Instrumentaciones de Programas Escritos en C para Interrelacionar las Vistas Comportamental y Operacional de los Sistemas de Software

Mario Beron[1], Germán Montejano[1], Pedro Henriques[2] y Maria J. Pereira[3]

[1] Universidad Nacional de San Luis, San Luis, Argentina
[2] Universidade do Minho, Braga, Portugal
[3] Instituto Politécnico de Bragança, Bragança, Portugal
{mberon,gmonte}@unsl.edu.ar
predrorangelhenriques@gmail.com
mjoao@ipb.pt

Resumen La *Comprensión de Programas* es una disciplina de la Ingeniería de Software cuyo principal objetivo es simplificar la comprensión de los sistemas de software. Una forma de alcanzar esta finalidad consiste en el desarrollo de estrategias de comprensión que permitan relacionar el Dominio del Problema, es decir el comportamiento del sistema, con el Dominio del Programa, o sea las construcciones del lenguaje de programación utilizadas en la implementación del programa. La creación de este tipo de estrategias requiere de la: i) Definición de representaciones de los Dominios del Problema y Programa; ii) Elaboración de un procedimiento que vincule ambas representaciones y iii) Recuperación de información estática y dinámica del sistema.
En este artículo, se describe una estrategia que recupera información dinámica con el propósito de facilitar la elaboración e implementación de estrategias de comprensión. Además, se presenta SVS (**S**imultaneous **V**isualization **S**trategy), un estrategia que permite relacionar los Dominios del Problema y Programa por medio de la utilización de información de tiempo de ejecución.

Palabras Claves: Comprensión de Programas, Dominio del Problema, Dominio del Programa, Instrumentación de Código.

1. Introdución

La *Comprensión de Programas* (CP) es una disciplina de la Ingeniería de Software cuyo objetivo principal es simplificar el análisis y el entendimiento de los sistemas de software. CP es útil para *Mantenimiento y Evolución de Software* (MES), *Ingeniería Inversa* (IR) y *Re-Ingeniería* (ReI).

MES es una tarea crítica porque implica actividades: *Perfectivas*, *Correctivas* y *Adaptativas*. La primera está relacionada con la incorporación de nuevas funcionalides. La segunda describe el proceso de eliminar los errores de programación. Finalmente, la tercera se interesa con la adaptación del sistema a nuevos contextos. Esas tres actividades consumen mucho tiempo y esfuerzos. 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 refiere a activades adaptativas. Finalmente, el 2.1 % hace referencia a otras actividades de MES. Por otra parte, la historia del MES muestra un incremento lineal, desde 1975 hasta 2005 en los costos de proyectos dedicados a MES [XPH07].

Otra área muy importante es IR. Esta disciplina se interesa con la elaboración de estrategias de extracción de la información desde diferentes fuentes. Por ejemplo: *Código Fuente*; *Makefiles*; *Documentación*; etc. Esta información se procesa con el objetivo de proporcionar: *Nueva Documentación*; la *Arquitectura del Sistema*; etc. IR implica grandes costos y consume mucho tiempo porque incluye actividades similares a MES.

ReI usa técnicas de IR para extraer la información del sistema y luego aplica estrategias para modificar el sistema de acuerdo a los requerimientos del programador. Por esta razón, ReI, de la misma manera que MES e IR, es un área donde se necesitan propuestas para minimizar costos y esfuerzos.

En todas las disciplinas mencionadas en los párrafos precedentes el programador debe entender grandes documentos con formalismos y metodologías diferentes. Además, debe comprender las funcionalidades del sistema en un alto nivel de abstracción.

Una forma de minimizar los costos y esfuerzos implicados en las disciplinas mencionadas previamente es por medio de la elaboración de estrategias de comprensión que permitan relacionar los Dominios del Problema (vista comportamental) y Programa (vista operacional) [11,9]. La elaboración de este tipo de estrategias requiere de la construcción de representaciones de los Dominios del Problema y Programa y de la recuperación de la información estática y dinámica del sistema.

