*
Rodolfo Pazos *(CENIDET, México)*
Francisco Pinheiro *(U. de Brasilia, Brasil)*
Ernesto Pimentel *(U. de Málaga, España)*
Isidro Ramos *(U.Politécnica de Valencia ,España)*
António Rito da Silva *(U. Técnica de Lisboa)*
Gustavo Rossi *(U. Nacional La Plata, Argentina)*
Francisco Ruiz *(U. Castilla-La Mancha, España)*
Miguel Toro *(U. de Sevilla, España)*
Ambrosio Toval *(U. de Murcia, España)*
Antonio Vallecillo *(U. de Málaga, España)*
Amparo Vila *(U. de Granada, España)*
Claudia Werner *(U. Federal de Rio, Brasil)*

Comité de Organización

Presidencia:

Luca Cernuzzi: *Universidad Católica "Nuestra Señora de la Asunción", Sede Regional Asunción*
Juan de Dios Garbett: *"Universidad Autónoma de Asunción"*

Miembros:

Universidad Autónoma de Asunción.
Hugo Correa
Ivey Díaz
Flavio Jodorcovsky
Blanca de Báez
Fernando Garbett
Universidad Católica "Nuestra Señora de la Asunción", Sede Regional de Asunción.
Vicente González Ayala
Norma Morales
Magalí González

Indice

Artículos Extendidos

Artículos Breves

www.uc.edu.py www.uaa.edu.py www.cyted.org

Una técnica de descripción formal para la generación y ejecución automática de casos de prueba de sistemas orientados a objetos

María del Mar Jiménez, Macario Polo, Mario Piattini
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4; 13071-Ciudad Real (Spain)
{Mar.Jimenez, Macario.Polo, Mario.Piattini}@uclm.es

Resumen

Se presenta un método para la automatización de la generación y la ejecución de casos de prueba (Test Cases, TC) de caja negra para Sistemas Orientados a Objetos (SOO). Para ello utilizamos una especificación formal del SOO desarrollada a partir del diagrama de clases del mismo y proponemos un algoritmo para la generación de TC. Para ejecutarlos se ha desarrollado una herramienta inicial que se irá completando a medida que se avance en la investigación.

1. Introducción.

La definición de '*Testing*' según el estándar IEEE de 1983 es "el proceso de realizar o evaluar un sistema o componente de un sistema de forma manual o automática para verificar que satisface los requisitos esperados, o para identificar diferencias entre los resultados esperados y los reales". El software debe ser probado para poder asegurar su correcto funcionamiento. La automatización de la fase de pruebas puede reducir significativamente el esfuerzo requerido para el desarrollo de pruebas adecuadas y reducir el tiempo de realización y ejecución de las mismas [12]. De acuerdo con [25], las investigaciones llevadas a cabo en este sentido son aún poco rigurosas, los programas y tipos de errores elegidos son poco representativos (por tamaño o número o tipos de errores), etc., lo que evidencia que hay realmente una fuerte necesidad de investigar en este sentido.

Actualmente, los sistemas de información de la mayoría de las compañías están creciendo mucho en complejidad. De hecho, las nuevas tecnologías en el desarrollo de software y las infraestructuras técnicas son cada vez más poderosas y nos permiten alcanzar un mayor grado de flexibilidad, por otro lado esto se traduce en soluciones más complejas para los sistemas software finales y su mantenibilidad se convierte en una tarea muy cambiante. Estos nos hace enfatizar los casos, actualmente muy comunes, donde los sistemas modernos necesitan coexistir con los antiguos. En este escenario, la realización de pruebas a nuevas funciones o a cambios de los antiguos sistemas ha llegado a ser una tarea muy demandada [18] .

Las pruebas software nos permiten comprobar cómo una serie de datos de entrada producen una serie de datos de salida en un determinado trozo de código [24]. La generación manual de algunos casos de prueba puede consumir bastante tiempo, y la fase de pruebas del software ha llegado a ser una de las más caras del ciclo de vida. La realización de las pruebas cuesta entre un 40 y un 75 por ciento del tiempo de las fases de desarrollo y mantenimiento del ciclo de vida software [4] [14]. A pesar de esto, la pruebas software han sido la técnica más usada para asegurar la calidad del software [2].

En este artículo se presenta un método para generar y ejecutar automáticamente casos de prueba a partir de la estructura estática de un sistema orientado a objetos. En la siguiente sección se hace un breve repaso a algunos trabajos relacionados; en la Sección 3 describimos los conceptos relevantes para nuestro objetivo mediante una técnica de descripción formal. La sección 4 presenta el método para generar automáticamente los casos de prueba apoyándose en la descripción anterior, mientras que en la sección 5 se presentan los pasos que hemos dado hacia la automatización de este método. Por último, incluimos nuestras conclusiones y las líneas de trabajo futuro.

2. Trabajos relacionados.

En los últimos veinte años se ha dedicado un gran esfuerzo a desarrollar métodos de prueba que detecten defectos en los programas con un bajo coste [13], existiendo dos enfoques principales para la realización de las pruebas: de ***caja blanca*** (en las que se dispone del código fuente del programa que se va a probar) y de *caja negra* (el código fuente no está disponible o no interesa). Ambos enfoques difieren en dos aspectos de manera importante [1]: el modo en que seleccionan los casos de prueba (basándose sólo en la interfaz, en las de caja negra, o teniendo además en cuenta la implementación de los algoritmos y de las estructuras de datos en las de caja blanca) y el modo en que se ejecuta la prueba (accediendo sólo a los miembros públicos en las pruebas de caja negra, o también a los privados en las de caja blanca, en las que se registra además la traza seguida por el programa).

