SymC++: Un depurador simbólico para C++

Proyecto Sistemas Informáticos

Clara Antolín García
Carlos Gabriel Giraldo García
Raquel Peces Muñoz

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2014

Documento maquetado con TEXIS v.1.0+.

SymC++: Un depurador simbólico para C++

Proyecto de Sistemas Infórmaticos

II/2014/6

Clara Antolín García
Carlos Gabriel Giraldo García
Raquel Peces Muñoz
Dirigido por:
Miguel Gómez Zamalloa-Gil

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2014

Autorización

Clara Antolín García, Carlos Gabriel Giraldo García y Raquel Peces Muñoz autores del presente documento y del proyecto "SymC++, una herramienta de depuración simbólica" autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.

Madrid, a 12 de Septiembre de 2014

Clara Antolín García Carlos Gabriel Giraldo García Raquel Peces Muñoz

A nuestros familiares
por creer siempre en nosotros
y a tí lector,
por interesarte en nuestro proyecto

Agradecimientos

Este trabajo no sería posible si no hubiésemos contado con todo el apoyo de todas las personas que nos rodean.

En primer lugar agradecer a nuestro tutor Miguel Gómez Zamalloa-Gil, el cual nos ha prestado todo su apoyo, interés y colaboración durante todo el desarrollo a lo largo de estos meses.

En segundo lugar a todos los profesores que nos han acompañado en nuestra formación desde el primer día que pusimos un pie en la facultad, sin ellos no habríamos adquirido los conocimientos que nos han permitido desarrollar este trabajo.

Por último y no menos importante, a nuestros familiares, amigos y compañeros, los cuales nos han soportado y apoyado en los momentos de desánimo y crispación.

A todos ellos, muchas gracias por creer en nosotros.

Resumen

La ejecución simbólica o evaluación simbólica es un modo de analizar programas para determinar qué entradas causan cada parte del programa a ejecutar. En este tipo de programas, un intérprete recorre el programa donde cada variable asume un valor simbólico acotado. Este recorrido dará como resultado expresiones en términos de dichos símbolos de expresiones y variables y las limitaciones en cuanto a los símbolos de los posibles resultados de cada rama condicional.

Dicho recorrido nos permite determinar las condiciones que deben ser verificadas por los datos de ingreso para que un camino particular se ejecute, y la relación entre los valores ingresados y producidos en la ejecución de un programa.

Durante este proyecto hemos desarrollado una herramienta llamada SymC++ que se basa en la ejecución simbólica para generar test en programas escritos en C++, escribiendo como parámetros el nombre del método, el intervalo de números enteros y la profundidad de los bucles, para ejecutar la función simbólicamente y obtener información de las diferentes ramas generadas. Por otro lado, se podrá obtener información a través de los XMLs generados por el árbol de ejecución o los resultados obtenidos.

El objetivo del proyecto es dotar a los estudiantes de iniciación a la programación de una herramienta que les permita depurar sus programas por un método diferente a la prueba y error.

Palabras clave

Ejecución simbólica, Evaluación simbólica, C++, XML, depuración, intérprete, valor simbólico.

Abstract

Symbolic execution or symbolic evaluation is a means of analyzing programs to determine what inputs cause each part of a program to execute. In this kind of programs, an interpreter follows the program where every variable assume a bounded symbolic value. It thus arrives expressions in terms of those symbols for expressions or variables, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

This path allows us to determine the conditions that have to be verified by the input data for a particularly branch is executed, and the relationship between input values and the produced by program execution.

During this project we have developed a tool called SymC++ which is based on symbolic execution to generate test programs written on C++. We have to write like parameters the method name, the range of integers and the deep to bucles, to execute it symbolically and get the information generated from different branches. On the other side, we could get information through the XMLs generated by the execution tree or results.

The project objective is to provide students of programming introduction a tool that allows them to debug their programs by a different method of trial and error.

Key words

Symbolic execution, Symbolic evaluation, C++, XML, symbolic value

Índice

Índice de figuras

Índice de Tablas

Índice de códigos

Capítulo 1

Introducción

1.1. Motivación de la propuesta

El proyecto nace de la idea y necesidad de utilizar una herramienta útil para los estudiantes de iniciación a la programación. El problema al que se enfrentan los estudiantes cuando los ejercicios a resolver crecen en complejidad, los programas también y normalmente no se comportan como deben para todos los casos. Escribir programas totalmente correctos requiere el uso de una serie de metodologías que en general resultan muy complejas para los estudiantes en los primeros cursos. La mayoría de ellos siguen una mecánica de prueba-error, es decir, escriben los programas sin pararse a pensar demasiado acerca de su corrección, y luego los prueban para ver si se comportan como esperan. Estas pruebas normalmente no son lo suficientemente exhaustivas y por lo tanto los estudiantes no son ni siquiera conscientes de las incorrecciones de sus programas. Desgraciadamente, los mecanismos y herramientas que ayudan en el testing y depuración de programas son en general demasiado avanzados, como para poder ser utilizados por programadores inexpertos.

Dado este problema aparece nuestro proyecto, como una herramienta sencilla e intuitiva que pueda mostrar de un modo natural el recorrido de un programa por sus diferentes ramas y los resultados que obtendría en cada uno de ellos. Esta herramienta ofrecerá una solución a la verificación de programas sencillos para estudiantes de iniciación, y de este modo acercar las herramientas de testing y depuración de un modo más fácil y visual.

En este caso concreto, la herramienta está desarrollada para el lenguaje de C++, puesto que es el utilizado en los primeros cursos de los grados de las Facultades de Informática, Estudios Estadísticos y Matemáticas, y concretamente es el utilizado en la asignatura de primer curso de Fundamentos de Programación en nuestra Facultad.

1.2. Estado del arte y trabajos relacionados

En esta sección se presentan los recursos y herramientas software disponibles que se han planteado como opciones a tener en cuenta a la hora de obtener apoyo en la realización de nuestro proyecto.

1.2.1. C++ y Clang

C++, un lenguaje con historia

C++ es un lenguaje de programación diseñado a mediados de los años 80 por Bjarne Stroustrup en los Laboratorios de Bell. C++ nació como extensión del lenguaje de programación C, al mismo tiempo que proporciona un cierto número de características que engalanan dicho lenguaje. El nombre de C++ fue propuesto por Rick Mascitti en el año 1983, cuando el lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Hasta ese momento se había utilizado el nombre de C con clases.

Se puede decir que C++ es un lenguaje que abarca tres paradigmas de programación: la programación estructurada, la programación genérica y la programación orientada a objetos. Es un lenguaje de programación intermedio, puesto que se puede utilizar tanto para el desarrollo rápido de aplicaciones, como para escribir software de bajo nivel, como drivers y componentes de sistemas operativos.

Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. Existen también algunos intérpretes, tales como ROOT.

Las principales características de este lenguaje son las facilidades que proporciona para la programación orientada a objetos y para el uso de plantillas o programación genérica. Además posee una serie de propiedades dificiles de encontrar en otros lenguajes como es la posibilidad de redefinir los operadores o poder crear nuevos tipos, que se comporten como tipos fundamentales.

La elección de desarrollar nuestra herramienta para el lenguaje de C++ se ha basado en dos motivos, el primero ha sido su popularidad, puesto que según el ranking TIBOE, el cual recoge los lenguajes de programación más usados, se encuentra en cuarta posición, y el segundo motivo ha sido el uso de este lenguaje en los primeros cursos de programación como ya hemos señalado anteriormente.

Clang y LLVM

Clang es un *front-end* de compilador para los lenguajes de programación C, C++, Objective-C y Objective-C++. Usa LLVM (anteriormente conocido como Low Level Virtual Machine, o Máquina Virtual de Nivel Bajo) como su *back-end* y ha sido parte del ciclo de lanzamiento de LLVM desde la versión 2.6. Está diseñado para ofrecer un reemplazo de GNU Compiler Collection (GCC). Es *open-source*, y varias compañías de software están involucradas en su desarrollo, incluyendo a Google y Apple. Su código fuente está bajo la licencia University of Illinois/NCSA. El proyecto Clang incluye además un analizador estático de software y varias herramientas de análisis de código.

Por otro lado, LLVM es una infraestructura para desarrollar compiladores, escrita a su vez en el lenguaje de programación C++, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado, el tiempo de ejecución y el tiempo ocioso en cualquier lenguaje de programación que el usuario quiera definir. Implementado originalmente para compilar C y C++, el diseño agnóstico de LLVM con respecto al lenguaje, y el éxito del proyecto han engendrado una amplia variedad de lenguajes, incluyendo Objective-C, Fortran, Ada, Haskell, bytecode de Java, Python, Ruby y otros.

LLVM suministra las capas intermedias de un sistema de compilado completo, tomando el código en formato intermedio (IF, en sus siglas en inglés) de un compilador y emitiendo un IF optimizado. Este nuevo IF puede ser convertido y enlazado en un código ensamblador dependiente de la máquina concreta para una plataforma objetivo. LLVM puede aceptar el IF generado por la cadena de herramientas GCC, permitiendo así que sea utilizado con todos los lenguages que a su vez son aceptados por GCC. LLVM también puede generar código máquina relocalizable en el momento de compilación o de enlazado, o incluso código máquina binario en el momento de ejecución.

1.2.2. Verificación y Análisis

Actualmente el software es uno de los productos más requeridos en el mundo. Tal demanda de software implica procesos de desarrollo más intensos, exhaustivos y menos propensos a errores. Siendo la corrección de fallos la etapa en la que más recursos se invierten y en la que hay más interés por mejorar su eficiencia.

A día de hoy en el mercado existen paquetes de herramientas capaces de ofrecer a los desarrolladores detección de errores y estudios sobre la ejecución de sus programas. Existen varias maneras de verificar la correctitud de un programa.

Lógicas

Sistemas de tipos

Asignar tipos de datos (tipificar) da significado a colecciones de bits. Los tipos de datos normalmente tienen asociaciones tanto con valores en la memoria o con objetos como con variables. Al igual que cualquier valor consiste en un conjunto de bits de un ordenador, el hardware no hace distinción entre dirección de memoria, código de instrucción, carácteres, enteros y números en coma flotante. Los tipos de datos informan a los programas y programadores cómo deben ser tratados esos bits.

Al proceso de verificar e imponer los límites por los tipos de datos, conocido como comprobación o checkeo de tipificación. Este proceso puede ocurrir tanto en la compilación (comprobación estática) o en la ejecución (comprobación dinámica). La elección entre sistemas de tipificación dinámico y estático requiere algunas contraprestaciones:

- El **tipado estático** busca errores en los tipos de datos durante la compilación. Esto debería incrementar la fiabilidad de los programas procesados. Sin embargo, los programadores, normalmente, están en desacuerdo en cómo los errores de tipos de datos más comunes ocurren, y en qué proporción de estos errores que se han escrito podrían haberse cazado con un tipado estático. Los defensores de los lenguajes fuertemente tipados han sugerido que casi todos los errores pueden ser considerados errores de los tipos de datos.

- El **tipado dinámico** permite a los compiladores e intérpretes ejecutarse más rápidamente, debido a que los cambios en el código fuente en los lenguajes dinámicamente tipados puede resultar en menores comprobaciones y menos código que revisar. Esto también reduce el ciclo editar-compilar-comprobar-depurar. El tipado dinámico típicamente hace que la metaprogramación sea más poderosa y fácil de usar.

Análisis estático

El análisis estático de software es un tipo de análisis que se realiza sin ejecutar el programa. En la mayoría de los casos se realiza sobre el código fuente y en otros casos se realiza sobre el código objeto.

Dicho término se aplica a los análisis realizados por parte de una herramienta automática sobre el programa sin que este se ejecute. La industria ha reconocido los métodos de análisis estático como elementos clave a la hora de mejorar la calidad de programas complejos. Un uso comercial creciente del

análisis estático es la verificación de las propiedades de software utilizadas en sistemas informáticos críticos para la seguridad y la localización de código vulnerable, pero también se incluyen una viariedad de métodos formales que verifican ciertas propiedades del programa. Por ejemplo, el uso de este tipo de análisis está muy extendido en los campos de la medicina, la aviación o la energía nuclear, donde no se pueden permitir errores ni riesgos.

Chequeo de modelos (Model checking)

La verificación de modelos (o Model checking) es un método automático de verificación de un sistema formal, en la mayoría de las ocasiones derivado del hardware o del software de un sistema informático. El sistema es descrito mediante un modelo, que debe satisfacer una especificación formal descrita mediante una fórmula, a menudo escrita en alguna variedad de lógica temporal.

El modelo suele estar expresado como un sistema de transiciones, es decir, un grafo dirigido, que consta de un conjunto de vértices y arcos. Un conjunto de proposiciones atómicas se asocia a cada nodo. Así pues, los nodos representan los estados posibles de un sistema, los arcos posibles evoluciones del mismo, mediante ejecuciones permitidas, que alteran el estado, mientras que las proposiciones representan las propiedades básicas que se satisfacen en cada punto de la ejecución.

Formalmente, el problema se representa de la siguiente manera: Dada una propiedad deseada, expresada como una fórmula en lógica temporal p, y un modelo M con un estado inicial s.

Los inventores del método, Edmund M. Clarke, E. Allen Emerson y Joseph Sifakis, recibieron el Premio Turing 2007 de la ACM, en reconocimiento de su fundamental contribución al campo de las ciencias de la computación.

Ejecución simbólica

La ejecución simbólica o evaluación simbólica es un modo de analizar programas para determinar qué entradas causan cada parte del programa a ejecutar. En este tipo de programas, un intérprete recorre el programa donde cada variable asume un valor simbólico acotado. Este recorrido dará como resultado expresiones en términos de dichos símbolos de expresiones y variables y las limitaciones en cuanto a los símbolos de los posibles resultados de cada rama condicional.

Dicho recorrido nos permite determinar las condiciones que deben ser verificadas por los datos de ingreso para que un camino particular se ejecute, y la relación entre los valores ingresados y producidos en la ejecución de un

programa.

Este procedicimiento aplicado al hardware recibe el nombre de simulación simbólica.

1.2.3. Testing

Paralelamente a las herramientas de verificación y análisis nacen las pruebas de software(o testing en inglés) son las investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto a la parte interesada.

Estas pruebas se diferencian de las herramientas de verificación y análisis en que las pruebas garantizan la correctitud de los programas para esos test, pero es imposible que dichos test abarquen todas las posibilidades o ramas de ejecución de un programa, de tal modo que el programa podría no estar correcto para todas sus posibilidades como si garantiza la verificación y el análisis. A continuación detallamos algunas de estas herramientas que ayudan a automatizar distintas fases del proceso de testing.

Frameworks xUnit

Durante su trabajo en los laboratorios de Xerox Park, Kent Beck desarrollo un framework para realizar test unitarios en el lenguaje de programación *Smalltalk* que estaba desarrollando. Dicho framework se llamaba sUnit. sUnit Permitía agrupar los tests en suites, el lanzamiento jerárquico de éstos y realizar para cada test la comparación entre el valor obtenido y el valor esperado para determinar la corrección de la funcionalidad testeada. Este framework ha sido llevado a una gran variedad de lenguajes de programación (*JUnit*, *XSLTUnit*, *PhpUnit*, *PyUnit*/ldots). xUnit es el nombre con el que se designa genéricamente a todos estos frameworks.

Una de las características de los tests xUnit es la expansión y el alto grado de utilización de esta herramienta por todo el mundo.

Generación de tests

El objetivo de las pruebas es presentar información sobre la calidad del producto a las personas responsables, por tanto la información requerida puede ser de lo más variada. Esto hace que el proceso de testing sea dependiente del contexto, no existen las *mejores pruebas*, toda prueba puede ser ideal para una situación o completamente inutil o perjudicial para otra. A continuación detallamos algunas de los diferentes enfoques para la generación de tests:

- **Pruebas de caja blanca** (white-box). Estas pruebas se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El testeador escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución y cerciorarse de que se devuelven los valores de salida adecuados. Puesto que están basados en una implementación concreta, si esta se modifica, normalmente también tendrán que rediseñarse las pruebas. A pesar de que este enfoque permite diseñar pruebas que cubran una amplia variedad de casos podría pasar por alto partes incompletas de la especificación o requisitos faltantes.

- **Pruebas de caja negra** (black-box). Estas pruebas se centran en el estudio de las entradas y las salidas generadas, independientemente de su funcionamiento interno, es decir, nos interesa su forma de interactuar entendiendo *qué es lo que hace* pero sin importarnos el *cómo lo hace*. Unas pruebas de test basadas en este sistema será más fácil de entender ya que permitirá dar una visión más clara del conjunto, el sistema también será más robusto y fácil de mantener, en caso de fallo este podrá ser aislado y abordado más ágilmente.

- **Testing aleatorio** o random. Buscando ampliar el ámbito de pruebas de unidad, se han aplicado diversas técnicas que van desde la automatización de pasos, hasta los enfoques de generación de objetos de manera aleatoria. En este caso concreteo, la generación de tests de forma aleatoria es un caso particular de las pruebas de caja negra, donde las pruebas se realizan con entradas generadas de manera aleatoria e independiente. Los resultados se comparan con las especificaciones del software en prueba para verificar si los resultados pasan el test o no.

1.2.4. Programación con restricciones

La programación con restricciones es un paradigma de la programación en informática, donde las relaciones entre las variables son expresadas en terminos de restricciones o ecuaciones. Este paradigma representa uno de los mayores desarrollos en los lenguajes de programación desde 1990 y ha sido identificada como una dirección estratégica en la investigación en computación.

Se trata de un paradigma de programación basado en la especificación de un conjunto de restricciones las cuales deben ser satisfechas por cualquier solución del problema planteado, en lugar de especificar los pasos para obtener dicha solución. El enfoque de este tipo de programación se basa principal-

mente en especificar un estado en el cual una gran cantidad de restricciones sean satisfechas simultaneamente. Un problema se define típicamente como un estado de la realidad en el cual existe un número de variables con valor desconocido. Un programa basado en restricciones busca dichos valores para todas las variables.

Actualmente existen muchos frentes de desarrollo relacionados con la programación con restricciones. Entre ellos destacan:

- Oz: Lenguaje multiparadigma y esotérico basado en la rama concurrente de la programación por restricciones. En el que se expresa música a partir de unas restricciones expresadas explícitamente por el programador. Se utiliza en proyectos tales como Mozart y Strasheela.

- Choco: Es una librería que añade satisfacción de restricciones a Java. Está construida en una estructura basada en la propagación de eventos. Ha sido utilizada en otros proyectos de ejecución simbólica como JsyX.

- Gecode: Es un proyecto abierto que cuenta con un conjunto de herramientas basado en C/C++ para el desarrollo de sistema y aplicaciones nativas que se apoyen en restricciones.

La programación con restricciones se relaciona mucho con la programación lógica, de hecho cualquier programa lógico puede ser traducido a un programa con restricciones y viceversa, en muchas ocasiones los programas lógicos son traducidos a programas con restricciones puesto que la solución es mucho más eficiente.

Los lenguajes de programación con restricciones son típicamente ampliaciones de otro lenguaje. El primer lenguaje utilizado a tal efecto fue Prolog. Por esta razón este campo fue llamado inicialmente Programación Lógica con Restricciones.

La programación lógica con restricciones es un caso particular de la programación con restricciones, en el cual la programación lógica se extiende para incluir conceptos de la programación con restricciones. Por tanto, un programa lógico con restricciones es un programa lógico que contiene restricciones en el cuerpo de sus claúsulas. Por ejemplo, la claúsula *A(X,Y) :- X+Y>0, B(X), C(Y)* es un ejemplo de una cláusula con restricciones donde *X+Y>0* es la restriccion y *B(X), C(Y)* son literales al igual que en programación lógica.

Al igual que en la programación lógica, los programas se ejecutan buscando demostrar un objetivo que puede contener restricciones además de los literales. Una prueba para un objetivo se compone de claúsulas cuyos cuerpos satisfacen las restricciones y literales que se pueden probar usando otras

claúsulas. La ejecución se realiza a través de un intérprete, que empieza desde un objetivo y recursivamente escanea el resto de claúsulas para probar el objetivo.

1.2.5. Trabajos relacionados

Como hemos mencionado anteriormente, la ejecución simbólica no es algo nuevo y ha adquirido protagonismo en los últimos años. Es por esto que podemos encontrar varios proyectos orientados al testing de diversos estilos, aquí describimos algunos de ellos para contextualizar nuestro proyecto.

1.2.5.1. PEX

PEX es una herramienta desarrollada por Microsoft para la generación automática de casos de prueba. Esta herramienta genera entradas de test para programas .NET, por tanto puede analizar cualquier programa que se ejecute en una máquina virtual .NET y soporta lenguajes como C#, Visual Basic y F#.

El enfoque de PEX se caracteriza por implementar la ejecución dinámica simbólica, pero también permite el uso de ejecucución concreta de valores para simplificar las restricciones, las cuales serían resueltas por el resolutor SMT. Por tanto, mediante un proceso iterativo, PEX realiza una ejecución concreta del método a analizar y examina la traza de ejecución buscando ramas no exploradas. Una vez ha encontrado una rama no explorada, usa la ejecu-ción simbólica y un sistema resolutor de restricciones para generar valores concretos que exploren dicha rama. Este proceso se repite hasta obtener el recubrimiento deseado.

El hecho de combinar la ejecución simbólica con la concreta permite en muchos casos incrementar la escalabilidad y tratar con situaciones donde la ejecución no depende sólo del código sino de factores externos. Se entiende por factores externos el uso de librerías nativas, llamadas al sistema operativo o interacciones con el usuario. Ante este tipo de situaciones, la ejecución simbólica presenta grandes limitaciones y en general, no se puede aplicar.

Esta herramienta se puede integrar en el entorno de desarrollo de Visual Studio facilitando su uso como un añadido (Addon).

Como proyecto interesante relacionado con PEX existe el juego Code Hunt también desarrollado por Microsoft. En este juego el jugador debe escribir código para avanzar en el juego. La relacióncon PEX es que lo que se muestra al jugador son las entradas y salidas, y mediante ejecución simbólica se evaluará el código escrito y de este modo poder avanzar.

1.2.5.2. PET

PET (Partial Evaluation-based Test Case Generator for Bytecode) es una herramienta cuyo propósito es generar casos de prueba de forma automática para programas escritos en bytecode (código de bytes) de Java. PET adopta el enfoque previamente comentado, esto es, ejecuta el bytecode simbólicamente y devuelve como salida un conjunto de casos de prueba (test-cases). Cada caso de prueba está asociado a una rama del árbol y se expresa como un conjunto de restricciones sobre los valores de entrada y una descripción de los contenidos de la memoria dinámica (o heap).

Las restricciones de la memoria dinámica imponen condiciones sobre la forma y contenidos de las estructuras de datos del programa alojadas en esta misma. PET utiliza de un resolutor de restricciones que genera valores concretos a partir de estas restricciones, permitiendo la construcción de los tests propiamente dichos.

PET puede usarse a través de una interfaz de línea de comandos o bien usando una interfaz web. Además soporta una variedad de opciones interesantes, como la elección de criterios de recubrimiento o la generación de tests jUnit. Estas opciones se describen con más detalle en el segundo capítulo.

1.2.5.3. jSYX

jSYX es un proyecto desarrollado el curso pasado como trabajo de Sistemas Informáticos. Este proyecto se basa en una máquina virtual de Java que permite la ejecución simbólica de archivos .class de Java, de este modo puede ser utilizado para el desarrollo automático de Tests.

El enfoque de jSYX es el uso de del Bytecode de Java como lenguaje de ejecución. Esta herramienta permite la ejecución recibiendo como parámetros una clase y su método, y de este modo permite la ejecución simbólicamente o de manera concreta. Por otra lado se podrá obtener información sobre el bytecode del archivo .class.

1.2.5.4. jPET

Al igual que el detallado en la sección anterior, jPET también se trata de un proyecto de Sistemas Informáticos, en este caso desarrollado durante el curso 2010/2011. Este proyecto se basa en PET, la herramienta detallada anteriormente, y es una extensión de dicha herramienta para poder utilizarse en programas Java de alto nivel e integrarse en Eclipse, con el objetivo de poder usar los resultados obtenidos por PET durante el proceso de desarrollo de software.

El enfoque de jPET es el tratamiento de la información generada por PET con el objetivo de presentarla al usuario de una manera más fácil, sencilla e intuitiva. Es por esto que incorpora un visor de casos de prueba para mostrar el contenido de memoria antes y después de la ejecución, una herramienta de depuración, para ver los resultados mostrando la secuencia de instrucciones que el caso de prueba ejecutaría, y es capaz de analizar sintácticamente precondiciones de métodos para evitar la generación de casos de prueba poco interesantes.

1.3. Objetivos, la herramienta SymC++

En este proyecto se estudiará y desarrollará una herramienta de "depuración simbólica" que permitirá estudiar el funcionamiento de programas informáticos sin ser ejecutados, a base de observar todas sus posibles ramas de ejecución (hasta un cierto nivel) así como los correspondientes pares entrada-salida. Dicha herramienta podría ayudar enormemente a los estudiantes de iniciación a la programación, y en general a programadores inexpertos, a la hora de razonar acerca de la corrección de sus programas.

Sólo se utilizará como forma de comprobación, no como solucionador de errores ni sintácticos ni semánticos, es decir, no realiza la función de compilador; mostrará los resultados de la ejecución para ciertas posibles entradas para comprobar si ésa es la solución que esperabas, de no ser así, el alumno sabrá que el código estará mal planteado. Los alumnos que están empezando a programar en C++ saben si su programa compila gracias al entorno que estén utilizando pero les es más difícil de comprobar si su programa está correcto sin la ayuda de una interfaz o de algúna otra herramienta. Ahí es donde entra nuestra herramienta.

La herramienta contará con una interfaz de usuario, proporcionando varias ventajas adicionales: el alumno podrá ver y actualizar cómodamente, en todo momento, el código que desee probar, podrá tanto escribir un código desde cero como importar un código ya creado, por otro lado, la herramienta marcará el recorrido que se ha realizado de forma que el alumno podrá corregir posibles errores más rápidamente, por ejemplo, al encontrar un recorrido que no era el deseado.

Para ilustrar mejor el funcionamiento de nuestra herramienta detallamos aquí un pequeño ejemplo para una función cuya estructura es un pequeño decodificador:

Código 1.1: Código de ejemplo

```
1  function deco(int a, int b){
2     if (a>0)
3        if (b>0)
4           return 0;
5        else
6           return 1;
7     else
8        if (b>0)
9           return 2;
10       else
11          return 3;
12 }
```

La ejecución del programa mostrado en el código anterior en nuestra herramienta generaría la siguiente tabla de resultados:

nÂº	Variable y valor	Traza
1	a=1 b=1 return=0	1,2,3
2	a=1 b=0 return=1	1,4,5
3	a=0 b=1 return=2	6,7,8
4	a=0 b=0 return=3	6,9,10

Tabla 1.1: Tabla de resultados para el código ejemplo

Esta tabla de resultados muestra en la primera columna el número correspondiente a la solución, en la segunda columna se muestra el valor de las variables de la solución, las cuales hacen referenciaa las variables de entrada de la función y la variable de salida o return, y en la tercera columna se muestra la traza de ejecución, o lo que es lo mismo, los números de línea que ha recorrido el programa para llegar a dicha solución. Por tanto, para el primer resultado, las entradas valen en este caso *"1, 1"* haciendo referencia a que son valores mayores que 0, y el valor del return es el valor de salida, por otro lado los números que indican la traza son los números de las líneas que se han recorrido del programa para esta solución, en este caso *"1, 2, 3"*, y asÃ sucesivamente para lo otros tres resultados.

Para llegar a esta solución, hemos desarrollado la herramienta denominada *SymC++*, el nombre surge de la relación con la ejecución simbólica y por estar desarrollada pensando en el lenguaje de programación C++.

Dicha herramienta hará uso de tres componentes principales, cuyo ciclo de ejecución detallamos en la siguiente figura:

Por tanto como se puede observar en la figura anterior, la herramienta SymC++ consta de tres partes diferenciadas:

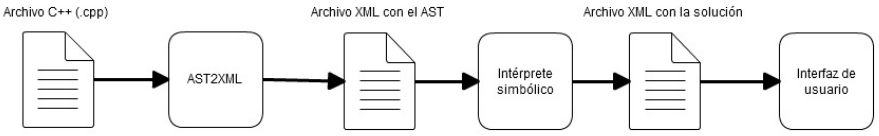

Figura 1.1: Figura que muestra el ciclo de ejecución de nuestra herramienta.

- **Ast2xml**: una herramienta desarrollada a través de Clang para obtener el arbol de sintaxis anotado con la información que nos interesa, la cual detallamos en el siguiente capítulo.
- **Intérprete simbólico**: la parte central del proyecto, es decir, el intérprete prolog que se encarga de obtener las posibles soluciones para cada una de las ramas del programa, así como la traza de ejecución a través de sus entradas y salidas, esta parte la detallaremos en el tercer capítulo.
- **Interfaz de usuario**: funciona como enlace entre las dos partes nombradas anteriormente y el usuario y la detallaremos en el cuarto capítulo de esta memoria.

Capítulo 2

Herramienta de Clang

La herramienta ast2xml conforma junto con el xmlinterpreter el cuerpo central del proyecto. Si bien ast2xml no cumple el papel más importante, suple las necesidades básicas de obtener a partir del código fuente una representación más manejable y sencilla de visualizar. Ha sido desarrollada a partir del proyecto abierto e internacional LLVM, pero se apoya principalmente en su compilador Clang.

2.1. AST2XMLtool

Al inicio del curso nos dimos cuenta de que nos podíamos aproximar al problema de distintas formas. Si bien, de una forma similar al proyecto Jsyx teniamos la posibilidad de basar nuestro proyecto en el uso de algo similar al bytecode de java pero relacionado con C++.

2.2. Historia de LLVM

LLVM nació como proyecto en el año 2000 en la Universidad de Illinois en Urbana-Champaign. En 2005, junto con Apple Inc., se empezó a adaptar el sistema para varios usos dentro del ecosistema de desarrollo de Apple. Actualmente LLVM está integrado en las últimas herramientas de desarrollo de Apple para sus sistemas operativos.

El un principio "LLVM" eran las iniciales de "Low Level Virtual Machine" (Máquina Virtual de Bajo Nivel), pero esta denominación causó una confusión ampliamente difundida, puesto que las máquinas virtuales son solo una de las aplicaciones de LLVM. Cuando la extensión del proyecto se amplió incluso más, LLVM se convirtió en un proyecto que engloba una gran varie-

dad de otros compiladores y tecnologías de bajo nivel. Por tanto, el proyecto abandonó las iniciales y actualmente, LLVM es la manera de referirse a todas esas utilidades.

El amplio interés que ha recibido LLVM ha llevado a una serie de tentativas para desarrollar front-ends totalmente nuevos para una variedad de lenguajes. El que ha recibido la mayor atención es Clang, un nuevo compilador que soporta otros lenguajes de la familia de C (Objective-C, C++, etc..). Apoyado principalmente por Apple, se espera que Clang sustituya al compilador del sistema GCC.

2.3. ¿Cómo funciona Clang?

En la mayoría de los casos Clang ejecutará el preprocesado y parseará el código formando un árbol de sintáxis abstracta. Esta estructura de árbol es más manejable que el propio código con el añadido de que cualquier subestructura mantiene referencias al código fuente.

2.4. Usando el árbol de sintaxis abstracta

La herramienta AST2XML se apoya en la adaptación del código, por parte de Clang, en su arbol de sintaxis abstracta (Abstract Syntax Tree). La representación del AST que emplea Clang se diferencia de la de otros compiladores en cuanto a que mantiene la semejanza con el código escrito y con el estándar de C++.

Un árbol de sintaxis abstracta (AST) es una representación de árbol de la estructura sintáctica abstracta (simplificada) del código fuente ampliamente utilizada en los distintos compiladores del mercado. Cada nodo del árbol denota una construcción que ocurre en el código fuente. La representación del AST que emplea Clang se diferencia de la de otros compiladores en cuanto a que mantiene la semejanza con el código escrito y con el estándar de C++.

Esta estructura es obtenida mediante el uso de las distintas utilidades incluidas en el apartado de desarrollo de Clang. Mediante dichas facilidades se recorre el AST de forma recursiva conservando los detalles más importantes del código en un fichero que se devuelve al usuario.

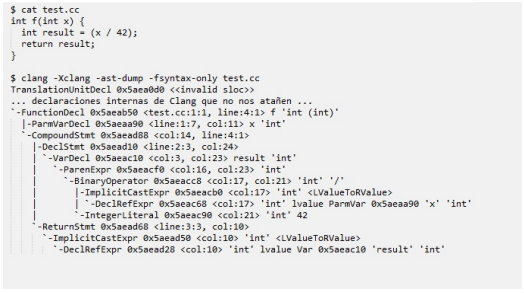

Figura 2.1: Figura Ejemplo del AST que emplea Clang a partir de un código específico.

2.5. Manipulación de un AST con clang

Como se puede apreciar en la figura, la representación que se obtiene directamente de clang puede llegar a parecer abrumadoramente compleja y redundante. En momentos iniciales del proyecto tomamos la decisión de que información como la ubicación de los elementos, los tipos de las variables, o incluso el tipo de los nodos no serían necesarias en cada momento. Decidimos pues optar por una representación más acotada, clara y concisa que, expresando lo mismo, fuera lo suficientemente sencilla de manipular por el intérprete de prolog.

2.5.1. LibTooling

Es una librería que tiene un papel muy importante en la arquitectura de la herramienta. Libtooling da soporte al diseño de herramientas autónomas basadas en Clang, proveyendo de la infraestructura necesaria para la realización de análisis sintácticos y semánticos de programas. En la implementación de nuestra herramienta destacan los roles de estos elementos que nos aporta la librería:

- CommonOptionParser: Es una utilidad que se encarga del análisis de los argumentos que recibe la llamada del ejecutable ast2xml. Aporta los mensajes por defecto de las utilidades básicas de las herramientas que se pueden implementar.
- ToXMLVisitor: Es el núcleo de la herramienta. Hereda de la interfaz RecursiveASTVisitor, que como su nombre indica, se encarga de hacer un recorrido recursivo y en profundidad de la estructura. Dentro de

nuestro visitor hay que implementar los diferentes métodos específicos que se amoldan a los distíntos nodos que vaya a visitar dentro del AST.

- ToXMLASTConsumer: Es la clase que desencadena la ejecución del visitor.
- ToXMLFrontendAction: Es la clase que encapsula el comportamiento de nuestra herramienta. Hereda de la clase abstracta ASTFrontendAction del paquete FrontendAction. Devuelve una instancia de nuestro consumidor en el momento de comenzar a recorrer el AST.

2.6. Estructura del AST

La estructura que emplea Clang para representar el AST de un código se basa principalmente en la interacción de dos clases base muy flexibles a partir de las cuales se construyen todas los demás: Decl (Declarations), que engloba toda las declaraciones (funciones, variables, templates, etc/ldots),Stmt (Statement) que abarca las instrucciones.

La mayoría de las clases que se derivan de ellas se explican por sí mismas, como por ejemplo: BinaryOperator (operador binario), FunctionDecl (declaración de función), etc/ldots

2.7. Recorriendo el AST

El recorrido que hace el RecursiveASTVisitor a lo largo del AST se realiza mediante llamadas a funciones Traverse. Estas funciones son específicas para cada clase que funciona como nodo del AST, aunque existen sus versiones genéricas: TraverseDecl() y TraverseStmt().

El comportamiento de ambas es muy similar.

2.8. Instrucciones como nodos de un fichero XML

El fichero que se le devuelve al usuario tiene formato XML y contiene información acerca de las funciones que, en el código fuente, haya declarado el usuario. Dichas funciones se encuentran englobadas en nodos function que almacenan el nombre, el número de línea, el tipo devuelto, los parámetros y el cuerpo de la función en cuestión. De los parámetros registramos su tipo y el nombre de su declaración.

```
<function name="foo" type="int" line="3">
    <params>
        <param type="int" name="a"/>
        <param type="int" name="b"/>

    </params>
    <body>
        ......
    </body>
</function>
```

Figura 2.2: Figura con ejemplo de la disposición básica de un nodo function.

En el desarrollo del proyecto hemos tenido que, de la misma forma, asignar a cada declaración, instrucción y expresión de C++ contempladas en nuestro dominio un nodo XML que mantuviera su significado y de la misma forma no incrementara la complejidad del texto.

2.9. Bloques de instrucciones

Una noción que existe en el AST de Clang es la del CompoundStatement que representa a un bloque de código delimitado por llaves (..). Puesto que esperamos que el usuario haga uso de ellos sólo en compañia de otras estructuras (if, while, for o funciones), en el XML se representa con el nodo body. Que contiene los nodos de las instrucciones programadas.

Declaraciones e inicializaciones

C + + es un lenguaje fuertemente tipado, y requiere que cada variable esté declarada junto con su tipo antes de su primer uso. En un caso práctico, esto informa al compilador el tamaño reservar en memoria para la variable y la forma de interpretar su valor. Si se declaran varias variables del mismo tipo, se puede realizar en una sola instrucción.

Cuando se declara una variable, adquiere un valor indeterminado hasta que se le asigne alguno explícitamente. En C++ se contemplan tres maneras de inicializar el valor de una variable, aunque en nuestro proyecto, de cara a el uso de nuestra herramienta por parte de alumnos que están aprendiendo a programar, admitimos únicamente la forma más sencilla.

Un nodo declarations contiene las declaraciones de las variables de un mismo tipo que han sido declaradas en la misma línea. Obtienen su información del nodo del AST de clang DeclStmt.

Figura 2.3: Figura utilizada para marcar una imagen por hacer.

Figura 2.4: Figura utilizada para marcar una imagen por hacer.

Los nodos declaration contienen información acerca del tipo de la variable, su nombre, la línea en la que se encuentra la declaración y la expresión cuyo valor se le va a asignar. La información surge de los nodos VarDecl que emplea el AST de clang.

Estructuras de control

Las instrucciones que producen ramificaciones en la ejecución de un programa, tales como: el if, el while o el for. Para poder realizar su ejecución correctamente mantenemos en sus respectivos nodos sus señas más características.

El nodo de la instrucción if contiene el número de línea, y los nodos correspondientes a la condición que dirige su ejecución, al cuerpo del then y al cuerpo del else (en caso de estar especificado en el código).

La instrucción while se traduce a un nodo que alberga como atributo el

Figura 2.5: Figura utilizada para marcar una imagen por hacer.

Figura 2.6: Figura utilizada para marcar una imagen por hacer.

número de línea donde comienza la instrucción, y los nodos de la expresión de su condición y el bloque de instrucciones de su cuerpo. Toda esta información es una fracción de la que contiene el nodo del AST de clang WhileStmt.

La estructura del nodo de la instrucción for mantiene el número de línea, y en su interior los nodos de la declaración de su variable de control, la condición, la instrucción de avance y el cuerpo del bucle. Obtiene su información a partir del nodo del AST de clang ForStmt.

Expresiones

Las expresiones escritas en C++ se traducen a nuestro formato en XML en nodos descritos por sus operadores, la variable a la que hacen referencia o la constante que representan.

En caso de que estos nodos hagan referencia a un operador, contienen información acerca del tipo de este (binario, unario, asignación, etc/ldots) y

Figura 2.7: Figura utilizada para marcar una imagen por hacer.

de las expresiones que contienen.

Las expresiones que se refieren a variables almacenan el nombre de estas y de forma equivalente las que representan constantes guardan su valor. Un caso particular de expresión sería el de las llamadas a funciones.

Operadores de asignación

Para representar las instrucciones de asignación nos expresamos mediante los nodos assignment y assignmentOperation, conteniendo el nombre de la variable, la línea y la expresión de la cual se calcula el valor.

A pesar de que esta distinción no existe como tal en Clang, la hacemos evidente para diferenciar las asignaciones corrientes (AssignmentOperator en Clang) de las asignaciones con operadores (CompountAssignmentOperator en Clang).

Esta distinción se hace con los estudiantes noveles de programación en mente con la finalidad de que, en caso de estudiar detenidamente el XML final de su código, tengan una visión más clara de su funcionamiento.

Operadores unarios

En C++ existen numerosos operadores unarios pero, considerando los conocimientos que hacen falta en etapas tempranas de la iniciación a la programación, nos hemos centrado en dar soporte a los operadores unarios de incremento, decremento, indicador de signo y el not lógico.

Distinguimos los operadores de incremento y decremento se presentan a los programadores en un principio como parte esencial de los bucles cum-

Figura 2.8: Figura utilizada para marcar una imagen por hacer.

Figura 2.9: Figura utilizada para marcar una imagen por hacer.

pliendo la función de instrucción de avance.

Por otro lado el indicador de signo cumple un papel importante a la hora de operar con valores negativos y de ayuda explícita al programador junior en caso de que quiera asegurar el valor de su variable.

El not lógico no aporta más expresividad pero permite simplificar programación de expresiones booleanas.

Figura 2.10: Figura utilizada para marcar una imagen por hacer.

Capítulo 3

Intérprete con restricciones

En este capítulo hablaremos sobre la parte principal de nuestro proyecto, el Intérprete Simbólico, que recibe el árbol sintáctico generado por la herramienta AST2XML, realiza una ejecución simbólica de éste devolviendo un conjunto de posibles valores de entrada con los correspondientes valores de salida generados.

3.1. El lenguaje Prolog y la librería clpfd

La Programación Lógica tiene sus orígenes más cercanos en los trabajos de prueba automática de teoremas de los años sesenta. J. A. Robinson propone en 1965 una regla de inferencia a la que llama resolución, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática. La resolución es una regla que se aplica sobre cierto tipo de fórmulas del Cálculo de Predicados de Primer Orden, llamadas cláusulas y la demostración de teoremas bajo esta regla de inferencia se lleva a cabo por reducción al absurdo. La realización del paradigma de la programación lógica es el lenguaje Prolog.

Prolog es un lenguaje de programación ideado a principios de los años 70 en la Universidad de Aix-Marseille I (Marsella, Francia). No tiene como objetivo la traducción de un lenguaje de programación, sino la clasificación algorítmica de lenguajes naturales. [1]

La idea es especificar formalmente los enunciados utilizando la lógica de predicados de manera que Prolog sea capaz de interpretar e inferir soluciones a partir de esos enunciados.

La lógica de predicados estudia las frases declarativas teniendo en cuenta la estructura interna de las proposiciones, es decir, un conjunto de cláusulas de la forma: “q :- p”, en la que si p es cierto entonces q es cierto. Una cláusula

pueden ser un conjunto de hechos positivos, por ejemplo de la forma "q :- p, r, s." ; una implicación con un único consecuente, "q :- p" ; un hecho positivo, "p." ; instrucciones con parámetros de Entrada/Salida "p(Entrada,Salida)." ; o incluso llamadas a otras funciones.

En Prolog, los predicados se contrastan en orden, la ejecución se basa en dos conceptos: la unificación y el backtracking. Una vez ejecutamos una función de Prolog se sigue ejecutando el programa gracias a las llamadas a predicados, si procede, hasta determinar si el objetivo es verdadero o falso. Todos los objetivos terminan su ejecución en éxito ("verdadero"), o en fracaso ("falso"). Si el resultado es falso entra en juego el backtracking, es decir, deshace todo lo ejecutado situando el programa en el mismo estado en el que estaba justo antes de llegar al punto de elección, ahí se toma el siguiente punto de elección que estaba pendiente y se repite de nuevo el proceso, de ahí la utilidad de prolog en nuestro proyecto ya que nos permite implementar un intérprete de ejecución simbólica de forma natural y prácticamente nativa.

La Programación por restricciones es un paradigma de la programación en informática donde las variables están relacionadas mediante restricciones (ecuaciones). Se emplea en la descripción y resolución de problemas combinatorios, especialmente en las áreas de planificación y programación de tareas (calendarización). La programación con restricciones se basa principalmente en buscar un estado en el cual una gran cantidad de restricciones sean satisfechas simultáneamente, expresándose un problema como un conjunto de restricciones iniciales a partir de las cuales el sistema construye las relaciones que expresan una solución.

Como ya se introdujo en el estado del arte, la programación con restricciones proporciona una herramienta en la que las variables están expresadas en términos de restricciones o ecuaciones. Para el desarrollo de esta herramienta hemos necesitado hacer uso de este paradigma de programación, el cual nos facilitaba mostrar restricciones como datos de salida.

La librería clpfd "Constraint Logic Programming over Finite Domains" (Programación lógica basada en restricciones sobre dominios finitos) proporciona una aritmética completa para variables restringidas a valores enteros o elementos atómicos, se puede utilizar para modelar y resolver diversos problemas combinatorios tales como las tareas de planificación, programación y asignación.

Para el desarrollo de nuestro intérprete, como ya indicamos anteriormente, optamos por el lenguaje de programación lógica Prolog, en gran parte debido a la familiaridad que nos supone, y al conjunto de facilidades relacionadas con el backtracking y la programación por restricciones que ya conocíamos de antemano.

3.2. Diseño e interfaz del intérprete

Como ya hemos indicado antes, AST2XML devuelve un archivo en formato XML con una versión simplificada del árbol sintáctico que emplea Clang con el que realizaremos la ejecución simbólica. Para llevar a cabo una ejecución simbólica es necesario tener un sistema capaz de establecer restricciones lógicas y matemáticas y que aparte las pueda resolver.

El intérprete está encapsulado en dos módulos para hacer más fácil su diseño: *VariablesTable.pl* e *Interpreter.pl*.

VariablesTable.pl

El intérprete irá guardando las variables y su respectivo valor en una tabla de variables representado en prolog mediante una lista, de forma que todas las posibles operaciones que se puedan aplicar sobre ella estén encapsuladas en un mismo módulo. Por ejemplo, funciones como añadir un elemento a la tabla (add), obtener el valor de una variable (getValue), modificar el nombre de una variable (updateNames), etc. son funciones que sólo se aplican sobre la tabla de variables.

Interpreter.pl

Por otro lado, este módulo representa el Intérprete que será llamado por la interfaz y el que devolverá la solución. Los parámetros de entrada son: fichero de entrada, fichero de salida, Inf, Sup, MaxDepth y nombre de función.

Fichero de entrada

Indica el nombre del fichero .xml donde está el código traducido por clang.

Código 3.1: Ejemplo de XML de entrada que recibe el intérprete

```
1  <function name="llamadaFactorial" type="int" line="79">
2     <params>
3        <param type="int" name="a"/>
4     </params>
5     <body>
6        <return line="80">
7           <callFunction name="factorial" type="int">
8              <arg>
9                 <variable name="a"/>
10             </arg>
11          </callFunction>
```

```
12          </return>
13       </body>
14    </function>
```

Fichero de salida

Indica el nombre del fichero .xml de salida que el intérprete devolverá con las soluciones para una determinada función.

Código 3.2: Ejemplo de XML obtenido por Clang

```
1  <caso>
2     <variable name="ret" value="-1"/>
3     <variable name="a" value="-5"/>
4     <data>
5        <traza> 76 77 83</traza>
6        <cin/>
7        <cout/>
8     </data>
9  </caso>
```

Inf, Sup y MaxLoop

Inf y Sup son los valores que representan el límite del dominio de los valores de entrada de la función a interpretar. MaxLoop representa el valor del número máximo de veces que puede ejecutarse una estructura de control: while y for.

Nombre de la función

Para indicar el nombre de la función que vamos a querer probar entre todas las funciones posibles contenidas en el fichero pasado en "Fichero de entrada".

3.3. Ciclo de ejecución

Como forma explicativa del ciclo de ejecución que sigue el intérprete pondremos el siguiente ejemplo en el de devolvemos el resultado obtenido por los dos parámetros de entrada:

Código 3.3: Ejemplo ilustrativo con función suma

```
1  int foo(int a, int b){
```

```
2     int c = a + b;
3     return c;
4  }
```

Como ya sabemos, el intérprete recibirá un fichero XML como parámetro de entrada, junto con otros parámetros que explicaremos más adelante. El código de nuestro ejemplo tendrá el siguiente aspecto una vez pasado por AST2XML:

Código 3.4: Árbol sintáctico de la función suma

```
1  <function name="foo" type="int" line="3">
2     <params>
3        <param type="int" name="a"/>
4        <param type="int" name="b"/>
5     </params>
6     <body>
7        <declarations>
8           <declaration type="int" name="c" line="4">
9              <binaryOperator type="arithmetic" operator="+">
10                <variable name="a"/>
11                <variable name="b"/>
12             </binaryOperator>
13          </declaration>
14       </declarations>
15       <return line="6">
16          <variable name="c"/>
17       </return>
18    </body>
19 </function>
```

El primer paso es la conversión de la estructura XML en una lista de elementos mediante la función load_xml_file(+File,-DOM) que nos aporta la librería sgml. Para nuestro ejemplo, le pasaremos por el parámetro File el XML con el código anterior y nos devolverá en el parámetro de salida DOM una lista de elementos de la forma:

Código 3.5: Elementos correspondientes a la función suma

```
1  [element(function,[name=foo,type=int,line=3],[
2     element(params,[],[
3        element(param,[type=int,name=a],[]),
4        element(param,[type=int,name=b],[])
5     ])
6     element(body,[],[
7        element(declarations,[],[
8           element(declaration,[type=int,name=c,line=4],[
```

```
9               element(binaryOperator,[type=arithmetic,
                    operator=(+)],[
10                  element(variable,[name=a],[]),
11                  element(variable,[name=b],[]),
12              ])
13          ])
14      ])
15      element(return,[line=6],[
16          element(variable,[name=c],[])
17      ])
18  ])
19 ])]
```

Cada element tiene tres argumentos: el nombre del nodo (*nombreNodo*), los atributos del nodo (*atributosNodo*) y el cuerpo del nodo (*cuerpoNodo*).

<nombreNodo atributosNodo>
// cuerpoNodo
</nombreNodo>

Otro parámetro que recibe el intérprete es el nombre de la función que queremos probar, tenemos que indicarlo puesto que en un mismo XML podremos tener más de una función. De esta forma buscaremos previamente la función que el usuario nos haya indicado e iremos pasando por cada una de las instrucciones del código de dicha función y ejecutándolas mediante las sucesivas llamadas a la función execute que se detallará más adelante.

El resto de parámetros que el intérprete recibirá son:

- **Outfile**: para indicar el nombre del archivo en el que se guardará el resultado de la ejecución, tanto las posibles entradas como sus respectivas salidas generadas.
- **Inf, Sup**: parámetros que nos servirá para delimitar el valor posible de las salidas.
- **MaxDepth**: para poner un número máximo de vueltas que puede llegar a hacer un bucle si la condición sigue siendo cierta en el momento en el que se alcanza dicho valor.

execute/3

Es un predicado encargado de ejecutar una instrucción. Sus argumentos son: el estado previo a la ejecución de unas instrucciones, la lista que contiene las instrucciones y el estado posterior a la ejecución. Este predicado se planteó

de forma que pudiéramos controlar cada una de las posibles instrucciones que puede llegar a haber en el código. Para conseguirlo en cada una de estas posibilidades expresamos explícitamente los casos de forma que la instrucción que se ejecuta en un momento dado es el primer elemento de la lista de instrucciones que recibe la función. Una vez se hayan completado los pasos que simulan el comportamiento de dicha instrucción, se hace una "llamada" recursiva a execute con el estado resultante de la ejecución, el resto de la lista de instrucciones y el estado posterior.

Esta forma de realizar los pasos de cómputo se asemeja, en un aspecto más teórico, a la semántica de paso corto (semántica operacional), aunque dicha similitud no es completa puesto que al trabajar con instrucciones puntuales llevamos a cabo muchos pasos intermedios.

Es importante destacar que el bactracking inherente de prolog se restringe en el predicado *execute*. Esto se debe a que el orden de las instrucciones es siempre el mismo. En el intérprete hemos limitado el backtracking a la evaluación de condiciones puesto que éstas son las que realmente definen el flujo del programa. Es por ello que, en las instrucciones de control, llegamos a necesitar la presencia de otros predicados que, además de dirigir el comportamiento de la ejecución, permiten limitar el número de repeticiones que realiza un bucle.

Iremos almacenando en cada paso el estado actual donde nos encontramos con la siguiente información:

- **Table**.
  Llevamos la tabla de variables correspondientes al estado en el que nos encontramos. En esta tabla contemplamos la posiblidad de estructurarla en ámbitos de forma que el acceso a las variables sea consistente en cada momento.

- **Cin, Cout**
  Listas con los valores de entrada (Cin) o salida (Cout) si procede.

- **Trace**
  Lista que conserva el número de las líneas de las instrucciones por las que hemos pasado para poder más tarde marcarlas en la interfaz.

3.4. Funciones, declaración y asignación de variables.

Definición de una función

Una función se define siguiendo el siguiente esquema:

<tipo_valor_retorno><nombreFuncion>(<lista_parámetros>)
[sentencias]

Código 3.6: Instrucción para la definición de una función

```
execute(Entrada,[('function'...CuerpoFuncion)|
    RestoInstrucciones],Salida
```

Ya sea el caso de la función principal o una llamada a una función esta instrucción se encargará de ejecutar el cuerpo de la función y después el resto de las instrucciones. Diferenciamos también si se trata de una función o procedimiento ya que en el caso de las funciones tiene que existir un valor de retorno, al contrario que los procedimientos. Como vemos en el esquema anterior, la función puede llegar a contener **parámetros** de entrada que leeremos mediante la instrucción:

Código 3.7: Intrucción que recibe los parámetros de entrada

```
execute(Entrada,[('params'...Parametros)|RestoInstrucciones
    ],Salida
```

Con esta instrucción insertaremos en la tabla de variables cada uno de los parámetros que formen parte de la variable Parametros. Una vez actualizada la tabla llamaremos a ejecutar al resto de las instrucciones para continuar con el programa.

Un **bloque** es un mero conjunto de sentencias, que puede estar vacío, o encerradas por llaves , ese conjunto lo recogemos con la instrucción correspondiente:

Código 3.8: Intrucción para el cuerpo de un bloque

```
execute(Entrada,[('body'...Body)|RestoInstrucciones],Salida
    )
```

Este es el caso en el que ejecutamos cualquier tipo de bloque, ya sea el cuerpo de una función, if, for, while, etc. Llamamos a ejecutar al cuerpo de

forma que creamos a su vez una nueva lista de variables para englobar las variables pertenecientes a ese ámbito.

Para resolver el caso de una **llamada a función** el intérprete hará uso del precicado *callFunction*, que buscará la función a la que se está referenciando y la ejecutaremos de forma que consigamos el valor devuelto por la función como resultado de la expresión.

[tipo] <variable>= <nombreFuncion>(<lista_parámetros>) ;

El tipo: [tipo] será vacío en el caso en el que la variable ya hubiese sido declarada con anterioridad.

Declaración de una variable

A la hora de declarar una variable primero se especifica el tipo y a luego una lista de variables seguidas de un punto:

<tipo><lista de variables>;

La lista de variables tiene que estar formada como mínimo por una variable. Nunca podrá ser una lista vacía. El tipo será siempre int ya que es el tipo de variable que contempla nuestra herramienta. Ya que se pueden declarar una o varias variables en la misma instrucción, diferenciamos la posibilidad de guardar en la tabla de variables una sola variable:

Código 3.9: Instrucción para la declaracion de una variable

```
execute(Entrada,[('declaration'...Nombre...
    CuerpoDeclaracion)|RestoInstrucciones],Salida)
```

y la posibilidad de guardar en la tabla de variables varias declaraciones a la vez:

Código 3.10: Intrucción para las declaraciones de varias variables en grupo

```
execute(Entrada,[('declarations'...Body)|RestoInstrucciones
    ],Salida)
```

En este último caso, internamente, se llamará a la función encargada de guardar una a una las variables con sus tipos asignados pero sin valor ya que se trata únicamente de una declaración y no de una asignación.

A continuación se seguirá con la ejecución del resto de las instrucciones

Asignación de una variable

Las sentencias de asignación responden al siguiente esquema:

<variable><operador de asignación><expresión>;

Éste es el momento en el que se le da valor a una variable anteriormente declarada y por tanto ya añadida anteriormente a la tabla de variables. Ejecutaremos el cuerpo de la asignación para obtener el valor que tomará la variable y luego la actualizaremos. Las asignaciones pueden expresarse de dos formas distintas dependiendo de la forma en que se ponga el operador de asignación.

En el caso en el que el operador de asignación sea de la forma:

<variable>= <expresión>;

la asignación se realizará de forma que se resuelve la expresión y luego se le asigna el valor resultante a la variable.

En el caso en el que el operador de asignación sea de la forma:

<variable>op= <expresión>; siendo op = +,-,*,/

Éste es un caso especial en el que la asignación en este caso no es otra cosa que hacer:

<variable>= <variable>op <expresión>;

Se sigue la ejecución con la llamada de *execute* del resto de las instrucciones.

3.5. Instrucciones aritméticas y condicionales.

Este apartado explica los predicados que se utilizarán a la hora de resolver expresiones aritméticas y sentencia de selección if / if...else.

resolveExpression/4

Este predicado servirá para resolver las distintas expresiones que existan. La estructura de llamada de este predicado es la que sigue:

Código 3.11: Instrucción para la resolución de las expresiones

```
resolveExpression(Entrada,CuerpoExpresion,ValorRetorno,
    Salida),
```

Entrada y Salida corresponden a la tabla de variables que entra y la tabla de variables actualizada después de haberse resuelto la expresión. CuerpoExpresion es la expresión concreta que queremos resolver y ValorRetorno el valor resultante devuelto. En este apartado nos centraremos únicamente en las expresiones aritméticas

Operadores aritméticos

Los operadores se pueden clasificar atendiendo al número de operandos que afectan. Según esta clasificación tenemos tres tipos de operadores, pueden ser unitarios, binarios o ternarios. Los primeros afectan a un solo operando, los segundos a dos y los ternarios a tres.

Operadores unitarios

Código 3.12: Operador unario

```
execute(Entrada,[('unaryOperator'...Operando,Operador)|
    RestoInstrucciones],Salida)
```

Los tipos unarios que reconoce la herramienta siguen la estructura:

- +<**expresion**>-<**expresion**> Para asignar valores positivos o negativos a los valores a los que se aplica
- <**variable**>++ <**variable**>- - Para incrementar o decrementar el valor de la variable ambos en una unidad

Actualizamos el valor de la variable en la tabla de variables y continuamos con la ejecución del resto de las instrucciones.

Operadores binarios

Los operadores de tipo binario siguen el esquema:

<variable>op <expresión>; siendo op los operadores: +,-,*,/

En este caso se resolverá cada expresión de forma independiente mediante la llamada recursiva:

Código 3.13: Instrucciones para la resolución de expresiones con operadores binarios

```
resolveExpression(Entrada1, Operando1, Resultado1, Salida1)

resolveExpression(Entrada2, Operando2, Resultado2, Salida2)
```

Combinará los dos resultados únicamente cuando éstos son expresiones sencillas de forma que ya hayan llegado al caso base, que se cumple cuando se tiene que la expresión es una variable o una constante de forma que ya no se pueda más que operar mediante el predicado work detallado más adelante.

Operadores lógicos

Nos servirán a la hora de definir la condición de una instrucción de control de flujo.

Los operadores lógicos de negación siguen el esquema:

! <expresion>

De forma que negará el valor resultante de la expresión.

Existen también otros tipos de operadores lógicos que siguen el esquema:

<expresion>op <expresion>

Siendo op los valores && o || ambos se resuelven en el predicado work detallado a continuación.

Código 3.14: Asignación de una variable

```
execute(Entrada,[('assignmentOperator'...Name...Operator...
    Cuerpo)|RestoInstrucciones],Salida)
```

Este es un caso especial de la asignación ya que se trata de una asignación mediante un operador binario. Esta ejecución la haremos con la llamada a resolveExpression (explicada más adelante) del cuerpo de la asignación. Como en el resto de los casos pasamos a continuación a ejecutar el resto de las instrucciones para seguir con el flujo del programa.

work/3

Código 3.15: Instrucción para la ejecución de expresiones

```
work(Operador, Resultado1, Resultado2,ResultadoFinal),
```

Cuya funcionalidad es únicamente la ejecución de la expresión *expresión1* Operador *expresión2* dejando la solución en *ResultadoFinal*.

En nuestro proyecto "Operador" únicamente podrá ser:

<	<=	>	>=	==	!=	&&	||	+	-	*	/

Tabla 3.1: Tabla con los operadores que reconoce el interprete.

Instrucción condicional IF, IF...ELSE

Las sentencias de selección permiten controlar el flujo del programa, seleccionando distintas sentencias en función de diferentes circunstancias.

if (<condición>) <sentencias>
else <sentencias>

La estructura del *if...else* se puede ver fácilmente con el siguiente diagrama:

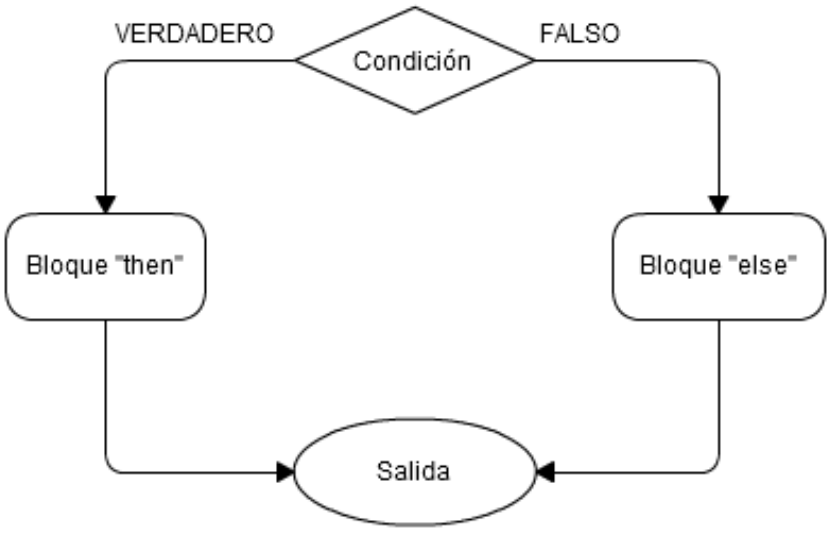

Figura 3.1: Figura que muestra la estructura del if...else.

El cuerpo del if y del else tiene que estar compuesto por una sentencia como mínimo, en el caso de que haya dos o más sentencias, éstas estarán delimitadas por llaves:

{ <sentencias>}

Se ejecutará el siguiente predicado cuando se trate de una instrucción condicional:

Código 3.16: Instrucción para la estructura condicional if

```
execute(Entrada,[('if'...Condicion...Bloque_1...Bloque_2)|
    RestoInstrucciones],Salida)
```

Primero se resuelve la condición y seguidamente se llamará a execute-Branch (detallada a continuación) con la solución obtenida, será ésta quien distinga si la condición ha salido verdadera o falsa. A continuación ejecutaremos el resto de las instrucciones para seguir con el resto del programa.

executeBranch/4

Es un predicado cuya funcionalidad extiende la del execute cuando se trata del if para el caso del then y del else. Igualmente recibe un estado de entrada y devuelve otro de salida, recibe la lista de instrucciones.

Código 3.17: Instrucción para la ejecución de estructuras condicionales

```
executeBranch(Entrada,[Then..Else..Body])|Instrucciones],
    ResulCond,Salida)
```

Aquí diferenciamos los dos casos posibles: *Then* y *Else*

- **Caso then:**
  Se ha cumplido la condición del if por lo que la variable ResulCond pasa a valer 1 realizando por tanto la ejecución del cuerpo del then y posteriormente se llamará a ejecutar al resto de las instrucciones.

  Código 3.18: Instrucción para la ejecución de la rama then

  ```
  executeBranch(Entrada,[...Then...)|RestoInstrucciones],1,
      Salida)
  ```

- **Caso else:**
  No se ha cumplido la condición del if por lo que la variable ResulCond pasa a valer 0, en este caso realizaremos la ejecución del cuerpo del else y posteriormente se llamará a ejecutar al resto de las instrucciones.

Código 3.19: Instrucción para la ejecución de la rama else

```
executeBranch(Entrada,[...Else...)|RestoInstrucciones],0,
    Salida)
```

3.6. Bucles y control de terminación.

Los bucles nos permiten realizar tareas repetitivas, y se usan en la resolución de la mayor parte de los problemas. En nuestro caso hablaremos únicamente del "while" y del "for". En ambos casos, el número de vueltas máximo que pueden llegar a hacer es el marcado por el valor MaxDepth, y se hará uso de él en el caso en el que la condición siga siendo cierta en el momento en el que se alcanza dicho valor, de esta forma controlaremos la terminación de los bucles.

Sentencia WHILE

La sentencia *while* sigue el siguiente esquema:

while (<condición>) <sentencia>

La estructura del while se puede entender de forma gráfica con el siguiente diagrama:

El predicado encargado de interpretar el bucle while es el que sigue:

Código 3.20: Instrucción para la ejecución del bucle while

```
execute(Entrada,[('while'...variableDeAvance,[Condicion,
    Operaciones])|RestoInstrucciones],Salida)
```

Como en cualquier instrucción de flujo resolveremos la condición, en el caso del while, a diferencia del if, llamaremos al predicado executeLoop (explicada más adelante) con el cuerpo del while, será él quien distinga si la condición salió verdadera o falsa para ejecutar realmente el cuerpo del bucle o no. Llevaremos una variable de avance que será actualizada en el cuerpo del bucle y será quien determine cuándo parar. A continuación ejecutaremos el resto de las instrucciones para seguir con el resto del programa.

Sentencia FOR

La sentencia *for* sigue el siguiente esquema:

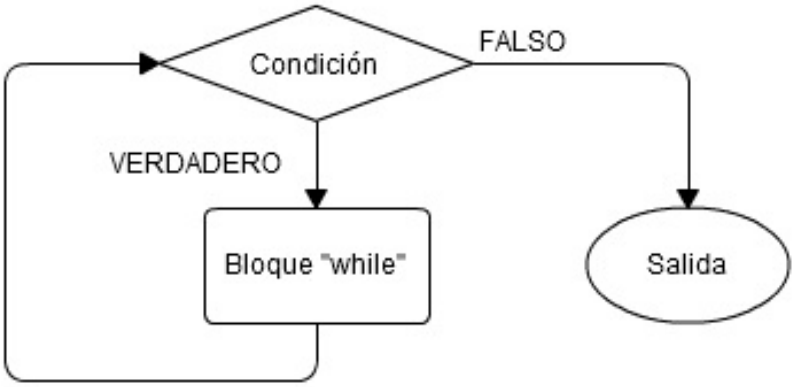

Figura 3.2: Figura que muestra la estructura del bucle while.

La estructura de la sentencia for se puede entender mejor de forma gráfica con el siguiente diagrama:

El predicado encargado de interpretar el bucle for es el que sigue:

Código 3.21: Instrucción para la ejecución del bucle for

```
execute(Entrada,[('for',variableDeAvance,[Variable,
    Condition,Avance,CuerpoFor])|RestoInstrucciones],Salida
    )
```

Es parecido al while ya que resolveremos la condición y llamaremos al predicado executeLoop (explicada a continuación) con el cuerpo del for, será él quien distinga si la condición salió verdadera o falsa para ejecutar realmente el cuerpo del bucle o no. A continuación ejecutaremos el resto de las instrucciones para seguir con el resto del programa.

executeBranch/4

El predicado executeLoop es llamado por execute cuando se encuentra una instrucción de bucle while o for. Igualmente recibe un estado de entrada, la lista de instrucciones, un valor numérico y devuelve el estado de salida. El valor numérico nos servirá para limitar el número de vueltas que realice el bucle, de forma que no haya posibilidad de recursión infinita, o un número de vueltas tan elevado que sea inviable devolver una solución. El número lo

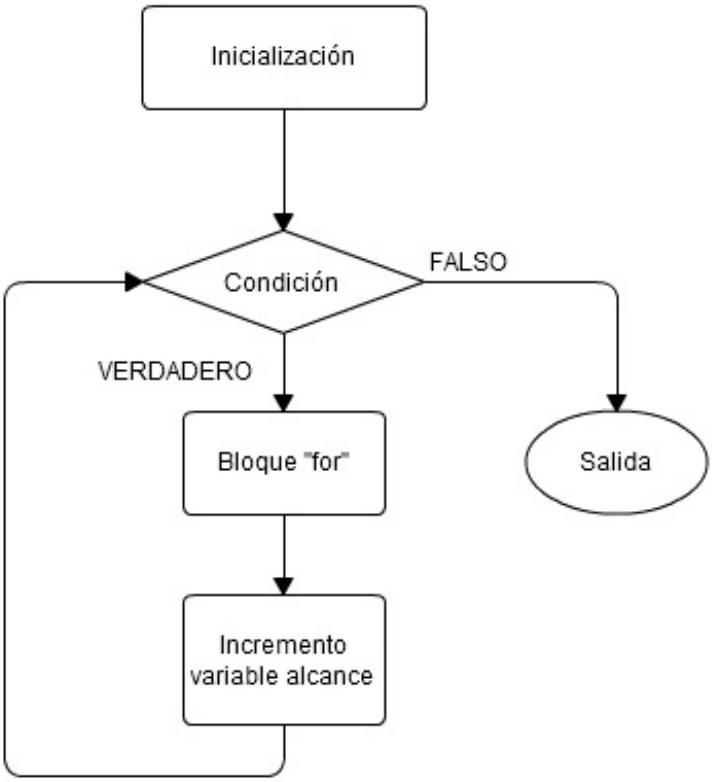

Figura 3.3: Figura que muestra la estructura del bucle for.

determina el usuario mediante la interfaz con valor del maxDepth como se dijo anteriormente.

Hemos limitado el backtracking a la evaluación de condiciones ya que éstas son las que realmente definen el flujo del programa.

Se pueden diferenciar 3 tipos de executeLoop:

- **Caso 1:** caso en el que se llega a la última vuelta permitida por la variable maxDepth.

  Código 3.22: Instrucción para el caso en el que bucle finaliza por la variable maxDepth

```
executeLoop(Entrada,[...Condicion...|RestoInstrucciones
    ],1,_,Salida)
```

  Cuando maxDepth alcanza el valor 1 comprobamos que la condición se cumple y llamamos a ejecutar el resto de las instrucciones.

- **Caso 2:** caso en el que la condición no se hace cierta, es decir hemos llegado al fin del bucle por lo que llamamos a ejecutar al resto de las instrucciones.

  Código 3.23: Instrucción para el caso en el que llegamos al fin del bucle

```
executeLoop(Entrada,[...|RestoInstrucciones],N,0,Salida)
```

- **Caso 3:** caso en el que la condición se hace cierta y aún no hemos llegado a la profundidad máxima permitida por lo que ejecutamos el cuerpo del bucle.

  Código 3.24: Instrucción para el caso en el que ejecutamos el cuerpo del bucle

```
executeLoop(Entrada,[...CuerpoBucle...|RestoInstrucciones
    ],N,1,Salida)
```

3.7. Instrucciones de Entrada/Salida.

Como ya dijimos anteriormente, iremos almacenando en cada paso el estado actual que está formado por: Table (la tabla de variables), Cin (Entrada por consola), Cout (Salida por consola), Trace (Traza o camino que llevamos ejecutado).

ConsoleIn (Entrada)

Para recoger una expresión sencilla que viene como entrada por consola lo haremos mediante el predicado:

Código 3.25: Instrucción para la ejecución de la consola de entrada

```
resolveExpression(EntryS,('consoleIn',[_=int],_),Value,OutS
    )
```

Recogeremos el valor contenido en la variable Value y lo concatenaremos al resto de la entrada que llevemos almacenado en Cin.

ConsoleOut (Salida)

Para devolver un valor por consola lo haremos mediante el predicado:

Código 3.26: Instrucción para la ejecución de la consola de salida

```
execute(Entrada,[('consoleOut'...Expresion)|
    RestoInstrucciones],Salida)
```

Nos sirve para sacar por pantalla el resultado de la expresión, por lo que llamaremos a la función *resuelveExpresión* con la cxpresión como parámetro y el resultado devuelto será incluido en el Cout anteriormente mencionado de forma que al final saldrá por pantalla todo aquello que se encuentre en esa variable. Ejecutamos el resto de las instrucciones.

3.8. Instrucción de retorno.

Instrucción RETURN

Ésta es la sentencia de salida de una función, cuando se ejecuta, se devuelve el control a la rutina que llamó a la función. Además, se usa para especificar el valor de retorno de la función. La instrucción de return sigue el siguiente esquema:

return [<expresión>] ;

La instrucción de retorno del valor de una función se recoge en el predicado:

Código 3.27: Instrucción para la ejecución de la sentencia return

```
execute(Entrada,[('return'...CuerpoReturn])|_],Salida)
```

En la tabla de variables llevamos reservada una variable que nos servirá como variable de retorno por lo tanto en este predicado lo único que haremos será ejecutar con la llamada a execute del cuerpo del return y con el valor devuelto se actualizará esa variable reservada. En este caso no llamaremos a ejecutar al resto de instrucciones ya que, como se trata de una instrucción de retorno, es la última instrucción que se debería de contemplar en esta función por tanto no deberíamos de seguir con el flujo del programa.

Capítulo 4

Interfaz gráfica de usuario de SymC++

En este capítulo explicaremos brevemente cual es la arquitectura de la interfaz desarrollada y su relación como núcleo central y enlace con todos los componentes de la herramienta y el usuario. Este capítulo está dirigido a aquellas personas interesadas en conocer las decisiones de diseño que hemos tomado, así como la estructura y funcionalidades implementadas.

Este capítulo no pretende ser un manual de usuario o de instalación, el cual se encuentra en el capítulo de apéndices.

4.1. Diseño

A la hora de desarrollar la interfaz para el cliente, el principal problema al que nos enfrentamos era la búsqueda de una interfaz intuitiva, fácil y que ofreciera todas las funcionalidades de depuración que buscamos para los estudiantes a los que está orientada esta herramienta. Es por esto que buscamos una vista de cliente, en la que toda la información resultante de la ejecución simbólica esté disponible en la vista principal. Para el desarrollo de la interfaz hemos utilizado una librería de "Synthetica Look And Feel", librería bajo licencia gratuita para usos no comerciales. Esta librería nos ha permitido dar una mayor vistosidad a la aplicación además de facilitarnos algunas cuestiones relativas al diseño. Por otro lado, todas las imágenes referentes a los botones también están bajo licencia gratuita para usos no comerciales.

Para la estructura de la interfaz, hemos utilizado "MigLayout", el cual se encuentra bajo licencia pública GNU GPL. Este layout nos facilita la distribución de la pantalla y del mismo modo, ofrece una mayor simplicidad y escalabilidad para la herramienta.

4.2. Funcionalidades

Las funcionalidades referentes a la interfaz del cliente son las básicas de cualquier editor de texto e incluso compilador de código. Hay que partir de la base, de que en nuestro proyecto lo más relevante no es la interfaz, sino todo el trabajo que hay detrás con la ejecución simbólica.

Como funcionalidades de depuración cabe destacar la visión en formato árbol de los XMLs generados tanto por Clang como por Prolog, los cuales contienen toda la información relativa al árbol sintáctico anotado del archivo C++ y al resultado de la ejecución simbólica respectivamente.

Referente a la visión del área de texto, para darle un aspecto de editor de código hemos añadido la visibilidad del número de línea.

Por tanto, para implementar las funcionalidades descritas, así como otras varias hemos utilizado las siguientes clases de código, todas bajo la licencia GNU, las cuales les hemos realizado las modificaciones necesarias para cumplir nuestro diseño :

- LineNumbers.java [1] (http://javaknowledge.info/jtextpane-line-number/)
- MultiLineCellRenderer.java [2] (http://www.java2s.com/Code/Java/Swing-Components/MultiLineCellExample.htm)
- XML2JTree.java [3] (cse.unl.edu/ reich/XML/DOM/XML2JTree.java)

4.3. Estructura de la herramienta

Para ilustrar mejor la estructura de la herramienta, así como para mostrar mejor la relación entre el usuario y la herramienta, y entre las diferentes secciones de la herramienta acompañamos la siguiente figura:

Como se puede observar, la herramienta SymC++ necesita un código en C++ sobre el que realizar la ejecucución y una serie de parámetros para el intérprete, y a partir de ahí la propia interfaz de usuario envía todo lo necesario tango al AST2XML como al Intérprete Simbólico, para que a través de la salida generada, sea procesada de nuevo por la interfaz y se pueda mostrar al usuario.

Cabe destacar, que para la comunicación entre la interfaz, tanto con AST2XML como con el Intérprete Simbólico, hemos creado dos scripts con comandos shell, los cuales son llamados desde java como procesos externos. Pero del mismo modo que estos comandos son lanzados desde la interfaz, también pueden ser ejecutados desde nuestro terminal de comandos, de este

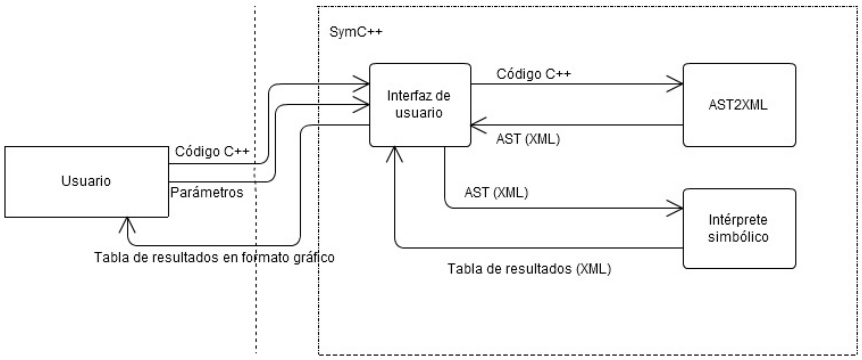

Figura 4.1: Figura que muestra las conexiones entre todas las partes de la herramienta.

modo, podemos generar tanto el XML del árbol sintáctico anotado, como el XML con la solución del intérprete simbólico, sin necesidad de usar la interfaz de usuario. Esta solución da una mayor portabilidad a ambas herramientas, así como aportar una mayor flexibilidad al usuario para que pueda usar el método con el que se sienta más comodo. Los comandos los detallamos a continuación:

- Comando para la ejecución de AST2XML.
- Comando para la ejecución del Intérprete Simbólico.

Hay que tener en cuenta, una serie de requisitos para que la herramienta funcione correctamente. Es necesario que el usuario seleccione al abrir por primera vez SymC++ dónde ha instalado que la herramienta de AST2XML, la cual se guardará en un archivo de configuración. Por tanto, al lanzar la ejecución, tomará dicho directorio como herramienta y será la encargada de generar el archivo XML del árbol de ejecución, en este caso es responsabilidad del usuario seleccionar correctamente tal archivo para una correcta ejecución. Por otro lado para la ejecución de SymC++ es necesario un archivo con el código C++, por tanto si el usuario abre un archivo con el código, antes de la ejecución se realizará un autoguardado, por otro lado, en el caso de que el usuario haya escrito el código completo en nuestra herramienta, la herramienta pedirá al usuario que guarde dicho archivo antes de poder ejecutar.

4.4. Directorio de archivos

El sistema de archivos de la aplicación está dividido en varios directorios:

- files : Directorio en el que se almacenan todos los archivos generados, tanto los XMLs generados por Clang y el intérprete, como el archivo de c++ que se va a ejecutar. En este directorio, se incluyen los archivos "BuiltinsIO.h" y "BuiltinsSTD.h" necesarios para el uso de las funciones de entrada y salida.
- tools: En este directorio se incluyen los archivos prolog necesarios para que el intérprete funcione y los comandos shell que realizan la llamada tanto a la herramienta Clang como al intérprete Prolog.
- img: Directorio en el que se incluyen todas las imágenes necesarias para la interfaz.
- libs: Directorio en el que se incluyen todas las librerías necesarias para que la interfaz funcione, las cuales son las referentes al estilo, synthetica, y al layout, miglayout.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Conclusiones sobre el trabajo realizado

Tras un curso desarrollando este proyecto podemos decir que hemos cumplido los objetivos fijados al inicio del curso, desarrollando una aplicación que realiza la ejecución simbólica de un conjunto limitado de operadores de C++. La herramienta evalúa todas las posibles ramas de funciones sencillas teniendo en cuenta todas las limitaciones de la ejecución.

Los resultados obtenidos por dicha ejecución se muestran de una forma sencilla y visual al usuario, además de mostrar las líneas que recorre dicho intérprete.

Conclusiones personales

Este proyecto nos ha aportado ciertas capacidades que no hemos podido desarrollar en otras asignaturas. En primer lugar, nos ha permitido enfrentarnos a un proyecto de investigación informática, mucho más teórico que práctico. Donde el mayor trabajo se ha centrado en la investigación de tecnologías que en el desarrollo de las mismas.

Por otro lado, el hecho de enfrentarnos nosotros mismos a un proyecto, a pesar de la tutorización, nos ha obligado a trabajar en equipo, coordinarnos y organizarnos para cumplir unos objetivos y plazos estipulados por nosotros mismos. Eso nos ha aportado responsabilidad y compromiso para con el proyecto.

A pesar que durante el presente curso hemos cursado otras asignaturas y

no hemos tenido la posibilidad de dedicarle más tiempo al proyecto, consideramos que el trabajo realizado ha sido muy productivo y nos ha aportado una gran experiencia académica y personal.

5.2. Ampliaciones potenciales y trabajo futuro

Nuestro proyecto puede ser utilizado como base para siguientes cursos para la asignatura de Sistemas Informáticos o proyecto de final de grado. Desde el primer día, nuestro tutor nos puntualizó que este es trabajo para toda una vida, y tener en cuenta toda la sintaxis de C++ es inviable en un curso académico.

Mayor repertorio de instrucciones

Como hemos comentado, en este proyecto hemos trabajado con un pequeño repertorio de las instrucciones de C++, puesto que abarcar toda su sintaxis era inviable. Es por esto que trabajamos con instrucciones básicas y el tipo de números enteros.

Si esta aplicación se quisiese usar como un depurador completo, sería necesario contemplar toda la sintaxis de C++ y todos los tipos de variables.

Otros sistemas operativos

Una de las limitaciones que tuvimos a la hora de desarrollar la aplicación es la instalación del compilador Clang en el sistema operativo de Windows, por tanto nos vimos obligados a limitarnos a desarrollar en Linux. Esta incompatibilidad limita la portabilidad de la herramienta y por tanto el número de usuarios que podrían utilizarla.

Si la apliacación quisiese abarcar a una mayor comunidad de usuarios sería necesario investigar el modo de compilar la herramienta de Clang en otros sistemas operativos como Windows o MacOSX.

Aplicacion web

En referencia a la sección anterior, una posible ampliación sería exportar esta aplicación al ámbito web. De este modo, no sería necesario instalación de librerías u otros programas que tengan incompatibilidades con el sistema

operativo, y por otro lado los usuarios no sólo se limitarían a los que conociesen la aplicación por la Universidad sino que la comunidad potencial es global.

Apéndice A

Manual de usuario

A.1. Requisitos previos

Como ya hemos indicado en las secciones de la memoria, nuestro proyecto actualmente sólo funciona en sistemas operativos LINUX, por tanto en lo siguiente supondremos que se trabaja bajo este sistema operativo.

Por otro lado para poder ejecutar el sistema que hemos desarrollado será necesario cumplir una serie de requisitos software previos. En concreto deberemos tener instalados los siguientes componentes:

- **"Java Runtime Environment"** o JRE de 32 bits, versión 1.6 o superior.
  Para ello desde la propia página recomiendan, en lo relativo a requisitos hardware, contar con un Pentium 2 a 266 MHz o un procesador más rápido con al menos 128 MB de RAM física.
  Enlace de descarga: `http://www.java.com/es/download/`

- **"Clang"**. Al comienzo del proyecto la última versión disponible era la 3.5, por lo que esta fue la que se utilizó.
  Para poder compilar nuestra herramienta de generación del AST2XML, cuyos pasos detallados de instalación comentamos en la próxima sección, es necesario tener instalado Clang en nuestro sistema. Puesto que la instalación y requisitos ya están detallados en la propia web de Clang no vamos a detenernos a comentarlos uno a uno.

  - **"Requisitos LLVM System"**:
    Enlace de requisitos: `http://llvm.org/docs/GettingStarted.html#requirements`

- **"Python"**. Para compilar Clang es necesario tener instalado Python en nuestro sistema.
  Enlace de descarga: `https://www.python.org/download`
- **"Compilacion de Clang"**
  Enlace con los pasos detallados `http://clang.llvm.org/get_started.html`

A.2. Instalación herramienta ast2xml

La herramienta ast2xml es necesaria compilarla en nuestro ordenador para poder utilizarla, puesto que es diferente para cada equipo. En esta sección damos por sentado que el usuario ya ha instalado Clang en su ordenador, a continuación detallamos los pasos para su compilación en nuestro sistema para luego poder utilizarla.

- Dirigirse a nuestro directorio LLVM.
  cd (directorio llvm)/tools/clang/tools
- Clonar el repositorio con nuestra herramienta en una carpeta del directorio.
  git clone `https://github.com/Si1314/AST2XML.git` **ast2xmltool**
- Copiar el Makefile.
- Ir al directorio de compilación.
  cd (directorio de compilación)/tools/clang/tools
- Crear un directorio específico para la herramienta (con el mismo nombre).
  mkdir ast2xmltool
- Copiar en él el archivo Makefile.
- Ir dentro del directorio.
  cd ast2xmltool
- Ejecutar el comando "make"
  La herramienta estará compilada y enlazada si no hay ningún problema
- Volver al directorio de compilación.
  cd (directorio de compilación)/Debug + Asserts/build

  Aquí debería encontrarse la herramienta, así como el resto de herramientas proporcionadas por el paquete de clang.

A.3. Tutorial de uso

Apéndice B

Gramática reducida C++

En éste apéndice mostramos la gramática de C++ reducida, con las reglas que utiliza nuestra herramienta.

B.1. Reglas gramáticales

- ElementosEntrada : ElementoEntrada
- ElementoEntrada : whitespace | comentario | token
- comentario : “/” “” caracter_ASCII - { “/” } “ ” “/” | “/” “/” caracter_ASCII - { “
n” }
- digito : 0 | 1 | 2 |3 | 4 | 5 | 6 | 7 | 8 | 9
- letra : a | b | c | d | e | f | g | h | i | j | k | l | m | n | ñ | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | Ñ | O | P | Q | R | S | T | U | V | W | X | Y | Z | Á | É | Í | Ó | Ú
- whitespace : espacio en blanco
- ascii Caracteres del estándar ASCII
- caracter_ASCII : ascii | Ñ | ñ
- token : identificador | palabra_reservada | literal | delimitador | operador
- identificador : “” [letra | dígito | “” | “$”]* letra [letra | dígito | “” | “$”]* “$” [letra | dígito | “” | “$”]*

- palabra_reservada : “class” | “else” | “for” | “if” | “int” | “return” | “void” | “while”

B.2. Literales y operadores

- literal : literal_integer
- literal_integer : [“+” | “-”] [digito - {0}]* digito* | [“+” | “-”] “0”
- delimitador : (|) | { | } | ; | , | .
- operador : operadorInfijo | operadorPrefijo | operadorPostfijo
- operadorInfijo : || | && | == | != | <| >| <= | >= | + | - | * | /
- operadorPrefijo : ! | + | -
- operadorPostfijo : ++ | –
- operadorAsignación : = | += | -= | *= | /=
- operadorUnario : & | * | + | | !

B.3. Tipos de datos

- tipo : tipo_primitivo identificador
- tipo_primitivo : “int”
- listaArgumentos : expresionAsignacion | listaArgumentos , expresionAsignacion

B.4. Expresiones

- expresión : expresionAsignacion | expresion , expresionAsignacion
- expresionAsignación : expresionCondicional | expresionUnaria operadorAsignacion expresionAsignacion
- expresionCondicional : expresionOrLogico
- expresionOrLogico : expresionAndLogico | expresionOrLogico || expresionAndLogico

- expresionAndLogico : expresionIgualdad | expresionAndLogico && expresionIgualdad
- expresionIgualdad : expresionRelacional | expresionIgualdad == expresionRelacional | expresionIgualdad != expresionRelacional
- expresionRelacional : expresionAditiva | expresionRelacional <expresionAditiva | expresionRelacional >expresionAditiva | expresionRelacional <= expresionAditiva | expresionRelacional >= expresionAditiva
- expresionAditiva : expresionMultiplicativa | expresionAditiva + expresionMultiplicativa | expresionAditiva expresionMultiplicativa
- expresionMultiplicativa : expresionConversion | expresionMultiplicativa * expresionConversion | expresionMultiplicativa / expresionConversion
- expresionConversion : expresionUnaria | (nombreTipo) expresionConversion
- expresionUnaria : expresionSufijo | ++ expresionUnaria | operadorUnario expresionConversion
- expresionSufijo : expresionPrimaria | expresionSufijo [expresion] | expresionSufijo (listaArgumentos?) | expresionSufijo ++
- expresionPrimaria : identificador | digito | literalCadena | (expresion)

B.5. Expresiones constantes

- expresionConstante : expresionCondicional

B.6. Sentencias

- sentencia : sentenciaCompuesta | sentenciaExpresion | sentenciaSeleccion | sentenciaIteracion | sentenciaSalto
- default : sentencia
- sentenciaCompuesta : { listaDeclaraciones? listaSentencias? }
- listaDeclaraciones : declaracion | listaDeclaraciones declaracion

- listaSentencias : sentencia | listaSentencias sentencia
- sentenciaExpresion : expresion? ;
- sentenciaSeleccion : if (expresion) sentencia | if (expresion) sentencia else sentencia
- sentenciaIteracion : while (expresion) sentencia | for (expresion? ; expresion? ; expresion?) sentencia
- sentenciaSalto : return expresion? ;

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes