UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Ejecución simbólica como herramienta de análisis

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel Pardo Pont

Tutora: Alicia Villanueva García

Julio 2014

Resumen

El análisis automático de software permite verificar que un programa cumpla una serie de propiedades deseables sin intervención humana, aportando agilidad al proceso de ingeniería y reduciendo los riesgos del software crítico. No obstante, para analizar un programa mediante un computador se requiere conocer las especificaciones formales a evaluar, no siempre al alcance del desarrollador. En este trabajo se elabora la base tecnológica de un sistema para inferir especificaciones automáticamente, aplicando ejecución simbólica sobre la semántica de un lenguaje derivado de C y facilitando así el proceso de verificación.

Palabras clave: análisis, automático, ejecución simbólica, especificaciones, formal, inferencia, semántica.

Abstract

Automatic software analysis allows developers to verify if a program satisfies a series of desirable properties without human intervention, accelerating the engineering process and reducing the risks of critical software. However, in order to perform the analysis of a program, its formal specifications are required to be known, and usually it is too difficult to produce them manually. In this paper we elaborate the technological basis for a system that performs automatic inference of specifications, applying symbolic execution over the semantics of a derivative language of C so that we can make easier the verification process.

Keywords: analysis, automatic, symbolic execution, specifications, formal, inference, semantics.

Tabla de contenidos

1. Introducción **7**

1.1. Marco conceptual 7

1.2. Objetivos del trabajo 8

1.3. Organización del documento 9

2. Conceptos preliminares **11**

2.1. El lenguaje KernelC 11

2.1.1. Introducción 11

2.1.2. Sintaxis y semántica estándar 14

2.2. Ejecución simbólica 19

2.2.1. Introducción 19

2.2.2. Extensión simbólica de KernelC 21

2.2.3. Gestión de estructuras dinámicas 25

3. Desarrollo del sistema **29**

3.1. Estructura de la aplicación 29

3.2. Descripción de la implementación 30

3.3. Prueba del sistema 39

4. Conclusiones **41**

5. Referencias **43**

1. Introducción

1.1. Marco conceptual

El análisis estático de software comprende las técnicas utilizadas para predecir, en tiempo de compilación, el comportamiento que un programa presentará cuando sea ejecutado (también conocido como *comportamiento dinámico*). Para ello, se emplean aproximaciones computables y seguras, es decir, que garanticen que el comportamiento que representan sea correcto aunque pueda no ser exacto. Conocer bien estas técnicas de análisis es fundamental en el campo de la ingeniería de software, pues permiten automatizar el proceso de verificación y hacer posible que los desarrolladores comprueben si el resultado obtenido satisface sus expectativas en muy poco tiempo.

Dichas expectativas se pueden definir en términos de una especificación. Las especificaciones establecen el comportamiento esperado de un programa informático, bien de manera semi-formal para comprensión humana, bien mediante lenguajes formales para ser interpretadas por un computador. Así pues, podemos decir que las especificaciones constituyen la base del análisis estático de programas, ya que al aplicar cualquier análisis realmente estamos comprobando si la aproximación al comportamiento real del programa se corresponde con aquél que define la especificación.

No obstante, las especificaciones formales son en su mayoría muy complejas para ser escritas y revisadas manualmente. Esto causa que, sin disponer de entrenamiento previo, los desarrolladores puedan no ser capaces de formular especificaciones adecuadas para su uso como entrada de herramientas de análisis de software. Incluso es posible que la documentación de los programas quede incompleta o no sea apropiada. La inferencia automática de especificaciones puede ayudar a mitigar estos problemas reduciendo el esfuerzo necesario para definir especificaciones de programas, y también resulta útil para obtener información cuando no disponemos del código fuente: a partir de los ejecutables, de software legado, para entender el funcionamiento de código malicioso, etc.

La inferencia automática de especificaciones computa, sin intervención del desarrollador, especificaciones aproximadas de un programa a partir usualmente del código fuente del mismo. Esta técnica tiene como objetivo principal entender el funcionamiento del programa y verificar que cumpla ciertas propiedades, y las especificaciones resultantes de su aplicación suelen estar destinadas a documentar el código o automatizar el proceso de pruebas.

Existen diversos métodos para analizar código fuente e inferir el comportamiento que tendrá en ejecución, pero entre todos ellos, la ejecución simbólica es uno de los más conocidos y efectivos [1]. Consiste en simular la ejecución del programa empleando valores simbólicos en lugar de datos concretos como información de entrada; a saber, todo campo o variable almacenará un valor simbólico al inicio de

la ejecución. Así pues, dado que el controlador de ejecución simbólica desconoce cuál de los caminos ejecutables del programa se seguirá, atendiendo a las dependencias de control (condicionales, etc.), se exploran todos ellos gestionando la información de configuración de manera independiente en cada uno: estado actual de las variables, del *heap*, condiciones cumplidas… Al terminar, se obtiene una lista de las distintas posibilidades de ejecución junto con las propiedades que han de satisfacer los datos de entrada para que se ejecute cada camino.

1.2. Objetivos del trabajo

La propuesta de este trabajo consiste en la implementación de un motor que, usado como base tecnológica, permita llevar a cabo inferencia de especificaciones. En concreto, permitirá la obtención automática de especificaciones formales vía ejecución simbólica de programas escritos en KernelC [2], que comprende un subconjunto de la semántica del lenguaje C. El uso de KernelC en lugar del propio lenguaje C se justifica porque el primero presenta una definición formal basada en reglas de lógica de reescritura, más sencilla de procesar por un programa informático.

El algoritmo básico que se aplicará para inferir las especificaciones de un procedimiento o función de entrada será el siguiente: en primer lugar, se leerá y procesará sintácticamente el código fuente completo del programa al que pertenece el procedimiento, para englobarlo en su contexto. Seguidamente, nuestro motor ejecutará simbólicamente el procedimiento y obtendrá los distintos caminos de ejecución, junto con las condiciones que se deben cumplir en cada uno de ellos. Esta lista de ramas se utilizará para, finalmente, inferir las especificaciones de la función empleando de nuevo el motor implementado. De esta manera, las especificaciones del programa serán más sencillas de entender por el usuario, dado que estarán expresadas en términos de la estructura de las funciones del programa original, la cual tiene sentido asumir que conoce (bien porque se trate del programador o porque forme parte del mismo equipo de desarrollo).

Más específicamente, podemos establecer para este trabajo los siguientes objetivos:

- Desarrollar un motor de ejecución simbólica destinado a ser la fase inicial de un sistema de inferencia de especificaciones.

- Establecer una interacción entre el sistema desarrollado y software externo para integrar tecnologías avanzadas y simplificar tareas como procesamiento sintáctico de código y resolución de restricciones.

- Integrar y poner en práctica los conocimientos teóricos adquiridos durante el grado. En concreto, aplicar los estudios referentes a análisis y validación

de programas, partiendo de la base de los métodos formales empleados en ingeniería de software.

1.3. Organización del documento

En los capítulos siguientes de este trabajo, abordaremos el siguiente contenido: en la sección 2 se describirán las técnicas y lenguajes que se han utilizado en el desarrollo, así como los fundamentos del método de ejecución simbólica para comprender mejor la estructura que presentará el motor implementado. Posteriormente, en la sección 3 comentaremos el proceso seguido para elaborar el sistema, y lo ilustraremos mostrando un ejemplo de ejecución del mismo. Finalmente, en la sección 4 expondremos las conclusiones obtenidas del trabajo realizado.

2. Conceptos preliminares

Antes de empezar a desarrollar el motor para inferencia automática de especificaciones, debemos establecer la base teórica que nos permitirá construir una solución efectiva al problema planteado. Esta base teórica la constituyen la plataforma sobre la cual trabajará el sistema, el lenguaje KernelC, y el método que implementará el motor: la ejecución simbólica.

2.1. El lenguaje KernelC

2.1.1. Introducción

Desde sus orígenes a principios de los setenta, el lenguaje de programación C ha sido uno de los más utilizados dentro del paradigma imperativo. Se produce constantemente una gran variedad de aplicaciones usando este lenguaje: procesadores de texto, bases de datos, videojuegos sencillos… e incluso sistemas operativos; de hecho, la mayor parte del sistema operativo UNIX está desarrollada en C. Esto supone un indicador importante de la potencia que posee [3, p. 40].

C presenta la mayoría de características propias de los lenguajes imperativos de alto nivel: portabilidad, creación de funciones, estructuras de control de flujo, soporte a importación de código en ficheros externos… No obstante, su mayor particularidad radica en que, además, proporciona facilidades para acceder a bajo nivel a la información en memoria. Esta dualidad dota a los programas escritos en C de alto rendimiento, puesto que permite aprovechar los recursos del computador de forma óptima; pero, para ello, se requiere hacer uso de operaciones como aritmética de punteros, *casting*, operaciones bit a bit, reserva y liberación de memoria explícitas, etc. [2], de modo que la responsabilidad de mantener la información en un estado estable recae sobre el desarrollador, complicando la programación y el posterior análisis. Pese a ello, suele prevalecer la eficiencia sobre la denominada *memory safety*, o protección contra violaciones de acceso a memoria, y por este motivo muchas aplicaciones donde las prestaciones y la optimización de recursos son críticas (videojuegos, por ejemplo) se siguen implementando en C o lenguajes derivados, como C# y C++[1].

Sin embargo, C es un lenguaje muy complejo para ser contemplado en su totalidad por un motor de ejecución simbólica estándar. La variedad de tipos de datos que es capaz de manejar y las librerías de funciones habitualmente utilizadas en los

---

[1] Recomendamos consultar [3] para obtener información más detallada sobre C++, y el siguiente sitio web para ampliar C#:

Microsoft Developer Network, «Guía de programación de C#,» [En línea]. Available: http://msdn.microsoft.com/es-es/library/67ef8sbd.aspx. [Último acceso: Junio 2014].

programas escritos en C, entre otros rasgos, producen que una aplicación que infiera especificaciones a partir de la semántica completa se torne inviable a nivel de coste de producción y ejecución, e incluso impredecible, dado que en muchos casos no se dispone de acceso al código de las librerías. Por ello, en su lugar, hemos optado por emplear un subconjunto denominado KernelC.

De acuerdo con [1], KernelC se define como un fragmento no trivial de C que incluye funciones, punteros, estructuras y primitivas de Entrada/Salida (E/S). En [2] se añade a esta definición también rutinas de reserva y liberación de memoria. La cantidad de tipos de datos respecto a C queda reducida a sólo uno: números enteros (*int* según la sintaxis de C), de modo que cada variable de programa y posición de memoria en KernelC podrá almacenar exactamente un entero. A su vez, la semántica formal de KernelC está constituida en términos de lógica de reescritura y, en concreto, bajo el *framework* K; de modo que aplicar dicha semántica sobre el código fuente de un programa dado se convierte en un proceso relativamente simple, sin ambigüedades.

Para comprender los fundamentos de la semántica de KernelC, a continuación introduciremos brevemente la lógica de reescritura. Una semántica, especificación, o de modo genérico: *teoría*, escrita en términos de lógica de reescritura, puede entenderse como un conjunto de ecuaciones y reglas en la forma $l \rightarrow r$, que definen la evolución de un sistema. Estas reglas se ejecutan siguiendo el principio *match-and-apply*, similar al método de sustitución en sistemas de ecuaciones lineales: cuando se obtiene un término t equivalente a la parte izquierda de una ecuación o regla, l, mediante una asignación θ de valores a sus variables, se sustituye dicho término por la parte derecha r de la ecuación o regla, manteniendo la asignación anterior [4]. Formalmente:

$$\left(l \rightarrow r \;\&\&\; t = \theta(l)\right) \rightarrow t ::= \theta(r)$$

Cabe destacar que esta fórmula, por simplicidad, sólo contempla el caso en que la sustitución se aplica a t en su totalidad, pero también puede aplicarse únicamente a un sub-término que forme parte de él. A modo de ejemplo, supongamos que tenemos el siguiente conjunto de reglas como definición de un sistema:

(1) $s(N) \rightarrow N{+}1$,
(2) $0 + N \rightarrow N$,
(3) $s(M) + N \rightarrow s(M{+}N)$,
(4) M, N, $s(M)$ y $s(N)$ son números naturales.

Si el estado inicial del sistema fuera la expresión $s(0) + s(s(0) + s(s(0)))$, una posible secuencia de reescrituras sería la siguiente:

- *Matching* con la regla 3, donde $M = s(0)$, $N = s(s(0) + s(s(0)))$. El término resultante de la sustitución sería $s(0 + s(s(0) + s(s(0))))$.

- *Matching* con la regla 1, donde $N = 0 + s(s(0) + s(s(0)))$. El término resultante de la sustitución sería $(0 + s(s(0) + s(s(0)))) + 1$.

- *Matching* con la regla 2, donde $N = s(s(0) + s(s(0)))$. El término resultante de la sustitución sería $(s(s(0) + s(s(0)))) + 1$. Nótese que en este caso ha coincidido con la parte izquierda de la regla sólo un fragmento del término actual; el sub-término no coincidente (+1) permanece invariable.

- *Matching* con la regla 1, donde $N = s(0) + s(s(0))$. El término resultante de la sustitución sería $(s(0) + s(s(0)) + 1) + 1$. Ídem a la sustitución anterior.

- *Matching* con la regla 3, donde $M = s(0)$, $N = s(s(0))$. El término resultante de la sustitución sería $(s(0 + s(s(0))) + 1) + 1$.

- *Matching* con la regla 2, donde $N = s(s(0))$. El término resultante de la sustitución sería $(s(s(s(0))) + 1) + 1$.

- *Matching* con la regla 1, donde $N = s(s(0))$. El término resultante de la sustitución sería $((s(s(0)) + 1) + 1) + 1$.

- *Matching* con la regla 1, donde $N = s(0)$. El término resultante de la sustitución sería $(((s(0) +1) + 1) + 1) + 1$.

- *Matching* con la regla 1, donde $N = s(0)$. El término resultante de la sustitución sería $((((0+1)+ 1) + 1) + 1) + 1$.

- Finalmente, *matching* con la regla 2, donde $N = 1$. El término resultante de la sustitución sería $((((1)+ 1) + 1) + 1) + 1$.

En este punto, habríamos obtenido una forma canónica de la expresión, dado que no puede aplicarse ninguna regla de la especificación del sistema. Normalmente, los motores de aplicación de lógica de reescritura (siendo el lenguaje Maude[2] uno de los más conocidos y utilizados) contienen también mecanismos de reducción de expresiones, por lo que el término de nuestro ejemplo se habría simplificado a 5.

La semántica de KernelC está constituida en base a este paradigma lógico, pero siguiendo las reglas y la estructura del *framework* K. Se trata de un marco de definición de semánticas ejecutables concebido para herramientas de análisis de software y lenguajes de programación, especialmente aquellos con soporte a concurrencia. Para definir un lenguaje, K establece un conjunto de información que se añade a las reglas de reescritura de la semántica, denominado *configuración*. La configuración representa el estado actual de la ejecución de un programa, dividido en subconjuntos que almacenan cada uno la información de una estructura interna: las instrucciones que aún quedan por ejecutar, los identificadores y valores de las

[2] Más ejemplos de lógica de reescritura escritos en Maude en el siguiente sitio web:

Department of Computer Science, University of Illinois, «Maude Tutorial and Tutorial Examples,» Marzo 2000. [En línea]. Available: http://maude.cs.uiuc.edu/maude1/tutorial/. [Último acceso: Junio 2014].

variables, el estado de la memoria dinámica, etc. Estos subconjuntos reciben la denominación de *celdas* [5].

2.1.2. Sintaxis y semántica estándar

Una vez disponemos de la base teórica sobre la que se sostiene KernelC, pasaremos a exponer la sintaxis abstracta que emplea y la semántica asociada a cada una de sus construcciones. Como comentamos anteriormente, KernelC es una simplificación del lenguaje C con un único tipo de dato: los números enteros. A este tipo de información primitiva se le añade soporte a números naturales, para representar las direcciones de memoria interna, y a cadenas de caracteres que almacenen los identificadores de las variables. La sintaxis contiene, además, operadores aritméticos y relacionales, construcciones básicas para control de flujo (condicionales *if-else* y bucles *while*), secuencias y bloques agrupadores de instrucciones. El acceso a memoria a bajo nivel se implementa mediante punteros y operaciones de reserva/liberación. Asimismo, el lenguaje acepta expresiones lógicas (*AND*, *OR* y *NOT*), condicionales unarios *if* y el valor *null*, aunque como operadores derivados, definidos en términos de otros operadores básicos. A continuación se presenta un resumen de la sintaxis completa del lenguaje KernelC, siguiendo la notación estándar *Backus-Naur Form* (*BNF*)[3] y obtenido a partir de [2].

Nat ::= N
Int ::= Z
Id ::= identificadores (cadenas de caracteres)
K ::= *Nat* | *Int* | *Id*
| !*K* | *K* && *K* | *K* || *K*
| *K* op *K*, op ϵ {+, -, *, /, <, <=, >, >=, ==, !=}
| *K* = *K*
| *K*; *K*
| {*K*} | {}
| *null*
| *malloc*(*K*)
| *free*(*K*)
| **K*
| *if* (*K*) *K* | *if* (*K*) *K else K*
| *while* (*K*) *K*

A su vez, los operadores derivados se interpretan de la siguiente manera:

[3] La descripción de las reglas de esta notación se puede consultar en el siguiente sitio web:

World Wide Web Consortium, «BNF Notation for syntax,» [En línea]. Available: http://www.w3.org/Notation.html. [Último acceso: Junio 2014].

null = 0
!*K* = *if* (*K*) 0 *else* 1
K1 && *K2* = *if* (*K1*) *K2 else* 0
K1 || *K2* = *if* (*K1*) 1 *else* K2
if (*K1*) *K2* = *if* (*K1*) *K2 else* {}

La configuración de un programa en KernelC, la cual etiquetaremos como *Cfg*, está formada por cuatro celdas: en primer lugar, el conjunto de instrucciones que quedan pendientes por ejecutar, al que hemos denominado *Cont* (*Continuation*). En segundo lugar, una lista con los identificadores de las variables del programa y su valor actual asociado, de nombre *Env* (*Environment*). Por último, dos conjuntos destinados a representar el estado de la memoria interna dedicada al programa en ejecución: *Mem* (*Memory*), que contiene las direcciones de memoria en uso y los valores asociados a ellas, y *Ptr* (*Pointers*), encargado de mantener la cantidad de posiciones de memoria consecutivas que se han asociado al puntero inicial correspondiente a cada bloque reservado mediante la operación *malloc*. Esto es, si se reservan las direcciones de memoria *P* y *P*+1, se añadirá una entrada a la celda *Ptr* tal que $\{P \rightarrow 2\}$.

Como ya hemos señalado anteriormente, la semántica original de KernelC está definida en forma de un sistema de reglas de lógica de reescritura [2]. No obstante, para la implementación del motor de ejecución simbólica seguiremos una simplificación de dicho sistema, especificada bajo la notación estándar de semánticas operacionales estructurales (*SOS*) [6]. En una semántica operacional, la ejecución de un programa se estructura siguiendo un modelo matemático que representa las computaciones efectuadas por un procesador. Sobre el papel, normalmente este modelo matemático toma la forma de un árbol, donde cada nodo simboliza la evaluación (computación individual de una construcción aceptada por la sintaxis) de una expresión completa, y sus nodos hijos, las evaluaciones de las sub-expresiones que la componen [7]. Por ejemplo, la evaluación de una expresión como (2+3) * (3-4) generaría un árbol de 3 niveles de profundidad:

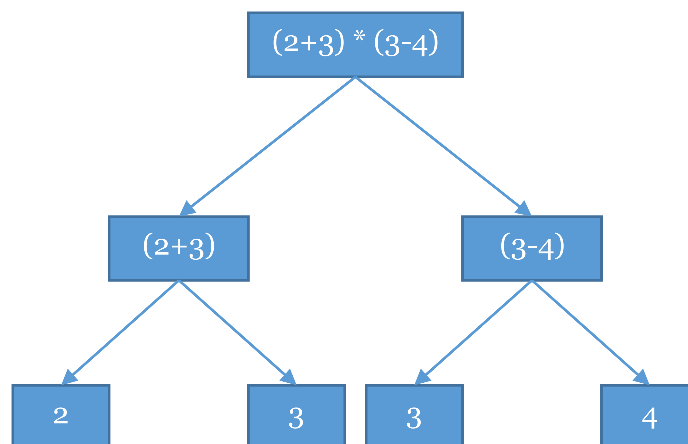

Figura 1. Ejemplo de evaluación de una expresión según la notación SOS

Así pues, la relación de evaluación de una expresión *K*, perteneciente a la sintaxis de KernelC, se define como sigue:

$$<K, Cfg> \rightarrow V,$$

donde *V* es un valor concreto que se obtiene como resultado de la evaluación. El efecto de esta relación sería el equivalente a la aplicación de una regla de reescritura, dado que una instancia de la parte izquierda $<K, Cfg>$ se sustituiría por la instancia correspondiente de la parte derecha *V*. Nótese, además, que las evaluaciones de las expresiones se llevan a cabo siempre bajo el contexto de una configuración de programa, incluso en aquellos casos en que *K* no contiene variables. Esto tiene sentido ya que nunca se va a computar una expresión de KernelC si no es dentro del proceso de la ejecución de un programa.

Llegados a este punto, nos encontramos en condiciones de especificar formalmente las reglas semánticas para las expresiones aritmético-lógicas incluidas en la sintaxis de KernelC. La evaluación de expresiones compuestas se muestra en forma de fracción; en la parte superior de la cual figurarán los cómputos intermedios necesarios para efectuar un paso de ejecución en la parte inferior. Podemos establecer una correspondencia con el ejemplo gráfico de la Figura 1: las hojas del árbol aparecerán en el nivel superior de la regla; debajo de ellas, los super-nodos que las generan, y así sucesivamente hasta alcanzar la raíz del árbol en el nivel inferior. Bajo esta notación, se definen las siguientes reglas:

- $K = Nat$, $K = Int$

$$<K, Cfg> \rightarrow N, \text{ donde } N \text{ es el valor numérico de } K$$

- $K = K1 \text{ op } K2$, donde op $\in \{+, -, *, /\}$

$$\frac{<K1, Cfg> \rightarrow N1\ ,\ <K2, Cfg> \rightarrow N2}{<K1 \text{ op } K2, Cfg> \rightarrow N}, \text{ donde } N = N1 \text{ op } N2$$

- $K = K1 \text{ op } K2$, donde op $\in \{<, <=, >, >=, ==, !=\}$

$$\frac{<K1, Cfg> \rightarrow N1\ ,\ <K2, Cfg> \rightarrow N2}{<K1 \text{ op } K2, Cfg> \rightarrow 0}, \text{ si } !(N1 \text{ op } N2)$$

$$\frac{<K1, Cfg> \rightarrow N1\ ,\ <K2, Cfg> \rightarrow N2}{<K1 \text{ op } K2, Cfg> \rightarrow N}, \text{ si } N1 \text{ op } N2, \text{ donde } N \mathrel{!=} 0$$

Las evaluaciones pueden, además, producir un cambio en la configuración como resultado de ejecutar una expresión. A este respecto, un ejemplo típico es la asignación de valores a variables; no se obtiene ningún valor numérico como

consecuencia de su evaluación, pero el elemento correspondiente del conjunto *Env* se ve alterado. En general, la ejecución de cualquier instrucción del lenguaje implicará una modificación de una o varias celdas de la configuración del programa. Por consiguiente, podemos ver la ejecución de un programa como una evaluación total donde *K* representa la secuencia de instrucciones que lo componen y *V* el valor de retorno del programa, si lo hubiere; la configuración asociada a la parte izquierda de la evaluación simbolizaría el estado inicial del programa, y su homóloga en la parte derecha, el estado final.

Para concluir, pasaremos a completar la definición de la semántica de KernelC añadiendo las reglas referentes a la ejecución de las instrucciones aceptadas por la sintaxis. El símbolo "□" que aparece en algunas de ellas representa el fin de programa y no tiene ninguna regla asociada, por lo que finalizaría la ejecución bajo la semántica operacional al alcanzar tal estado. También emplearemos el símbolo "•" para indicar que el valor de un elemento está indefinido o se desconoce, bien sea una variable o una posición de memoria.

- Acceso a una variable *K*, donde *K* = *Id*

  $<K, Env, Mem, Ptr> \rightarrow Env(K)$, donde *Env*(*K*) representa el valor numérico almacenado en la variable *K* cuando la ejecución se encuentra en el estado *Env*.

- Acceso a un puntero **P*

  $<*P, Env, Mem, Ptr> \rightarrow Mem(P)$, donde *Mem*(*P*) representa el valor numérico almacenado en la dirección de memoria a la que referencia el puntero *P* cuando la ejecución se encuentra en el estado *Mem*.

- Asignación *K1* = *K2*. La asignación de un valor a una variable actualiza el conjunto *Env* eliminando el par {*K1* → •}, si existe, e insertando el nuevo valor resultante de evaluar *K2*.[4]

$$\frac{<K2, Env, Mem, Ptr> \rightarrow N2}{<K1 = K2, Env, Mem, Ptr> \rightarrow <\square, (Env \setminus \{K1 \rightarrow \bullet\}) \cup \{K1 \rightarrow N2\}, Mem, Ptr>}$$

- Bloques de instrucciones {*K*}, {}

$$\frac{<K, Env, Mem, Ptr> \rightarrow <K', Env', Mem', Ptr'>}{<\{K\}, Env, Mem, Ptr> \rightarrow <K', Env', Mem', Ptr'>}$$

$$<\{\}, Env, Mem, Ptr> \rightarrow <\square, Env, Mem, Ptr>$$

[4] Los símbolos "\" y "U" representan las operaciones de diferencia simétrica y unión de conjuntos, respectivamente.

- Secuencia *K1*; *K2*. Para esta expresión debemos distinguir dos casos: cuando la ejecución de *K1* termina, comienza a ejecutarse *K2*; pero si la ejecución de *K1* genera una nueva expresión *K1'*, como es el caso de los condicionales y bucles, entonces debe ejecutarse *K1'* antes de *K2*.

$$\frac{<K1, Env, Mem, Ptr> \to <\square, Env', Mem', Ptr'>}{<K1;K2, Env, Mem, Ptr> \to <K2, Env', Mem', Ptr'>}$$

$$\frac{<K1, Env, Mem, Ptr> \to <K1', Env', Mem', Ptr'>}{<K1;K2, Env, Mem, Ptr> \to <K1';K2, Env', Mem', Ptr'>}$$

- Reserva de memoria dinámica *malloc*(*K*). Se añade al conjunto *Mem* un puntero con valor indefinido para cada bloque de memoria a reservar, y a *Ptr* una asociación entre la dirección de memoria inicial y la cantidad de celdas contiguas que se han reservado.

$$\frac{<K, Env, Mem, Ptr> \to N}{<malloc(K), Env, Mem, Ptr> \to <\square, Env, Mem \cup \{P \to \bullet,\ P+1 \to \bullet, \ldots, P+N-1 \to \bullet\}, Ptr \cup \{P \to N\}>}$$

No se considera el caso en que *malloc* falla, es decir, se asume capacidad de memoria infinita.

- Liberación de memoria dinámica *free*(*P*). Se trata de la operación contraria a *malloc*, por tanto, se eliminan los pares referentes a *P* en *Mem* y *Ptr*.

$$<free(P), Env, Mem, Ptr> \to <\square, Env, Mem \setminus \{P \to \bullet,\ P+1 \to \bullet, \ldots, P+N-1 \to \bullet\}, Ptr \setminus \{P \to N\}>$$

- Asignación a un puntero **P* = *K*. Este caso es similar a la asignación a variables, salvo que en esta regla se actualiza el conjunto *Mem*, permaneciendo *Env* inalterado.

$$\frac{<K, Env, Mem, Ptr> \to N}{<*P = K, Env, Mem, Ptr> \to <\square, Env, (Mem \setminus \{P \to \bullet\}) \cup \{P \to N\}, Ptr>}$$

- Condicional *if* (*E*) *K1 else K2*. Se consideran dos posibilidades: la primera, si la condición *E* se evalúa a un valor distinto de 0, el cual KernelC y, en general, la familia de lenguajes C interpretan como un valor de verdad cierto. En este caso, se avanza un paso de ejecución y se dispone *K1* para ser evaluada. Por otro lado, si *E* se evalúa a 0 (valor de verdad falso), en el siguiente paso de ejecución se evaluará *K2*.

$$\frac{<E, Env, Mem, Ptr> \rightarrow I}{<if\ (E)\ K1\ else\ K2, Env, Mem, Ptr> \rightarrow <K1, Env, Mem, Ptr>},$$

si I != 0

$$\frac{<E, Env, Mem, Ptr> \rightarrow I}{<if\ (E)\ K1\ else\ K2, Env, Mem, Ptr> \rightarrow <K2, Env, Mem, Ptr>},$$

si I == 0

- Bucle *while* (*E*) *K*. Este caso es similar al condicional, con la salvedad de que un valor de verdad falso en la condición no computará nada, alcanzándose el fin del subprograma; mientras que un valor cierto ocasionará la ejecución de *K* y, a continuación, la reevaluación del bucle.

$$\frac{<E, Env, Mem, Ptr> \rightarrow I}{<while(E)\ K, Env, Mem, Ptr> \rightarrow <K; while(E)\ K, Env, Mem, Ptr>},$$

si I != 0

$$\frac{<E, Env, Mem, Ptr> \rightarrow I}{<while(E)\ K, Env, Mem, Ptr> \rightarrow <\square, Env, Mem, Ptr>},$$

si I == 0

2.2. Ejecución simbólica

2.2.1. Introducción

Dado un lenguaje de programación, definido por una sintaxis y una semántica, la ejecución estándar o concreta de un programa escrito en dicho lenguaje consiste en aplicar las reglas de la semántica hasta alcanzar un estado final, a partir de un estado inicial determinado. Por ejemplo, supongamos que disponemos del siguiente programa en KernelC, al que denominaremos *P*:

$$if(X < 3)\ Y = 2\ else\ Y = 6;$$

$$X = 5;$$

En el lenguaje C las variables no se inicializan a ningún valor por defecto, a diferencia de otros lenguajes de programación; sin embargo, para simplificar el

ejemplo, asumiremos que las variables X e Y contienen ambas el valor 0 cuando el flujo del programa alcanza la instrucción condicional. De este modo, el árbol resultante de la ejecución de P, aplicando la semántica operacional que hemos definido para KernelC (véase 2.1.2), sería el siguiente:

- Primer paso de ejecución:

$$\cfrac{\cfrac{\cfrac{<X, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow 0 \;,\; <3, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow 3}{<X < 3, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow 1}}{<\mathit{if}\,(X < 3)\; Y = 2\; \mathit{else}\; Y = 6, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow <Y = 2, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}>}}{<\mathit{if}\,(X < 3)\; Y = 2\; \mathit{else}\; Y = 6; X = 5, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow <Y = 2; X = 5, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}>}$$

- Segundo paso de ejecución:

$$\cfrac{\cfrac{<2, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow 2}{<Y = 2, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow <\cdot, \{X \rightarrow 0, Y \rightarrow 2\}, \{\}, \{\}>}}{<Y = 2; X = 5, \{X \rightarrow 0, Y \rightarrow 0\}, \{\}, \{\}> \rightarrow <X = 5, \{X \rightarrow 0, Y \rightarrow 2\}, \{\}, \{\}>}$$

- Tercer y último paso de ejecución:

$$\cfrac{<5, \{X \rightarrow 0, Y \rightarrow 2\}, \{\}, \{\}> \rightarrow 5}{<X = 5, \{X \rightarrow 0, Y \rightarrow 2\}, \{\}, \{\}> \rightarrow <\cdot, \{X \rightarrow 5, Y \rightarrow 2\}, \{\}, \{\}>}$$

La ejecución estándar de P obtiene en tres pasos de ejecución un único estado final, en el cual $\{X = 5, Y = 2\}$. No obstante, este método resulta insuficiente para llevarlo a cabo en un análisis automático; al terminar una ejecución real del programa, Y podría tener como valor 2 o 6 en función del valor de entrada de X, pero el análisis sólo consideraría un caso. Si nuestra intención es obtener automáticamente una descripción de la estructura del programa en forma de especificaciones, necesitaremos basarnos en un procedimiento capaz de obtener información de todos los caminos de ejecución posibles y que sea independiente de los parámetros de entrada. La ejecución simbólica cumple tales requisitos.

En el paradigma de ejecución simbólica, no se asume ningún valor inicial para las variables del entorno; en su lugar, se asigna un valor simbólico a cada una de ellas. Acorde con el estándar, el valor simbólico asociado a la variable X se mostrará bajo la forma $?X$. Los valores simbólicos son expresiones de contenido indeterminado, es decir, no pueden identificarse con un valor concreto (salvo que las restricciones del programa indiquen lo contrario). Por este motivo, las dependencias de control se resuelven explorando todos los caminos posibles y acumulando las condiciones que se satisfacen en cada uno de ellos. Al término de la ejecución, se obtiene una lista con todos los estados finales posibles y los correspondientes conjuntos de restricciones sobre los datos de entrada que permiten alcanzarlos. Estos grupos de propiedades asociados a cada camino ejecutable reciben el nombre de *path conditions*.

2.2.2. Extensión simbólica de KernelC

Partiendo de la definición y características del método de ejecución simbólica, debemos extender la configuración de programa previamente definida para KernelC (véase 2.1.2) añadiendo una quinta celda que almacene la *path condition* actual. Esta configuración simbólica la denotaremos como Cfg_s. Por añadidura, también será necesario ampliar la sintaxis de KernelC para que, de forma sistemática, se acepten expresiones simbólicas en cada construcción del lenguaje. Así pues, la sintaxis completa que deberemos considerar para el análisis de programas en KernelC es la siguiente:

Nat ::= N
Int ::= Z
Id ::= identificadores (cadenas de caracteres)
SNat ::= ?Id_N
SInt ::= ?Id_Z
SExp ::= *Nat* | *Int* | *Id* | *SNat* | *SInt*
  | !*SExp* | *SExp* && *SExp* | *SExp* || *SExp*
  | *SExp* op *SExp*, op ϵ {+, -, *, /, <, <=, >, >=, ==, !=}
K ::= *SExp*
  | *Id* = *SExp*
  | *K*; *K*
  | {*K*} | {}
  | *null*
  | *malloc*(*SExp*)
  | *free*(*Id*)
  | **Id* = *SExp*
  | *if* (*SExp*) *K* | *if* (*SExp*) *K else K*
  | *while* (*SExp*) *K*

Obsérvese que se ha restringido la sintaxis de las asignaciones a variables y punteros, de *malloc* y *free* y de las condiciones en *if* y *while*. De esta manera se evitan incoherencias como, por ejemplo, intentar asignar una expresión condicional a una variable o que una condición esté compuesta por una secuencia de expresiones.

Con la inclusión de las expresiones simbólicas, debemos redefinir las reglas de la semántica relativas al control de flujo y a la evaluación de condiciones, puesto que ya no se puede establecer en todos los casos un valor de verdad como resultado de la ejecución de expresiones lógicas. Así pues, la evaluación de bucles y condicionales producirá una bifurcación del camino de ejecución actual, de manera que se examinen tanto la situación en que la condición es cierta como el caso en que es falsa. Para ello, deberá agregarse a la *path condition* una nuevo término que indique bajo qué circunstancias el flujo de ejecución alcanzaría el camino actual. Este término no es más que el resultado de evaluar la condición, de modo que las variables que aparecen en ella quedan sustituidas por sus valores concretos o simbólicos correspondientes. La semántica actualizada queda como sigue:

- *K1* op *K2*, donde op ∈ {<, <=, >, >=, ==, !=}

$$\frac{\langle K1, Cfg_s\rangle \rightarrow N1\ ,\ \langle K2, Cfg_s\rangle \rightarrow N2}{\langle K1 \text{ op } K2, Cfg_s\rangle \rightarrow 0}, \text{ si } !(N1 \text{ op } N2)$$

$$\frac{\langle K1, Cfg_s\rangle \rightarrow N1\ ,\ \langle K2, Cfg_s\rangle \rightarrow N2}{\langle K1 \text{ op } K2, Cfg_s\rangle \rightarrow N}, \text{ si } N1 \text{ op } N2, \text{ donde } N \mathrel{!=} 0$$

$$\frac{\langle K1, Cfg_s\rangle \rightarrow N1\ ,\ \langle K2, Cfg_s\rangle \rightarrow N2}{\langle K1 \text{ op } K2, Cfg_s\rangle \rightarrow N1 \text{ op } N2}, \text{ si } N1 \text{ op } N2 \text{ está indefinido}$$

- *if* (*E*) *K1 else K2*

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle if\ (E)\ K1\ else\ K2, Env, Mem, Ptr, PC\rangle \rightarrow \langle K1, Env, Mem, Ptr, PC\rangle}$$

si *PC* implica *I*

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle if\ (E)\ K1\ else\ K2, Env, Mem, Ptr, PC\rangle \rightarrow \langle K2, Env, Mem, Ptr, PC\rangle}$$

si *PC* implica !*I*

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle if\ (E)\ K1\ else\ K2, Env, Mem, Ptr, PC\rangle \rightarrow \langle K1, Env, Mem, Ptr, PC\ \wedge\ I \mathrel{!=} 0\rangle}$$

y, en un camino independiente,

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle if\ (E)\ K1\ else\ K2, Env, Mem, Ptr, PC\rangle \rightarrow \langle K2, Env, Mem, Ptr, PC\ \wedge\ I == 0\rangle}$$

si *PC* no implica *I* ni !*I*

- Bucle *while* (*E*) *K*

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle while(E)\ K, Env, Mem, Ptr, PC\rangle \rightarrow \langle K;while(E)\ K, Env, Mem, Ptr, PC\rangle}$$

si *PC* implica *I*

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \rightarrow I}{\langle while(E)\ K, Env, Mem, Ptr, PC\rangle \rightarrow \langle \square, Env, Mem, Ptr, PC\rangle}$$

si PC implica $!I$

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \to I}{\langle while(E)\ K, Env, Mem, Ptr, PC\rangle \to \langle K; while(E)\ K, Env, Mem, Ptr, PC \wedge I\ != 0\rangle}$$

y, en un camino independiente,

$$\frac{\langle E, Env, Mem, Ptr, PC\rangle \to I}{\langle while(E)\ K, Env, Mem, Ptr, PC\rangle \to \langle \square, Env, Mem, Ptr, PC \wedge I == 0\rangle}$$

si PC no implica I ni $!I$

Obsérvese que se contemplan dos casos particulares cuando, a partir de las restricciones de la *path condition*, se puede deducir si la condición es cierta o falsa. Esto implicaría que, independientemente de los valores dados a los parámetros de entrada al programa, siempre se ejecutaría el fragmento de código asociado a la cláusula *if* o a la cláusula *else*, o bien un número invariable de iteraciones de un bucle, que podría ser cero. Por esta razón, un análisis mediante ejecución simbólica sería también capaz de detectar código muerto en un programa; código que nunca llegará a ejecutarse.

Como hemos indicado a lo largo de esta sección, la ejecución simbólica de un programa o procedimiento explora todos los posibles caminos ejecutables del mismo, obteniendo un estado final para cada uno de ellos (asumimos que todas las ejecuciones terminarán y adoptaremos un sistema para evitar bucles infinitos en el análisis). Por consiguiente, una traza de ejecución simbólica se puede visualizar como un árbol, donde el nodo raíz representa el estado inicial y los nodos hoja simbolizan los posibles estados finales. Cada rama que llega a un nodo hoja desde la raíz se corresponde con un camino de ejecución; por ello se suelen emplear indistintamente los términos *rama* y *camino* para hacer referencia a los distintos recorridos que puede tomar el flujo de ejecución de un programa.

A continuación ilustraremos el proceso de ejecución simbólica empleando como ejemplo el programa P definido en 2.2.1. La *path condition* en el nodo raíz se inicializa a *true*, para poder concatenar mediante la operación lógica *AND* las condiciones que se van asumiendo satisfechas. Por su parte, las variables del conjunto *Env*, $\{X, Y\}$, se inicializan a valores simbólicos diferenciados tal que $\{X \to ?X, Y \to ?Y\}$. La primera bifurcación se produce al efectuar el primer paso de ejecución, ya que la instrucción inicial es de tipo *if-else*. En la rama donde consideramos que la condición es cierta, se produce la siguiente secuencia de evaluaciones:

- Primer paso de ejecución:

$$\cfrac{\cfrac{\cfrac{\langle X, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, true\rangle \to ?X\ ,\ \langle 3, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, true\rangle \to 3}{\langle X < 3, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, true\rangle \to ?X < 3}}{\langle if\ (X < 3)\ Y = 2\ else\ Y = 6, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, true\rangle \to \langle Y = 2, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, ?X < 3\rangle}}{\langle if\ (X < 3)\ Y = 2\ else\ Y = 6; X = 5, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, true\rangle \to \langle Y = 2; X = 5, \{X \to ?X, Y \to ?Y\}, \{\}, \{\}, ?X < 3\rangle}$$

- Segundo paso de ejecución:

$$\dfrac{\dfrac{\langle 2, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X < 3\rangle \rightarrow 2}{\langle Y = 2, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X < 3\rangle \rightarrow \langle \square, \{X \rightarrow ?X, Y \rightarrow 2\}, \{\}, \{\}, ?X < 3\rangle}}{\langle Y = 2; X = 5, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X < 3\rangle \rightarrow \langle X = 5, \{X \rightarrow ?X, Y \rightarrow 2\}, \{\}, \{\}, ?X < 3\rangle}$$

- Tercer y último paso de ejecución:

$$\dfrac{\langle 5, \{X \rightarrow ?X, Y \rightarrow 2\}, \{\}, \{\}, ?X < 3\rangle \rightarrow 5}{\langle X = 5, \{X \rightarrow ?X, Y \rightarrow 2\}, \{\}, \{\}, ?X < 3\rangle \rightarrow \langle \square, \{X \rightarrow 5, Y \rightarrow 2\}, \{\}, \{\}, ?X < 3\rangle}$$

Paralelamente, la traza de ejecución de la rama en la cual la condición se ha evaluado como falsa es la siguiente:

- Primer paso de ejecución:

$$\dfrac{\dfrac{\dfrac{\langle X, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, true\rangle \rightarrow ?X \quad \langle 3, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, true\rangle \rightarrow 3}{\langle X < 3, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, true\rangle \rightarrow ?X < 3}}{\langle if\ (X < 3)\ Y = 2\ else\ Y = 6, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, true\rangle \rightarrow \langle Y = 6, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X \geq 3\rangle}}{\langle if\ (X < 3)\ Y = 2\ else\ Y = 6; X = 5, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, true\rangle \rightarrow \langle Y = 6; X = 5, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X \geq 3\rangle}$$

- Segundo paso de ejecución:

$$\dfrac{\dfrac{\langle 6, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X \geq 3\rangle \rightarrow 6}{\langle Y = 6, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X \geq 3\rangle \rightarrow \langle \square, \{X \rightarrow ?X, Y \rightarrow 6\}, \{\}, \{\}, ?X \geq 3\rangle}}{\langle Y = 6; X = 5, \{X \rightarrow ?X, Y \rightarrow ?Y\}, \{\}, \{\}, ?X \geq 3\rangle \rightarrow \langle X = 5, \{X \rightarrow ?X, Y \rightarrow 6\}, \{\}, \{\}, ?X \geq 3\rangle}$$

- Tercer y último paso de ejecución:

$$\dfrac{\langle 5, \{X \rightarrow ?X, Y \rightarrow 6\}, \{\}, \{\}, ?X \geq 3\rangle \rightarrow 5}{\langle X = 5, \{X \rightarrow ?X, Y \rightarrow 6\}, \{\}, \{\}, ?X \geq 3\rangle \rightarrow \langle \square, \{X \rightarrow 5, Y \rightarrow 6\}, \{\}, \{\}, ?X \geq 3\rangle}$$

Como resultado del proceso hemos obtenido dos configuraciones de programa finales, que podemos interpretar como sigue: si el valor inicial de la variable X, $?X$, es menor que 3, el programa terminará con los valores $X = 5$, $Y = 2$. Si, por el contrario, $?X$ es mayor o igual que 3, al término del programa las variables contendrán los valores $X = 5$, $Y = 6$. Nótese que el análisis no aporta ninguna información sobre el valor concreto inicial de Y, ya que la expresión simbólica asociada $?Y$ no interviene en las condiciones del programa, y por tanto no influye en el resultado obtenido.

2.2.3. Gestión de estructuras dinámicas

Los lenguajes de la familia C ofrecen soporte a definición y uso de estructuras de datos, que permiten agrupar una serie de variables bajo un nombre de manera que se unifica el acceso a sus valores. Dichas estructuras son heterogéneas, es decir, pueden asociar entre sí datos de distintos tipos. Por este motivo se suelen emplear como base para la construcción de estructuras de datos dinámicas, como listas enlazadas, donde cada nodo contiene un conjunto de datos y una referencia a su nodo sucesor[5]. Por ejemplo, los nodos de una lista dinámica de números enteros se definirían en C de la siguiente manera [3, p. 306]:

```
struct t_nodo {
   int numero;
   t_nodo *siguiente;
}
```

Para utilizar una estructura en un programa, primero debe declararse indicando los elementos que la componen, también denominados *campos*, y el nombre que la identificará. Posteriormente, se podrán crear variables del tipo de la estructura dentro del programa, y almacenar datos en sus campos. Estas variables de tipo estructura pueden verse como punteros, ya que, en la práctica, una estructura se materializa como un bloque de memoria contigua cuya longitud viene determinada por la cantidad y tipo de campos que contiene. De este modo, cada posición de memoria donde se almacena uno de sus campos se traduce a una dirección base, correspondiente con el nombre de la estructura, y un desplazamiento respecto de esa dirección base.

En nuestro caso, especificaremos el desplazamiento en memoria en términos de elementos, a diferencia de la notación de C, donde se suelen expresar en *bytes* de datos. Esto se debe a que KernelC limita la información a un tipo de dato, y aunque se trate de números enteros que en realidad ocupan 4 *bytes*, se unifica el espacio ocupado por cada elemento para simplificar. Así pues, continuando con el ejemplo anterior, si se declara una variable de la estructura *t_nodo* con nombre *nodo_inicial*, el acceso a sus dos componentes *numero* y *siguiente* podría considerarse como dos punteros que referencien a las direcciones de memoria *nodo_inicial*+0 y *nodo_inicial*+1, respectivamente.

De acuerdo con esta visión de las estructuras de datos, podemos observar que se encuentran indirectamente aceptadas en la sintaxis de KernelC, aplicándose las mismas reglas que con los accesos a punteros convencionales. No obstante, las variables de tipo estructura presentan la posibilidad de declararse como punteros o como abstracciones de alto nivel, y si bien en el primer caso sería necesario inicializar memoria dinámica mediante *malloc* para poder almacenar valores en sus campos, en el segundo se reserva la memoria sistemáticamente tras la declaración. Por ello, deberán proporcionarse mecanismos para interpretar la lectura de la

[5] Para ampliar la información sobre la sintaxis relativa a las estructuras en C, recomendamos consultar [3, pp. 220-229], y para más información sobre el papel que juegan en las estructuras de datos dinámicas: [3, pp. 298-317].

declaración de una variable estructura como un *malloc* de tantas direcciones de memoria como campos la compongan.

El soporte al manejo de punteros y, especialmente, de estructuras de datos dinámicas, complica la ejecución simbólica puesto que un puntero *P* del programa podría no estar inicializado (en cuyo caso apunta a la dirección de memoria 0x0 o *null*) o referenciar al mismo bloque de memoria que otro puntero del programa, *Q*, en cuyo caso cuando se asigne un valor nuevo a **P*, la lectura de **Q* proporcionará ese valor recién asignado. Este efecto se suele denominar *aliasing*. Para que la semántica de KernelC considere estas posibilidades de forma eficaz (evitando ocasionar una explosión de estados), aplicaremos una variante de la técnica denominada *lazy initialization*.

Conceptualmente, el patrón de *lazy initialization* consiste en retrasar en el tiempo la instanciación de un objeto en memoria hasta su primer uso, con el objetivo de mejorar el rendimiento del sistema y reducir los requerimientos de memoria del programa [8]. Es decir, la inicialización de los datos de un sistema se efectúa bajo demanda, de modo que si un objeto determinado nunca llega a ser usado no se consume esfuerzo de computación para darle un valor. Esto es crítico si el objeto en cuestión requiere una gran cantidad de memoria para mantenerlo actualizado; por ejemplo, si se trata de una estructura compuesta por varios vectores o si la inicialización de un elemento implica establecer una conexión con una base de datos. La mayor parte de los lenguajes de programación modernos proporciona mecanismos en forma de librerías para hacer uso de esta técnica.

En el contexto de la ejecución simbólica, implementar el patrón de *lazy initialization* implica que sólo se tendrán en cuenta los casos especiales de la inicialización de punteros cuando se utilicen por vez primera, a través de bifurcaciones en el árbol de ejecución para evaluar un camino que explore cada una de las posibilidades. Sin embargo, en nuestro caso práctico, los objetos que deben inicializarse son punteros cuya dirección de memoria puede modificarse manualmente. Por ende, se considerarán los casos especiales previamente comentados tras asignar un valor a un puntero.

A continuación redefiniremos la regla de la semántica de KernelC referente a la asignación de valores a punteros para extenderla con la aplicación simplificada de la *lazy initialization*. Se creará una rama por cada dirección base distinta en el conjunto *Mem*, es decir, que al asignar un valor a un campo de una estructura no se intentará igualar su dirección de memoria a la del resto de campos de la misma variable de tipo estructura. Además, se recorrerá un camino adicional en el cual no se produzca *aliasing*, de manera que el puntero al cual se asigna el valor no sea equivalente a ninguna otra dirección de memoria incluida en *Mem*. El caso en que el puntero no está inicializado no producirá bifurcación, pues asumiremos que cuando el controlador de ejecución simbólica detecte que se intenta asignar un valor a una dirección de memoria no almacenada en *Mem*, se producirá un error y se detendrá la ejecución del programa. La extensión de KernelC con *lazy initialization* queda como sigue:

- Asignación a un puntero $*P = K$

$$\frac{\langle K, Env, Mem, Ptr, PC\rangle \rightarrow N}{\langle *P = K, Env, Mem, Ptr, PC\rangle \rightarrow \langle \square, Env, (Mem \setminus \{P \rightarrow \bullet\}) \cup \{P \rightarrow N\}, Ptr, PC\rangle}$$

y, en caminos independientes,

$$\frac{\langle K, Env, Mem, Ptr, PC\rangle \rightarrow N}{\langle *P = K, Env, Mem, Ptr, PC\rangle \rightarrow \langle \square, Env, (Mem \setminus \{P \rightarrow \bullet\}) \cup \{P \rightarrow N\}, Ptr, PC \wedge P = ?P_i\rangle}$$

$\forall\ ?P_i$ tal que $?P_i \in Mem \wedge ?P_i \mathrel{!=} P$

3. Desarrollo del sistema

En esta sección de cariz práctico, explicaremos qué herramientas se han utilizado y qué pasos se han seguido para implementar un prototipo de motor de ejecución simbólica objeto de este trabajo. Nos centraremos especialmente en describir la estructura de la aplicación y cómo se plasman sobre el código fuente los conceptos teóricos establecidos en el capítulo 2. Finalizaremos la construcción del sistema con una prueba de su ejecución sobre un ejemplo de programa real, mostrando los resultados obtenidos.

3.1. Estructura de la aplicación

A un alto nivel de abstracción, nuestro motor de ejecución simbólica consistirá en un programa escrito en lenguaje C y compilado para su ejecución en consolas de comandos UNIX. El sistema recibirá como entrada del usuario el nombre de un fichero de código fuente en C (con extensión .c), y el identificador de la función incluida en dicho archivo de la que se desea inferir una especificación formal. El usuario obtendrá como salida de la aplicación una lista impresa en la misma terminal de comandos, donde se le proporcionará la información recopilada en el análisis sobre cada camino ejecutable del programa. Para un primer prototipo, esta información se compondrá de: el valor que se retornó al alcanzar el estado final, si lo hubiera; la *path condition* acumulada al final de la ejecución de la rama, y un conjunto de valores sugeridos para los parámetros de entrada que garanticen la ejecución del camino concreto.

Internamente, el sistema se compondrá de tres módulos diferenciados, de manera que, ejecutándolos secuencialmente, se puedan obtener las especificaciones de la función a partir de su código fuente. En primer lugar, se dispondrá de un módulo de procesamiento sintáctico del programa o *parsing*, que leerá el fichero del programa y lo traducirá a estructuras de datos específicas que representen el conjunto de instrucciones y de variables que forman la función, respectivamente. Partiendo de estas estructuras e inicializando tres conjuntos vacíos para *Mem*, *Ptr* y la *path condition*, se obtendrá la configuración inicial para la ejecución del programa. A continuación, un módulo dedicado puramente a la ejecución simbólica del programa accederá progresivamente a la estructura de datos que contiene la abstracción de las instrucciones, interpretará la información y actualizará la configuración, bifurcando según sea necesario hasta alcanzar todos los estados finales posibles. Por último, cada vez que se alcance un estado final se activará el módulo de resolución de restricciones que consultará a un *SAT solver*, en concreto Z3, la satisfacibilidad de la *path condition* acumulada y obtendrá ejemplos de valores que cumplan sus propiedades. También se consultará a Z3 para podar bifurcaciones en el árbol de ejecución que no se correspondan con ejecuciones

reales. Desde este módulo, además, se formateará la información de salida obtenida del resolutor y se mostrará al usuario. Gráficamente, esta estructura se puede representar de la siguiente manera:

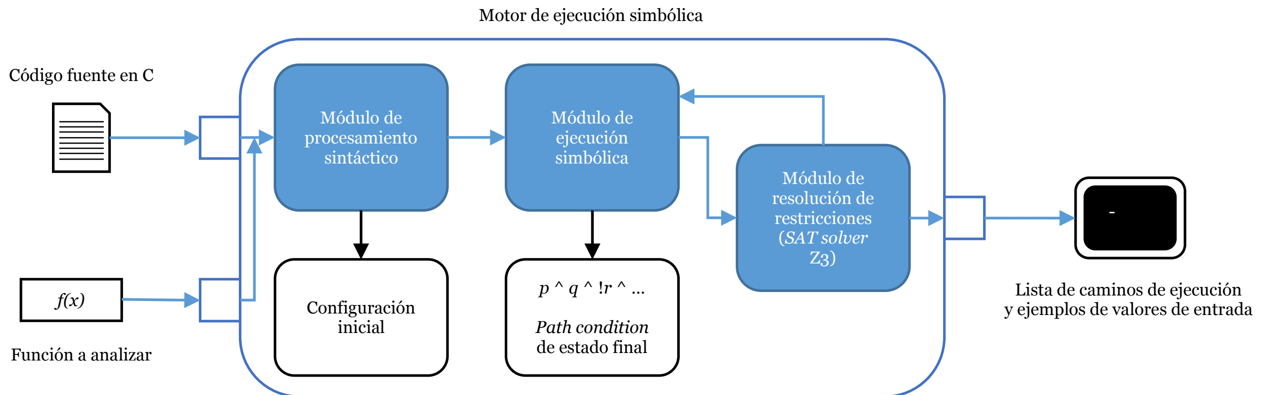

Figura 2. Representación gráfica de la estructura del motor de ejecución simbólica

3.2. Descripción de la implementación

Para implementar la estructura previamente definida, hemos utilizado el sistema de ficheros representado en la Figura 3. A lo largo de este apartado, se situarán los fragmentos de código que usemos para ilustrar las explicaciones en su fichero correspondiente, haciendo referencia a él directamente por su nombre.

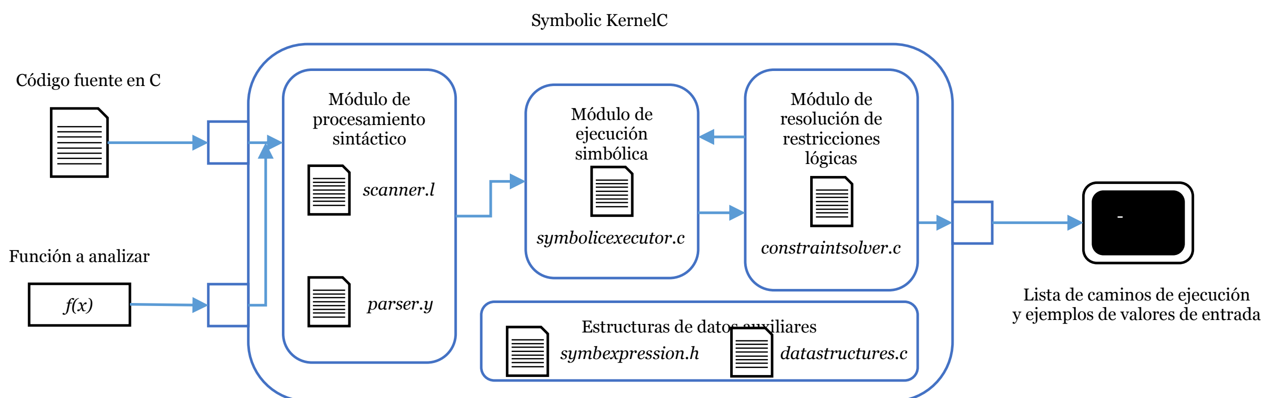

Figura 3. Estructura de ficheros de la implementación del motor de ejecución simbólica

La viabilidad de nuestro sistema es altamente dependiente de cómo se organicen los datos para representar las expresiones del programa de forma fácilmente comprensible para el computador; el grado de similitud de las estructuras de datos empleadas a las reglas de la sintaxis del lenguaje afecta directamente al rendimiento y complejidad del sistema en su conjunto. Cuanto más sencillo sea identificar una variable interna del motor con una expresión del lenguaje, la aplicación de las

reglas de la semántica será, a su vez, más simple. Por este motivo, hemos decidido adoptar como tipo de dato una estructura tal que cada una de sus instancias se corresponda biunívocamente con una expresión de la sintaxis simbólica de KernelC, al que hemos denominado *SExp* por consistencia con la terminología del estándar. Su definición en lenguaje C, contenida en *symbexpression.h*, es la que sigue:

```
typedef struct Expression{
      SExpType type;
      SExpOperationType operationType;
      ConcreteValue value;
      struct Expression* factor1;
      struct Expression* factor2;
}SExp;
```

Los tipos de expresión, operación y valores concretos que se contemplan para esta estructura de datos son los siguientes:

```
typedef enum {
      SEXP_SKIP,
      SEXP_NATURAL,
      SEXP_INTEGER,
      SEXP_ID,
      SEXP_OPERATION,
      SEXP_ASSIGNMENT,
      SEXP_MALLOC,
      SEXP_FREE,
      SEXP_POINTER,
      SEXP_STRUCT_POINTER,
      SEXP_IF,
      SEXP_WHILE,
      SEXP_NOT,
      SEXP_RETURN
}SExpType;


typedef enum{
      SEXP_OP_NO_OP,
      SEXP_OP_ADDITION,
      SEXP_OP_SUBTRACTION,
      SEXP_OP_PRODUCT,
      SEXP_OP_DIVISION,
      SEXP_OP_EQUALS,
      SEXP_OP_NOT_EQUALS,
      SEXP_OP_LESS,
      SEXP_OP_LESS_EQUAL,
      SEXP_OP_GREATER,
      SEXP_OP_GREATER_EQUAL,
      SEXP_OP_AND,
      SEXP_OP_OR
}SExpOperationType;
```

```
typedef union{
      unsigned int concreteNatural;
      int concreteInteger;
      char* concreteId;
}ConcreteValue;
```

El elemento *SEXP_SKIP* se añade para simbolizar el fin de ejecución de un programa o subprograma (de manera equivalente al símbolo "□" que hemos empleado en la semántica), así como el tipo de operación *SEXP_OP_NO_OP*, que utilizaremos como valor neutro para las expresiones que no sean operaciones binarias. Por su parte, se diferencian los punteros de tipo entero de los punteros de tipo estructura por la distinta forma de acceder a sus valores. Recordemos que los campos de una estructura en memoria se referencian por una dirección base, determinada por el identificador de la variable estructura, y un desplazamiento, dado por el identificador del campo concreto.

Con respecto a la unión *ConcreteValue*, se incluye para poder almacenar números naturales y enteros no simbólicos, así como para referenciar a los valores simbólicos simples *SNat* y *SInt* asociados a los parámetros del programa (véase 2.2.2). El tipo de datos *union* de C es similar a *struct*, con la salvedad de que todos sus miembros comparten el mismo espacio en memoria, por lo que sólo se puede tener almacenado un valor a la vez. La interpretación de este valor dependerá del alias mediante el cual se accede (el campo de la unión).

El hecho de que *SExp* se haya definido como un tipo de datos recursivo evita limitar el alcance de las reglas más de lo que estipula la sintaxis, y proporciona abstracción acerca de los subtérminos de cada expresión. En otras palabras, permite representar, por ejemplo, una operación de suma cuyos factores sean el identificador de una variable y el producto de dos números enteros. Si nos fijamos en las expresiones reconocidas por KernelC (véase 2.2.2), podemos observar que son intrínsecamente recursivas; a modo de ejemplo, la expresión simbólica genérica contiene en su definición la suma de expresiones simbólicas: "*SExp* ::= [...] *SExp* + *SExp* [...]".

A continuación nos dispondremos a construir el procesador sintáctico para convertir el texto plano del código fuente en una estructura de datos interna empleando el tipo que acabamos de definir. Sin embargo, el reconocimiento automático de construcciones léxico-semánticas es un proceso muy complejo de implementar desde cero, puesto que implica el uso de autómatas de pila o técnicas similares para relacionar los elementos léxicos (*tokens*) y, por ende, conocimientos específicos de teoría de lenguajes formales a muy alto nivel de detalle. Por este motivo, hemos optado por generar automáticamente el código C encargado del *parsing* del programa empleando las herramientas Flex y Bison, las cuales proporcionan una abstracción para definir de manera sencilla escáneres y analizadores sintácticos de texto a partir de una lista de *tokens* y una gramática determinadas.

Flex[6] y Bison[7] efectúan el reconocimiento de construcciones léxicas y sintácticas, respectivamente, mediante *matching* con reglas formales que se asocian a acciones especificadas en código C. El proceso es, pues, muy similar a una evaluación vía lógica de reescritura (véase 2.1.1): cuando la estructura formal de un elemento leído coincide con una regla de la gramática del lenguaje, se ejecuta la acción correspondiente de forma sistemática. Así pues, podemos construir una representación en memoria del conjunto de instrucciones del programa, si añadimos una regla sintáctica para cada expresión aceptada en el lenguaje KernelC (ya sea aritmética, relacional, lógica o instrucción), y asociamos a ella la creación de un nuevo elemento de tipo *SExp*.

Así pues, definiremos un escáner léxico bajo la notación de Flex (*scanner.l*) y un analizador semántico con el lenguaje específico de Bison (*parser.y*), los cuales darán lugar a dos ficheros de código C generados automáticamente, con nombres *lex.yy.c* y *parser.tab.c* (asignados por las propias herramientas). Pero, para construir una representación en memoria de la configuración de ejecución simbólica de un programa, e inicializar sus celdas a sus valores iniciales, debemos establecer previamente una estructura de datos acorde a la información que almacenará. En nuestro caso particular, hemos definido dicha estructura en el fichero *datastructures.c* de la siguiente manera:

```
typedef struct{
      SExp** continuation; //Cont
      SymbolicVariable** environment; //Env
      int numVariables;
      MemoryPointer** memory; //Mem
      int memLength;
      ReservedMemoryPointer** pointers; //Ptr
      int ptrLength;
      SExp** pathCondition; //PC
      int numPathConstraints;
}Configuration;
```

Obsérvese que se incluye un indicador de la cantidad de elementos de cada celda salvo por el conjunto de instrucciones a ejecutar; esto se debe a que no es necesario, pues previamente definimos el tipo *SEXP_SKIP* para representar el fin de un programa. Los tipos de datos auxiliares *SymbolicVariable*, *MemoryPointer* y *ReservedMemoryPointer* incluyen todos un identificador del componente y un valor asociado, bien sea de tipo *SExp* para los conjuntos *Env* y *Mem*, o bien un número natural en el caso de *Ptr*.

---

[6] Para más información acerca del generador de analizadores léxicos Flex, se puede consultar su documentación oficial en:

V. Paxson, W. Estes y J. Millaway, «Lexical Analysis with Flex, for Flex 2.5.37,» Julio 2012. [En línea]. Available: http://flex.sourceforge.net/manual/. [Último acceso: Mayo 2014].

[7] Para más información acerca del generador de analizadores sintácticos Bison, y su integración con Flex, se puede consultar su documentación oficial en:

Free Software Foundation, Inc., «Bison 3.0.2,» Octubre 2013. [En línea]. Available: http://www.gnu.org/software/bison/manual/bison.html. [Último acceso: Mayo 2014].

Una vez el módulo de procesamiento sintáctico ha obtenido la representación en memoria de la configuración inicial, se invocará al módulo de ejecución simbólica, encargado de explorar todos los caminos ejecutables posibles y obtener los conjuntos de propiedades que satisfacen los datos de entrada en cada uno de ellos. Este módulo (implementado en el fichero *symbolicexecutor.c*) consistirá en una función principal, que denominaremos *symbolicExecute*, encargada de gestionar el avance del flujo del programa y la bifurcación de caminos; y una función subordinada, llamada *executeExpression*, cuyo propósito será la evaluación de expresiones simbólicas en forma de variables de tipo *SExp*. El procedimiento principal *symbolicExecute* consistirá en un bucle principal que, en cada iteración, extraerá un elemento de la lista de instrucciones del programa y lo enviará a *executeExpression* para evaluarlo. Este bucle principal se ejecutará hasta alcanzar una expresión de tipo *SEXP_SKIP* o *SEXP_RETURN*, en cuyo caso se detectará fin de programa y la función terminará.

Por su parte, *executeExpression* se presenta como una función recursiva que recibirá como parámetro una expresión simbólica, reconocerá de qué tipo de expresión se trata a través del campo *type* de la estructura *SExp* y aplicará la regla de la semántica de KernelC correspondiente. Esta aplicación puede suponer bien la sustitución del identificador de una variable por su valor actual (para *SEXP_ID*, *SEXP_POINTER* y *SEXP_STRUCT_POINTER*), bien la obtención de otra expresión simbólica derivada (*SEXP_OPERATION*, *SEXP_NOT*) o la modificación de un elemento de la configuración del programa (expresiones de última instancia como *SEXP_ASSIGNMENT*, *SEXP_MALLOC* o *SEXP_FREE*). La evaluación de los sub-términos de una expresión, almacenados en los campos *factor1* y *factor2*, se efectuará mediante llamadas recursivas a *executeExpression* hasta que se obtenga el resultado de evaluar la instrucción completa.

A modo de ejemplo, veamos el código asociado a la evaluación de operaciones binarias (tipo SEXP_OPERATION):

```
SExp* executeExpression(SExp* expression){
      SExp* operationInterpreted;
      if(!expression) return 0;
      switch(expression->type){
      case SEXP_OPERATION:
            operationInterpreted = malloc(sizeof(SExp));
            operationInterpreted->type = SEXP_OPERATION;
            operationInterpreted->operationType =
            expression->operationType;
            operationInterpreted->factor1 =
            executeExpression(expression->factor1);
            operationInterpreted->factor2 =
            executeExpression(expression->factor2);
            return operationInterpreted;
      }
}
```

La ejecución de expresiones de tipo *SEXP_IF* y *SEXP_WHILE* constituye casos especiales de los que se hace cargo la propia función principal, dado que implica una bifurcación en el árbol de ejecución simbólica. No obstante, la especificación actual de *SExp* no contempla la inclusión de las instrucciones asociadas a las cláusulas *if* y *else* de una expresión condicional, ni tampoco el cuerpo de un bucle *while*. Ambos tipos emplearán el elemento *factor1* para almacenar la condición a evaluar cuando se ejecuten, pero requieren campos adicionales para poder diferenciar las instrucciones específicas de cada una de las ramas que generan. De esta manera, debemos añadir al tipo *SExp* previamente definido dos sub-listas que contendrán las expresiones a ejecutar en cada camino abierto tras evaluar la condición. Denotaremos tales sub-listas con los identificadores *trueBranch* y *falseBranch*.

A este respecto, hay otra particularidad que debemos considerar: la bifurcación constante como resultado de evaluar un bucle puede ocasionar una explosión de estados o incluso que el proceso de ejecución simbólica nunca termine, si nunca se alcanza una situación en que la guarda sea necesariamente falsa. Por ello, es obligatorio adoptar algún mecanismo de terminación que garantice que un bucle no se evalúe un infinito número de veces. Existen diversas técnicas que permiten establecer límites a la ejecución simbólica de bucles sin perder completitud en el análisis; entre ellas, destaca la subsunción de estados abstractos[8]. Sin embargo, en nuestro caso se asume que los programas que se analizarán serán relativamente simples, por lo que podemos admitir una sub-aproximación acotando el número de iteraciones a evaluar. Así pues, nuestro módulo de ejecución simbólica dispondrá de un contador como variable global y, definido como una constante, el número máximo de veces que se ejecutará un mismo bucle.

De este modo, *symbolicExecute* generará nuevas ramas en el árbol de ejecución en los siguientes casos: cuando la condición de una sentencia *if* o *while* no se pueda determinar cierta o falsa, y tras la asignación de un valor a un puntero, ya sea de tipo entero o estructura. Las bifurcaciones en el árbol de ejecución se traducirán en llamadas recursivas a *symbolicExecute*, previa adición de la restricción correspondiente a la *path condition*. La expresión a añadir será bien el resultado de evaluar la condición de *if* o *while* o bien la igualación entre el puntero al cual se asigna un valor y otro puntero del conjunto *Mem*. Cuando se efectúen estas llamadas, se tendrá que indicar como parámetro el punto del programa (posición dentro de la lista) desde el cual debe comenzar a ejecutarse la rama actual, con el propósito de evitar que se reevalúen instrucciones. Además, para poder restaurar el estado cuando finalice la ejecución de cada camino, se implementará un sencillo algoritmo de *backtracking*, guardando físicamente en memoria una copia de la configuración antes de llevar a cabo la llamada recursiva.

---

[8] Para ampliar información sobre la subsunción abstracta, recomendamos consultar la siguiente referencia:

S. Anand, C. S. Pasareanu y W. Visser, «Symbolic Execution with Abstract Subsumption Checking,» de *SPIN*, 2006. [En línea]. Available: http://cs.stanford.edu/people/saswat/research/spin06.pdf. [Último acceso: Junio 2014].

A modo de ejemplo, el caso particular de las instrucciones if-else está cubierto por el siguiente código. La variable *i* que se usa para el paso por parámetro simboliza la posición de la expresión actual dentro de la lista de instrucciones.

```
case SEXP_IF:
SExp* condition = executeExpression(subProgram[i]->
factor1);



if(1 == checkBranchViability(condition)){
/*CASE 1: THE PC IMPLIES THE CONDITION*/
      symbolicExecute(subProgram[i]->trueBranch, 0);

}else if(-1 == checkBranchViability(condition)){
/*CASE 2: THE PC IMPLIES NOT THE CONDITION*/
      symbolicExecute(subProgram[i]->falseBranch, 0);

}else{
/*CASE 3: THERE IS NO IMPLICATION; WE MUST TRY BOTH
BRANCHES*/
      struct Conf* lastState = copyConfiguration();

      configuration->pathCondition[configuration->
      numPathConstraints] = condition;
      configuration->numPathConstraints++;

      symbolicExecute(subProgram[i]->trueBranch, 0);
      symbolicExecute(subProgram, i+1);

      configuration = lastState;

      SExp* reversedCondition = malloc(sizeof(SExp));
      reversedCondition->type = SEXP_NOT;
      reversedCondition->factor1 = condition;

      configuration->pathCondition[configuration->
      numPathConstraints] = reversedCondition;
      configuration->numPathConstraints++;

      if(subProgram[i]->falseBranch){/*If there is an
      else clause*/

            symbolicExecute(subProgram[i]->falseBranch,
            0);
      }
```

La función *symbolicExecute* invoca al módulo de resolución de restricciones cuando alcanza una instrucción de tipo *SEXP_RETURN* e inmediatamente después de evaluar una condición, antes de ramificar la ejecución. La funcionalidad principal

de este módulo radica en la obtención de valores de ejemplo que satisfagan las condiciones de la *path condition*, así como de dar formato a la información e imprimirla por pantalla para mostrarla al usuario. Pero, además, este motor lógico debe ser capaz de comprobar si una expresión determinada es necesariamente cierta o falsa de acuerdo con la *path condition* acumulada en el momento de ejecutar una instrucción condicional. O, formulado en otros términos, si las expresiones lógicas que componen la path condition implican que la condición siempre se cumplirá o no, independientemente de los valores concretos de los términos que intervienen. De este modo podremos optimizar las computaciones y podar ramas imposibles, que no se puedan ejecutar en ningún caso.

Dado que la implementación de un motor lógico de estas características resulta bastante compleja, y que los fundamentos teóricos de la resolución de restricciones no forman parte de los objetivos del presente trabajo, emplearemos un resolutor de problemas de satisfacibilidad booleana, o *SAT solver*, previamente creado y probado. En concreto, nos hemos decantado por utilizar Z3, un sistema desarrollado por Microsoft Research que proporciona una API[9] (en forma de librerías) escrita en varios lenguajes, permitiendo su inclusión en código fuente de programas en C, C++, Python, etc.

Internamente, Z3 mantiene una pila de fórmulas y términos (variables) proporcionados por un usuario o un programa bajo un contexto determinado. Estas fórmulas ejercen la función de aserciones lógicas que definen propiedades de un sistema. Un conjunto de aserciones puede satisfacerse si existe al menos una interpretación, denominada *modelo*, bajo la cual todas las fórmulas de la pila se cumplen [9]. Nuestro objetivo es, mediante las funciones de librería[10], trasladar las restricciones de la *path condition* en formato *SExp* a un conjunto de fórmulas lógicas evaluables por Z3, comprobar si todas ellas pueden satisfacerse y, en caso afirmativo, obtener del propio motor el modelo que se ha aplicado para interpretar las aserciones. Este modelo nos proporcionará un valor para cada expresión simbólica que interviene en las fórmulas especificadas, de manera que si se aplica el modelo a los parámetros de entrada del programa se garantizará que el flujo de ejecución tome el camino asociado a dicha *path condition*.

En nuestra implementación, la encargada de efectuar esta transformación que obtiene unas sugerencias de valores de entrada a partir de una lista de expresiones simbólicas es la función *checkPathConstraint*, incluida en el fichero *constraintsolver.c*. El código que ejecuta esta función es el siguiente:

---

[9] La documentación de la API para lenguaje C, que se ha empleado en este proyecto, puede consultarse en el siguiente recurso web:

Microsoft Corporation, «Z3: C API,» Noviembre 2012. [En línea]. Available: http://research.microsoft.com/en-us/um/redmond/projects/z3/group__capi.html. [Último acceso: Junio 2014]

[10] Se pueden consultar ejemplos de uso de la librería de Z3 con código C en el siguiente enlace:

Microsoft Corporation, «Z3 Source Code Examples,» Noviembre 2012. [En línea]. Available: http://z3.codeplex.com/sourcecontrol/latest#examples/c/test_capi.c. [Último acceso: Junio 2014].

```
void checkPathConstraint(){

   Z3_config z3conf = Z3_mk_config();
   Z3_context context = Z3_mk_context(z3conf);
   Z3_del_config(z3conf);


   Z3_ast currentConstraint;

   int i;
   for(i=0; i< configuration->numPathConstraints; i++){

      currentConstraint = makeConstraint(configuration->
      pathCondition[i], context);
      Z3_assert_cnstr(context, currentConstraint);

   }

   Z3_model resolutionModel = 0;
   Z3_lbool  result  =  Z3_check_and_get_model(context,
   &resolutionModel);

   switch(result){

      case Z3_L_FALSE:
            printf("UNSATISFIABLE: There are no possible
            entry values for this path!\n\n");
            break;
      case Z3_L_UNDEF:
            printf("UNDEFINED: There is no guarantee that
            this path will always be executed with these
            values:\n");
            printf("%s\n\n",  Z3_model_to_string(context,
            resolutionModel));
            break;
      case Z3_L_TRUE:
            printf("SATISFIABLE:  These  are  the  values
            suggested to execute this path:\n");
            printf("%s\n\n",  Z3_model_to_string(context,
            resolutionModel));
            break;
   }

   Z3_del_context(context);
```

```
}
```

Las tres primeras líneas de la función, así como la última, tienen como único objetivo la creación y borrado de información interna de Z3, pero son necesarias para el correcto funcionamiento del motor lógico. La función auxiliar makeConstraint interpreta la información almacenada en una restricción en formato SExp y la convierte al tipo de dato que emplea Z3 para expresar fórmulas y términos, Z3_ast. Nótese que Z3 podría no ser capaz de determinar con certeza si una path condition puede satisfacerse; en cuyo caso advertirá de este hecho y a continuación mostrará igualmente el modelo que ha deducido.

3.3. Prueba del sistema

A continuación, mostraremos un ejemplo de ejecución del motor implementado para ilustrar los resultados obtenidos. Partiremos de un código fuente sencillo, pero que contiene varios tipos diferentes de expresiones de KernelC para maximizar la cobertura de la prueba. El código que se va a analizar empleando la aplicación desarrollada es el siguiente:

```
struct complex
{
      int real;
      int img;
};


int addComplex(int aReal, int bReal, int aImg, int
bImg);


int addComplex(int aReal, int bReal, int aImg, int bImg)
{
   struct complex a;
   struct complex b;
   struct complex c;

   a.real = aReal;
   b.real = bReal;
   a.img = aImg;
   b.img = bImg;

   c.real = a.real + b.real;
   c.img = a.img + b.img;
```

```
        if ( c.img >= 0 )
              return c.real + c.img;
        else
              return c.real;
    }
```

El programa hace uso tanto de expresiones aritméticas y relacionales como de asignaciones a punteros y, para finalizar, un condicional. Así pues, debería construirse una *path condition* que contenga tanto equivalencias entre direcciones de memoria de punteros (expresadas como valores simbólicos) como la condición o la negación de la misma, según la rama en que nos encontremos.

La ejecución del motor proporcionándole como parámetros el nombre del fichero de código y el nombre de la función *addComplex* produce el siguiente resultado:

```
Output ×
SymbolicKernelC (Build, Run) ×   SymbolicKernelC (Run) ×

EXECUTABLE PATH 1: RETURN c->real + c->img
Path Constraint: ?a == ?b ^ ?b == ?c ^ ?aImg + ?bImg >= 0
SATISFIABLE: These are the values suggested to execute this path:
?b -> 1
?bImg -> 0
?c -> 1
?aImg -> 0
?a -> 1


EXECUTABLE PATH 2: RETURN c->real
Path Constraint: ?a == ?b ^ ?b == ?c ^ !(?aImg + ?bImg >= 0)
SATISFIABLE: These are the values suggested to execute this path:
?b -> 1
?bImg -> (- 1)
?c -> 1
?aImg -> 0
?a -> 1
```

Figura 4. Fragmento de la salida obtenida tras la ejecución simbólica de la función addComplex

Por limitación en la extensión de la salida impresa, ajustada al tamaño de la consola, se han seleccionado dos casos significativos. Nótese cómo en las restricciones de la *path condition*, efectivamente, se produce la equivalencia de punteros dos a dos y se incluye la evaluación de la condición de la sentencia *if*, en este caso, la sustitución del puntero *c.img* (que, a su vez, corresponde a la dirección de memoria ?*c* + 1) por su valor almacenado en memoria, ?*aImg* + ?*bImg*. Se puede observar también que, en el estado actual del prototipo, el resolutor Z3 trata las direcciones de memoria simbólicas como valores numéricos enteros, lo cual no es del todo erróneo, pero sí sería mejorable; se plantea, para versiones posteriores del motor, la interpretación de memoria simbólica como números en formato hexadecimal que tomen valores típicos del direccionamiento en UNIX. De esta manera, los resultados serán más realistas y familiares a los usuarios.

4. Conclusiones

En este trabajo nos hemos centrado, fundamentalmente, en conocer y comprender técnicas de actualidad en el campo de la ingeniería de software: semánticas formales, ejecución simbólica, resolución de restricciones lógicas... Todas ellas, integradas adecuadamente, poseen un gran potencial para simplificar el proceso de verificación de programas informáticos, apoyando y facilitando en gran medida la labor de los desarrolladores. En nuestro caso, las hemos enfocado a automatizar la obtención de especificaciones por la vía del análisis sintáctico y semántico de código fuente.

Partiendo de estos conocimientos, hemos sido capaces de realizar dos aportaciones al campo de la ingeniería de software. En primer lugar, hemos elaborado una nueva formalización de la semántica de un lenguaje de programación, en concreto KernelC, hasta ahora definida únicamente mediante fórmulas de lógica de reescritura. Para ello, se ha empleado la notación de semánticas operacionales estructurales o *SOS*, más próxima al paradigma imperativo en el que se enmarca el lenguaje en cuestión. Y, en segundo lugar, hemos aplicado esta redefinición en conjunto con el resto de técnicas aprendidas para desarrollar un motor de ejecución simbólica que se aplique sobre programas escritos en KernelC, e infiera de él especificaciones sencillas de manera automática.

Sin embargo, el motor desarrollado corresponde a la primera fase de un proyecto más ambicioso, por lo que se pueden plantear todavía varias extensiones y mejoras. Por ejemplo, debería dotarse de la capacidad de poderlo ejecutar a partir de unas restricciones iniciales, que se añadan a las propiedades intrínsecas del propio programa. Asimismo, se ha de considerar la posibilidad de almacenar los resultados obtenidos en ficheros de texto, proporcionando una mayor accesibilidad de la información, así como de optimizar el formato de dichos resultados, ya que en su estado actual puede resultar confuso en algunos aspectos. También sería conveniente mejorar las prestaciones ofrecidas y, aunque sea menos relevante, interesaría eliminar algunas restricciones menores sobre el lenguaje que el motor es capaz de interpretar, las cuales se han impuesto para este primer prototipo por cuestión de simplicidad.

Al inicio del trabajo nos marcamos como objetivos, además del desarrollo del motor y la aplicación de técnicas de vanguardia, la integración de los conocimientos obtenidos durante los estudios de grado en un contexto práctico. Tales conocimientos proceden, en su mayoría, de asignaturas fundamentalmente teóricas, por lo que la realización de este trabajo ha supuesto una oportunidad excelente para aplicarlos en un proyecto real, reforzando la comprensión de los contenidos y ampliándolos más allá de la línea base marcada en sus guías docentes. Las asignaturas más influyentes en este sentido han sido Análisis, Validación y Depuración de software, que aportó la base de la ejecución simbólica y el análisis estático, y Métodos Formales en la ingeniería de software, proporcionando los conceptos sobre semánticas de lenguajes y especificaciones formales; puesto que, sin esos conocimientos centrales, la materialización del proyecto habría sido

imposible. Empero, también se han aplicado importantes contenidos de Diseño de software (programación modular), de Lenguajes, Técnicas y Paradigmas de programación (clasificación de lenguajes de programación e identificación de notaciones idóneas para la semántica según su paradigma), de Teoría de Autómatas y Lenguajes formales junto con conocimientos básicos de teoría de compiladores (procesamiento sintáctico de código)…

En definitiva, esperamos que este trabajo sea útil como punto de partida para la investigación e implementación de herramientas que puedan acoplarse al proceso de desarrollo de software en general, y al subproceso de verificación y pruebas en particular. Los rápidos avances tecnológicos que se producen día a día proporcionan a los desarrolladores nuevas oportunidades para optimizar la calidad del software, pero a su vez elevan los niveles de exigencia hasta tal punto que resulta necesario recurrir a soporte automático para producir software de alta calidad y coste razonable. Por este motivo nuestra investigación es tan crucial. Quizá, en unos años, la automatización de procesos que hoy nos parece lejana se haya consolidado con un uso generalizado en la industria.

5. Referencias

[1] M. Alpuente, M. A. Feliú y A. Villanueva, Automatic Inference of Specifications using Matching Logic.

[2] G. Rosu, W. Schulte y T. F. Serbanuta, Runtime Verification of C Memory Safety, Microsoft Research, 2009.

[3] M. Á. Acera García, Manual Imprescindible C/C++, Anaya Multimedia, 2012.

[4] Formal Systems Laboratory, «Term Rewriting and Rewriting Logic,» [En línea]. Available: http://fsl.cs.illinois.edu/index.php/Term_Rewriting_and_Rewriting_Logic. [Último acceso: Junio 2014].

[5] «K Framework,» Diciembre 2013. [En línea]. Available: http://www.kframework.org/index.php/Main_Page. [Último acceso: Junio 2014].

[6] G. D. Plotkin, «A structural approach to operational semantics,» *The Journal of Logic and Algebraic Programming,* vol. 60 61, pp. 17-139, 2004.

[7] «Operational Semantics,» Abril 2002. [En línea]. Available: http://www.cs.jhu.edu/~scott/pl/lectures/opsem.html. [Último acceso: Junio 2014].

[8] Microsoft Network, «Lazy Initialization,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/vstudio/dd997286%28v=vs.100%29.aspx. [Último acceso: Junio 2014].

[9] Microsoft Corporation, «Getting started with Z3: A Guide,» 2014. [En línea]. Available: http://rise4fun.com/z3/tutorial/guide. [Último acceso: Junio 2014].