El uso de métodos formales es una alternativa que ofrece excelentes resultados, ya que se ajustan muy bien a la realización de pruebas automáticas de programas. De acuerdo con Hierons [17], es sin embargo desafortunado que estos lenguajes se utilicen casi exclusivamente para el desarrollo de sistemas críticos. Además, Bowen y Hinchey [5] indican que sigue siendo necesario probar el sistema final desarrollado porque pueden surgir errores al generar el código y porque son raros los sistemas desarrollados exclusivamente con métodos formales. Por otro lado, el abaratamiento de los costes de la fase de pruebas se debe a que éstos han sido desplazados a las de diseño y desarrollo [5].

Un enfoque más liviano sugiere el uso de lenguajes formales para especificar algunas características del componente que se está probando:

- Doong y Frankl [9] describen la estructura y comportamiento de las clases mediante un lenguaje formal, de manera que pueden generar automáticamente valores y secuencias de métodos para ser utilizados como casos de prueba.
- Uno de los problemas del método anterior es que pueden generarse secuencias de métodos sin sentido y que no tendrán lugar en tiempo de ejecución. Para resolverlo, Kirani y Tsai [20] proponen incluir en cada clase un conjunto de expresiones regulares que representen las posibles secuencias de métodos correctas. Puesto que el código del método puede contener, por ejemplo, instrucciones condicionales que no pongan al objeto en el estado deseado, esta solución mejora la anterior pero no es tampoco una garantía de éxito.
- Kung et al. [22] traducen la especificación formal de la clase a una máquina de estados y genera secuencias de eventos compatibles con ésta. Ambos elementos son procesados por una herramienta de ejecución simbólica, que puede detectar algunos errores. La ejecución simbólica se encuentra con grandes dificultades al procesar bucles, instrucciones condicionales, llamadas a rutinas, punteros o recursividad ([7], [8]), por lo que tiende a ser cada vez menos utilizada.

De acuerdo con Edwards [10], las técnicas anteriores están muy limitadas en la generación de valores de prueba cuyos tipos de datos son complejos.

Recientemente han aparecido nuevas técnicas que proponen el uso de bibliotecas de clases específicamente diseñadas para la realización de pruebas [3], que son la base para el desarrollo del sistema ("el software se prueba antes de tenerlo").

Cuando se modifica un producto software debe ser sometido a *pruebas de regresión*, en las que se le somete a casos de prueba superados por la versión original del programa con objeto de comprobar que la versión modificada no contiene errores que antes no existían. Las pruebas de regresión tienen la ventaja de que los casos de prueba o bien su descripción se encuentran archivados y no es entonces necesario dedicar recursos a construirlos [27]. Aún así, la reejecución de todos los casos de prueba de un programa de 20.000 líneas de código puede requerir siete semanas para su ejecución [31], por lo que se trabaja en dos enfoques principales para reducir este esfuerzo: la *reducción del conjunto de casos de prueba* y la *priorización de casos*.

De acuerdo con Meudec [23], los esfuerzos en la automatización de la verificación y prueba del software deben dedicarse principalmente a las tres siguientes áreas, si bien las dos últimas son las que se encuentran en un estado más atrasado:

1. Automatización de tareas administrativas, mediante las cuales se mantiene el control del proceso de pruebas, permitiendo la generación de informes, etc.
2. Automatización de tareas de generación de casos de prueba, mediante las cuales se generan automáticamente casos de prueba.
3. Automatización de tareas mecánicas, en las que se ejecutan los casos de prueba.

Las herramientas comerciales para la realización de pruebas de programas constan de pseudolenguajes que permiten escribir pequeños *test scripts*, en los que el ingeniero de programas especifica la operaciones que se ejecutarán, instrucciones de traza, bucles, etc. [12]. De acuerdo con estos autores, la escritura de *scripts* es, al igual que su ejecución, bastante costosa.

3. Especificación formal del sistema

Los sistemas orientados a objetos se pueden representar con diagramas estructurales (de clases, objetos, componentes, paquetes,...) y con diagramas de comportamiento (de secuencia, colaboración, interacción, estados, actividad,...). La mayoría de las técnicas de generación de casos de pruebas existentes en la literatura se basan en los diagramas de comportamiento para generarlos, en particular, en los diagramas de estado y para ello es necesario definir primero el comportamiento de los objetos. Nuestro método se basará en una especificación formal obtenida a partir del diagrama de clases del sistema, por lo que no será necesario definir el dinamismo del sistema para generar los casos de prueba.

La representación de los diagramas de clases usando una representación formal se basa en una descripción matemática de algunos de los elementos existentes en un sistema orientado a objetos. Para ello hemos considerado las ideas propuestas por [21] y hemos utilizado la notación algebraica descrita posteriormente por los mismos autores en [29].

También nuestras ideas probablemente podrían ser puestas en práctica utilizando alguno de los entornos descritos anteriormente, como son Object-Z [32], OCL [33], TROLL [19] u OASIS [30]. Sin embargo, nuestra representación está más cerca de las ideas de Manfred Broy [6], para quien la Ingeniería del Software, al igual que otras disciplinas, necesita descripciones matemáticas y teóricas y métodos de desarrollo; pero tales teorías y descripciones no deberían ser demasiado complejas, ya que de lo contrario podrían no ser de ayuda para los ingenieros software. La teoría debería darnos semánticas para técnicas de descripción usuales (como diagramas de clases, diagramas de estados, etc.) y debería definir métodos desde una racionalidad matemática, más del estilo de "Técnicas de Descripción formales" que de "Métodos Formales". Esto significa una descripción formal de las técnicas usuales de la ingeniería del software a través del uso de las matemáticas, como por ejemplo [11].

Teniendo en cuenta estas consideraciones, una posible descripción textual de la estructura estática (modelo de clases) de un sistema orientado a objetos sería que es una quintupla formada por: clases, interfaces, asociaciones, agregaciones y dependencias [24].

La descripción formal del sistema utilizando la notación algebraica elegida sería:

$$S=(C, I, As, Ag, D)$$

Donde C es el conjunto de clases e I es el conjunto interfaces, As es el conjunto de asociaciones, Ag el de agregaciones y D el de dependencias. Siendo una relación de asociación de la forma $A_s \subseteq CX\{C \cup I\}$, una agregación de la forma $A_g \subseteq CX\{C \cup I\}$ y una dependencia de la forma $D \subseteq \{C \cup I\}X\{C \cup I\}$.

Una clase k es un elemento de C ($k \in C$), tal que: $k=$*(visibilidad, nombre, Atributos, Constructores, Métodos, Padres)*, donde *visibilidad* puede ser *público, privado, protegido,* donde *nombre* es la palabra que

designa a esa clase, *Atributos* es el conjunto de los atributos de la clase; *Constructores* es el conjunto de métodos que nos permiten construir instancias de k; *Métodos* es el conjunto de sus métodos; *Padres* es el conjunto de clases padre ($Padres \subseteq C$).

Una interfaz i es un elemento de I ($i \in I$), tal que: $i=$*(visibilidad, nombre, Atributos, Métodos, Padres)*, cuyos métodos son abstractos y están implementados por alguna $k \in C$.

Cada uno de estos elementos puede ser descrito con más profundidad, dando detalles sobre la definición de sus atributos, constructores y métodos:

- $a \in Atributos$; $a =$ *(nombre, clase, visibilidad, Modificadores)*, donde *nombre*, *clase*, *visibilidad* y *Modificadores* respectivamente son el nombre, clase, visibilidad (público, privado, protegido) y modificadores (*estático, abstracto, final,* etc.) del atributo.
- $c \in Constructores$; $c =$ *{nombre, visibilidad, Parámetros, Modificadores)*, donde *nombre* es el mismo nombre de la clase y *Parámetros* es el conjunto de parámetros de este constructor, siendo $p \in Parámetros=$*(nombre, clase)*, donde *clase* es el tipo del argumento.
- $m \in Métodos$; $m =$ *{nombre, clase, visibilidad, Argumentos, Modificadores}*, donde *clase* es el tipo del resultado devuelto por el método, y los *Modificadores* podrían ser *static, abstrac, final, synchronized, volatile, native*, etc.

Una asociación *as* ($as \in As$), agregación *ag* ($ag \in Ag$) o dependencia d ($d \in D$) son tipos de relaciones que pueden existir entre clases, y son de la forma: $as=$*(origenAs* $\in C$, *destinoAs* $\in C \cup I$*)*, $ag \in Ag=$*(origenAg* $\in C$, *destinoAg* $\in C \cup I$*)*, $d \in D=$*(origenD* $\in C \cup I$, *destinoD* $\in C \cup I$*)*, podríamos haber incluido otros elemento de las relaciones como podría ser la cardinalidad, pero no lo hemos considerado necesario ya que nuestro método realiza pruebas de caja negra sobre cada una de las clases, no teniendo mucha importancia a priori las relaciones entre ellas.

Con esta descripción, la representación del sistema orientado a objetos del que partimos queda como sigue:

$S=(C, I, As, Ag, D)=(\{K_1,...K_{n1}\}, \{I_1,...,I_{n2}\}, \{As_1,...,As_{n3}\}, \{Ag_1,...,Ag_{n4}\}, \{D_1,...,D_{n5}\})$
$(\{(visibilidad_1, nombre_1, Atributos_1, Constructores_1, Metodos_1, Padres_1),...$
$(visibilidad_{n1}, nombre_{n1}, Atributos_{n1}, Constructores_{n1}, Metodos_{n1}, Padres_{n1})\},$
$\{(visibilidad_1, nombre_1, Atributos_1, Metodos_1, Padres_1),...$
$(visibilidad_{n2}, nombre_{n2}, Atributos_{n2}, Metodos_{n2}, Padres_{n2})\},$
$\{(origenAs_1, dDestinoAs_1),...,(origenAs_{n3}, destinoAs_{n3})\},$
$\{(origenAs_1, destinoAg_1),...,(origenAs_{n4}, destinoAg_{n4})\},$
$\{(origenAs_1, destinoD_1),...,(origenAs_{n5}, destinoD_{n5})\})$

Por consecuencia, cualquier sistema orientado a objetos, escrito en cualquier lenguaje de programación o en cualquier lenguaje de modelado, puede ser descrito usando esta simple representación formal. Es posible definir algunas funciones de transformación en estos conjuntos para desarrollar algunas tareas propias de la ingeniería del software. La generación y ejecución de casos de prueba son dos de estas tareas.

