View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Biblioteca Digital do IPB

Simplificando la Comprensión de Programas a través de la Interconnexión de Dominios

Mario M. Berón
Universidad Nacional de San Luis - Departamento de Informática
San Luis - Argentina
mberon@unsl.edu.ar
Pedro R. Henriques
Universidad de Minho - Departamento de Informática
Braga - Portugal
pedrorangelhenriques@gmail.com
Maria J. Varanda
Instituto Politécnico de Bragança - Departamento de Informática
Bragança - Portugal
mjoao@ipb.pt
Roberto Uzal
Universidad Nacional de San Luis - Departamento de Informática
San Luis - Argentina
ruzal@sinectis.com.ar

Resumen

La Comprensión de Programas (CP) es un área de la Ingeniería del Software cuyo objetivo es el estudio y creación de modelos, métodos, técnicas y herramientas con la finalidad de facilitar el entendimiento de los sistemas de software. La CP es útil para mantenimiento, reingeniería, ingeniería reversa, entre otras tantas aplicaciones. Actualmente existen muchas herramientas destinadas a facilitar el entendimiento del software. Sin embargo, la gran mayoría carece de algunos aspectos fundamentales para alcanzar este objetivo. Uno de tales aspectos es: *La Relación existente entre los Dominios del Problema y del Programa*. Alcanzar esta relación es una tarea compleja porque es necesario: i) proponer una representación del problema, ii) elaborar una representación del programa y iii) desarrollar un procedimiento para vincular ambas representaciones.

En este artículo, se presentan dos de estrategias destinadas a relacionar estos dominios. La primera de ellas, SVS (**S**imultaneous **V**isualization **S**trategy) es una estrategia *En Vida* (*alive*) porque a medida que el sistema se ejecuta visualiza las componentes del programa usadas para producir la salida. La segunda, BORS (*Behavioral-Operational Relation Strategy*) es *Post Mortem* porque aplica procedimientos para alcanzar esta relación después que el programa se ejecutó. Finalmente, el *Análisis comportamental* es una aproximación para comprender un sistema a partir de los conceptos involucrados en su implementación.

Palabras Claves: Comprensión de Programas, Métodos, Técnicas, Herramientas.

1. Introducción

La Comprensión de Programas (CP) es una disciplina de la Ingeniería del Software (IS) cuyo objetivo es ayudar al programador a entender programas. CP es útil para mantenimiento y evolución del Software (MES), Ingniería Inversa (IR), Re-Ingeniería (ReI)y Educación [RD94, MMW02, VMV95]. En las tres primeras disciplinas, la CP ayuda a reducir costos y esfuerzos. En la última asiste al estudiante en el proceso de entender algoritmos.

MES es una tarea crítica porque implica tres importantes actividades: *Perfectiva*, *Correctiva* y *Adaptativa*. La primera está relacionada con la incorporación de nuevas funcionalidades. La segunda describe el proceso de detección y elminación de errores. La tercera está centrada en el estudio de técnicas para adaptar el sistema a nuevos contextos, por ejemplo, usar nuevos formatos de archivos, cambiar la organización del código, etc. Estas tres principales actividades consumen mucho tiempo y recursos. Por ejemplo, Xi y Hassan en [XPH07] declaran que el 39 % de las actividades son perfectivas, 56.7 % son correctivas y el 2.2 % se corresponden con actividades adaptativas. Finalmente, el 2.1 % está relacionado con otras actividades de MES. Por otra parte, la historia del MES muestra un incremento lineal, desde 1975 hasta 2005, en los costos de proyecto destinados a MES [XPH07]. Esta información justifica el interés de la comunidad científica de reducir los esfuerzos dedicados a estas tres principales tareas.

Otra área muy importante es la Ingeniería Reversa (IE). Esta disciplina estudia la creación de estrategias de extracción de la información desde diferentes fuentes. Por ejemplo, código fuente, archivos makefile, documentación, etc. Luego esta información se procesa para elaborar documentación, arquitectura, etc. La IR implica grandes costos y tiempo porque generalmente incluye actividades similares a MES.

La Re-Ingeniería (ReI) usa técnicas de IR para extraer información del sistema, luego aplica estrategias para modificar el sistema de acuerdo a los requerimientos del usuario. Por esta razón, la ReI, al igual que MES y IR, es un área donde se necesitan propuestas para minimizar costos.

En todas las disciplinas (MES, IR, y ReI) mencionadas en los párrafos precedentes, el programador debe entender grandes documentos con diferentes formalismos y metodologías. Además, él debe comprender como el sistema lleva a cabo sus funcionalidades en un alto nivel de abstracción. La Comprensión de Programas asiste en esas tres principales áreas. Esta actividad se concretiza a través de métodos, estrategias y herramientas diseñadas para reducir las tareas del programador [Sto05]; y como un efecto colateral de ésta última pecularidad, los costos se reducen.

Corrientemente hay muchas herramientas de CP con sofisticadas técnicas de exploración de código [Sto98]. Esas herramientas funcionan adecuadamente, sin embargo algunas tareas de entendimiento son todavía muy complejas. Por una parte, algunas herramientas sólo usan análisis estático para recuperar información del sistema. Después de eso, esta información se representada visualmente para proveer diferentes vistas del sistema. De esta forma, el proceso de CP se simplifica. No obstante, el usuario está forzado a encontrar el código fuente usado por el sistema para producir salidas específicas. Otra clase de herramientas analizan el flujo de ejecución del sistema y proveen otro tipo de visualización de la información. Esas herramientas frequentemente muestran características de bajo nivel que son útiles para estados avanzados de CP, como por ejemplo para comprender rutinas específicas del sistema. Finalmente, otra categoría de herramientas combinan información estática y dinámica, pero su salida consiste de interesantes y complejos análisis de cada clase de información.

En términos generales es posible afirmar que las herramientas de CP olvidan algunos aspectos imporntantes de la CP. Por ejemplo, la relación entre la salida del sistema y las componentes

del programa usadas para producir dicha salida. Esta relación es conocida como *La Interconexión entre los Dominios del Problema y Programa* [Bro78, LF94].

La característica mencionada en el párrafo precedente ayuda a la exploración del sistema porque el usuario sólo inspecciona las partes del sistema relacionadas con una funcionalidad específica. Para ser más claro, suponga que el usuario está interesado en modificar la funcionalidad F del sistema S. Además el tamaño de S es considerable. En este caso, la primer tarea consiste en encontrar las funciones que implementan F y después de eso algunas técnicas de inspección se pueden usar para estudiar aquellas funciones detalladamente. Es importante remarcar que la primer tarea consume tiempo y esfuerzo; además si la misma se puede evitar entonces muchas ventajas serían obtenidas. Una forma de salvar este inconveniente consiste en usar estrategias para interconectar los dominios del problema y programa. Esto se debe a que dicha relación identifica en forma parcial o total las funciones que implementan una característica específica del sistema.

Este artículo presenta estrategias destinadas a reducir el tiempo requerido para identificar las funciones que conforman una tarea específica del sistema. Esta tarea se lleva a cabo a través de la recuperación de la relación existente entre el dominio del problema y programa.

La organización de este artículo se describe a continuación. La sección 2 expone algunas fortalezas y debilidades de las herramientas de comprensión de programas actuales. La sección 3 presenta diferentes estrategias para relacionar los dominios del problema y programa, el objetivo principal de estas estrategias consiste en permitir al usuario idenficar las funciones usadas por el sistema para llevar a cabo sus funcionalidades. La sección 4 explica los pasos necesarios para aplicar las estrategias de interconexión de dominios usando PICS una herramienta para facilitar la comprensión de programas escritos en lenguaje C. La sección 5 tiene como objetivo mostrar la utilidad de las estrategias de interconexión de dominios expuestas en este trabajo usando como caso práctico el comando *agrep* de Linux. Finalmente, la sección 6 presenta las conclusiones.

2. Trabajos Relacionados

Existen muchas herramientas de comprensión de programas con procedimientos muy útiles para simplificar el entendimiento del software. Muchas de ellas, quizás las más completas se basan en técnicas de análisis estático para inspeccionar el código fuente del sistema. En esta clase de herramientas se pueden mencionar: Understand C/C++, CodeSurfer, Imagix 4D, CScope, SHriMP, SeeSoft, etc [BHU07]. La característica común a estas herramientas es la facilidad con que el usuario puede acceder a las distintas componentes del sistema y las estrategias de visualización utilizadas. Como característica saliente de esta clase de herramientas se pueden mencionar las sofisticadas representaciones de programas usadas para obtener información detallada del código fuente. En este aspecto se puede decir que la propuesta presentada por CodeSurfer es muy completa porque representa el programa usando un grafo que refleja las dependencias existentes entre todos los elementos del programa (variables, sentencias, procedimientos, funciones, módulos, etc.). Esta representación hace posible aplicar operadores de grafos que computan diferentes tipos de slicing estático y permiten acceder fácilmente a los elementos del sistema. Las otras herramientas generalmente representan el código fuente del sistema como un Árbol de Sintáxis Abstracta y luego definen barridos sobre esos árboles para recuperar la información deseada. Es importante remarcar que estas herramientas poseen un atractivo e intuitivo sistema de visualización que posibilita observar claramente la información. Otras herramientas recuperan información estática y dinánica para mostrar aspectos del código

fuente y del flujo de ejecución. Algunas herramientas que cumplen con esta características son: ALMA y JGrasp entre otras [BHU07]. Estas herramientas poseen formas de ver las funciones y datos usados en tiempo de ejecución lo cual da un paso mas en la compleja tarea de comprender programas. Desafortunadamente las herramientas investigadas no relacionan el dominio del problema con el dominio del programa y por consiguiente no solucionan el problema descripto en la sección 1.