En este artículo se presenta una estrategia que posibilita la extracción de información de tiempo de ejecución que es útil para la elaboración de estrategias de comprensión que vinculen la salida del sistema (Dominio del Problema) con su operación (Dominio del Programa). Además presenta SVS (Simultaneous Visualization Strategy), un estrategia que vincula los Dominios del Problema y Programa por medio de la utilización de la información dinámica del sistema. El artículo está organizado como sigue. La sección 2 describe el esquema de Instrumentación de Código para diferentes construcciones del lenguaje de programación (en este caso C). La sección 3 explica SVS y muestra ejemplos de su aplicación a *xfig* (un sistema de dibujos comunmente proporcionado por las distribuciones del sistema operativo Linux). Finalmente, la sección 4 presenta las conclusiones.

2. Instrumentación de Código

En esta sección se describe una estrategia que permite recuperar información dinámica desde el sistema denominada *Instrumentación de Código* [5]. La idea básica de esta aproximación consiste en insertar sentencias dentro del código fuente del sistema con el propósito de recuperar información de tiempo de ejecu-

ción que pueda ser utilizada para: *La elaboración de estrategias de comprensión*; *Cálculo de métricas*; *Explopración de trazas de ejecución*; etc [7,2,13,6,12].

En las subsecciones siguientes se presenta la aproximación seguida para instrumentar diferentes construcciones del lenguaje de programación.

2.1. Instrumentación de Funciones

La estrategia usada para instrumentar las funciones del sistema consite en insertar funciones de inspección dentro del código fuente. Esas funciones tienen como finalidad capturar el flujo de ejecución del programa y otra información que el usuario desee conocer. Para llevar a cabo esta tarea se deben tomar dos decisiones importantes: i) ¿Cuáles son los puntos de verificación? y ii) ¿Qué información se debe recuperar para conocer el flujo de ejecución del programa a nivel funciones?

Teniendo presente el primer ítem, el sistema se puede conceptualizar como una máquina de estados. En esta máquina, los estados están compuestos por las variables globales y aquellas declaradas en la función *main*, y sus transiciones están definidas por las funciones. El segundo, considera la recuperación de estos datos. Un punto interesante para llevar a cabo esta tarea es el comienzo y fin de cada función, porque en esos lugares se pueden conocer el estado actual y el próximo. Las sentencias insertadas son funciones de inspección (o también conocidas como *inspectores*) que procesan y muestran esta información.

En este contexto, cada vez que el sistema comienza su ejecución invoca sus funciones y la primer y última sentencia ejecutada por ellas son los inspectores. Esas funciones pueden inicialmente imprimir el nombre de la función ejecutada y otra información que el usuario quiera conocer, como por ejemplo parámetros, variables globales, etc.

Aunque las funciones de inspección insertadas en el final no son estrictamente necesarias, son incluídas en el código fuente porque proporcionan la información requerida para construir algunas estructuras de datos interesantes empleadas por las estrategias que intentan relacionar los Dominios del Problema y Programa [4,8]. A continuación se presentan diferentes casos que se deben considerar cuando se desea instrumentar las funciones siguiendo la idea descripta en los párrafos precedentes.

1. Las funciones pueden tener dos o más puntos de salida: la introducción de esta sección describe la mejor situación cuando las funciones tienen un punto de entrada y un punto de salida. Sin embargo, en sistemas reales, las funciones no siempre poseen esta característica[4]. En esos casos, la función de inspección de salida se debe insertar antes de cada sentencia *return*.
2. Las funciones pueden tener puntos de salida dentro de selecciones e iteraciones: este problema se resuelve transformando la sentencia *return* en una

[4] Esta característica aparece cuando las sentencias *return* se encuentran dentro de selecciones e iteraciones o cuando se usan sentencias *goto*. Las dos primeras se describen en otros ítems de este trabajo.

sentencia compuesta con las siguientes características: i) La primer sentencia es la invocación a la función de inspección de salida y ii) La última es la sentencia *return*.

3. Las funciones no tienen un punto de salida explícito: en esos casos, la estrategia debería insertar las funciones de inspección de salida al final de esas funciones.
4. Si los puntos de salida no están correctamente instrumentados el flujo de control del programa puede no ser correcto: para resolver este problema, se debe especializar la técnica de instrumentación en los puntos de salida. Esta tarea se lleva a cabo transformando la sentencia *return* en una sentencia compuesta siguiendo los pasos descriptos a continuación: i) Declarar una variable cuyo tipo es el tipo de retorno de la función; ii) Generar una sentencia de asignación. Esta sentencia tiene en su parte izquierda (*lvalue*) la variable definida en el paso previo y su parte derecha (*rvalue*) es la expresión encontrada en la sentencia *return*; iii) Insertar una función de inspección de salida antes de la sentencia *return* y iv) Cambiar la expresión dentro de la sentencia *return* por la variable definida en el primer paso.
5. La función puede tener puntos de salida explícitos e implícitos: en este caso se deben aplicar los pasos descriptos en el ítem anterior a los retornos de función explícitos. Además de eso, es necesario la inserción de una función de inspección al final de la función o procedimiento.

Luego de la discusión de las distintas alternativas que se pueden presentar cuando se desean instrumentar funciones es posible afirmar que el esquema de instrumentación de estas construcciones de los lenguajes de programación debe: i) Insertar el inspector de entrada al comienzo de cada función; ii) Instrumentar las sentencias *return* siguiendo la aproxiamción descripta en el ítem 4 e iii) Insertar un inspector de salida al final de cada función. La Figura 1 muestra un ejemplo de instrumentación de las funciones.

```
int f (int x, int y) {
     ...........
     ............
     ..............
     return h(x^100);
     .....
}
```

```
int f (int x, int y) {
    INSPECTOR_DE_ENTRADA("f");
    ..............
    ............
    {
      int nruter;
       nruter=h(x^100);
       INSPECTOR_DE_SALLIDA("f");
       return nruter;
    }
    ...........
    INSPECTOR_DE_SALIDA("f");
}
```

Figura 1. Instrumentación de Funciones

2.2. Instrumentación de Iteraciones

En la práctica, la estrategia de instrumentación de las funciones tiene un problema: *El número de funciones registradas puede ser muy grande* [3]. Trabajar con todas las invocaciones no es apropiado porque: i) Dificulta la comprensión [1,3] debido a la sobrecarga de información y ii) La implementación de algoritmos eficientes de análisis y exploración de trazas de ejecución es muy complicada por la gran cantidad de datos que manipulan. Este problema emerge cuando las funciones se invocan dentro de las iteraciones. Esta es la razón principal para controlar el número de veces que se deben mostrar las funciones encontradas dentro de este tipo de construcciones de los lenguajes de programación [10,6].

```
for(inic.; cond.; acc.)
 acciones
```

```
1
for(inic.; cond.; acc.)
  {
       acciones
       2
  }
 3
```

Figura 2. Instrumentación de una Iteración For

Para resolver el problema descripto en el párrafo precedente, se distinguen tres puntos principales en las iteraciones, identificados en la Figura 2 por los números 1, 2 y 3. En 1 y 3, se inserta una función de control llamada *showFunction(value)*. Esta función tiene un doble propósito: i) Cuando se encuentra al inicio de una sentencia de iteración, inserta en el tope de la pila un valor que indica la cantidad de veces que los inspectores deben reportar la información de la función invocada dentro de la iteración y ii) Cuando se encuentra al final, elimina el valor que se encuentra en el tope de la pila con la finalidad de recuperar la cantidad de veces que se deben mostrar las funciones invocadas en la iteración anterior. En 2, se coloca otra función de control, denominada *dec()*, que decrementa el valor almacenado en el tope de la pila. Cuando este valor es cero, las funciones de inspección de entrada (ubicada al comienzo de la función invocada) y salida (ubicada antes de cada sentencia *return* utilizada por la función invocada) no muestran la información asociada a la función.

Los párrafos previos presentan una idea genérica de como llevar a cabo la instrumentación de las sentencias de iteración. No obstante, se deben considerar diferentes casos:

1. La iteración puede contener solamente una sentencia dentro de su cuerpo: en esta situación se coloca el cuerpo de la iteración dentro de una sentencia compuesta junto con una invocación a la función *dec()* ubicada al final de la sentencia compuesta.