4. Generación automática de casos de prueba

Una clase es el bloque más básico para la construcción de software orientado a objetos [28]. Por tanto, el la unidad más usada para la realización de pruebas. La noción de clase como unidad de pruebas ha sido utilizada en muchos reportes de proyectos en SOO [26]. El proceso de probar una clase consiste en la construcción de una instancia de la clase (a través de la invocación de uno de sus constructores) y la ejecución, en esta instancia, de una secuencia de algunos de sus métodos. El constructor y los métodos seleccionados pueden tomar parámetros, que pueden ser de cualquier tipo. Suponemos un constructor o un método que toma un entero como parámetro, este podría ser invocado utilizando valores aleatorios, valores límite (cercanos al cero), valores muy grandes y valores muy pequeños, etc. como entradas para el caso de prueba. Si otra operación toma una cadena de caracteres como parámetro, podría ser invocado usando la cadena vacía, la cadena null, una cadena de caracteres especiales, una cadena de letras y números, etc. como entradas para el caso de prueba. Si hubiese otra operación que toma un entero y una cadena como parámetros, podrían ser invocadas todas las combinaciones de todos los valores mencionados anteriormente. Cuando el parámetro no

es un tipo de dato simple (su tipo es una clase compleja, como "Persona", "Coche", etc.), es posible generar a priori instancias de estas clases que luego nos sirvan como datos de prueba de la clase a probar.

Así, inspirados en las ideas de Doong y Frankl (1994) [9] sobre "secuencias de prueba", y apoyándonos en el concepto de "*test* script", definimos formalmente un test script (ts $\in$ *TS*) [12] para la clase *k* como un par formado por una llamada a un constructor y una lista de llamadas a métodos. Un script de prueba por tanto es: $ts(k \in C)=\{c \in k.Constructores, \{m_1, m_2, ..., m_n, m_i \in k.Métodos\}\}$

Tanto los constructores como los métodos toman parámetros, que son instancias de clases o valores de tipos primitivos. Para un script dado, pueden construirse multitud de casos de prueba dependiendo de los valores que pasemos a los parámetros de su constructor y sus métodos. Por ejemplo, aplicando la técnica de *conjetura de errores* [4], podríamos pasar a los parámetros de tipo *String* los valores *null*, *new String()*, "", "ñáè", y *-1, 0, +1, 32767, -32768* a los de tipo *int*. Estos valores "propensos a error" serían almacenados para ser asignados a los parámetros de sus respectivos tipos o clases de datos. Es decir, que dispondríamos de los siguientes conjuntos predefinidos de Conjetura de Error *(CE)*:

CE(String)={ null, new String(), "", "ñáè"}
CE(int)={ -1, 0, +1, 32767, -32768}

Podríamos utilizar estos conjuntos para generar casos de prueba para el siguiente script de prueba:

Persona p=new Persona(String) // Constructor
p.setEdad(int) // Método

Los casos de prueba que generaríamos serían un total de 20 (4 procedentes de *CE(String)* x 5 procedentes de *CE(int)*), algunos de los cuales son:

Persona p=new Persona(null) *p.setEdad(-1)*	*Persona p=new Persona(null)* *p.setEdad(0)*	*Persona p=new Persona(null)* *p.setEdad(+1)*	...
Persona p=new Persona("ñáè") *p.setEdad(-1)*	*Persona p=new Persona("ñáè")* *p.setEdad(0)*	*Persona p=new Persona("ñáè")* *p.setEdad(+1)*	...
...	...	...	...

Tabla 1. Casos de prueba generados para un script de prueba en función de éste y de *CE*

Como se observa, la generación de casos de prueba para un script de prueba utilizando la técnica de conjetura de errores se reduce a ir asignando a los parámetros el producto cartesiano de los conjuntos de Conjetura de Error *CE* correspondientes a los tipos de los argumentos del script. Para rellenar el conjunto de Conjetura de Error *CE* de clases complejas o nuevas (pertenecientes al enunciado del problema, por ejemplo), deben crearse instancias de estas clases que se grabarán en disco mediante serialización; más adelante, estos objetos podrán ser recuperados y pasados como parámetros a los constructores y métodos que los utilicen.

Por lo tanto, los datos de prueba se generarán en función del tipo de dato que sea, si es un tipo básico la generación se hará con valores aleatorios y con valores problemáticos, si es un tipo complejo, la generación se hará creando a priori una instancia de esa clase, es decir, probándola antes.

4.1. Algoritmo de generación de casos de prueba

Nuestro algoritmo está basado en la siguiente idea: comenzamos generando un conjunto de T *est Scripts (TS)* para la clase que queremos probar, mediante la técnica de *Conjetura de Error CE* descrita anteriormente. Para parámetros de métodos y constructores que sean objetos de otras clases usaremos los objetos de las mismas almacenados anteriormente en ficheros mediante serialización.

Más formalmente, siendo $tc \in TC$ un caso de prueba, es decir, el constructor y el conjunto de métodos de los que se conoce el valor que van a tomar sus parámetros, el algoritmo de generación y ejecución automática quedaría como sigue:

1. *Generación del conjunto de Test Scripts (TS)*

   a. *Para cada clase generamos el siguiente número de test scripts:*

   $$|k.Constructores| \prod_{i=1}^{n} |k.Métodos|^n$$

   Para ello generamos combinaciones de un constructor, un constructor y un método, el mismo constructor y dos métodos, ... el mismo constructor y n métodos y hacemos esto para cada uno de los constructores; y obtendremos de este modo el número de test scripts resultantes, siendo n es la longitud máxima del test script.

   $$|k.constructor| + |k.constructor| \cdot |k.método| +$$
   $$|k.constructor| \cdot |k.método|^2 + ...$$
   $$... + |k.constructor| \cdot |k.método|^n = |k.constructor| \prod_{i=0}^{n} |k.método|^i$$

   Obviamente, si n=0, los test scripts que se generen consistirán inicialmente en llamadas a los constructores.

   b. *Cada ts constará de un constructor y como máximo n métodos, siendo formalmente un ts:*

   $$ts(k \in C)=(c \in k.Constructores, \{m_1, m_2, ..., m_n, m_i \in k.Métodos\})$$

2. *Generación de test cases (tc) para cada test script (ts):*

   a. *Para cada ts obtendremos el siguiente número de test cases:*

   $$\subseteq_{i=1}^{n} |CE(a_k)|$$

   Los casos de prueba vendrán dados en función del número de valores que obtengamos por la técnica de conjetura de error de cada uno de los parámetros del constructor y los métodos que forman el ts.

