Información Tecnológica
Vol. 25(3), 115-124 (2014)
doi: 10.4067/S0718-07642014000300014

Una Revisión sobre la Ejecución Simbólica de Programas Computacionales

Cristian L. Vidal[(1)], Rodolfo F. Schmal[(2)], Sabino Rivero[(2)] y Rodolfo H. Villarroel[(3)]
(1) Escuela de Ingeniería Informática, Facultad de Ingeniería y Administración, Universidad Bernardo O'Higgins, Avenida Viel 1497, Ruta 5 Sur, Santiago-Chile (e-mail: vidalsil@msu.edu)
(2) Escuela de Ingeniería Informática Empresarial, Facultad de Ciencias Empresariales, Universidad de Talca Campus Lircay, Avenida Lircay S/N, Talca-Chile (e-mail: rschmal@utalca.cl; srivero@utalca.cl)
(3) Escuela de Ingeniería Informática, Facultad de Ingeniería, Pontificia Universidad Católica de Valparaíso, Avenida Brasil 2241, Valparaíso-Chile (e-mail: rodolfo.villarroel@ucv.cl)

Recibido Oct. 16, 2013; Aceptado Nov. 28, 2013; Versión final recibida Ene. 16, 2014

Resumen

El objetivo de este artículo es presentar la ejecución simbólica de programas y su extensión, ejecución simbólica generalizada, para señalar las mejoras necesarias a la ejecución simbólica para que llegue a ser un enfoque práctico de verificación de programas. El análisis de programas permite determinar niveles de correctitud de software o cumplimiento de los requerimientos de usuario. Existen dos enfoques para la verificación de programas, analítica y dinámica, y en medio de ellas, existe la ejecución simbólica la que estáticamente analiza el código fuente de programas, y dinámicamente simula la ejecución de las instrucciones ejecutables de programas por medio de datos de entrada simbólicos. En este trabajo se describe los conceptos de verificación de programas, la propuesta original de ejecución simbólica junto a sus ventajas y desventajas, y las principales características de ejecución simbólica generalizada. Finalmente, se resumen las principales áreas de investigación que se abren relacionadas con la ejecución simbólica.

Palabras clave: verificación de software, ejecución simbólica, EjeSim, ESG

A Review about Symbolic Execution of Computer Programs

Abstract

The objective of this paper is to present the symbolic execution of programs and its extension, generalized symbolic execution, to indicate the necessary improvements to symbolic execution so that it becomes a practical approach for program verification. Program analysis allows determining levels of software correctness or compliance with the user requirements. There are two approaches for program verification, analytic and dynamic, and, between them, symbolic execution exits which statically analyzes the program source code, and dynamically simulates the execution of executable instructions of programs by means of symbolic input data. In this paper, concepts of program verification, the original proposal of symbolic execution along with their advantages and disadvantages, and the main features of generalized symbolic execution are described. Finally, the main open research areas related to symbolic execution are summarized.

Keywords: software verification, symbolic execution, SymExe, GSE

INTRODUCCIÓN

Los usuarios buscan productos de software que sean confiables, robustos y libres de errores de ejecución, que funcionen como esperan y sin errores inesperados en tiempo de ejecución. Tradicionalmente, las técnicas usadas para verificar propiedades de calidad del programas son a) Verificación estática, o análisis de software que incluye el chequeo de modelos o model checking, y b) Verificación dinámica, también conocida como verificación experimental, o prueba de software. El análisis de programas de software, mediante predicciones y análisis de código fuente, busca asegurar que una aplicación software completamente satisface los requerimientos de usuario previamente establecidos, condición no siempre alcanzable (Ghezzi et al., 2002). El análisis de una aplicación software caracteriza una clase de posibles ejecuciones de dicha aplicación, mientras que una prueba de software caracteriza a una ejecución individual. Como una forma moderna de análisis de software, el chequeo de modelos permite de manera automática analizar sistemas (Ghezzi et al., 2002; Khurshid et al., 2003). La prueba de programas de software es un proceso que busca determinar si en la práctica una aplicación software funciona de acuerdo a sus requerimientos (Ghezzi et al., 2002). Principalmente, las pruebas de software detectan la diferencia entre las condiciones de funcionamiento actuales y requeridas, dada la presencia de fallas de ejecución o bugs. Aun cuando las pruebas de software pueden ser muy útiles para la verificación práctica de software, es necesario considerar que: "las pruebas de software puede ser muy efectivas para mostrar la presencia de errores o bugs, pero ellas no son adecuadas para mostrar la ausencia de ellos" (Dijkstra, 1972). Así, las pruebas de software pueden ser usadas para mostrar la presencia de errores, pero no su ausencia.

Entre verificación estática y dinámica, existe otra forma de verificación de programas de software denominada ejecución simbólica (EjeSim) propuesta por King (1976). EjeSim es una técnica de verificación que integra elementos de pruebas de software y pruebas de correctitud para analizar y evaluar el software mediante el uso de entradas de datos simbólicas en lugar de valores concretos. Así, los valores de variables de un programa ejecutado simbólicamente son expresiones simbólicas, y sus salidas son funciones de sus entradas simbólicas. Cuando se encuentra un error en el código del programa que EjeSim está analizando, los valores simbólicos son instanciados a valores concretos que presentan las condiciones del error de ejecución. En general, las salidas de EjeSim son útiles para la generación automática de casos de prueba (Păsăreanu y Visser, 2009; Cadar et al., 2011; Cadar y Sen, 2013). Puesto que hay propuestas de nuevas versiones de EjeSim tal como ejecución simbólica generalizada (ESG) (Khurshid et al., 2003), así como también nuevas aplicaciones de EjeSim (Belt et al., 2011; Păsăreanu, et al., 2013). El objetivo de este artículo es analizar los elementos potencialmente clave para que EjeSim llegue a ser un enfoque práctico para la verificación de software. Para estos efectos se presentan conceptos de verificación de programas; se revisan los elementos y algoritmos de EjeSim clásica; se describen los elementos, características, y algoritmos ESG, una extensión de EjeSim; y se resume una aplicación general de EjeSim.

VERIFICACIÓN DE PROGRAMAS DE SOFTWARE

La verificación de software persigue una completa correctitud, esto es, correctitud en el proceso y en cada uno de los productos del desarrollo de software. Esta meta no es alcanzable del todo principalmente por factores humanos y técnicos tales como la diversidad de personas en las etapas de desarrollo de software, y las cualidades del software a ser verificadas (Ghezzi et al., 2002). A continuación, se describen las técnicas de verificación estática y dinámica de software.

Verificación Estática

El análisis de programas de software, formal o no formal, es el arte de hacer predicciones acerca del comportamiento en la ejecución de un programa, a partir de su código fuente para verificar si el software cumple sus requerimientos y si el software produce fallas (Holzmann, 2002). De esta forma, la principal meta del análisis de software es la detección de errores y defectos en el código fuente del software. 'Marchar sobre el código' e 'Inspecciones de código' son ejemplos de técnicas de análisis informal de software, mientras 'Pruebas de correctitud' es un ejemplo de técnica de análisis formal de software. 'Marchar sobre el código' emula el funcionamiento de piezas de software en busca de fallas, en tanto que 'Inspecciones de código' analiza el código de software en búsqueda de errores conocidos y definidos, como por ejemplo, el acceso a variables no inicializada. En 'Pruebas de correctitud' se usa una notación de pre- y post-condiciones, y por medio de substitución hacia atrás, se intenta probar que el programa analizado garantiza la veracidad de sus post-condiciones al final de su ejecución (Ghezzi et al., 2002).

El chequeo de modelos es una técnica actual de análisis de software que consiste en representar un sistema como una máquina de estados finitos (MEF), y dada una propiedad del software expresada en un definido formalismo, entonces verificar si el comportamiento del sistema satisface la propiedad deseada (Ghezzi et al., 2002). Este último paso de chequeo de modelos es automático, lo que es una de las grandes

virtudes de esta forma de análisis. Dado que en una MEF puede ocurrir una explosión en su espacio de estados, el chequeo de modelos aplica técnicas de reducción sobre el conjunto de estados: reducción de orden parcial y reducción simétrica (Baier y Katoen, 2008).

Verificación Dinámica

La verificación dinámica o pruebas de software (usualmente un proceso manual) es la forma más común de verificar una pieza de software. Esta forma de verificación consiste en operar la pieza de software en una situación representativa con datos de entradas ejemplo, y verificar si el comportamiento obtenido es el esperado, es decir, si los datos de salida representan la salida esperada (Ghezzi et al., 2002). Así, es relevante definir casos de prueba que exhiban el comportamiento deseado de un sistema con casos no testeados previamente. En la práctica, para mejorar el rango de cobertura de pruebas de software, usualmente los casos de prueba se generan de manera aleatoria. Este esquema presenta dos claras debilidades: casos de prueba redundantes, y baja probabilidad de generar casos de prueba que produzcan fallas de ejecución (Williams, 2011; Sen et al., 2005). Por su parte, Khurshid et al. (2003) argumentó que las pruebas de software no son útiles para encontrar errores o fallas de ejecución en problemas concurrentes debido al no determinismo.

El principal objetivo de las pruebas de software es buscar por fallas de ejecución mientras que el del análisis de software es buscar fallas para definir niveles de correctitud. Así, para comparar análisis y pruebas de software, se pueden definir características tales como el grado de correctitud obtenible por cada técnica de verificación, así como también el costo de aplicación de cada técnica, y comparar sus resultados para seleccionar la más adecuada. En relación al grado de correctitud, según Ghezzi et al. (2002), el análisis de software permite obtener altos niveles de correctitud.

Chequeo de modelos garantiza un resultado de verificación para una propiedad dada, en un sistema modelado como una MEF. Sin embargo, el algoritmo de chequeo de modelos suele ser de alta complejidad debido a la explosión combinatoria del número de estados de una MEF. Por ello, en la práctica, el chequeo de modelos es más aplicado para verificar sistemas que presentan un pequeño número de estados tales como diseño de hardware y de protocolos de comunicación. Últimamente, elementos de software también son modelados como instancias de MEF (Ghezzi et al., 2002; Wagner et al., 2006). Para verificación dinámica, sólo una prueba exhaustiva de software, esto es, bajo todas las posibles circunstancias y valores, entrega certeza respecto a la correctitud. Pruebas exhaustivas son usualmente realizadas sobre sistemas de software de ámbito pequeño, esto es, con variables de entrada y salida de dominio y rango finito. Por esta razón, en proyectos con variables no finitas, las pruebas exhaustivas de software no son prácticas por sus tiempos y costos asociados (Ghezzi et al., 2002; Williams, 2011).

En cuanto al costo de aplicación, Ghezzi et al. (2002) objeta el uso de 'Pruebas de correctitud' debido al alto uso de pruebas formales y matemáticas así como también la velocidad y memoria requerida por los dispositivos de computación asociados. Además, una prueba formal, en su número de pasos, es frecuentemente aún más grande y más compleja que su programa analizado. La verificación dinámica no presenta estas objeciones. Respecto al costo de pruebas de software, Williams (2011) señala que el proceso de pruebas de software utiliza la mitad del costo de desarrollo de software y mantención. Por su parte, Ghezzi et al. (2002) observa que datos empíricos de proyectos industriales muestran que el costo de remover errores después que el software ha sido desarrollado es mucho mayor que el costo de removerlos con anterioridad.

De esta manera, si bien un análisis de software bien aplicado permite alcanzar altos niveles de correctitud, es oneroso. Por otra parte, las pruebas de software no garantizan un alto grado de correctitud, pero su aplicación es más económica. Por ello, verificar software usando una mezcla de análisis y prueba de software podría proveer resultados con alto grado de correctitud. El grado de aplicación de cada técnica de verificación dependerá del tipo de software a ser verificado. Por ejemplo, para software de uso crítico, tal como uno para medir y controlar la temperatura de pacientes de un hospital, análisis de software debería ser aplicado para verificar que el software teóricamente funciona. De la misma forma, pruebas de software deberían usarse para verificar el correcto funcionamiento del software en casos prácticos, hasta que se considere que casos generales y críticos han sido verificados.

Puesto que todas las cualidades del software deberían ser verificadas, principios de separación de incumbencias pueden ser usadas para la verificación de software, principalmente para coordinar análisis y diseñar casos de prueba. En Ghezzi et al. (2002) se presentan ejemplos de casos de prueba específicos tales como prueba de sobrecarga y test de confianza de software, los que se enfocan en incumbencias específicas. Así, de manera similar, para verificar una funcionalidad X de una aplicación software, se puede analizar el software y diseñar una prueba particular para la función X.

EJECUCIÓN SIMBÓLICA

King (1976) propuso EjeSim para obtener programas de software confiable. Para ello EjeSim inspecciona el código fuente de los programas y emula su ejecución como una técnica de análisis, utilizando datos simbólicos como datos de entrada, pero datos simbólicos en lugar de datos concretos, para producir software parcialmente correcto. De allí que, EjeSim sea una síntesis de los enfoques de verificación de software estática y dinámica. Además, de acuerdo a Ghezzi et al. (2002), un programa p es parcialmente correcto para pre- y post-condiciones P y Q respectivamente, si siempre que p inicia su ejecución en un estado que satisface P, si p termina, este lo hace en un estado que satisface Q.

Como se ha indicado, para la verificación de programas, EjeSim analiza la ejecución de los programas con entradas simbólicas. De esta manera, los programas analizados por EjeSim presentan estados simbólicos. El estado de un programa analizado por EjeSim incluye el valor simbólico de sus variables α, una ruta de ejecución β, y una condición de ruta PC, siendo esta última una fórmula booleana libre de cuantificadores sobre entradas simbólicas. Así, dado un estado simbólico $\sigma = <\alpha, \beta, PC>$de un programa en ejecución simbólica, EjeSim trabaja como sigue:

i) En cada instrucción o estamento no condicional que genera un nuevo valor para una variable del programa, α es actualizado.

ii) En cada instrucción o estamento condicional “IF e THEN S1 ELSE S2”, hay dos restricciones o condiciones de ruta. Así, PC es actualizado a PC = PC ∧ σ(e), y una nueva condición de ruta PC’= PC’∧ (! σ(e)) es creada. Para estas fórmulas, σ(e) entrega el valor simbólico de la expresión e, mientras PC y PC’ son condiciones de ruta que determinan la satisfacibilidad para una ejecución de asignar valores concretos para las variables simbólicas asociadas. En estas fórmulas lógicas, el símbolo ∧ es la conjunción lógica mientras ! se corresponde a la negación lógica.

iii) Si PC o PC’ no es viable, la ejecución simbólica por esta ruta termina.

iv) Si una ejecución simbólica alcanza una instrucción de salida o término, o un error, por ejemplo, el programa falla o viola una aserción, la correspondiente instancia de ejecución simbólica termina y una asignación que satisface la actual condición simbólica de ruta es generada, usando un solucionador de condiciones comercial o un procedimiento de decisión propio.

Como se aprecia en este algoritmo, EjeSim es muy útil para la detección de errores de ejecución o bugs, y para la generación de casos de prueba para probar la corrección parcial de programas. En ambos casos, por conmutatividad de EjeSim, las entradas concretas de los casos de prueba generados alcanzan la misma terminación encontrada por EjeSim (King, 1976). Un árbol de EjeSim caracteriza las rutas de ejecución seguidas durante la ejecución simbólica de un programa, y sus nodos representan estados del programa, los cuales se conectan por medio de transiciones del programa (ejecución de las instrucciones). Para construir un árbol, EjeSim usa una exploración primero en profundidad de las rutas y vuelta atrás (backtracking). En este plano, King (1976) implementó un sistema denominado EFFIGY para EjeSim.

En consecuencia, EjeSim es una técnica para el análisis de programas, y de esta forma producir programas confiables dado que por medio de deducción aritmética, se puede concluir que se satisface la especificación de un programa analizado por EjeSim (Ghezzi et al., 2002). Por ejemplo, el árbol de EjeSim de la figura 2 asociado al código de programa en C de la figura 1 muestra que, el procedimiento analizado funciona correctamente para todos los casos posibles. Como indica Ghezzi et al. (2002), muchos de los pasos de EjeSim pueden ser realizados tal como la substitución hacia atrás es realizada en pruebas de correctitud de programas

```
Double Total, Precio;

1. if (Total > Precio)
   {
 2.   Total = Total - Precio;

 3.   if (Total == 0)
 4.       assert(false);
    }
```

Fig. 1: Código de Programa a Ser Analizado por EjeSim.

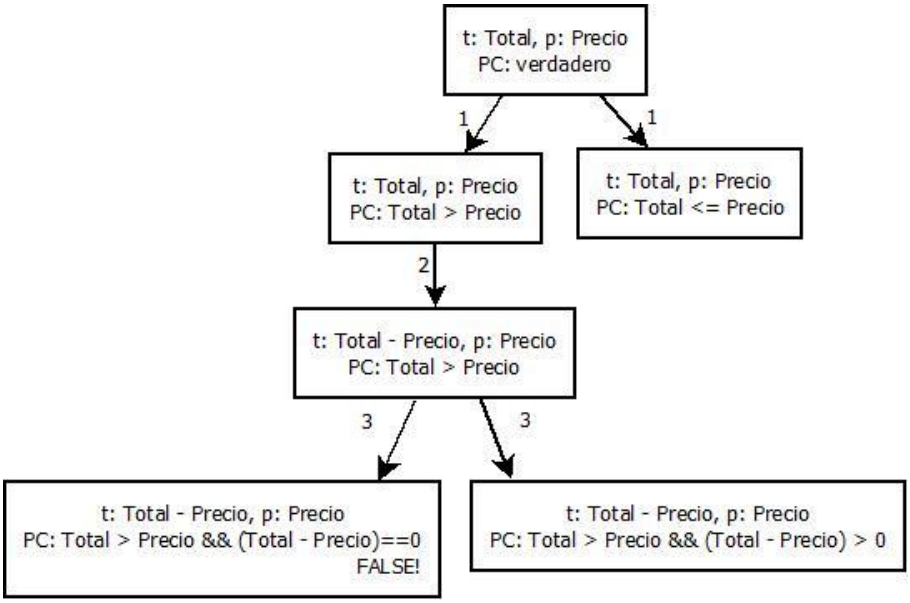

Fig. 2: Árbol de EjeSim de Programa de Figura 1.

Se destaca que EjeSim permite analizar programas secuenciales, pero sin ciclos, esto es programas con loops o recursivos, que incluyan una condición de término completamente simbólica, es decir con variables simbólicas sin recibir valores concretos con una asignación explícita. Programas con ciclos con una condición de término simbólica usualmente producen árboles de EjeSim infinitos. Dado que el número de rutas que EjeSim explora puede crecer exponencialmente, esto se considera un problema de término o escalabilidad. Una solución para evitar la ejecución infinita es limitando la profundidad de la búsqueda de EjeSim (King, 1976). En la práctica, EFFIGY permite una elección interactiva de la ruta a seguir en todo estamento condicional así como también de limitar la búsqueda exhaustiva de EjeSim mediante la facilidad TEST.

También se debe señalar que EjeSim trabaja con sólo variables enteras, esto es, EjeSim no permite trabajar con variables que representan estructuras, referencias, número reales y cadenas de caracteres, entre otras. Se trata de una importante limitación puesto que estructuras y variables de referencia son elementos básicos de programación. EjeSim no resuelve todos los tipos de fórmulas que PC puede contener (Cadar y Sen, 2013), puesto que EjeSim trabaja sólo con expresiones enteras lineales. Es decir, fórmulas con números reales o con operadores matemáticos diferentes a suma y resta no son tratables o solucionables para EjeSim. Este es un asunto clave de EjeSim (King,1976). Siguiendo el principio de separación de incumbencias, si existiera alguna incumbencia o propiedad posible de verificar por medio del análisis del código fuente de programas no verificable por EjeSim original, se puede extender EjeSim con el fin de analizar el código fuente para trabajar con esta incumbencia. Claro está que la incumbencia puede ser relativa a algún patrón de ejecución no soportado originalmente por EjeSim, como lo es trabajar con expresiones aritméticas no enteras.

EJECUCIÓN SIMBÓLICA GENERALIZADA

Como se revisó en la sección previa, EjeSim permite chequear ciertos programas secuenciales con variables enteras solamente. Khurshid et al. (2003) propuso una ejecución simbólica generalizada (ESG) con dos objetivos: primero, aplicar EjeSim sobre programas Java; y segundo, instrumentar el código fuente de los programas a ser analizados por ESG, para usar un chequeador de modelos, Java Pathfinder (JPF), para explorar las rutas de ejecución simbólica.

Puesto que Java, como lenguaje de programación orientada a objetos, permite definir variables de referencias u objetos a instancias de clase, además de soportar concurrencia, gracias al uso de JPF, ESG analiza programas concurrentes, y además permite la ejecución simbólica de métodos o subrutinas de clase que aceptan argumentos de referencias gracias al algoritmo de *inicialización tardía* (Khurshid et al., 2003). Además, ESG permite definir pre-condiciones de métodos como una forma de restringir ejecuciones no deseadas.

El algoritmo de inicialización tardía trabaja sobre campos usados por el método analizado, donde un campo puede representar a uno de sus argumentos o un atributo de un objeto usado en el código del método. La figura 3 muestra el algoritmo de inicialización tardía el que distingue entre campos de referencia y campos primitivos no inicializados, y considera tres formas no determinísticas para inicializar un campo de referencia. Nótese que este algoritmo, en el caso de un campo de referencia, evalúa si alguna pre-condición es violada para producir una vuelta atrás. Finalmente, en el caso de un campo primitivo no inicializado, este recibe un nuevo valor simbólico del tipo apropiado.

```
si f no está inicializado entonces
    si f es un campo de referencia de tipo T entonces
        no determinísticamente inicializar f a
        1. null
        2. un nuevo objeto de clase T (con valores de campos no inicializados)
        3. un objeto creado durante una inicialización previa de un campo de tipo T
        si alguna pre-condición es violada entonces
            vuelta atras()
        fin si
    fin si

    si f es un campo primitivo (o string) entonces
        inicializar f a un nuevo valor simbólico del tipo apropiado T
    fin si
fin si
```

Fig. 3: Algoritmo de Inicialización Tardía de ESG (de Khurshid et al., 2003)

```
class Nodo{
  int elemento;
  Nodo siguiente;

  Nodo cambiarNodos(){

1.  if (siguiente!=null)
2.     if (elemento - siguiente.elemento > 0){
3.        Nodo temp = siguiente;
4.        siguiente = temp.siguiente;
5.        temp.siguiente = this;
6.        return temp;
          }
7.     return this;
     }
}
```

Fig. 4: Código de Clase Nodo y Método cambiarNodos

Cuando el algoritmo de ESG encuentra una condición sobre campos primitivos, esa condición o su negación es agregada no determinísticamente a la correspondiente condición de ruta, y el algoritmo chequea la factibilidad de solución para la condición de ruta. En el caso de que una condición de ruta no sea solucionable, el algoritmo realiza una vuelta atrás si hay rutas aún no exploradas, y termina en caso contrario.

En la figura 4 se muestra el código de un programa Java, y la figura 5 muestra el árbol ESG para la ejecución simbólica del método cambiarNodos de este código Java. En este árbol, el valor "?" para un campo elem indica que el campo no es accedido, y, de la misma forma, la "nube" indica que el campo siguiente no es accedido. Esta figura no ilustra valores nulos. Además, el árbol de la figura 5 asume el uso de una pre-condición: la entrada debe ser una estructura de datos acíclica. Como una consecuencia, en este árbol de ESG, no se exploran las transiciones marcadas con una "X".

Junto con aplicar EjeSim sobre programas Java, el objetivo adicional de ESG es instrumentar el código fuente de programas a ser analizados para el uso de JPF como el analizador para explorar rutas de ejecución simbólicas. Para lograrlo, primero, se define una traducción de código a código para la instrumentación de un programa, la cual habilita realizar la ejecución simbólica del programa usando JPF, de manera de no necesitar un analizador propio.

En ESG, el chequeador de modelos JPF revisa el programa instrumentado usando la técnica de exploración del espacio de estados. Un estado incluye una configuración del heap de ejecución de Java, una condición de ruta sobre datos primitivos, y una ordenación cronológica de la ejecución de hilos. Cuando una condición de ruta es actualizada, esta es analizada para determinar su viabilidad de solución usando un procedimiento de decisión adecuado, tal como la librería Omega para restricciones enteras lineales. Si la condición de ruta no es solucionable, el chequeador de modelos JPF prosigue con una vuelta atrás (Khurshid et al., 2003).

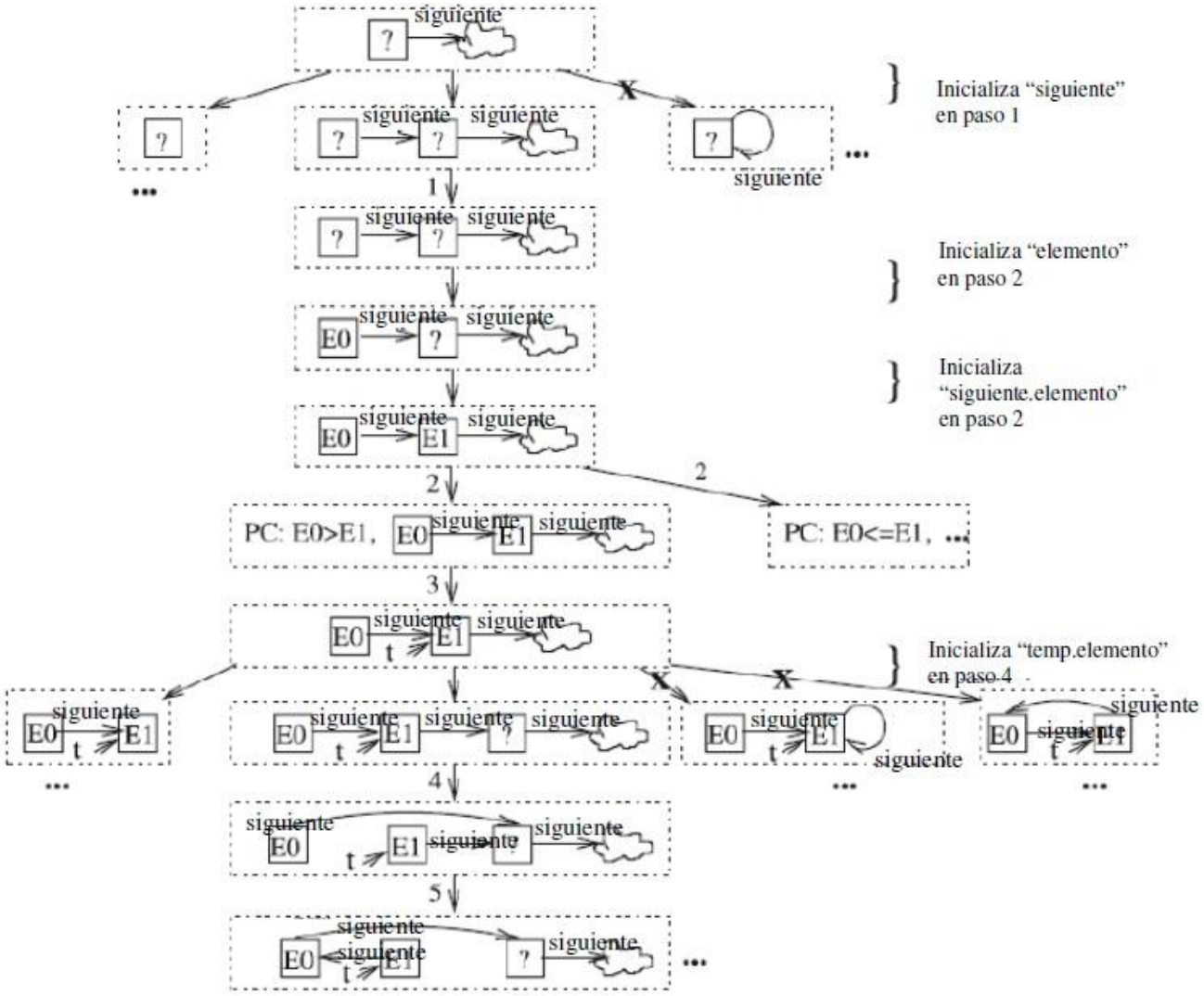

Fig. 5: Árbol de ESG de Método swapNode de Figura 4.

Entre los beneficios de llevar a cabo una ejecución simbólica con un chequeador de modelos para ambas herramientas (Khurshid et al., 2003) se incluyen:

i) Direccionar el problema de explosión del espacio de estado del chequeado de modelos gracias al uso de datos simbólicos que representan datos de grandes dominio, de manera que no es necesario revisar conjuntos de datos concretos;

ii) Lograr modularidad debido al uso de inicialización tardía, es posible analizar elementos de compilación como métodos de manera individual;

iii) Verificar la correctitud de programas concurrentes que usan entradas de dominios no restringidos con estructuras complejas;

iv) Explotar las capacidades propias de un chequeador de modelos tales como sus diferentes estrategias de búsqueda, como son la búsqueda heurística, el chequeo de propiedades temporales, y el uso de reducción de orden parcial y de simetría.

Khurshid et al. (2003), indicó que las principales ventajas de esta integración entre EjeSim y un chequeador de modelos en ESG son los multihilos, el uso de ciclos, el uso de recursión, y la invocación de métodos, características soportadas por JPF. En general, tal como EjeSim, el marco de trabajo de ESG permite chequear por la correctitud de programas y generar datos de entrada para pruebas (Khurshid et al., 2003). Por tanto, realizar ejecución simbólica durante el chequeo de modelos es un poderoso enfoque para analizar software dados los beneficios de esta integración. Nótese que, dado que la ejecución simbólica generalizada trabaja con la ejecuciones de métodos de clase y de instancias de clase definidos, las asociaciones entre clases e instancias de clase no son afectadas por dicha ejecución. Sin embargo, ESG se define para trabajar sobre métodos que trabajan con datos sólo de la clase.

CONCLUSIONES

Del análisis en torno a EjeSim y ESG para la verificación de software, se deduce la necesidad de un mejorado solucionador de condiciones, de enfoques para manejar el crecimiento exponencial de rutas en ejecución simbólica, de una integración con chequeo de modelos, y de extender la ejecución simbólica para verificar elementos de otros enfoques o metodologías de programación.

Un Mejorado Solucionador de Condiciones

Un solucionador de condiciones es un importante componente de EjeSim. Dado un dominio de aplicación y un número de condiciones a ser solucionadas, la disponibilidad de procedimientos de decisión adecuados para integrar varios procedimientos de decisión destinados a solucionar condiciones (Păsăreanu et al., 2009). Además, dada una condición de ruta p1, antes de enviar p1 a un procedimiento de decisión, se sugiere, si es posible, la aplicación de simplificaciones sobre p1 para así facilitar su solución. Así, la integración de procedimientos de decisión con la revisión e integración de simplificaciones sobre condiciones de ruta para EjeSim son importantes áreas de investigación futura. Usualmente, las aplicaciones de software requieren resolver complejas condiciones matemáticas no lineales, difíciles de ser resueltas. Así, es necesario el desarrollo de nuevas heurísticas, que una vez desarrolladas, pueden ser parte de un solucionador de condiciones de EjeSim.

En general, avances algorítmicos y computacionales son siempre importantes para la aplicación de nuevas versiones de ejecución simbólica. Por ejemplo, Păsăreanu et al. (2009) remarcó que, dados los avances algorítmicos tales como inicialización tardía de ESG, una variedad de nuevos y rápidos procedimientos de decisión y solucionador de condiciones tales como solucionadores SMT; y la existencia de dispositivos de computación más poderosos, es posible la aplicación de ejecución simbólica sobre grandes programas así como también el descubrimiento de faltas internas y fallas de ejecución en aplicaciones software comúnmente usadas. Por tanto, mejoramientos orientados a solucionar condiciones de EjeSim representa una prominente área para trabajos futuros.

Enfoques para Manejar la Explosión de Rutas

El manejo de un número exponencial de condiciones de ruta representa una significante meta de escalabilidad para la ejecución simbólica. Avances en técnicas de composición, de eliminación de rutas redundantes, y heurísticas de búsqueda son elementos necesarios para enfrentar la explosión de condiciones de ruta de la ejecución simbólica (Păsăreanu et al., 2009). Además, la solución paralela de condiciones puede mejorar aún más el rendimiento de la ejecución simbólica ya que estas condiciones pueden ser evaluadas de manera independiente. Así, la revisión de técnicas algorítmicas para enfrentar la explosión de rutas, así como la integración de paralelismo para la resolución de condiciones de ruta constituyen áreas de trabajo actual y futuro en la ejecución simbólica.

Una Integración entre Ejecución Simbólica y Chequeo de Modelos

Al revisar ESG, es posible apreciar los beneficios de la integración entre el chequeo de modelos y la ejecución simbólica porque el desarrollo de un analizador para la ejecución simbólica no se hace necesario puesto que un chequeador de modelos es encargado de explorar las rutas de ejecución en el programa analizado y de aplicar técnicas algorítmicas para obtener eficientes resultados. Además, un chequeador de

modelos permite el chequeo de programas concurrentes, y, algunos chequeadores de modelos trabajan sobre números reales. Por ejemplo, JPF permite analizar programas concurrentes en Java, y actualmente JPF utiliza el solucionador de condiciones Choco para resolver condiciones enteras o reales lineales o no lineales (JPF, 2013). De esta forma, la integración de la ejecución simbólica y chequeadores de modelos provee claros beneficios.

Para una integración completa y general de chequeadores de modelos y ejecución simbólica, se debe considerar la existencia de diferentes chequeadores de modelos con diferentes lenguajes de modelación. Entonces, para cada chequeador de modelo, sería necesario definir rutinas de instrumentación para adaptar o transformar el código fuente de programas a código equivalente en el lenguaje de modelación asociado. De esta forma, a futuro será necesario revisar las rutinas de instrumentación para así generalizarlas, de manera que la ejecución simbólica pueda ser integrada con diferentes chequeadores de modelos.

Extender la Ejecución Simbólica para Elementos de Otras Metodologías de Programación

Puesto que hay diferentes metodologías de programación como programación estructurada, orientada a objetos, orientada a aspectos, concurrente, y paralela; y dado que es una tarea necesaria verificar programas con elementos propios de estas metodologías, extender la ejecución simbólica para analizar programas en lenguajes de cada una de estas metodologías es una prominente área de trabajo futuro. Además, por lo revisado de ESG, puesto que ya existen chequeadores de modelos encargados de verificar propiedades de diferentes metodologías de programación, por el principio de separación de incumbencias, la extensión de la ejecución simbólica para otros enfoques de programación puede ser lograda con la definición de rutinas generales de instrumentación de código, y de esta forma adaptar el código de programas a ser analizados por chequeadores de modelos. Revisar esta integración general es un área de trabajo actual de los autores.

REFERENCIAS

Baier, C. y Katoen, J., Principles of Model Checking (Representation and Mind Series), MIT Press, ISBN 026202649X, USA (2008).

Belt, J., y otros autores 6 autores, Bakar, Kiasan: Flexible Contract Checking for Critical Systems Using Symbolic Execution, Actas de Third International Conference on NASA Formal Methods, NFM'11, Berlin, Heidelberg 58-72, Abril (2011).

Cadar, C., y otros 6 autores, Symbolic Execution for Software Testing in Practice: Preliminary Assessment, Actas de 3rd International Conference on Software Engineering, ICSE '11, New York, NY, USA, ACM,1066-1071, Mayo (2011).

Cadar, C. y Sen, K., Symbolic Execution for Software Testing: Three Decades Later, Comunicaciones de la ACM, 56(2), 82-90, Febrero (2013).

Dijkstra, E. W., The Humble Programmer, Comunicaciones de la ACM, 15(10), 859-866, Octubre (1972).

Ghezzi, C., Jazayeri, M. y Mandrioli, D., Fundamentals of Software Engineering, SegundaEdición, Prentice Hall PTR, Upper Saddle River, NJ, USA (2002).

Holzmann, G. J., Software Analysis and Model Checking, Actas de 14th International Conference on Computer Aided Verification, CAV '02, London, UK, Springer-Verlag, 1-16, Julio (2002).

JPF, Java Pathfinder, the swiss army knife of Java verification, http://javapathfinder.sourceforge.net/ extensions/symbc/doc/. Acceso: 22 de Agosto (2013).

Khurshid, S., Păsăreanu, C. y Visser, W., Generalized Symbolic Execution for Model Checking and Testing, Actas de 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'03, Berlin, Heidelberg, Springer-Verlag, 553-568, Abril (2003).

King, J. C., Symbolic Execution and Program Testing, Comunicaciones de la ACM, 19(7), 385-394, Julio (1976).

Păsăreanu, C. S. y Visser, W., A Survey of New Trends in Symbolic Execution for Software Testing and Analysis, International Journal on Software Tools for Technology Transfer, 11(4), 339-353, Octubre (2009).

Păsăreanu, C. S., Visser, W.,Bushnell, D., Geldenhuys, J., Mehlitz, P. y Rungta, N., Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis, Automated Software Engineering, 20(3) , Springer-Verlag, 391-425 (2013).

Sen, K., Marinov, D., y Agha, G., CUTE: a Concolic Unit Testing Engine for C, Actas de 10th European Software Engineering Conference junto con 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE, New York, NY, USA, ACM, 263-272, Septiembre (2005).

Wagner, F., Schmuki, R., Wagner, T. y Wolstenholme, P., Modeling Software with Finite State Machines: A Practical Approach, Auerbach Publications, Technical Report Santos-tr2009-09-25, Mayo (2006).

Williams, L., A (Partial) Introduction to Software Engineering: Practices and Methods, NCSU CSC326 Course Pack, Universidad del Estado de Carolina del Norte, Ciencia de la Computación, Raleigh, NC, (2011).