3. Estrategias para Relacionar los Dominios del Problema y Programa

En esta sección se proponen procedimientos que permiten conocer cuales son las componentes usadas por el sistema para producir su salida. En otras palabras, las estrategias descriptas en esta sección permiten relacionar el dominio del problema con el dominio del programa.

3.1. SVS: Simultaneous Visualization Strategy

La idea subyacente a la estrategia SVS (**S**imultaneous **V**isualization **S**trategy) consiste en ejecutar en forma paralela de dos sistemas: i) el sistema de estudio y ii) un monitor que permite visualizar las funciones usadas en tiempo de ejecución. Una propuesta interesante para concretizar SVS consiste en instrumentar el sistema, es decir insertar sentencias útiles dentro del código fuente para extraer información de tiempo de ejecución (ver [BHVU06, BHU07] para más detalles del esquema de instrumentación). Para llevar a cabo esta tarea, es importante decidir: i) que información recuperar y ii) como se debe instrumentar el sistema para producir el comportamiento deseado.

Teniendo presente el primer ítem se puede decir que conocer que funciones del sistema se utilizan en una ejecución específica permite identificar las funciones utilizadas para un objetivo particular del sistema.

El segundo ítem considera la recuperación de la información dinámica. En este contexto, los puntos interesantes para alcanzar este objetivo son el comienzo y eventualmente el fin de cada función.

El primero de ellos posibilita conocer que función está siendo invocada y el segundo es útil para construir ciertas estructuras de datos que pueden ser usadas para interconectar los dominios del problema y programa. Las sentencias insertadas deben ser funciones destinadas a procesar y mostrar esta información. Esas funciones son llamadas *Inspectores* o *Funciones de Inspección*, las cuales forman parte del monitor.

Como fue mencionado en el comienzo de esta sección, SVS se basa en la ejecución paralela del sistema de estudio y el monitor. A medida que el sistema se ejecuta el monitor va indicando cuales son las funciones que el sistema está utilizando para producir su salida.

La sección 5 muestra algunos ejemplos de SVS aplicada al comando *agrep* de Linux.

3.2. BORS: Behavioral-Operational Relation Strategy

Para relacionar las vistas comportamental y operacional, se puede usar la información de los objetos del dominio del problema y el flujo de ejecución del programa (dominio del programa).

La primer tarea se lleva a cabo observando la salida del sistema y aplicando algunas técnicas para recuperar información de los objetos.

La segunda se obtiene instrumentando el código fuente como se describió en la sección 3.1. Mezclando el conocimiento acerca de los dominios del problema y el programa, una estrategia para alcanzar nuestro propósito fue desarrollada.

En la sección 3.1, las funciones son registradas como una lista. Para proveer más información semántica la lista de funciones de tiempo de ejecución puede ser vista como un árbol. Con esta estructura de datos, la relación llamador-llamado se representa más adecuadamente. Esta forma de mirar a las funciones es mejor que una simple lista de funciones porque es posible identificar claramente el trabajo realizado por cada función. Por ejemplo, si F es una función del sistema, entonces su funcionalidad esta dada por todas las funciones llamadas directa o indirectamente por ella. Esta información se almacena en el sub-arbol cuya raíz es la función F (ver [Abd05, BHU07] para más información acerca del árbol de ejecución de funciones).

Esta característica justifica nuestra preferencia de representar las funciones de tiempo de ejecución como un árbol en lugar de una lista. Este árbol se denomina *fe-Tree* (**F**unction **E**xecution **T**ree).

Un *fe-Tree* es un árbol con aridad r donde:

1. La raíz del fe-Tree es la primer función ejecutada por el sistema (normalmente llamada main), y
2. Para cada nodo (función) n, sus descendientes son las funciones invocadas directamente por n en tiempo de ejecución.

Con el fe-Tree, se puede explicar cualquier función del sistema. Además, es posible conocer los diferentes contextos donde las funciones fueron invocadas. Por esta razón, se usa el fe-Tree para inspeccionar los aspectos del sistema considerados importantes por el usuario. Esos aspectos están constituidos por un conjunto de funciones y sus explicaciones están compuestas por la descripción de cada función.

En los siguientes párrafos se describe BORS (**B**ehavioral-**O**perational **R**elation **S**trategy), una estrategia destinada a relacionar las vistas comportamental y operacional del software. Esta estrategia se basa en la observación de los objetos del dominio del problema y en la información de tiempo de ejecución.

BORS tiene tres pasos claramente definidos:

- Detectar las funciones relacionadas con cada objeto del Dominio del Problema: consiste en descubrir los objetos del dominio del problema y sus funciones asociadas. La primer tarea se lleva a cabo observando la salida del sistema. La segunda puede ser tan simple como leer el código fuente o tan compleja como aplicar estrategias para detectar tipos de datos abstractos. Todas las funciones seleccionadas en este paso son almancenadas en una lista.

- Construir un fe-Tree con las funciones usadas en tiempo de ejecución: usa información de tiempo de ejecución para construir el fe-Tree. Estos datos se recuperan por medio de la instrumentación del código fuente.

- Explicar las funciones encontradas en el paso 1 usando el árbol construidos en el paso 2: Este paso se implementa aplicando un barrido por niveles sobre el fe-Tree. Cuando el nombre del nodo visitado se encuentra en la lista de funciones construida en el paso 1, BORS

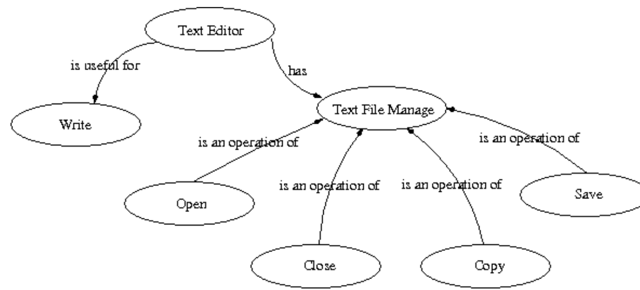

Figura 1: Basic Text Editor Conceptual Map

reporta: el camino desde la raíz del árbol hasta el nodo corriente y el correspondiente sub-árbol.

El lector interesado puede leer [BHU07] para obtener más detalles acerca del funcionamiento de la estrategia BORS.

3.3. Análisis Comportamental

Otra interesante aproximación para alcanzar una relación comportamental-operacional consiste en reproducir cada comportamiento del sistema y registrar las operaciones realizadas. Después de eso, la información se debe procesar con el objetivo de recuperar conocimiento.

Por una parte, se puede usar instrumentación de código para recuperar las operaciones de tiempo de ejecución del sistema. Por otra parte, el usuario debe ejecutar el sistema y usar todas sus funcionalidades. Para cada funcionalidad, se salva cada traza de ejecución y se asignaa cada una de ellas un nombre semántico. Cuando todas las tareas explicadas arriba están hechas se pueden analizar las trazas de ejecución [HL05].

Con esta propuesta, el usuario sólo tiene funcionalidades del sistema aisladas con las componentes del programa asociadas. Sin embargo, es posible mejorar esta relación pormedio de la introducción de representaciones del dominio del problema. Por ejemplo, se pueden usar los mapas conceptuales para representar el comportamiento del sistema y ellos pueden ser decorados con las funciones del sistema empleadas para implementar cada concepto. Por ejemplo, asuma que el sistema bajo estudio es un editor de textos básico y que su mapa conceptual es el que se encuentra en la Figura 1.

Después (antes o al mismo tiempo) de construir esta representación el usuario puede ejecutar el sistema instrumentado para concretizar cada concepto. En otras palabras, el usuario puede usar el sistema para: escribir, abrir archivo, etc. Para cada operación, se registran las trazas de ejecución y esta información se asocia a cada concepto. El resultado es una representación del dominio del problema decorada con las componentes del programa empleadas para llevar a cabo cada concepto.

El mismo procedimiento se puede aplicar usando herramientas estándar de Ingeniería del Software tales como diagramas de caso de uso o el modelo de dominio de UML. La ventaja de esta última propuesta consiste en la facilidad de integrar los resultados obtenidos en los proyectos de desarrollo de software.

4. PICS

PICS (**P**rogram **I**nspection and **C**omprehension **S**ystem) es una herramienta de comprensión de programas cuya finalidad es facilitar la comprensión de sistemas escritos en lenguaje C. PICS tiene implementadas las estrategias SVS y BORS y un conjunto de funciones de inspección que se pueden ser usar después de haber aplicado las estrategias que relacionan los dominios del problema y programa.

Para poder aplicar SVS y BORS los siguientes pasos son necesarios:

- Saltar las directivas al preprocesador: Los sistemas escritos en lenguaje C poseen el inconveniente de usar dos lenguajes, ellos son: el lenguaje del preprocesador y el lenguaje C propiamente dicho. La implementación de las técnicas de interrelación de dominios requiere la construcción de un analizador sintáctico para insertar las sentencias que permiten recuperar la información dinámica. Construir un analizador sintáctico que reconozca dos lenguajes a la vez es complejo, por lo tanto para eliminar este inconveniente se pueden usar dos aproximaciones: i) preprocesar el código fuente o ii) comentar las directivas al preprocesador. La primera de ellas es quizás la más adecuada, no obstante es el caso general que los archivos utilizados para compilar sistemas grandes (Makefiles) son muy complejos y dificiles de entender. Además de eso, el código preprocesado es muy diferente del código fuente original porque contiene el código correspondiente a la implementación de los archivos de cabecera (headers). Esta peculiaridad no ayudan a comprender el sistema. La otra solución inhibe las directivas al procesador usando comentarios. Esta aproximación presenta el inconveniente de requerir la intervención del usuario cuando el análisis sintáctico no puede finalizar con éxito. Para finalizar este ítem es importante mencionar que el usuario decide que camino seguir de acuerdo a su experiencia con el lenguaje C.

- Detectar tipos no declarados: Los sistemas normalmente usan librerias que definen tipos de datos no estándares. Usualmente, estas definiciones no se pueden encontrar porque no se dispone del código fuente. Para solucionar este inconveniente todos los tipos no declarados son incorporados en una base de tipos a medida que se analiza el programa.

- Extraer información estática: En esta etapa se han superado todos los problemas de análisis sintáctico por tanto se aplican técnicas de compilación tradicionales para recuperar información estática tales como variables, tipos, funciones, etc. Esta información es importante porque puede ser usada luego de haber detectado las funciones utilizadas por una ejecución específica del sistema.

- Instrumentar el código fuente: En esta fase se insertan las funciones de inspección que indican el comienzo y fin de las funciones del sistema. Además se incorporan al código fuente otras sentencias destinadas a controlar la cantidad de veces que las funciones invocadas dentro de los ciclos se deben mostrar.

- Insertar las declaraciones del monitor: Generalmente, para compilar el sistema de estudio, se debe insertar en cada módulo un archivo de cabecera que contiene las declaraciones de las funciones implementadas en el monitor (funciones de inspección y variables necesarias para controlar la ejecución del sistema). Esta tarea es simple de realizar porque implica incorporar al inicio de cada módulo el archivo de cabecera correspondiente.

- Compilar el sistema: El monitor usa librerias gráficas que son utilizadas con el objetivo de visualizar adecuadamente la información. Por esta razón, para compilar el sistema de estudio se requiere el pasaje de parámetros específicos al compilador. Para realizar esta tarea, se debe modificar el archivo que compila al sistema (usualmente un Makefile) incorporando los parámetros necesarios para lograr una compilación exitosa del sistema. Es importante notar que generalmente este paso es simple debido a que los Makefiles poseen variables que almacenan las banderas que son pasadas como parámetros al compilador. No obstante, cuando este no es el caso, se requiere de experiencia en la inspección de esta clase de archivos.
- Ejecutar el sistema: En esta etapa el usuario puede ejecutar el sistema y observar, a medida que el sistema se ejecute (SVS) o después de su ejecución (BORS), la relación entre los dominios del problema y programa.

5. Caso de Estudio: El Comando agrep de Linux

El comando *agrep* es similar al comando *grep* o *fgrep* pero es mucho más general y rápido. Este comando busca cadenas de caracteres dentro de los archivos especificados en su entrada. La salida de dicho comando son las líneas de los archivos que contienen la cadena de caracteres recibidas como entrada. Es importante notar que las cadenas de caracteres pueden ser especificadas por expresiones regulares. Esta caracteristica permite expresar modelos de cadenas de caracteres muy complejos.

Para analizar este programa su utilizó la herramienta PICS brevemente descripto en la sección 4. En las subsecciones siguientes se describirán los resultados obtenidos en cada una de las fases necesarias para la utilización de SVS y BORS.

5.1. Saltar las directivas al preprocesador

La aplicación de esta etapa al programa *agrep* no presentó mayores inconvenientes porque las directivas al preprocesador, usadas por la implementación de *agrep*, son simples; es decir son del tipo $\#include < nombre.h >$. Este tipo de directivas pueden ser comentadas sin problemas. Por otra parte, las directivas no se usan para la generación de código para diferentes arquitecturas u otros objetivos.

5.2. Detectar tipos no declarados

Afortunadamente este comando está implementando usando estrictamente las funciones provistas por C sin librerias adicionales. Los tipos no estándares encontrados son definidos por el usuario a través de declaraciones *typedef*. Dichas declaraciones se pueden detectar por medio de técnicas de compilación. Por los motivos antes mencionados, esta etapa fue superada sin problemas adicionales.

5.3. Extraer información estática

El problema principal en esta etapa se presenta por las directivas al preprocesador utilizadas y por los tipos no declarados. Como las directivas se utilizan en una forma básica y no hay en

el sistema tipos no declarados pertenecientes a librerias u otro artefacto de software, entonces se puede extraer la información estática sin problemas.

5.4. Instrumentar el código fuente

Las funciones de inspección y las sentencias que controlan las iteraciones se pudieron insertar sin inconvenientes. No obstante, la implementación del comando *agrep* presenta el inconveniente de que muchas de sus funciones finalizan usando una sentencia *exit(n)*. Esta peculiaridad permite registrar el ingreso y finalización de algunas funciones mientras que otras no. Este problema impide el funcionamiento de la estrategia BORS. Esto es porque no se puede recuperar la información necesaria para construir el árbol de ejecución de funciones (fe-tree). Para solucionar este problema se requiere la definición e implementación de un procedimiento de verificación de trazas de ejecución. En los siguientes párrafos se describe, en términos generales, los pasos necesarios para salvar este inconveniente.

La información reportada por las funciones de inspección para una función específica del sistema consiste de las siguientes cadenas de caracteres:

```
INSPECTOR_ENTRADA F
INSPECTOR_SALIDA F
```

ambas cadenas indican el comienzo y finalización de la función inspeccionada. Cuando la función termina con una sentencia *exit(n)* la cadena INSPECTOR_SALIDA *función* no se registra. En esas situaciones se necesita ejecutar la rutina de verificación de trazas de ejecución. Esta rutina utiliza una pila para la verificación. Cada vez que se encuentra una cadena INSPECTOR_ENTRADA *función* el nombre de la función se inserta en la pila. Cuando aparece una cadena INSPECTOR_SALIDA *función* con el mismo nombre de función que el encontrado en la pila, se elimina el elemento del tope de la pila. Si todas las cadenas de entrada fueron procesadas y la pila contiene elementos (funciones sin finalización) entonces se generan automáticamente las correspondientes cadenas INSPECTOR_SALIDA *función*. Esta aproximación hace posible el uso de la estrategia BORS y como un efecto colateral permite descubrir las posibles finalizaciones del sistema de estudio por condiciones de error o anormales.

5.5. Insertar las declaraciones del monitor

Esta tarea es dependiente del sistema, en muchas situaciones sólo se debe insertar en cada módulo del sistema los prototipos de las funciones implementadas en el monitor. Por otra parte, se deben incorporar en el módulo que contiene la función main (el módulo principal) las definiciones de cada una de las funciones. No obstante, en otras situaciones las definiciones también se deben insertar en diferentes partes del sistema de estudio. En el caso de *agrep* fue necesario insertar en el módulo *main.c* las definiciones de las funciones del monitor y en los módulos restantes los prototipos. Esta tarea se pudo realizar sin inconvenientes.

5.6. Compilar el sistema

Como fue declarado en la sección 4 las funciones de inspección hacen uso de librerias gráficas para mostrar la información recuperada adecuadamente. Para poder compilar el monitor y el sistema de estudio se deben incorporar los parámetros necesario para ligar las rutinas de la libreria gráfica con el sistema. Afortunadamente, el archivo Makefile de *agrep* es sencillo y

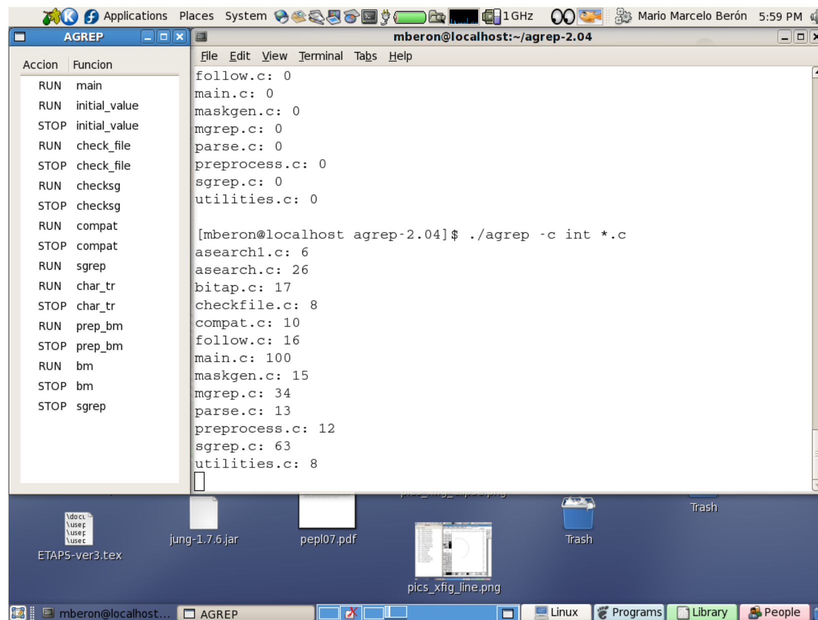

Figura 2: SVS aplicado al comando agrep

la incorporación de estos parámetros se realizó modificando la variable correspondiente a los parámetros del compilador (CFLAGS) como se muestra a continuación:

```
CFLAGS        = ‘pkg-config gtk+-2.0 --cflags --libs‘ -lpthread -O
```

en esa de línea se incorporó ‘pkg-config gtk+-2.0 –cflags –libs‘ y -lpthread. La primera de ellas permite el uso de las librerias gtk+-2.0 para la visualización de la información. La segunda se utiliza para la ejecución paralela del sistema y del monitor. Con estas modificaciones el comando *agrep* se pudo compilar sin inconvenientes.

5.7. Ejecutar el sistema

En este momento se posee una versión modificada del comando *agrep* que permite visualizar las funciones usadas para una búsqueda de cadenas específica. A continuación se muestran algunos ejemplos del funcionamiento de SVS y BORS.

5.7.1. SVS aplicado a agrep

Un ejemplo de uso de SVS consiste en detectar que funciones son utilizadas por el comando *agrep* para contabilizar el número de veces que una cadena aparece en cada uno de sus módulos .c . La forma de invocar a *agrep* es la siguiente:

```
$>agrep -c int *.c
```

esta invocación de *agrep* debe retornar como resultado la cantidad de veces que la cadena *int* aparece en cada uno de los módulos con extención C. La Figura 2 muestra el resultado obtenido luego de aplicar SVS al comando *agrep*.

El lector puede observar la relación directa entre los dominios del problema y programa. Además, se puede ver que algunas funciones no finalizan. Esto se debe al uso de sentencias *exit(n)*.

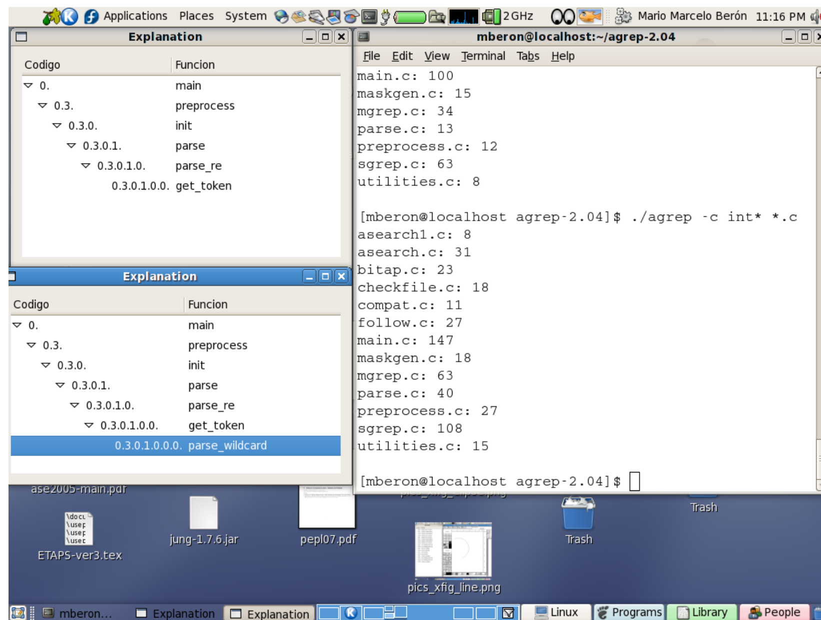

Figura 3: BORS aplicado al comando agrep

.

5.7.2. BORS aplicado a agrep

BORS utiliza la misma información que SVS pero después de la ejecución del comando *agrep*. En este caso, la implementación de BORS retorna como resultado los contextos donde la función que se desea explicar fue invocada.

Para aplicar esta estrategia se deben indicar que funciones se desean explicar. Luego de realizado este trabajo se construye el árbol de ejecución de funciones y se aplica el procedimiento descripto en sección 3.2.

Suponga que el usuario desea analizar que funciones del módulo *parse.c* se ejecutaron para el mismo ejemplo presentado para SVS. En este caso las funciones que BORS recibe como entrada son: *parse_wildcard*, *parse_chlit*, *parse_cset*, *get_token*, *parse_re*, *mk_leaf*, *mk_alt*, *parse*, *wrap* y *cat2*. La Figura 3 muestra el resultado obtenido por BORS luego de la contrucción del árbol de ejecución de funciones y de la aplicación del barrido por niveles.

El lector puede observar que el contexto donde las funciones bajo estudio se invocaron se identifica claramente. Finalmente, es importante notar que para la aplicación exitosa de BORS se debió usar previamente la rutina de verificación de trazas de ejecución.

6. Conclusión

En este artículo se presentaron diferentes aproximaciones para facilitar la comprensión de sistemas de software. Básicamente las propuestas descriptas relacionan los dominios del problema y programa. Esta característica es una debilidad de las herramientas de comprensión de programas actuales cuyas principales fortalezas se basan en la implementación precisa de técnicas de análisis estático y dinámico. Por esta razón este artículo presenta importantes contribuciones en el contexto de la Comprensión de Programas.

Las estrategias presentadas relacionan los dominios del problema y programa de dos maneras

diferentes. La primera de ellas, SVS, lo hace mientras el sistema se está ejecutando. La segunda, BORS, realiza una tarea similar pero después de la ejecución del sistema. Ambas estrategias utilizan técnicas de análisis estático tradicionales, instrumentación de código para recuperar información dinámica y un conjunto de pasos definidos por los autores para alcanzar el objetivo.

Las estrategias mencionadas en el párrafo precedente y algunas técnicas de inspección de código fueron implementadas en PICS una herramienta para comprender sistemas escritos en lenguaje C. Esta herramienta fue utilizada para probar ambas estrategias (SVS y BORS) tomando como caso de estudio el comando *agrep* del sistema operativo Linux. Los resultados fueron positivos porque fue posible identificar claramente las funciones utilizadas por *agrep* para realizar algunas de sus tareas específicas.

Finalmente se describió una estrategia alternativa denominada *Análisis Comportamental*. Este procedimiento es semi-automático y permite relacionar una descripción del dominio del problema, contruida por el usuario, con diferentes trazas de ejecución. Pensamos que esta última aproximación es muy interesante porque permite relacionar los conceptos utilizados en el sistema con las componentes usadas en su implementación. No obstante, se necesita la definición de algunos procedimientos para extraer más información desde las trazas de ejecución. Creemos que esta tarea se puede llevar a cabo a través del uso de técnicas de análisis de trazas de ejecución tales como: resumir las trazas, extracción de conceptos, etc.

Referencias

[Abd05] H. Abdelawahab. The Concept of Trace Summarization. *Program Comprehension through Dynamic Analysis*, 1:43–46, 2005.

[BHU07] M. Berón, P. Henriques, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Technical Report*, 2007.

[BHVU06] M. Beron, P. Henriques, M. Varanda, and R. Uzal. A Language Processing Tool for Program Comprehension. *Congreso Argentino de Ciencias de la Computacion (CACIC06)*, 2006.

[Bro78] R. Brook. Using a behavioral theory of program comprehension in software engineering. *Proceedings of the 3rd international conference on Software engineering*, pages 196–201, 1978.

[HL05] Abdelwahab Hamou-Lhadj. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. PhD thesis, Ottawa-Carleton Institute for Computer Science - University of Ottawa, 2005.

[LF94] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *ACM Conference on Computers and Human Interface*, Denver, Colorado, April 1994.

[MMW02] K. Mens, T. Mens, and M. Wermelinger. Supporting software evolution with intentional software views. *Proceedings of the International Workshop on Principles of Software Evolution*, pages 138–142, 2002.

[RD94] S. Rifkin and L. Deimel. Applying Program Comprehension Techniques to Improve Software Inspections. *Proceedings of the 19th Annual NASA Software Engineering Laboratory Workshop, Greenbelt, MD, Nov*, 1994.

[Sto98] Margaret A. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, 1998.

[Sto05] M.A. Storey. Theories, methods and tools in program comprehension: past, present and future. *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.

[VMV95] A. Von Mayrhauser and AM Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[XPH07] Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining software engineering data. In *Proc. 29th International Conference on Software Engineering (ICSE 2007), Companion Volume, Tutorial*, pages 172–173, May 2007.