   Para un ts concreto de un constructor y n métodos (donde n es la longitud máxima del test script) para cada uno de ellos habrá que probar las combinaciones de sus parámetros tomando todos los valores obtenidos por conjetura de error.

   Para un ts concreto con un constructor y n métodos y suponiendo que cada uno de los parámetros tiene tres posibles valores obtenidos mediante conjetura de error, obtendremos el siguiente número de combinaciones para cada uno de ellos, dependiendo del número de parámetros que tengan:

   $c() = 1;\ m_1(p_1) = 3;\ m_2(p_1,p_2) = 3^2;\ ....\ m_n(p_1, p_2, ..., p_N) = 3^n$

   Si luego combinamos estos resultados entre ellos obtendremos que:

   $c(p_1,...,p_j);\ m_1(p_1,...,p_j);\ ...\ m_n(p1,...,pk)$

   Por lo que obtendremos $3^1+3^j+...+3^k$ combinaciones. De forma general:

   $c(a_1,....a_j);\ m_1(a_{i+1},...a_j);\ ...\ m_n(a_{j+1}...a_k)$ *podruce:*

$$\sum_{i=1}^{k} |CE(a_i)| \text{ combinaciones}$$

Siendo CE (tipo) el conjunto de valores propensos a error del tipo y |CE(tipo)| el número de valores propensos a error del tipo.

b. *Los datos serán obtenidos según la técnica de CE en el caso de tipos básicos y utilización instancias de las clases serializadas en caso contrario.*

$$f(ts \in TS) = tc \in TC$$

f es la función que se encarga de generar estos tc a partir de un ts.

Un tc será un ts a cuyos parámetros les han sido asignados valores:

$$tc(ts \lfloor TS) = (\overline{pc}, \overline{pm_1}, \overline{pm_2}, \ldots, \overline{pm_n})$$

Siendo $\overline{pc}$ el conjunto de parámetros del constructor, y $\overline{p_1}, \ldots, \overline{pm_n}$ el conjunto de parámetros de cada uno de los métodos.

3. *Por último ejecutamos los casos de prueba para cada una de las clases ejecutando el constructor y los métodos correspondientes a cada uno de los tc para esa clase, utilizando como parámetros los obtenidos en el paso anterior para cada uno de los tc.*

Este algoritmo podría alcanzar niveles de complejidad altos en clases complejas, actualmente estamos trabajando en una herramienta que nos permita generar estos casos de prueba de manera automática, o que de la posibilidad al usuario de elegir el mismo los valores que desea probar.

4.2. Ejemplo

Vamos a considerar un ejemplo que ha sido tomado de la literatura existente de casos de prueba para poder comprobar el correcto funcionamiento de nuestro algoritmo [16]. Tiene dos clases: VendingMachine y Dispenser, como se muestra en la figura 1 y el código de las mismas se muestra en la figura 2.

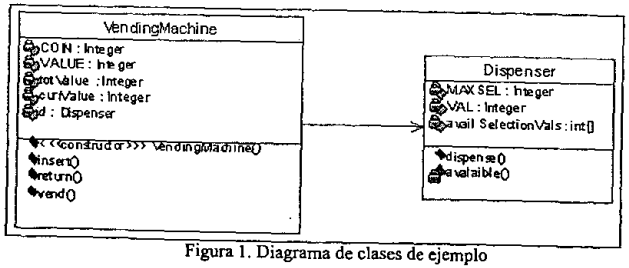

Figura 1. Diagrama de clases de ejemplo


```
1. public class VendingMachine {
2. final private int COIN = 25;
3. final private int VALUE = 50;
4. private int totValue;
5. private int currValue;
6. private Dispenser d;
7. public VendingMachine() {
8. totValue = 0;
9. currValue = 0;
10. d = new Dispenser();
11. }
12. public void insert() {
13. currValue += COIN;
14. System.out.println("Current
    value = " + currValue );
15. }
16. public void return() {
17. if ( currValue == 0 )
18. System.err.println( "no coins to
    return" );
19. else {
20. System.out.println("Take your
    coins");
21. currValue = 0;}
22. }
23. public void vend( int selection)
    {
24. int expense;
25. expense = d.dispense( currValue,
    selection );
26. totValue += expense;
27. currValue -= expense;
28. System.out.println( "Current
    value = " + currValue );
29. }
30. } // class VendingMachine
```

```
31. public class Dispenser {
32. final private int MAXSEL =
33. final private int VAL = 50;
34. private int[] availSelectio
    {2,3,13};
35. public int dispense( int cr
    int sel ) {
36. int val=0;
37. if ( credit == 0 )
38. System.err.println("No coin
    inserted");
39. else if ( sel > MAXSEL )
40. System.err.println("Wrong
    selection "+sel);
41. else if ( !available( sel )
42. System.err.println("Selecti
    "+sel+" unavailable");
43. else {
44. val = VAL;
45. if ( credit < val )
46. System.err.println("Enter
    credit)+" coins");
47. else
48. System.err.println("Take
    selection"); }
49. return val;
50. }
51. private boolean available(
    ) {
52. for (int i = 0;
    i<availSelectionVals.lengt
53. if (availSelectionVals[i]
    return true;
54. return false;
55. }
56. } // class Dispenser
```