2. Las iteraciones pueden encontrarse dentro de selecciones u otras iteraciones: en este caso la sentencia de iteración se coloca dentro de una sentencia compuesta y se aplica el esquema descripto en el ítem previo.
3. El cuerpo de la iteración puede tener sentencias de salto incondicional: en los próximos párrafos, se describen los pasos requeridos para la instrumentación correcta de las iteraciones cuando en su cuerpo se encuentran este tipo de sentencias.
   - *break*: el objetivo de la sentencia *break* es pasar el flujo de control a la primera sentencia después de la iteración. Este efecto no modifica la estrategia de control de las iteraciones porque la próxima sentencia después de la ejecución del *break* es una función *showFunction(valor)* que realiza una operación *pop* sobre la pila de control. De esta manera, se recupera el número de veces que la iteración previa debe mostrar las funciones invocadas en su interior. Es importante notar que este efecto mantiene la intención del esquema de instrumentación de las iteraciones.
   - *continue*: un caso diferente aparece con la sentencia *continue*. El objetivo de esta primitiva es producir un salto incondicional a la condición de la iteración. Esta situación viola la estrategia de control de las iteraciones porque algunas funciones se pueden mostrar muchas veces. Para solucionar este problema, es necesario el cambio de la sentencia *continue* por una sentencia compuesta que contenga: i) Una invocación a la función *dec()* que decremente el número que se encuentra almacenado en el tope de la pila y ii) La correspondiente sentencia *continue*.
   - *return*: cuando la sentencia *return*[5] está dentro de una iteración se debe sustituir con una sentencia compuesta que realize las siguientes operaciones: i) Suprimir todos los elementos de la pila y ii) Aplicar la transformación expuesta en la sección 2.1 para la sentencia *return* .
   - *goto*: generalmente las sentencias *goto* promueven malas prácticas de programación, por esta razón no se consideran en este esquema de instrumentación.

Luego del análisis presentado en los ítems previos es posible afirmar que el esquema de instrumentación de las iteraciones debe: i) Colocar la sentencia de iteración dentro de una sentencia compuesta; ii) Insertar como primer y última sentencia de la sentencia compuesta una invocación a la función *showFunction(valor)*; iii) Colocar el cuerpo de la iteración dentro de una sentencia compuesta que contiene: el cuerpo de la iteración bajo análisis y una invocación a la función *dec()*.

La Figura 3 muestra un ejemplo del esquema de instrumentación de las iteraciones para una sentencia *for*. Es importante notar que el esquema de instrumentación presentado en esta sección también se aplica para iteraciones implementadas con las sentencias *while* y *do-while*.

[5] El uso de sentencias *return* dentro de las iteraciones no es considerada una buena práctica de programación y por consiguiente es de poco interés desde el punto de vista de las instrumentaciones de código.

```
for (acc1; acc2; acc3)          {
  {                               showFunction(valor);
    ..........                    for(acc1; acc2; acc3)
  }                               {
                                    ..................
                                    dec();
                                  }
                                   showFunction(valor);
                                 }
```

Figura 3. Instrumentación de Iteraciones

2.3. Instrumentación de las Sentencias de Selección

La instrumentación de funciones y la estrategia de control de las iteraciones son importantes porque proveen información de alto nivel y reducen el volumen de los datos. Sin embargo, algunas veces es útil obtener más información que solamente el nombre de las funciones y valores de variables. Por ejemplo, conocer la parte de las sentencia de selección (*then* o *else*) que se ejecutó puede ser una ayuda para comprender el comportamiento de funciones específicas. Por esta razón, esta sección tiene como objetivo la descripción de una estrategia para instrumentar esta clase de sentencias.

Instrumentación de la Sentecia IF-THEN-ELSE La finalidad de la instrumentación de la sentencia *if* es conocer cual de sus partes (*then* o *else*) se ejecutó [2].

Para alcanzar el objetivo mencionado con anterioridad en cada parte de la sentencia *if* (*then*, *else*) se debe colocar dentro de una sentencia compuesta cuyo contenido es: i) Una invocación a un *inspector* que indica que parte de la sentencia se ejecutó y ii) El cuerpo de la sentencia *if*.

La idea se presenta en la Figura 4. El lector puede observar que la sentencia *if* y las sentencias correspondientes a las partes *then* y *else* se colocan dentro de sentencias compuestas. El uso de estas sentencias compuestas evita dos problemas: i) Introducir errores sintácticos y ii) Cambiar la semántica del programa.

De la misma forma que las sentencias descriptas en las secciones previas, la sentencia *if* puede estar dentro de iteraciones. Por esta razón, se debe controlar la cantidad de información que recuperan las funciones de inspección. En este caso se puede usar un esquema similar al definido para las iteraciones.

```
{if (condición) then  {IF_INSPECTOR(CONDICIÓN-VERDADERA); sentencias;}
       else {IF_INSPECTOR(CONDICIÓN-FALSA); sentencias;}}
```

Figura 4. Instumentación de la Sentencia IF-THEN-ELSE

La utilización de una misma pila para controlar las iteraciones y las sentencias de selección puede causar algunos problemas, por ejemplo algunas sentencias *if* se pueden ignorar por no encontrarse dentro de las iteraciones. Para evitar este problema es necesario que las funciones que insertan y suprimen valores en la pila de control se inserten al inicio y al final de la sentencia *if* lo cual puede generar códigos complejos de entender cuando se usan iteraciones y selecciones al mismo tiempo. Una posible forma de solucionar este problema es usar otra pila, definir inspectores específicos para la sentencia *if* y luego aplicar una idea similar a la utilizada para las iteraciones (ver sección 2.2). Este esquema, además de proporcionar una administración clara de las estructuras de datos utilizadas en la instrumentación, posibilita: i) La impresión de la información correspondiente a las selecciones ubicadas fuera de las iteraciones y ii) Controlar el número de veces que se debe mostrar la información de la sentencia *if* dentro de una iteración.

Instrumentación de la Sentencia SWITCH La instrumentación de la sentencia *switch* sigue una aproximación similar a la utilizada para la sentencia *if* con la diferencia que las funciones de inspección se insertan en cada opción del *switch*. El resto del esquema, es decir el control de las iteraciones es igual al descripto para la sentencia *if*.

3. SVS: Simultaneous Visualization Strategy

SVS es una estrategia que relaciona los Dominios del Problema y Programa por medio de la utilización del esquema de instrumentación definido en la sección 2.

La puesta en funcionamiento de SVS requiere: i) La instrumentación del sistema de estudio; ii) La definición de un Monitor que contiene la implementación de las funciones de inspección y iii) La inserción de primitivas que permitan la ejecución paralela del sistema instrumentado y el Monitor.

Como se puede observar la idea es sencilla. No obstante, la implementación del esquema de instrumentación utilizado por SVS es muy compleja porque requiere de mucha precisión para evitar cambiar la semántica del sistema. El lector interesado en estudiar una descripción más acabada de SVS y otras estrategias que posibilitan relacionar los Dominos del Problema y Programa puede ver [4].

SVS fue aplicado a *xfig*, un sistema que permite realizar gráficos. En términos generales, las tareas que se pueden hacer con este sistema son las siguientes: Dibujar círculos, cuadros, líneas, etc; Importar y exportar imágenes en diferentes formatos; Crear, suprimir, mover y modificar objetos; Seleccionar cada uno de los atributos de los objetos de diferentes maneras; Almacenar las figuras en un formato propio (*.fig*) o convertirlas a formatos tales como *PostScript*, *GIF*, *JPEG*, *HP*, etc; Imprimir las imágenes con extensión *.fig.* u otros formatos tales como *PostScript*, *GIF*, *JPEG*, *HP*; etc. Esta aplicación consta de aproximadamente 96.000 líneas de código y presenta muchas construcciones de programación que permiten probar la robustez del esquema de instrumentación.

La Figura 5 muestra la utilización de SVS para la detección de las funciones utilizadas por *xfig* para graficar un círculo. El lector puede observar la relación directa entre el Dominios del Problema (ventana de *xfig*) y el Dominio del Programa (ventana con nombres de funciones en su interior).

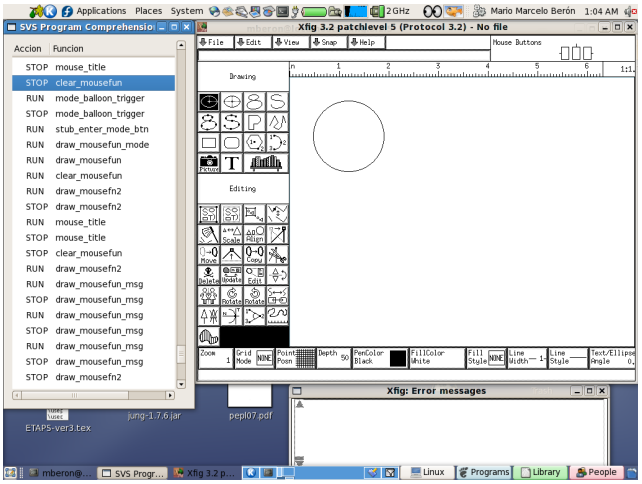

Figura 5. Funciones ejecutadas para hacer un círculo

Este simple ejemplo exterioriza dos ventajas principales del trabajo presentado en este artículo: i) La utilidad de estrategias de comprensión que relacionen los Dominios del Problema y Programa para disminuir los costos y esfuerzo humano invertido en tareas de MES, IR y ReI y ii) La pertinencia de la información recuperada por el esquema de instrumentación.

4. Conclusión

En este artículo se presentó un esquema de instrumentación de código que permite recuperar información dinámica de los sistemas. Esta estrategia se basa en la inserción de funciones de inspección, al inicio y al final de cada función del sistema, que reportan el nombre de la función ejecutada y otra información que el usuario considere apropiada para facilitar la elaboración de estrategias de entendimiento y el proceso de comprensión de software. Además, el esquema propone una forma de controlar las iteraciones que posibilita la reducción de la información provista por las funciones de inspección cuando las invocaciones de funciones se encuentran dentro de las iteraciones. Por otra parte, cuando se desea explorar el funcionamiento de una o varias funciones específicas del sistema es posible obtener información de más bajo nivel instrumentando las sentencias de selección. Como paso siguiente se presentó SVS una estrategia de comprensión

cuya finalidad es simplificar el proceso de comprensión a través de la interrelación de los Dominios del Problema y Programa. Es importante notar que este tipo de estrategias no son frecuentemente implementadas en las herramientas de comprensión actuales (en [4] se puede encontrar el sustento a esta afirmación), por consiguiente el trabajo presentado en este artículo representa una contribución interesante en el contexto de la Comprensión de Programas. Como trabajo futuro se planifica la elaboración de estrategias de instrumentación que permitan controlar la recursión y la definición de representaciones del Dominio del Problema más completas que la utilizada por SVS que consistió en la salida del sistema en sí misma.

Referencias

1. H. Abdelawahab. The Concept of Trace Summarization. *Program Comprehension through Dynamic Analysis*, 1:43–46, 2005.
2. R. Akers. Using Build Process Intervention to Accomodate Dynamic Instrumentation of Complex Systems. *Program Comprehension through Dynamic Analysis*, 1:1–5, 2005.
3. C. Bennet, D. Myers, M. A. Storey, and D. German. Working with Monster Traces: Building a Scalable, Usable Sequence Viewer. *Program Comprehension through Dynamic Analysis*, 1:1–5, 2007.
4. M. Beron, P. Henriques, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Technical Report*, 2007.
5. 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.
6. B. Cornelissen and L. Moonen. Visualizing Similarities in Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:6–10, 2007.
7. M. Denker, O. Greevy, and O.Ñierstrasz. Supporting Feature Analysis with Runtime Annotations. *Program Comprehension through Dynamic Analysis*, 1:29–33, 2007.
8. R. Fonseca, P. Henriques, and M. Pereira. Webappviewer: Uma ferramenta para compreender aplicações web. *Licenciate Thesis at University of Minho*, 2006.
9. J. Ka-Yee and Y. Gueheneuc. Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. *Program Comprehension through Dynamic Analysis*, 1:34–42, 2007.
10. A. Kuhn, O. Greevy, and T. Girba. Applying Semantic Analysis to Feature Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:48–53, 2005.
11. 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.
12. A. Lienhard, T. Girba, O. Greevy, and O.Ñierstransz. Exposing Side Effects in Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:10–17, 2007.
13. A. Rohatgi, A. Hamou Lhadjm, and J. Rilling. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. *International Conference on Program Comprehension (ICPC08)*.