Ingeniare. Revista Chilena de Ingeniería
ISSN: 0718-3291
facing@uta.cl
Universidad de Tarapacá
Chile

Altuna Castillo, Enrique José; Guibert Estrada, Lisandra

Generación de pistas durante el aprendizaje de la programación para concursos usando el análisis estático y dinámico de las soluciones

Ingeniare. Revista Chilena de Ingeniería, vol. 21, núm. 2, agosto, 2013, pp. 205-217
Universidad de Tarapacá
Arica, Chile

Disponible en: http://www.redalyc.org/articulo.oa?id=77228591005

Generación de pistas durante el aprendizaje de la programación para concursos usando el análisis estático y dinámico de las soluciones

Hint generation for learning programming for competitions using static and dynamic analysis of solutions

Enrique José Altuna Castillo[1] Lisandra Guibert Estrada[1]

Recibido 20 de marzo de 2012, aceptado 19 de marzo de 2013
Received: March 20, 2012 Accepted: March 19, 2013

RESUMEN

La enseñanza de la programación es una tarea difícil, y la calidad del aprendizaje que se obtiene por los estudiantes está influida, entre otros factores, por el soporte que reciben mientras resuelven problemas. Para generar pistas que ayuden a los estudiantes a solucionar los errores cometidos se han aplicado varias estrategias, principalmente enfocadas en encontrar diferencias en las salidas obtenidas y esperadas, además de ofrecer información proveniente del análisis estático. En el presente trabajo se expone un método para la generación de pistas que además de usar las técnicas anteriores incluye la realización del análisis dinámico apoyado en una especificación de las entradas y salidas de los problemas que permite al sistema razonar en función de los tipos de datos, rangos, distribución y correspondencia entre pares entrada-salida. Se realizó además un estudio donde se demostró cómo la aplicación del método desarrollado influyó positivamente en la cantidad de intentos que necesitaron los estudiantes para solucionar correctamente los ejercicios planteados.

Palabras clave: Sistemas tutores inteligentes, análisis dinámico, análisis estático, programación, concursos.

ABSTRACT

The teaching of programming is a difficult task, and the quality of learning obtained by the students is influenced, among other factors, by the support they receive as they solve problems. To generate hints that help students to fix the mistakes, various strategies have been implemented, primarily focused on finding differences in the obtained and expected outputs, as well as providing information from the static analysis. A method for generating hints, that besides using the above techniques includes performing dynamic analysis, supported in a specification of the inputs and outputs of the problems that allows the system to reason in terms of data types, range, distribution and correspondence between input-output pairs are presented. It was also conducted a study that has demonstrated how the developed method positively influences the number of attempts needed by the students to solve correctly the given exercises.

Keywords: Intelligent tutoring systems, dynamic analysis, static analysis, programming, contest.

INTRODUCCIÓN

La necesidad de programas con niveles de complejidad que se incrementan con los años provocó que algunas empresas crearan planes de formación y las universidades crearan currículos orientados a egresar profesionales en carreras relacionadas con la informática. La Asociación de Máquinas Computadoras (ACM), una de las organizaciones con más prestigio a nivel internacional en la educación

[1] Centro de Tecnologías para la Formación. Universidad de las Ciencias Informáticas. Carretera a San Antonio km 2 ½, Boyeros. La Habana, Cuba. E-mail: ejaltuna@uci.cu; lguibert@uci.cu

en temas relacionados con la informática, divide el estudio de la misma en cinco perfiles para carreras de nivel universitario:

- Tecnologías de la información
- Sistemas de información
- Ingeniería de software
- Ciencia de la computación
- Ingeniería de computadoras

Según los informes [1-5] emitidos por la ACM sobre las metas del currículo para estas cinco carreras, los profesionales de la informática deben ser capaces de diseñar, implementar y probar programas de computadoras trabajando en equipo.

En los estudios realizados por [6-8] se describe como los estudiantes presentan dificultades con el aprendizaje en las asignaturas de la Disciplina de Programación. En el trabajo de [8] se plantea que una posible solución se encuentra en fomentar el entrenamiento en programación para competencias usando sistemas de evaluación automática de soluciones para la ejercitación. En estos sistemas los ejercicios deben ser resueltos mediante un programa de tipo consola implementado con un lenguaje de programación permitido por el sistema, que obtenga datos de la entrada estándar del sistema e imprima hacia la salida estándar. La solución se valida mediante la comparación de la salida generada por el programa con la definida por el creador del ejercicio para cada entrada y se envía una respuesta instantánea al estudiante sobre el resultado obtenido.

Sin embargo, la capacidad de retroalimentación de estos sistemas es limitada, lo que provoca que los estudiantes tengan poco apoyo mientras resuelven ejercicios y muchas veces les sea imposible corregir los errores cometidos, principalmente mientras se inician en la actividad. La situación antes mencionada conlleva a que se pierda la motivación, se limite el aprendizaje y aumente el porcentaje de estudiantes que no continúan preparándose en programación para competencias luego de las primeras sesiones. Una posible solución a este problema se puede encontrar en el trabajo de [9], donde mediante el desarrollo de un sistema que aporte pistas en función de los errores cometidos se apoya el aprendizaje en forma similar a como lo haría un tutor humano.

Muchos son los trabajos que se han escrito sobre la aplicación de herramientas creadas con técnicas de análisis estático y análisis dinámico de código fuente con el objetivo de encontrar errores cometidos por los desarrolladores durante la implementación de sistemas informáticos, como Valgrind [10], Stub4JSC [11], Pin [12]. Sin embargo, son escasas las referencias a sistemas que hayan aplicado estas técnicas de forma exitosa con el objetivo de apoyar a estudiantes principiantes en el aprendizaje de la programación y no se encontró ninguna referencia en la programación para competencias.

En el presente trabajo se describe la concepción, construcción y aplicación de Asistente, un sistema para la generación de pistas al estudiante sobre los errores cometidos durante la solución de ejercicios típicos de competencias de la programación de nivel de dificultad bajo. Para la generación de las pistas se usarán técnicas de análisis estático y dinámico de las soluciones. La versión actual del sistema se considera un prototipo y solo cuenta con soporte para el lenguaje de programación Java. Por lo tanto, los procedimientos aquí expuestos se corresponden al desarrollo relacionado con este lenguaje, si bien los principios son extrapolables a cualquier otro, es posible que dadas las características del lenguaje sea necesario implementar alguna de las herramientas que aquí se usan.

La principal novedad del trabajo se encuentra en la aplicación de técnicas de análisis dinámico para la generación de pistas con mayor información sobre las características de los programas a los estudiantes, de forma que con el análisis de la información ofrecida se soporte el proceso de resolución de problemas. Además, se expone una especificación para la descripción de las entradas y salidas que permite al sistema razonar en función de ciertas características del problema.

El resto del documento se estructura como sigue: en la próxima sección se expone una visión general del sistema, en las dos siguientes se exponen las dos principales técnicas aplicadas en el trabajo para la generación de pistas, luego se describe el experimento realizado, se discuten los resultados, se analizan sistemas similares, se concluyen los resultados obtenidos en la investigación y se finaliza con algunas apreciaciones de los autores acerca de líneas de trabajo futuras.

DESCRIPCIÓN GENERAL DEL SISTEMA

El sistema funciona como una aplicación web sobre la Plataforma Java Edición Empresa (Java EE) [13], usando el soporte que brinda JavaServer Pages para generar contenido dinámico para la web que pueda ser visualizado en un navegador. Se desarrolló con la intención de que fuese desplegado por el servidor Apache Tomcat, aunque se puede usar en cualquier otro que cumpla con la especificación JSR 154 [14]. Se utilizó una variación de la arquitectura clásica propuesta para el desarrollo de sistemas tutores inteligentes [15], como se puede observar en la Figura 1. El **módulo tutor** se encarga de coordinar el trabajo del sistema, el **módulo dominio** es el encargado de generar la retroalimentación en forma de pistas para los errores que cometa el estudiante y será el centro de este trabajo. El **módulo estudiante** almacena la información relacionada con los perfiles del estudiante y la interfaz de usuario se encarga de la presentación de la información y la comunicación con el estudiante. Las inclusiones principales en la arquitectura son las adiciones del **evaluador automático** y el **repositorio de problemas**. El **evaluador automático**, como se describirá en detalle posteriormente, realiza la comprobación inicial a las soluciones. Mientras que el **repositorio de problemas** tiene la función de almacenar los datos de los problemas incluidos en el sistema, información que permite al **módulo dominio** razonar sobre los errores en función del problema. Esta división se hizo con el objetivo de no mezclar la información relativa al conocimiento general del dominio de resolución de problemas de programación para competencias con la perteneciente a cada problema en particular. De forma que en futuras versiones del sistema la gestión de los problemas pueda ser externalizada sin un impacto significativo en la arquitectura del sistema y en el funcionamiento del resto de los módulos.

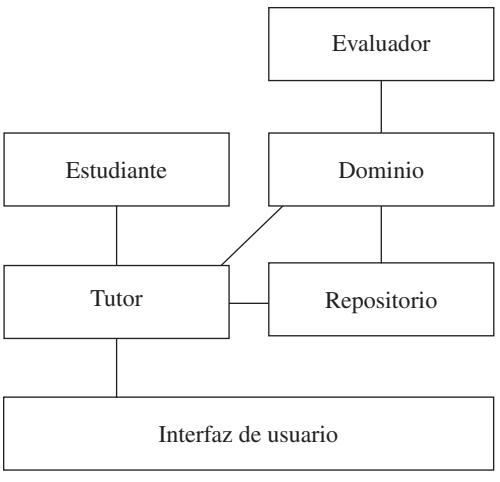

Figura 1. Arquitectura del sistema.

Una vez autenticado en el sistema, el estudiante accede a la interfaz con el listado de los problemas que no ha resuelto, de cada uno se muestra el nivel dificultad y temática a la que pertenece. Selecciona un problema y se le muestra la especificación de la tarea a realizar, donde se incluye información sobre el formato en que se reciben los datos de entrada por parte del programa y el formato en que se espera su salida. El problema es implementado por el usuario en su estación de trabajo y el código fuente resultante enviado al sistema para su evaluación.

Al recibir el sistema una solución para un problema, el módulo tutor solicita al módulo evaluador la calificación de la misma. Esta se compila con el compilador correspondiente al lenguaje seleccionado (en el caso de Java la herramienta **javac** que se distribuye como parte del JDK) y se ejecuta con los datos de prueba. El sistema de evaluación automática puede arrojar varios resultados:

- Aceptado: la solución es correcta.
- Error de compilación: existe un error en la sintaxis del código fuente de la solución que no permite su compilación exitosa.
- Límite de tiempo excedido: la ejecución de la solución se extendió por más tiempo del permitido para el problema.
- Límite de memoria excedido: la ejecución de la solución consumió más memoria que la permitida para el problema.
- Error en tiempo de ejecución: la ejecución de la solución terminó de forma inesperada.
- Respuesta incorrecta: la salida generada por la solución para los datos de prueba no coincide con la esperada por el sistema.

Si la respuesta es **Aceptado** el módulo tutor informa el resultado y muestra soluciones y comentarios seleccionados por el profesor, se almacena en el perfil del estudiante que el ejercicio ha sido completado.

Las demás respuestas se consideran no satisfactorias y al ser entregadas se complementan con pistas que se obtienen del módulo de dominio. En este trabajo se entiende por pista como información contextual que es suministrada al ocurrir un error e indica posibles causas que provocan su aparición. Adicionalmente, pueden ofrecer datos que faciliten la solución del error por parte del usuario.

En el sistema las pistas se organizan atendiendo a niveles de profundidad ascendentes y su intención está dirigida a que la corrección del error venga acompañada de un proceso de descubrimiento del conocimiento, logrado mediante indicaciones generales y al evitar entregar respuestas directamente.

Al cometer un error se brinda al estudiante la oportunidad de corregirlo sin ayuda del sistema, lo que permite que solucione errores tipográficos o descuidos sobre algún aspecto de la solución, lo que limita el impacto en la evaluación de pequeños errores desafortunados que no se corresponde con el estado real de conocimiento del estudiante. Cada problema puede ser enviado sin cantidad de oportunidades límite; en cada ocasión el estudiante puede mantenerse en el nivel actual de ayudas o solicitar el próximo nivel. Las ayudas están clasificadas en cinco niveles, donde para cada nivel se muestran las ayudas correspondientes a este y a los niveles anteriores:

- Nivel 0: Solo el enunciado original. Sin ayuda.
- Nivel 1: Ayudas provenientes del análisis estático de la estructura del programa e información general del análisis dinámico.
- Nivel 2: Ayudas provenientes del análisis dinámico.
- Nivel 3: Se muestran los juegos de datos que provocan el error en que se incurre.
- Nivel 4: Se muestran las soluciones y comentarios seleccionados por el profesor.

La distribución por niveles anterior se aplica de forma general para todos los tipos de respuesta que se obtienen del evaluador automático. Aunque en el caso de **Error de compilación** sólo existe un nivel de ayuda, que entrega los mensajes de error del compilador. La información que se muestra al usuario al enviar una solución se gestiona de forma asíncrona, al enviarse la solución y esta ser calificada por el motor de calificación automática se muestra la primera versión de la retroalimentación con la respuesta del evaluador. El proceso de generación de las pistas puede tomar desde unos pocos segundos hasta varios minutos, y al obtenerse los resultados se actualiza la interfaz de usuario. La actualización de la interfaz de usuario de forma asíncrona y sin tener que solicitar la página nuevamente al servidor es posible con el uso de la técnica de desarrollo web Ajax, que permite que el cliente mantenga una comunicación con el servidor, la que se realiza en segundo plano, al tiempo que el usuario realiza otras acciones.

ANÁLISIS ESTÁTICO

Se conoce como análisis estático al procedimiento realizado mediante varias técnicas para la obtención de información acerca del posible comportamiento en ejecución de un programa, realizado sobre el código fuente, código intermedio, código de máquina o cualquier abstracción de alguna de las opciones anteriores. Su uso, a diferencia del análisis dinámico, se orienta a encontrar violaciones a buenas prácticas en el desarrollo y no a comprobar si el código cumple con su especificación. Este tipo de análisis ha sido aplicado para encontrar errores, malas prácticas y estilos de programación inadecuados; usando diferentes niveles en las posibilidades de configuración, tipos de errores detectados y profundidad, con excelentes resultados [16-19].

Para la detección de errores mediante el análisis estático se desarrollaron sistemas como Broadway [20], PMD, Metal [18] y Findbugs [21- 22] y con ellos se han encontrado defectos en sistemas en producción, por ejemplo en el núcleo de Linux [23]. Algunas de las técnicas más comúnmente usadas pertenecientes a este tipo son el chequeo del modelo [24], análisis del flujo de datos [25], análisis del flujo de control [26], interpretación abstracta [27] y demostración de teoremas [28].

Sobre la base de los estudios realizados por [29] y [30] acerca de las herramientas existentes para el análisis estático en Java y apoyado en el estudio de la documentación pública de estas herramientas, se llegó a la conclusión sobre la factibilidad de utilizar Findbugs en su versión 2.0.0 como la mejor alternativa para basar el análisis estático de las soluciones, atendiendo a los criterios de tipo de licencia que usa, eficiencia, extensibilidad,

precisión en los errores detectados y capacidad para ser integrado en otros programas. FindBugs es una herramienta de propósito general que usa la librería BCEL para generar el grafo de control de flujo y realizar el análisis del flujo de datos intraprocedural, diseñada para ser usada con programas de cualquier tamaño y sin restricciones. Sin embargo, las soluciones que se envían en competencias de programación tienen características particulares dado que se deben escribir en un único archivo de código, usar un hilo de ejecución y solo es posible usar un subconjunto de las bibliotecas de clases provistas por el lenguaje de programación. Algunas de las reglas que se usan en Findbugs se enfocan en encontrar abrazos fatales y otras condiciones que no se adaptan a las soluciones de competencias de programación, por lo que fue necesario un proceso de personalización, posible por la flexibilidad que brinda la herramienta en cuanto a las reglas que se aplican en el análisis de forma declarativa.

El **módulo dominio** invoca la herramienta FindBugs mediante la línea de comandos, indicando los ficheros que debe analizar y la ruta donde debe escribir el fichero XML con los resultados. Estos resultados son preparados para su presentación en forma de pistas al estudiante. Para la generación de las pistas con el análisis estático de la estructura estática del programa, correspondientes al **Nivel 1** se utilizaron 128 reglas clasificadas en las categorías: mala práctica, rendimiento, estilo y código confuso (construcciones que están escritas en forma que puede llevar a cometer errores). Las pistas generadas permiten al estudiante encontrar errores tales como secciones de código inalcanzables, errores en la conversión dinámica de tipos, problemas de rendimiento, ciclos infinitos, variables no usadas, entre otras. De igual forma se fomenta el uso de buenas prácticas de codificación que aumentan la calidad del código que se genera al hacerlo menos propenso a errores.

```
1  int a = calculateA();
2  int b = calculateB();
3
4  //Resultado incorrecto
5  double result = a/b;
```

Figura 2. Fragmento de código que puede llegar a ser un resultado no deseado.

```
Línea 5: La división de enteros
trunca el resultado al valor
entero más cercano a cero. El
hecho que el resultado sea
convertido a doble sugiere que
esta precisión debería ser
retenida. Esto podría indicar que
ambos operandos deberían ser
convertidos a double antes de
realizar la división. Ejemplo:

int x = 2;
int y = 5;

// Incorrecto: resultado 0.0
double v1 =  x / y;

// Correcto: resultado 0.4
double v2 = (double)x/(double)y;
```

Figura 3. Ejemplo de pista generada mediante el análisis estático.

En la Figura 2 se puede observar un fragmento de código que puede generar un resultado que no se corresponda con el deseado por desarrollador y que se informa por el análisis estático que realiza FindBugs. La sentencia de la línea 5 ubica el resultado de la división de dos enteros (que da como resultado un entero) en una variable de tipo flotante. Ante este error el sistema ofrece la siguiente pista como se observa en la Figura 3.

ANÁLISIS DINÁMICO

El análisis dinámico permite observar el comportamiento del código mientras se ejecuta. De forma general el programa es procesado de acuerdo con un escenario de ejecución y se obtiene información sobre la ejecución de la aplicación, conocida como traza de ejecución, que puede ser o no procesada en tiempo real.

Varias estrategias se han aplicado para la obtención de esta información. Entre las más trabajadas en la bibliografía son la instrumentación del programa [31-32], el uso de depuradores para escuchar y filtrar eventos durante la ejecución [33-34] y la realización de modificaciones a la máquina virtual de Java para que genere información adicional en los reportes [35].

La instrumentación ha demostrado ser un soporte efectivo para la obtención de información sobre la ejecución programas, ya que permite niveles de granularidad importantes con menos esfuerzo en el desarrollo que otras técnicas usadas con el mismo propósito [11]. Su aplicación se basa en tomar un programa o una parte de este e insertar sentencias que generan datos sobre la ejecución de las sentencias originales al momento que se ejecute el código instrumentado. Puede ser realizada sobre el código fuente o archivos compilados, aunque la primera opción es preferible, ya que como indica el trabajo de [36] durante el proceso de compilación se pierde parte de la información semántica y de forma general en cuanto a exactitud.

Las técnicas usadas en la bibliografía consultada para realizar la instrumentación varían desde la aplicación de programación orientada a aspectos [37], herramientas para el análisis y modificación de programas compilados(en el caso de Java se refiere a Bytecode) [38] y herramientas para el análisis y modificación de código fuente [39].

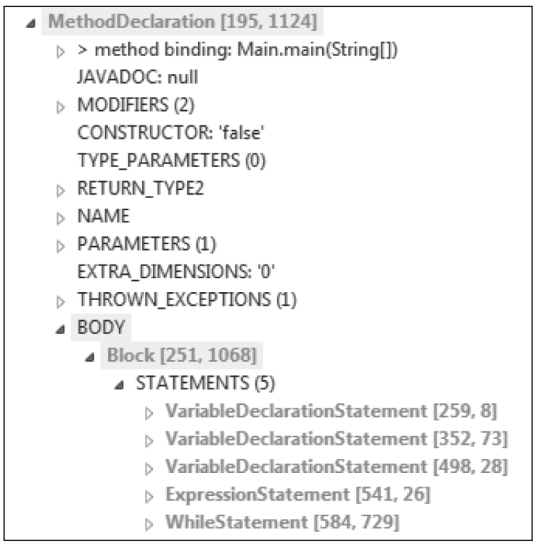

Figura 4. Fragmento del AST de una solución.

En la concepción de Asistente se seleccionó para instrumentación el soporte provisto por Java development tools (JDT) [39] que representa el código fuente a través de su Árbol de Sintaxis Abstracta (AST), una representación abstracta de la estructura sintáctica que se obtiene de procesar el código fuente con las reglas de la gramática del lenguaje de programación en que fue implementado. El AST obtenido tiene una estructura jerárquica donde la raíz se corresponde con la unidad de compilación y el resto de los nodos internos y externos representan el resto de las estructuras del programa, en la Figura 4 se puede observar parte del AST de un programa simple.

En el sistema desarrollado al solicitarse por el usuario el **Nivel 1** o superior, se inicia el análisis dinámico por el **módulo de dominio**. Como se describe a continuación, el módulo de dominio cuenta con un componente de instrumentación desarrollado para los propósitos del presente trabajo. Este instrumentador realiza el procedimiento ilustrado en la Figura 5 para lo que se apoya en el soporte que brinda JDT para la generación y modificación del AST de un programa.

El instrumentador toma como entrada el código fuente de la solución y genera el AST con JDT. El AST es recorrido para insertar en él las instrucciones que generan las trazas necesarias para el seguimiento de la solución en el momento que se ejecute. A pesar de que es posible recorrer el árbol a través de sus niveles de forma recursiva y en cada paso comprobar el tipo de nodo que se alcanzó, no es recomendado. La mejor opción consiste en hacer uso del patrón Visitador [40]. Para este propósito las clases de los nodos del árbol, subclases de ASTNode, cuentan con los métodos visit() y endVisit(). Al utilizar el patrón Visitador se extiende la clase ASTVisitor y para cada uno de los nodos que se desea procesar se implementan los métodos preVisit(ASTNode node) y postVisit(ASTNode node). Al iniciarse el recorrido con el Visitador, en cada nodo se llamarán los métodos en este orden:

1. preVisit(ASTNodenode)
2. visit()
3. se invoca recursivamente a los nodos hijos
4. endVisit()
5. postVisit(ASTNode node)

Durante el recorrido al árbol los cambios que se desean aplicarle son programados con ASTRewrite, que permite que se mantenga una versión del AST que luego puede ser escrito hacia un AST modificado y este se puede convertir en una versión instrumentada del código fuente original, listo para ser compilado y ejecutado, como se ilustra en la Figura 5.

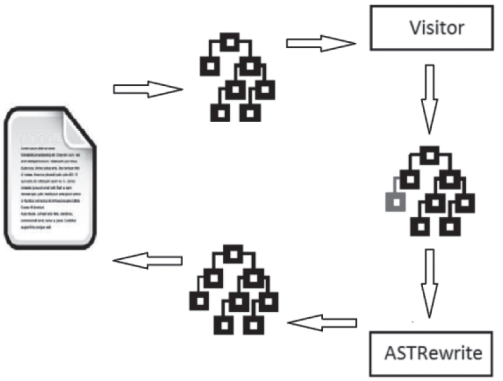

Figura 5. Proceso de instrumentación.

Una decisión común que afronta el investigador durante la concepción de la instrumentación consiste en determinar cuáles estructuras del lenguaje deben ser informadas, teniendo en cuenta que el costo computacional de la ejecución del programa instrumentado será directamente proporcional a la cantidad de información que se genere en el proceso. Durante la conceptualización de Asistente se realizó un análisis inicial sobre la información de tiempo de ejecución necesaria para generar pistas. A partir de esta versión inicial, refinada a través de un proceso de prueba y error, se llegó a un propuesta final donde se generan avisos simples de ejecución para las sentencias fundamentales del programa. Para un subconjunto de estas sentencias se generan además otras informaciones que se usan para comprender el funcionamiento del programa. Por ejemplo, los resultados de evaluación de expresiones, variables implicadas en la evaluación de expresiones, resultados de decisiones condicionales, ejecución de iteraciones para bloques iterativos, argumentos pasados a invocaciones de métodos, invocador actual del método, tiempo de ejecución de una llamada a método, tipo de datos de una declaración, datos recibidos por el programa desde la entrada estándar, datos impresos por el programa hacia la salida estándar, entre otros.

Normalmente, los programas que se generan para los ejercicios de programación para concursos no exceden a un minuto en tiempo de ejecución. Un caso especial ocurre con los ejercicios previamente calificados por el motor de evaluación automática como tiempo límite excedido, donde se impone un límite de cinco minutos para terminar la ejecución, dado que existe la posibilidad de que la ejecución no termine al encontrarse en un ciclo infinito o que en función del orden computacional de la solución y la magnitud de los datos de entrada tarde un tiempo demasiado grande. Mantener la ejecución durante un tiempo superior al límite impuesto se considera poco práctico al balancear las ventajas y desventajas, ya que durante este se priva al usuario de retroalimentación y acapara los recursos del sistema.

Las pistas con información general del análisis dinámico que se generan para el **Nivel 1** son:

- Cantidad de veces que se invoca cada método, se agrupan además por invocadores.
- Mayor y menor tiempo de ejecución obtenido para método invocado.
- Para las estructuras iterativas se muestran los números mínimo y máximo de iteraciones que se realizaron durante la ejecución del programa.
- Cantidad de ocasiones en las que se ejecutó cada instrucción, resaltando las que no fueron ejecutadas.

De forma específica, para los errores en tiempo de ejecución se muestra información detallada sobre el contexto donde se generó el error, así como los métodos que se encontraban en la pila de llamadas en ese momento.

En cuanto al **Nivel 2**, las ayudas se generan únicamente con el análisis dinámico, la idea para su desarrollo fue encontrar situaciones que se consideran como sospechosas durante la ejecución de la solución. Algunos ejemplos de situaciones que se detectan son:

- Expresiones numéricas fuera de rango. Expresiones que en su evaluación se obtiene un valor que no es posible almacenar en el tipo de dato previsto, por lo que se ajusta por el entorno de ejecución, truncando el valor original.
- Ejecución de un bloque switch que carece de bloque *default* y no se encontró una correspondencia con ninguna de las opciones.
- Falta de correspondencia entre los datos que se imprimen con los tipos que se esperan (Ejemplo: se espera un entero y se imprime un número con coma flotante).
- Elementos de la entrada que no son usados para calcular ninguna de las salidas.

- Diferencias en las entradas que se usan para calcular cada salida, encontradas al comparar la solución errónea con otras que fueron calificadas como correctas anteriormente para otros usuarios.

El análisis dinámico en este segundo nivel se apoya en la especificación sobre la entrada/salida del problema y en algunos casos en soluciones aceptadas por otros usuarios para el ejercicio que se resuelve. Para que el sistema tenga conocimiento sobre las entradas y salidas se desarrolló un lenguaje de especificación, donde el creador del problema representa la distribución, los tipos y los rangos posibles sobre los datos que se proporcionan y esperan. Esta información, de igual forma que el resto de la específica de cada problema, es almacenada en el **repositorio de problemas**.

```
program{
  typedef{
    &_COUNT AS &INT32[2,500]
  }
  input{
    $END=COUNT
    $TEST_COUNT=1
    test_case{
      &_COUNT &_COUNT
    }
  }
  output{
    &_COUNT
  }
}
```

Figura 6. Especificación de las entradas y salidas.

En la Figura 6 se muestra la especificación de entrada/salida para un problema con un único caso de prueba, constituido por dos números enteros, que pueden tener como valor mínimo 2 y máximo 500. La salida para el caso de prueba es un número entero con las mismas restricciones.

En la Figura 7 se muestra un fragmento del código fuente necesario en la solución correcta a un problema simple que se corresponde con la especificación de entrada/salida mostrada en la Figura 7. Con el mismo se resuelve el problema de sumar dos números enteros leídos de la entrada estándar e imprimir el resultado.

```
Scanner in = new Scanner(System.in);
int a = in.nextInt();
int b = in.nextInt();
System.out.println(a + b);
```

Figura 7. Fragmento de una solución correcta al problema de la suma de dos enteros.

A continuación, en la Figura 8 se presenta una versión incorrecta de solución para el mismo problema.

```
Scanner in = new Scanner(System.in);
int a = in.nextInt();
int b = in.nextInt();
System.out.println(a + 5);
```

Figura 8. Fragmento de una solución incorrecta al problema de la suma de dos enteros.

El usuario envía la solución y el sistema califica la solución mediante el mecanismo de evaluación automática. Como la solución es incorrecta se generan pistas para los niveles 1 y 2. En este caso el sistema determinará la existencia de una pista de nivel 2, perteneciente al tipo “Diferencias en las entradas que se usan para calcular cada salida, encontradas al comparar la solución errónea con otras que fueron calificadas como correctas anteriormente para otros usuarios”. Para su generación el sistema instrumenta el programa mediante el procedimiento descrito anteriormente para obtener la información de ejecución. Se compara la información de ejecución del programa con la generada por soluciones correctas, igualmente mediante la instrumentación. Tomando como ejemplo la solución correcta de la Figura 7, con la información sobre entrada/salida el sistema conoce que se ejecutará un único caso, donde se leen dos valores enteros y encuentra una correspondencia entre ambas entradas y el valor entero que se imprime.

Este procedimiento se realiza para un subconjunto de las soluciones correctas (se planea implementar en un futuro cercano un procedimiento que clasifique las soluciones más representativas) y cada vez se compara con la correspondencia entre entradas y salidas que se evidencia para la solución que se evalúa. Luego del sistema no encontrar una solución

correcta de acuerdo con el patrón de correspondencia de entradas y salidas se genera una pista donde se indica como información general "Al comparar su solución con soluciones correctas se evidencia que en su programa existe una correspondencia diferente en cuanto a las entradas usadas para calcular algunas salidas". Como información específica se muestra la siguiente:

```
Entrada:
Entero_1 Entero_2

Salida:
Entero_3

Incorrecta:
Entero_3 -> Entero_1,Entero_2

Correcta(s):
Entero 3 -> Entero 1
```

Figura 9. Pista que se genera mediante el análisis dinámico del nivel 2, con el uso de la especificación de entrada/salida.

EXPERIMENTO

Se llevó a cabo un estudio para comprobar la efectividad del sistema desarrollado y el método aplicado. El mismo se realizó en diciembre del año 2011 sobre 117 estudiantes que cursaba el primer semestre de la carrera Ingeniería en Ciencias Informáticas, en la Universidad de las Ciencias Informáticas, Cuba. En ese momento los mismos se encontraban finalizando la asignatura de Introducción a la Programación, donde se introducen en los conceptos básicos de las estructuras del lenguaje, la programación orientada a objetos y se desarrollan programas simples usando el lenguaje de programación Java. Para la realización del experimento que a continuación se describe se realizó una división de la población de 117 estudiantes en dos grupos. El primero compuesto por 60 estudiantes y el segundo compuesto por 57 estudiantes. La selección de los estudiantes que pertenecen a cada grupo se realizó de forma aleatoria, usando una tabla de números aleatorios.

Durante 10 días los estudiantes de ambos grupos se enfrentaron a los 10 ejercicios seleccionados para el experimento, los mismos para cada grupo. Cada día a las 8:00 AM se liberó un ejercicio que podía ser resuelto desde ese momento hasta el final del décimo día. Los ejercicios se presentaron ordenados ascendentemente según el nivel de dificultad y para poder resolver un ejercicio es necesario haber resuelto los anteriores. El primer grupo, que llamaremos **Ge,** será el grupo experimental y al enviar soluciones incorrectas a los problemas se le provee asistencia en forma de pistas generadas automáticamente, como se describe anteriormente en este trabajo. El segundo grupo **Gc** es el grupo de control, que al enviar soluciones incorrectas recibe solo la retroalimentación estándar generada por el mecanismo de evaluación automática de soluciones e información generada del análisis estático. Durante la ejecución del experimento para todos los estudiantes se desactivó el **Nivel 4** de ayudas en el sistema, que muestra las soluciones y comentarios realizados por los profesores, con el objetivo de no afectar los resultados.

RESULTADOS Y DISCUSIÓN

Al analizar los datos obtenidos del experimento se puede observar cómo de los 117 estudiantes que iniciaron el experimento solo 60 lograron resolver los 10 ejercicios propuestos. En total en los diez días se enviaron 3.450 soluciones, de ellas 956 correctas y 2.494 incorrectas, para un 28 por ciento de efectividad. De los 956 problemas solucionados, 145 lo hicieron al primer intento y 811 con al menos un intento. En la Figura 10 se muestra el comportamiento de la cantidad de veces que se solucionó un problema después de haber cometido N intentos incorrectos, para los especímenes de ambos grupos. En la Figura 11 y Figura 12 se puede observar el comportamiento de las variables cantidad de resueltos y cantidad de resueltos al primer intento para cada problema, en ese orden.

De las 811 veces que se resolvieron ejercicios después de realizar al menos un intento incorrecto, 440 corresponden a los sujetos del grupo experimental y 371 al grupo de control. Con el objetivo de comprobar si el nivel de pistas ofrecido afecta la cantidad de intentos incorrectos necesarios antes de lograr resolver el ejercicio de forma exitosa se aplicaron varias pruebas estadísticas. Esta creencia se encontraba apoyada por la diferencia observada entre las medias de la variable cantidad de intentos entre los grupos que recibieron tratamientos diferentes.

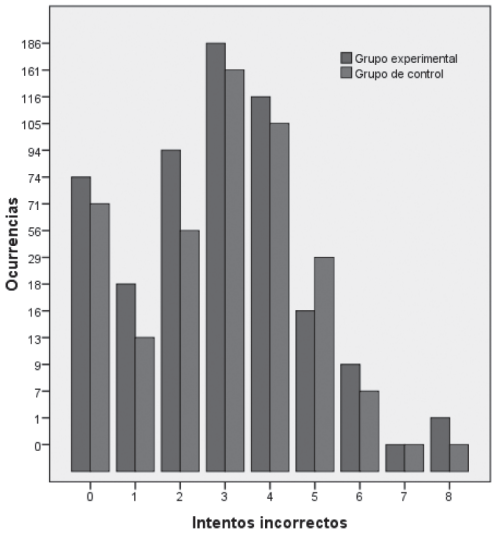

Figura 10. Cantidad de veces que se solucionó un ejercicio después de un número de intentos incorrectos.

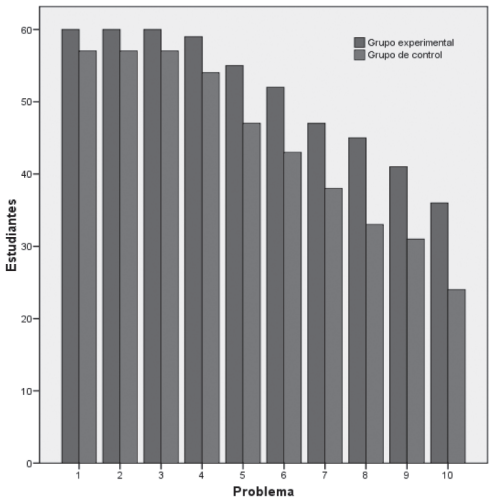

Figura 11. Estudiantes que resolvieron cada problema.

Al aplicar la prueba Kolmogorov-Smirnov a los datos se comprobó que los mismos no siguen una distribución normal. Este hecho obligó a que fuese necesaria la aplicación de la prueba U de Mann-Whitney. La misma arrojó que los datos realmente procedían de conjuntos con medias diferentes con una significación asintótica bilateral de 0,013. Hecho que permite corroborar la hipótesis de que el nivel de pistas empleado influyó en la cantidad de intentos que necesitó el estudiante para solucionar un problema.

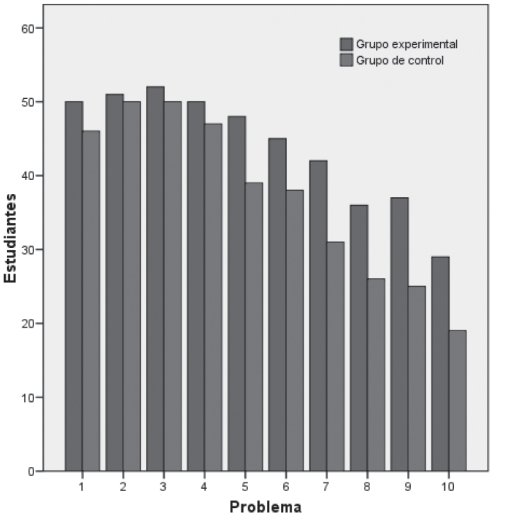

Figura 12. Estudiantes que resolvieron cada ejercicio después de al menos un intento incorrecto.

En el análisis anteriormente explicado se excluyeron de los resultados los casos en los que el estudiante solucionó un problema al primer intento sin enviar soluciones incorrectas, dado que en este comportamiento no influyen las pistas que genera la aplicación. Por el motivo expuesto incluir estos datos en el estudio podría afectar considerablemente la validez interna del mismo.

SISTEMAS RELACIONADOS

En la revisión realizada por [41] se pueden exponer los sistemas TRY[42] y CourseMaker[43] que hacen uso de la evaluación automática de las soluciones de los estudiantes, con el propósito de determinar si son correctas o no. En los trabajos de [44-45] se presentan otros, donde mediante información obtenida del análisis estático se brinda soporte a los estudiantes con el objetivo que solucionen errores y favorecer el uso de buenas prácticas en la programación. Estos sistemas al no contar con otra información dinámica que los flujos de salida del programa y no contar con otra información específica al problema tienen una limitada capacidad para ofrecer ayuda a los estudiantes.

CONCLUSIONES

El uso del análisis dinámico de soluciones en combinación en el estático permitió se generara información sobre la solución creada por el

estudiante con un mayor nivel de profundidad, que facilitara el proceso de autodiagnóstico de errores. Con la técnica desarrollada se mejoró en cuanto al soporte que provee el sistema durante la resolución de ejercicios, lo que provocó que los estudiantes fuesen capaces en un mayor grado de solucionar los errores cometidos, según se demostró en el experimento realizado.

La especificación introducida para la representación de información acerca del formato de las entradas y salidas permitió al sistema razonar en función de ciertas características del problema, que son generales para todas las soluciones posibles del mismo.

TRABAJO FUTURO

En próximos trabajos sería interesante estudiar el impacto real en la calidad del aprendizaje del método expuesto integrado a un ambiente de aprendizaje. De igual forma es necesario investigar desde una perspectiva pedagógica el impacto del mal uso que pueden realizar los estudiantes del sistema de pistas expuesto, donde pueden solicitar todos los niveles de pistas al primer intento incorrecto, sin detenerse a analizar con los datos que se obtienen de cada nivel.

REFERENCIAS

[1] ACM. "Curriculum Guidelines for Undergraduate Degree Programs in Information Technology". 2008. Date of visit: September 12, 2011. URL: http://www.acm.org/education/curricula/IT2008%20Curriculum.pdf

[2] ACM. "Curriculum Guidelines for Undergraduate Degree Programs in Information Systems". 2010. Date of visit: September 12, 2011. URL: http://www.acm.org/education /curricula/IS%202010%20ACM%20final.pdf

[3] ACM. "Curriculum Guidelines for Graduate Degree Programs in Software Engineering". 2009. Date of visit: September 12, 2011. URL: http://www.gswe2009.org/fileadmin/files/ GSwE2009_Curriculum_Docs/GSwE2009_version_1.0.pdf

[4] ACM. "Computer Science Curriculum 2008: An Interim Revision of CS 2001". 2008. Date of visit: September 12, 2011. URL: http://www.acm.org/education/ curricula/ComputerScience2008.pdf

[5] ACM. "Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering". 2004. Date of visit: September 12, 2011. URL: http://www.acm.org/education/education/curric_vols/CE-Final-Report.pdf

[6] C. Madoz y A. DeGiusti. "Vinculación de un curso interactivo multimedial". Memorias del Congreso Argentino de Ciencias de la Computación CACIC97. 1997.

[7] F.A. Salgueiro, G. Costa, Z. Cataldi, R. García y F.J. Lage. "Sistemas inteligentes para el modelado del tutor". GCETE'2005, Global Congress on Engineering and Technology Education. 2005.

[8] T.O. Junco, E.J. Altuna y J.A. Soria. "El Jurado Online como alternativa para la evaluación de las materias de programación en la enseñanza universitaria". IV Conferencia Científica Internacional Universidad de Holguín "Oscar Lucero Moya". 2009.

[9] D. Perkins. "La escuela inteligente". Editorial GEDISA. 1997.

[10] N. Nethercote and J. Seward. "Valgrind: A program supervision framework". 3rd Workshop on Runtime Verification. 2003.

[11] C. Huajie, Z. Tian, B. Lei and L. Xuandong. "An Instrumentation Tool for Program Dynamic Analysis in Java". Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion. IEEE Computer Siciety. 2011.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser and G. Lowney. "Pin: Building customized program analysis tools with dynamic instrumentation". ACM SIGPLAN Conference on Programming Language Design and Implementation. 2005.

[13] K. Mukhar, C. Zelenak, J.L. Weaver and J. Crume. "Beginning Java EE 5: From Novice to Professional". Apress, pp. 5-21. New York, United States of America. ISBN: 1-59059-470-3. 2006.

[14] JCP. JSR 154: JavaTM Servlet 2.4 Specification. 2007. Date of visit: January 19, 2012. URL: http://jcp.org/ jsr/detail/154.jsp

[15] J.R. Carbonell. "AI in CAI: An artificial intelligence approach to computer assisted instruction". IEEE transaction on Man

Machine System. Vol. 11, Issue 4, pp. 190-202. 1970.

[16] U. Shankar, K. Talwar, J.S. Foster and D. Wagner. "Detecting format string vulnerabilities with type qualifiers". USENIX Security Symposium. 2001.

[17] D. Evans and D. Larochelle. "Improving security using extensible lightweight staticanalysis". IEEE Software. Vol. 19, Issue 1, pp. 42-51. February, 2002. DOI: 10.1109/52.976940.

[18] K. Ashcraft and D. Engler. "Using programmer-written compiler extensions to catch security holes". IEEE Symposium on Security and Privacy. 2002.

[19] G. Wasserman and Z. Su. "Sound and precise analysis of web applications for injection vulnerabilities". ACM SIGPLAN Conference on Programming Language Design and Implementation. 2007.

[20] S.Z. Guyer. "Incorporating Domain-Specific Information into the Compilation Process". Doctoral Dissertation. Austin, Texas: The University of Texas at Austin; 2003.

[21] N. Ayewah, D. Hovemeyer, D. Morgenthaler, J. Penix and W. Pugh. "Experiences Using Static Analysis to Find Bugs". IEEE Software. Vol. 25, Issue 5, pp. 22-29. October, 2008. DOI: 10.1109/MS.2008.130.

[22] N. Ayewah, W. Pugh, D. Morgenthaler, J. Penix and Y. Zhou. "Evaluating Static Analysis Defect Warnings On Production Software". 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2007.

[23] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton and S. Hallem. "A few billion lines of code later: Using static analysis to find bugs in the real world". Communications of the ACM. Vol. 53, Issue 2, pp. 66-75. February, 2010. DOI: 10.1145/1646353.1646374.

[24] E.M. Clarke, O. Grumberg and D.A. Peled. "Model Checking". MIT Press. 1999.

[25] J.C. Huang. "Detection of data flow anomaly through program instrumentation". IEEE Transactions on Software Engineering. Vol. 5, Issue 3, pp. 226-236. May, 1979. DOI: 10.1109/TSE.1979.234184.

[26] L. Hubert, N. Barre, F. Besson, D. Demange, T. Jensen and V. Monfort. "Sawja: Static Analysis Workshop for Java". Formal Verification of Object-Oriented Software. 2010.

[27] P. Cousot and R. Cousot. "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". ACM Symposium on Principles of Programming Languages (POPL'77). ACM Press. 1977.

[28] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe and R. Stata. "Extended Static Checking for Java". ACM SIGPLAN Conference on Programming Language Design and Implementation. 2002.

[29] C.N. Christopher. "Evaluating Static Analysis Frameworks". Carnegie Mellon University. 2006. Date of visit: February 19, 2012. URL: http://www.cs.cmu. edu/~aldrich/ courses/654-sp09/tools/ christopher-analysis-frameworks-06.pdf

[30] I.L. Codesido. "Comparación de analizadores estáticos para código Java". Master Thesis. Universidad Abierta de Cataluña. 2011.

[31] X. Baoxi. "Instrumentation technique and its applications in software analysis and debug". Doctoral Thesis. Nanjing University. 2009.

[32] G. Xu and A. Rountev. "Precise memory leak detection for Java software using container profiling". 30th International Conference on Software Engineering. 2008.

[33] A. Zaidman. "Scalability solutions for program comprehension through dynamic analysis". Doctoral Thesis. 2006.

[34] D. Insa and J. Silva. "An Algorithmic Debugger for Java". 2010 IEEE International Conference on Software Maintenance (ICSM). 2010.

[35] R. Keller and U. Hölzle. "Binary Component Adaptation". ECCOP '98 Proceedings of the 12th European Conference on Object-Oriented Programming. 1998.

[36] Y. Li-Qian, W. Lin-Zhang, L. Bin, Z. Jian-Hua and L. Xuan-Dong. "Combined Static and Dynamic Immutability Analysis of Java Program". Chinese Journal of Computers. Vol. 33, Issue 4, pp. 736-746. May, 2010. DOI: 10.3724/SP.J.1016.2010.00736.

[37] K.C. Foo, J. Guo and Y. Zou. "Verifying Business Processes Extracted from E-Commerce Systems Using Dynamic Analysis". International Workshop on Program Comprehension through Dynamic Analysis. 2007.

[38] M. Dahm. "Byte code engineering". Java-Informations-Tage. 1999.

[39] T. Kuhn, O. Thomann. "Abstract Syntax Tree". 2006. Date of visit: September 20, 2011. URL: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

[40] E. Gamma, R. Helm, R. Johnson and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison Wesley. 1994.

[41] K.M. Ala-Mutka. "A Survey of automatic assessment approaches for programming assignments". Computer Science Education. Vol. 15, Issue 2, pp. 83-102. April, 2005.

[42] K.A. Reek. "The TRY system -or- how to avoid testing student programs". SIGCSE technical symposium on Computer science education. 1989.

[43] C. Higgins, G. Gray, P. Symeonidis and A. Tsintfsias. "Automated assessment and experiences of teaching programming". ACM Journal on Educational Resources in Computing. Vol. 5, Issue 5, pp. 1-21. September, 2005. DOI: 10.1145/ 1163405.1163410.

[44] N. Truong, P. Roe and P. Bancroft. "Static Analysis of Students' Java Programs". Sixth Australian Computing Education Conference. Australian Computer Society, Inc., Dunedin, New Zealand. 2004.

[45] M. Hristova, A. Misra, M. Rutter and R. Mercuri. "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students". SIGCSE '03. 2003.