Figura 2. Código de las clases VendingMachine y Dispenser

La representación algebraica de este sistema es:

$S = \{K_1, K_2, As_1\}$

K_1= {Public, VendingMachine, {a_1, a_2, a_3, a_4, a_5}, c_1, {m_1, m_2, m_3} }	K_2={Public, Dispenser,{a_1, a_2 , a_3 } VendingMachine}
a_1= {COIN, Integer, Private}	a_1= {MAXSEL, Integer, Private}
a_2= { VALUE, Integer, Private }	a_2= { VAL, Integer, Private }
a_3= { totValue, Integer, Private }	a_3= { availSelectionVals, int[], Private }
a_4= { currValue, Integer, Private }	m_1= {dispense, Integer, Public, , p_1, p_2},
a_5= {d, Dispenser, Private}	p_1= {credit, Integer}
c_1= {VendingMachine, Public}	p_2= {sel, Integer}
m_1= {insert, void, Public, Static}	m_2= {avalaible ,boolean, Private, p_1, Sta
m_2= {return, void, Public, Static }	p_1= {sel, Integer}
m_3= {vend, void, Public, p_1, Static }	
p_1= {selection, Integer}	

$As_1 = (K_1, K_2)$

Una vez que tenemos una descripción formal de las clases, podemos generar los ca utilizando nuestro algoritmo de generación de casos de prueba y los valores obtenidos anteriormente serializados.

Obtendríamos los siguientes ts para cada una de las clases:

Para K_1:	$ts_{11}=(c_1)$; $ts_{12}=(c_1, m_1)$; $ts_{13}=(c_1, m_1, m_2)$; $ts_{14}=(c_1, m_1, m_2, m_3)$
Para K_2:	$ts_{21}=(new)$; $ts_{22}=(new, m_1)$; $ts_{23}=(new, m_1, m_2)$

Obtendríamos los siguientes tc para cada uno de los ts anteriores:

Para ts_{11}: $tc_{111}=(c_1())$; *Para ts_{12}: $tc_{121}=(c_1(), m_1())$;* *Para ts_{13}: $tc_{131}=(c_1(), m_1(), m_2())$;* *Para ts_{14}: $tc_{141}=(c_1(), m_1(), m_2(), m_3(-1))$;* $tc_{142}=(c_1(), m_1(), m_2(), m_3(0))$; $tc_{143}=(c_1(), m_1(), m_2(), m_3(1))$; $tc_{145}=(c_1(), m_1(), m_2(), m_3(32768))$; $tc_{146}=(c_1(), m_1(), m_2(), m_3(-32768))$;	$tc_{2212}=(new, m_1(1,0))$; $tc_{2213}=(new, m_1(1,1))$; $tc_{2214}=(new, m_1(1,32768))$; $tc_{2215}=(new, m_1(1,-32768))$; $tc_{2216}=(new, m_1(32768,-1))$; $tc_{2217}=(new, m_1(32768,0))$; $tc_{2218}=(new, m_1(32768,1))$; $tc_{2219}=(new, m_1(32768,32768))$; $tc_{2220}=(new, m_1(32768,-32768))$; $tc_{2221}=(new, m_1(-32768,-1))$; $tc_{2222}=(new, m_1(-32768,0))$; $tc_{2223}=(new, m_1(-32768,1))$; $tc_{2224}=(new, m_1(-32768,32768))$; $tc_{2225}=(new, m_1(-32768,-32768))$;
Para ts_{21}: $tc_{211}=(new)$; *Para ts_{22}: $tc_{221}=(new, m_1(-1,-1))$;* $tc_{222}=(new, m_1(-1,0))$; $tc_{223}=(new, m_1(-1,1))$; $tc_{224}=(new, m_1(-1,32768))$; $tc_{225}=(new, m_1(-1,-32768))$; $tc_{226}=(new, m_1(0,-1))$; $tc_{227}=(new, m_1(0,0))$; $tc_{228}=(new, m_1(0,1))$; $tc_{229}=(new, m_1(0,32768))$; $tc_{2210}=(new, m_1(0,-32768))$; $tc_{2211}=(new, m_1(1,-1))$;	*Para ts_{23}: $tc_{231}=(new, m_1(-1,-1), m_2(-1))$;* $tc_{232}=(new, m_1(-1,0), m_2(-1))$; $tc_{233}=(new, m_1(-1,1), m_2(-1))$; $tc_{234}=(new, m_1(-1,32768), m_2(-1))$; $tc_{235}=(new, m_1(-1,-32768), m_2(-1))$; ... $tc_{2375}=(new, m_1(-32768,-32768), m_2(32768))$;

4.3. Función de transformación f:soo → {tci}

La función de transformación se encarga de obtener un conjunto de casos de prueba a partir de un SOO:

$$F: S \blacklozenge TC^n$$
$$soo \blacklozenge \{tc_i\}$$

Siendo S un SOO de la forma *S={C, I, As, Ag, D}*, ts una secuencia de prueba o *script* de prueba de la forma $ts(k \in C)=(c \in k.Constructores, \{m_1, m_2, ..., m_n, m_i \in k.Métodos\})$ y siendo tc un caso de prueba de la forma $tc(ts\lfloor TS) = \{\overline{pc}, \overline{pm_1}, \overline{pm_2}, ..., \overline{pm_n}\lfloor ts.P\}$.

5. Automatización

Para ejecutar los distintos casos de prueba desarrollamos una herramienta inicial en la que había que introducir los casos de prueba manualmente. Para la ejecución de los casos de prueba la herramienta debería tener en cuenta el orden de generación de los mismos, ya que para probar una clase normalmente es necesario tener almacenados objetos de otras clases con las que se relaciona. Cuando un caso de prueba (*tc*) es ejecutado, produce una instancia de una clase que puede ser guardada para hacer posteriormente pruebas de

regresión y para continuar haciendo pruebas de caja negra sobre nuestro sistema. En un entorno real, esto se consigue gracias a la serialización.

Una vez hayamos ejecutado distintos casos de prueba sobre la clase, tendremos algunos objetos como resultado de ello, que almacenaremos, ya que nos servirán como entradas para futuros casos de prueba. En la figura 3 se muestran algunos objetos que han sido guardados utilizando esta característica y que posteriormente podrán ser usados para las pruebas.

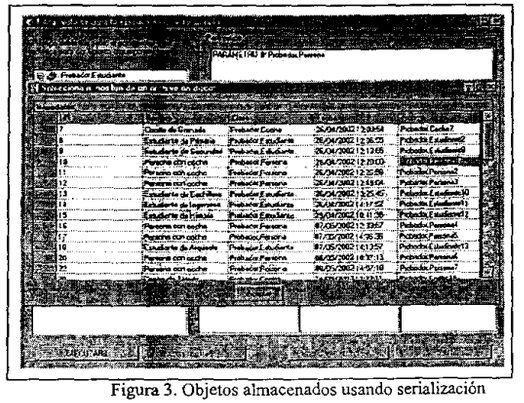

Figura 3. Objetos almacenados usando serialización

En la última columna de la figura anterior aparece el nombre del archivo en el que guardamos el objeto. Por el momento, en nuestra herramienta, hemos decidido que sea el usuario el que introduzca manualmente la clase y construya también manualmente el test script, como se muestra en la figura 4. Si para alguno de los métodos no existen objetos creados que sirvan como entradas para probarlo, nos dará un mensaje indicándonos cual es la clase que debemos probar primero antes de probar la actual. Con esto garantizamos que no se quedará sin probar ninguna clase por olvido del usuario.

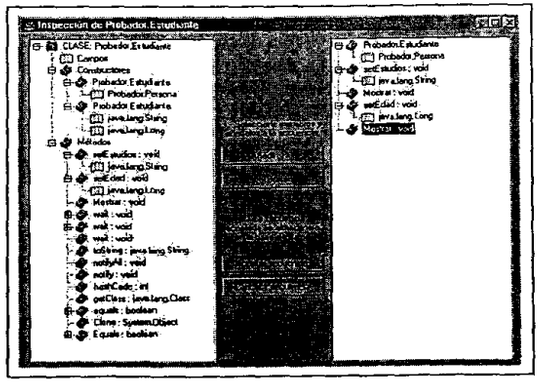

Figura 4. Herramienta para la ejecución de los casos de prueba

Tras construir el test script (*ts*) y el test case (*tc*) (de momento manual), el usuario puede ejecutar el tc, archivar el ts, ver el resultado de la ejecución y guardar la instancia para usarla potencialmente (figura 5).

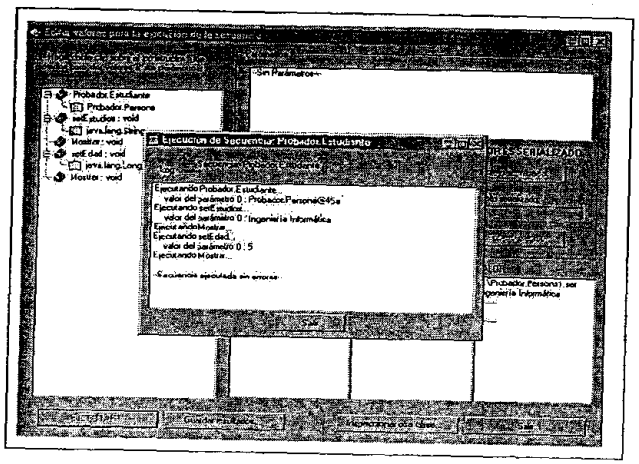

Figura 5. Pantalla mostrada después de una ejecución sin errores.

6. Conclusiones y trabajos futuros

Este artículo presenta un método para la automatización de la generación y ejecución de casos de prueba para sistemas orientados a objetos. La principal idea es el uso de una especificación formal del sistema para representar cualquier sistema orientado a objetos y, entonces, aplicando funciones algebraicas sobre esta descripción formal generar los deseados casos de prueba. El método presentado es de gran potencia partiendo de pocos recursos; y consigue un abaratamiento importante respecto de los lenguajes formales que abaratan la fase de pruebas encareciendo las fases de diseño y de codificación, debido al grado de complejidad que alcanzan estos lenguajes. Por lo que podemos concluir que es un método sencillo y fácilmente ampliable pero con un gran potencial.

Actualmente estamos trabajando en el refinamiento del algoritmo para que sea capaz de crear los casos de prueba más adecuados. Se pretende limitar el número de casos de prueba (*tc*) generados; es decir, seleccionar sólo los casos de prueba con un cierto nivel de calidad, para lo que nos basaremos en la propuesta de Kirani y Tsai [20], que proponen el uso de expresiones regulares en cada una de las clases del sistema para representar las secuencias de métodos correctas. También se pretende medir el nivel de cubrimiento de los casos de prueba, para lo que seguiremos un enfoque similar al propuesto por Ghost y Mathur [15], que proponen la mutación de interfaces. En nuestro caso generaremos automáticamente mutantes usando reflexión. La herramienta se ampliará para que pueda soportar totalmente la técnica descrita anteriormente, soportando más lenguajes OO y consiguiendo que la generación y ejecución de los casos de prueba sea totalmente automática.

7. Agradecimientos

Este trabajo está parcialmente financiado por el proyecto TAMANSI (Técnicas Avanzadas para el mantenimiento del Software), financiado por la Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha (PBC-02-001).

8. Referencias

[1] Ball T., Hoffman D., Ruskey F., Webber R. y White L. (2000). State generation ar testing. *Soft. Testing, Verification and Reliab.*, (10). 149-170.

[2] Bashir, I., Goel, A.L. *Testing Object-Oriented Software*. Springer, 1999.

[3] Beck K. (1999). Embracing change with Extreme Programming. *Computer*, 32(10), 7(

[4] Beizer B. *Black-Box Testing: Techniques for Functional Testing of Soft. and Systems.*

[5] Bowen JP. y Hinchey MG. (1995). Ten Commandments of Formal Methods. *IEEE So*

[6] Broy, M. (2001). Toward a Mathematical Foundation of Software Engineerin *Transactions on Software Engineering*, 27(1), 42-57.

[7] Clarke LA y Richardson DJ (1985). Applications of symbolic evaluation. J. of S. and

[8] Coward PD (1991). Symbolic execution and testing. Information and Soft. Technolog

[9] Doong RK. y Frankl PG. (1994). The ASTOOT approach to testing object-orient *Transactions on Software Engineering and Methodology*, 3(2):101-130.

[10] Edwards SH (2000). Black-box testing using flowgraphs: an experimental assessm and automation potential. *Software Testing, Verification and Re-liability*, 10, 249-262

[11] Feijs LMG., Krikhaar, RL. Relation Algebra with Multi-Relations. *I. J. Comp. Math.*

[12] Fewster M., Graham D. *Software Test Automation*. Addison-Wesley, ACM Press Bo

[13] Forgács I. y Bertolino A. *Preventing untestedness in data-flow based test* Verification and Reliability, (12), 29-58, 2002.

[14] Ghiassi, M., K.I.S. Woldman, "Dual Programming Approach to Soft. Testing," *S. Q.*

[15] Ghosh S. y Mathur AP. (2001). Interface Mutation. *Soft. Testing, Verif. and Reliabili*

[16] Harrold, M. J., Orso, A., Rosenblum, D., Rothermel, G., Soffa, M., Do, H. Using C for Support Regression Testing of Component-Based Software. *Technical Rep* College of Computing, Georgia Institute of Technology, 2001.

[17] Hierons, R. (2002). Editorial: Formal methods and testing. *Soft. Testing, Veri. and R*

[18] Interesse, M. Test Manager: The test Automation Component for the Mainten Systems. *International Conference on Software Maintenance ICSM 2002*, October

[19] Jungclaus R., Saake G., Hartmann T., and Sernadas C. (1991). Object-Orien Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Brauns

[20] Kirani S. y Tsai WT. (1994). Method sequence specification and verification o *Object-Oriented Programming*, 7(6):28–38.

[21] Krikhaar R.L. *Software Architecture Reconstruction*. PhD thesis, University of Ams

[22] Kung D., Suchak N., Gao J., Hsia P., Toyoshima Y. y Chen C. (1994). *On object* IEEE COMPSAC'94. IEEE Computer Society Press, pp. 222–227.

[23] Meudec C. (2001). ATGen: automatic test data generation using constraint log symbolic execution. *Software Testing, Verification and Reliability*, 11(2), 81-96.

[24] Michael, CC., McGraw, G., Schatz MA. (2001). Generating Software Test Data *Transactions on Software Engineering*, 27(12), 1085-1109.

[25] Moreno A. y Vegas S. (2002). *Limitaciones sobre el conocimiento empírico de* Actas de las VII Jornadas de Ingeniería del Software y Bases de Datos.

[26] Murphy, G.C., Townsend, P., Wong, P.S. Experiences with Cluster *Communications of the ACM*, 37(9):39-47, Septiembre, 1994.

[27] Onoma K., Tsai WT., Poonawala M. y Suganuma H. (1988). Regression Te environment. *Communications of the ACM*, 41(5), 81-86.

[28] Overbeck, J., Testing Object-Oriented Soft.: State of the Art and Research Direc *of the First Euro. Inte. Conf. of Soft. Testing, Analysis and Review*, London/UK, O

[29] Reinder J., Loe M. G. Feijs, André Glas, René L. Krikhaar, Thijs Winter (2000). towards support at the architectural level. *Journal of Software Maintenance*, 12(3)

[30] Pastor Ó, Insfrán E, Pelechano V, Romero J y Merseguer J (1997). OO-METH *Production Environment Combining Conventional and Formal Methods*. Proc. on Advanced Information Systems Engineering (Cai-SE), pp. 145-158.

[31] Rothermel G., Untch RH., Chu Ch. y Harrold MJ. (2001). Prioritizing Test Testing. *IEEE Transactions on Software Engineering*, 27(10), 929-948.

[32] Spivey, J.M., *The Z Notation: A reference Manual.* International Series in Comp.

[33] Warmer J y Kleppe A (1998). The Object Constraint Language. Addison-Wesley: