Instrumentación de Programas Escritos en Java para Interconectar los Dominios del Problema y del Programa

Hernán Bernardis

Departamento de Informática - Facultad de Ciencias Físico-Matemáticas y Naturales
Universidad Nacional de San Luis
Ejército de los Andes 950 – San Luis - Argentina
hernanbernardis@gmail.com

Resumen La Comprensión de Programas (CP) es una disciplina de la Ingeniería de Software cuyo objetivo es facilitar el entendimiento de los sistemas; mediante el desarrollo de Métodos, Técnicas, Estrategias y Herramientas que permiten comprender las funcionalidades del sistema de estudio. Uno de los principales desafíos en CP es establecer una relación entre los Dominios del Problema y del Programa. El primero se relaciona con el comportamiento del sistema de estudio; mientras que el segundo se centra en las componentes del programa para producir dicho comportamiento. Una forma de construir esta relación consiste en elaborar una representación para cada dominio y luego establecer un procedimiento de vinculación entre ambas representaciones. La tarea anterior implica extraer información de ambos dominios, para lo cual existen múltiples técnicas. En este artículo se describe un esquema de extracción de información dinámica desde el dominio del programa, que es muy útil para la implementación de estrategias de comprensión.

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

1. Introducción

La Comprensión de Programas [3,4,10,15,18] es una disciplina de la Ingeniería del Software destinada a elaborar Modelos, Métodos, Técnicas y Herramientas, basadas en procesos de aprendizaje y en procesos específicos de ingeniería, con el fin de llegar a un conocimiento profundo sobre un sistema de software. Este aprendizaje es primordial para poder establecer una relación entre el Dominio del Problema y el Dominio del Programa [9,11]. Es decir, la relación real entre los aspectos relacionados al comportamiento del sistema en estudio y las componentes del mismo utilizadas para producir dicho comportamiento, respectivamente. Para reproducir la relación entre ambos dominios, mencionada anteriormente, se debe:

1. Construir una representación para el dominio del problema,

2. Construir una representación para el dominio del programa,
3. Establecer una vinculación entre ambas representaciones.

En la Figura 1 se muestra el modelo para la reconstrucción de la relación entre el Dominio del Problema y el Dominio del Programa, definido en los pasos anteriores.

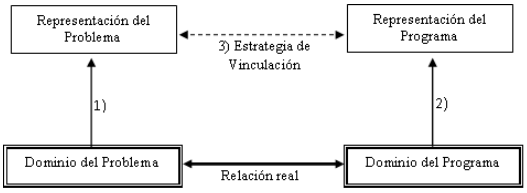

Figura 1. Modelo de la relación real existente entre el Dominio del Problema y el Dominio del Programa.

Los pasos 1) y 2) descriptos en el procedimiento de los párrafos precedentes implican la extracción de la información de ambos dominios (Cabe mencionar que estos pasos se pueden realizar de forma independiente). En este contexto entran en juego un conjunto de técnicas desarrolladas, las cuales son utilizadas para este fin. Las existentes aplicadas al Dominio del Programa se pueden clasificar en estáticas y dinámicas. Las primeras son similares a las utilizadas para la construcción de compiladores. Es decir, reciben como entrada un código fuente, realizan un análisis sintáctico sobre el mismo y producen, generalmente como salida, un Árbol de Sintaxis Abstracta (AST). Sobre este árbol, se definen diferentes tipos de recorridos para extraer la información deseada del sistema de estudio y realizar diferentes tipos de chequeo semántico. Por ejemplo, si el objetivo es construir una tabla de símbolos, se pueden definir recorridos para recuperar identificadores de variables, funciones y tipos; entre otros tantos objetos y relaciones entre ellos que se pueden extraer desde el código fuente.
La aproximación descripta en el párrafo precedente es más compleja que realizar un análisis sintáctico y semántico sin construir explícitamente el AST [1]. No obstante, el uso de AST, permite no solo recuperar mucha más información sino también realizar transformaciones de código muy complejas e importantes para: la Comprensión de Programas, Ingeniería Reversa, Optimización de Código, etc. El lector interesado en conocer diferentes técnicas de análisis estático puede leer [1,5,14].

Las técnicas de análisis dinámico usan generalmente, aunque no siempre es el caso, técnicas de análisis estático con el fin de realizar una transformación del código fuente del sistema. Esta transformación, no modifica la semántica

[1] Esta técnica se suele usar cuando las operaciones que se desean realizar en el código fuente no son tan complejas.

del mismo, pero permite obtener información respecto del comportamiento del programa en tiempo de ejecución. El punto central en este tipo de técnicas es la detección de los lugares estratégicos en donde realizar la transformación del código. Esto se debe a que la información de tiempo de ejecución de un programa es enorme y difícilmente se pueda administrar eficientemente. Por esta razón, el diseñador de este tipo de transformaciones debe tener muy en claro qué desea conocer del sistema y cuáles son los puntos estratégicos para realizar la transformación. Es importante mencionar que no se ha podido establecer un patrón genérico de transformaciones, ya que estas son muy dependientes de la información que se necesita obtener. El lector interesado en profundizar el estudio sobre las técnicas dinámicas de extracción de información puede leer [2,8,19].

En este artículo, se describen las tareas llevadas a cabo en el contexto del desarrollo de una tesis de Licenciatura en Ciencias de la Computación. Dicha tesis se centra en el análisis de una técnica dinámica de extracción de la información desde el Dominio del Programa. Esta técnica se conoce con el nombre de *"Instrumentación de Código"*; y será aplicada al Paradigma Orientado a Objetos, más precisamente sobre el lenguaje Java.

La organización de este artículo se expone a continuación. La siguiente sección describe todos aquellos aspectos relacionados a la implementación del esquema de instrumentación, aplicado a métodos e iteraciones. La sección 3 define una estrategia para la interconexión de los dominios del programa y del problema. Finalmente, en la sección 4 se exponen las conclusiones y trabajo futuro.

2. Instrumentación de Programas en Java

La Instrumentación de Código (IC) es una estrategia que consiste en la inserción de sentencias dentro del código fuente de un programa, con el propósito de extraer información de tiempo de ejecución [3,6,19]. La inserción de instrucciones debe realizarse de manera cuidadosa y controlada para:

- No afectar el comportamiento original del programa ni su lógica subyacente: al insertar sentencias se pueden producir, de manera accidental, cambios en los componentes del programa que modifiquen la idea original del mismo.
- No extraer información errónea: si las sentencias no son las adecuadas o no se colocan en los lugares apropiados del código, puede existir una inconsistencia entre la información extraída y el comportamiento real del programa. Este problema es abordado en la sección siguiente.
- No afectar demasiado el tiempo de ejecución del programa original: actualmente, los avances de la tecnología han incrementado la velocidad y el tamaño de los programas que pueden ser ejecutados. Sin embargo, no es una característica deseable que los cambios provocados por una IC retrasen demasiado los tiempos de ejecución del programa original.

En las próximas secciones se define el esquema de instrumentación mencionado en los párrafos anteriores.

2.1. Instrumentación de Métodos

Un método es un bloque de código cuya principal finalidad es realizar una tarea con un fin específico para un sistema particular. En el contexto del lenguaje Java, estos tienen la siguiente forma:

```
ModAcceso tipoRet nbreMet(/*parámetros*/){
   //sentencias del cuerpo del método
}
```

Figura 2. Estructura Genérica de un método en el lenguaje Java.

Dentro de la Programación Orientada a Objetos, la comunicación entre objetos se realiza mediante el pasaje de mensajes, los cuales provocan la ejecución de un determinado método perteneciente al objeto receptor del mensaje. Si se pretende conocer el funcionamiento de un programa, es de suma importancia conocer cuáles métodos son ejecutados y en qué orden. Por esta razón, se aplica IC para poder obtener esta información, insertando sentencias[2] dentro del cuerpo del programa fuente que indiquen la entrada y salida a cada uno de los métodos[3]. Dicha tarea puede realizarse en base a la siguiente idea: *Colocar inspectores dentro del cuerpo de cada método, al comienzo del mismo para indicar el ingreso, y al final para la salida.* Esta idea es sobre la cual se desarrolla esta investigación, mostrando todos aquellos aspectos involucrados a la misma.

A continuación se realiza el desarrollo del esquema de instrumentación propuesto en los párrafos precedentes, para lo cual se presenta una forma de instrumentar la construcción descripta en la Figura 2 (ver Figura 3).

```
ModAcceso tipoRet nbreMet(/*parámetros*/){
      IC.inspectorDeEntrada("nbreMet");
      //sentencias originales del método
      IC.inspectorDeSalida("nbreMet");
}
```

Figura 3. Instrumentación genérica de métodos en Java.

[2] En el contexto de este trabajo, se hace referencia a las sentencias de extracción de la información mediante el nombre de *inspectores*.

[3] En este caso, se realiza la impresión por pantalla de la entrada y salida, pero se podría indicar de otra forma.

De esta manera, cuando la ejecución alcanza una llamada al método e ingresa al código del mismo, se invoca al inspector de entrada. Este indica que dicho método se encuentra dentro del flujo de ejecución, de ese escenario particular, y que la misma ha comenzado. De esta forma, se obtiene valiosa información que permite realizar un seguimiento del comportamiento del programa, el cual en este caso, continúa con la ejecución de las sentencias originales del cuerpo del método. Luego, se llega al inspector de salida que se encuentra como última instrucción, el cual revela que la ejecución del método ha finalizado. Mediante estas sentencias de extracción de información, se logra establecer una IC para llevar un control de los métodos involucrados en una determinada ejecución para un determinado escenario.
La solución mostrada en la Figura 3, parece correcta, no obstante esta observación no es cierta porque, la aproximación usada presenta una serie de inconvenientes, los cuales son abordados en las siguientes secciones.

Sentencia de Retorno con Expresiones en su Interior. Una sentencia de retorno puede contener una expresión que haga que la información, obtenida desde la IC, sea falsa o inconsistente. Un ejemplo de este problema surge cuando existe un pasaje de mensajes dentro de la sentencia de retorno. Observe la siguiente figura:

```
public int g(/*parámetros*/){
     IC.inspectorDeEntrada("g");
     //sentencias originales del método
     IC.inspectorDeSalida("g");
     return(b.f());
}
```

Figura 4. Invocación a método dentro de sentencia *return*.

Cuando se invoca el método g, la sentencia *IC.inspectorDeEntrada(g)* reporta el inicio de ejecución de este. Luego, la misma procede hasta encontrar la sentencia *IC.inspectorDeSalida(nbreMet)*, la cual informa que la ejecución de dicho método *g* finalizó. Sin embargo, eso no es verdadero ya que aún falta ejecutar la sentencia *return*, la cual posee una llamada a *b.f()* que se ejecuta como parte de la ejecución de *g*. Al iniciar la ejecución de *b.f()*, su inspector de entrada indicará su comienzo, pero esto sucederá luego de que se indicó el fin de la ejecución de *g*. Esta información obtenida desde la instrumentación no es correcta debido a que la ejecución de *b.f()* se realiza aún no finalizada la de *g*. Esta inconsistencia se encuentra planteada en la Figura 5. En la parte a) se muestra el verdadero funcionamiento del programa, mientras que en la parte b) se expone el flujo de ejecución obtenido mediante la instrumentación.

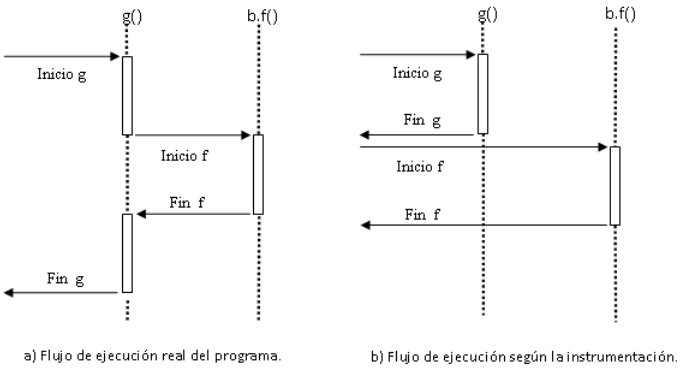

Figura 5. Comparación entre el flujo de ejecución obtenido de la instrumentación y el real.

Otro problema, similar al explicado en los párrafos anteriores, se puede presentar cuando la sentencia *return* contiene operaciones. El cálculo de las mismas debe realizarse antes de producirse el retorno y, en caso de que una falla ocurra durante su cálculo, la información extraída no será consistente. Esto se debe a que al fallar, se termina la ejecución del programa dentro del método que realiza el retorno. Sin embargo, la instrumentación indicaría que la falla se produjo luego de salir de este, debido a que el inspector de salida se ejecuta antes de que lo haga el *return*.

```
public int g(.....){
    IC.inspectorDeEntrada("g");
    //sentencias originales del cuerpo del método
    ....
    IC.inspectorDeSalida("g");
    return(( X*5)/Y );

}
```

Figura 6. Código genérico de método Java con operaciones en la sentencia *return*.

Al observar la Figura 6 se puede ver que en caso de producirse una falla con la operación dentro de la sentencia *return*, como una división por 0, el inspector de salida ya se ejecutó. Por lo tanto, la información extraída por la IC indica que el método g finalizó y luego se produjo la falla, mientras que g fue quien falló. Cabe mencionar que también se puede dar el caso particular en el cual, dentro de una sentencia *return*, exista una combinación de invocaciones y operaciones. Todos estos problemas con la instrumentación, producidos por la presencia de

expresiones dentro de la sentencia de retorno, pueden ser solucionados mediante la aplicación de una transformación del código. Dicha transformación no afecta la lógica y funcionalidad subyacente del programa, y consiste en:

1. Crear una variable del mismo tipo de retorno del método,
2. Asignar, a la variable creada en el paso previo, el valor devuelto por la expresión que se encuentra en la sentencia return,
3. Colocar la variable dentro de la sentencia de retorno,
4. Crear una sentencia compuesta formada por las sentencias definidas en los items anteriores.

Figura 7. Transformación realizada a las sentencias *return* que contienen expresiones.

Con esta transformación, la sentencia de retorno sólo contendrá una variable y las expresiones se evaluarán antes de ella. Para comprender mejor la idea expuesta en la Figura 7 se tiene:

```
ModAcceso tipoRet nbreMet(/*...*/){
   //sentencias del método


   return (expresion);
}
```

a) Código genérico original.

```
ModAcceso tipoRet nbreMet(/*...*/){
   //sentencias del  método
   {
    vbleRet = expresion;
    return (vbleRet);
   }
 }
```

b) Código genérico transformado.

Figura 8. Transformación realizada a instrucciones *return* que contienen expresiones como parámetro.

En la Figura 8 se pueden observar los cambios que implica la transformación. En la parte a) de la misma, se muestra el código genérico de un método en java con una expresión en su sentencia *return*. En la parte b) se expone el código del mismo método de a), pero con la transformación indicada en la Figura 7 aplicada. De esta manera, el problema provocado por las expresiones dentro de las sentencias de retorno se elimina. Esta solución provoca que los *return* sólo contengan variables cuyos valores ya han sido calculados y asignados anteriormente. En caso de que ocurra algún problema con estos cálculos, se detecta antes de la instrucción que indica la salida del método, y la información extraída es consistente respecto de la ejecución.

Métodos sin Sentencia de Retorno. Hasta ahora se realizó el análisis de aquellos casos en que los métodos contienen sentencias de retorno, pero la pre-

sencia de los mismos no es obligatoria en Java. Pueden existir métodos que no contengan sentencias *return* y su salida simplemente se dé por el fin del código de los mismos[4]. Este caso es simple de instrumentar debido que existe un único punto de salida y no requiere transformación. Sólo basta colocar el *inspector* que controla la salida como última sentencia del cuerpo del método, justo antes de la llave de cierre (}) del mismo, como se muestra en la siguiente figura:

```
public int g(.....){
     IC.inspectorDeEntrada("g");
     //sentencias originales del cuerpo del método
     IC.inspectorDeSalida("g");
}
```

Figura 9. Código genérico de método Java sin *return*, transformado e instrumentado.

Para los casos planteados en esta sección y las anteriores, basta buscar el final del método y realizar la instrumentación teniendo en cuenta los casos posibles presentados. Sin embargo,puede darse el caso en el que esto no sea suficiente. Este problema junto con su posible solución se trata en la siguiente sección.

Análisis para Múltiples Instrucciones *return*. Hasta ahora se han considerado aquellos casos en los cuales los métodos contienen un único punto de salida, con o sin sentencia de retorno. Sin embargo, un método puede presentar varios puntos de finalización del mismo. Esto es producido por la presencia de múltiples sentencias de retorno dentro del cuerpo del método, tal como se muestra en la Figura 10 parte a). Si no se realiza la instrumentación sobre todos ellos, la extracción de la información sería incompleta e inconsistente. La salida del método podría darse por un punto no instrumentado y no ser registrada, con lo cual la información obtenida no serviría demasiado para aplicar procedimientos de comprensión del comportamiento del programa [4,12,17].

En los casos planteados en la sección anterior, se debía encontrar el fin del método y luego, en base a la existencia o no de *return*, realizar la instrumentación adecuada. Ahora, con la presencia de múltiples sentencias de retorno se requiere un estudio más detallado del código. El análisis se debe extender sobre todo el cuerpo del método para localizar todos los puntos de salida del mismo. Una vez que se han identificados dichos puntos, para cada caso particular se aplica la técnica, descripta en las secciones precedentes, que sea adecuada. Lo expuesto en este párrafo se muestra en la Figura 10 parte b). Observando la Figura 10 también se puede ver que la primer sentencia de retorno (línea 5), que simplemente contiene una variable para retornar, no recibe transformación debido a que la misma no es necesaria. Por otro lado, en la línea 16 de la parte a), la sentencia

[4] Esto ocurre cuando la ejecución llega a la llave de cierre (}) final del cuerpo del método.

```
ModAcceso tipoRet nbreMet(/*...*/){
1
2    ... // sentencias sin return
3
4
5    return(z);
6
7    ... // sentencias sin return
8
9
10
11   return(j/3);
12
13   ... // sentencias sin return
14   if (condicion)
15
16      return(i);
17
18   ... // sentencias sin return
19
20
21
22
23   return(3* g(X));
24
}
```

a) Método Java genérico.

```
ModAcceso tipoRet nbreMet(/*...*/){
1    IC.inspectorDeEntrada("nbreMet");
2    ... // sentencias sin return
3    {
4      IC.inspectorDeSalida("nbreMet");
5      return(z);
6    }
7    ... // sentencias sin return
8    {
9      tipoRetorno retorno;
10     retorno = j/3;
11     return(retorno);
12   }
13   ... // sentencias sin return
14   if (condicion){
15     IC.inspectorDeSalida("nbreMet");
16     return(i);
17   }
18   ... // sentencias sin return
19   {
20     tipoRetorno retorno;
21     retorno = 3* b.g(X);
22     IC.inspectorDeSalida("nbreMet");
23     return(retorno);
24   }
  }
```

b) Método Java genérico instrumentado.

Figura 10. Códigos genéricos de métodos Java con múltiples instrucciones *return*, con y sin instrumentación.

return esta dentro de una sentencia condicional. En ese caso, la instrumentación procede como si fuera cualquier otra sentencia de retorno a transformar, sólo que se deben realizar las acciones correspondientes, dentro del cuerpo del condicional. El resultado del caso expuesto anteriormente se muestra en las líneas 14 a 17 de la parte b) de la misma Figura. Otro caso a considerar, se muestra en la sentencia de retorno ubicada en la línea 23 de la parte a), en donde existe una invocación dentro de una operación. La manera de resolverlo, al igual que para el resto de los casos con expresiones, radica en realizar la transformación de la Figura 7. El resultado de esto se muestra en la parte b) de la Figura 10, en las sentencias 19 a 24, en donde se inserta una sentencia compuesta en la cual: se crea la variable retorno, se asigna el resultado de la expresión a la misma y se la coloca dentro de la sentencia *return*.

Sintetizando todo lo expuesto en esta sección y en las anteriores, se puede establecer un procedimiento general para la instrumentación de métodos, como se muestra a continuación:

- Colocar al inspector de entrada como primera instrucción en el cuerpo del método.
- Identificar todos los puntos de salida del método, para lo cual se requiere analizar todo el código fuente,
- Si la salida es por sentencia *return* se debe:
  - Si la sentencia contiene una expresión, aplicar la siguiente transformación:
    1. Crear una variable del mismo tipo de retorno del método,
    2. Asignar, a la variable creada en el paso previo, el valor devuelto por la invocación que se encuentra en la sentencia *return*,
    3. Colocar la variable dentro de la sentencia de retorno.
    4. Crear una sentencia compuesta formada por las sentencias definidas en los items anteriores.
  - Si la sentencia no contiene una expresión, colocar el inspector de salida justo antes de la misma.
- Si la salida es por fin de método, insertar el inspector de salida antes de la llave de cierre ('}') del método.

Este esquema de IC provee información de los métodos, indicando cuáles y en qué orden se ejecutan. A su vez, garantiza que la información extraída es correcta y consistente respecto comportamiento real del programa. Esto se debe a que durante la ejecución del mismo, identifican aquellos métodos que son ejecutados y su orden, lo cual es fundamental para comprender el funcionamiento del programa.

El esquema de instrumentación definido provee mucha información. Si no se realiza un control adecuado, puede que esta cantidad de información sea demasiada para poder realizar un buen manejo de la misma y sacar provecho. Este problema se explica en la siguiente sección, junto con la posible solución al mismo.

2.2. Instrumentación de Iteraciones

En las secciones anteriores se realizó el análisis respecto a la extracción de información de los métodos, pero ahora surge el problema de cómo tratar esa información y cómo controlar su cantidad.
Dentro de un programa existen diferentes sentencias de control del flujo de ejecución, entre las cuales se encuentran las iteraciones. Estas sentencias permiten repetir la ejecución de un bloque de instrucciones un determinado número de veces. Estos bloques de código pueden contener pasaje de mensajes a métodos instrumentados, los cuales por cada iteración devuelven información. El problema surge cuando las iteraciones se repiten un gran número de veces, provocando que la cantidad de información que provee la IC del o de los métodos invocados dentro del bloque sea demasiado grande (Incluso puede provocar que el programa no funcione correctamente, lo cual es inaceptable). La IC no debe afectar el funcionamiento del sistema analizado, sino que sólo se debe ejecutar en conjunto para obtener datos de la misma.
Considerando el esquema de instrumentación de métodos definido previamente en este capítulo, se puede observar de manera intuitiva en la Figura 11, el problema al cual se hace referencia.

```
ModAcceso tipoRet nbreMet(/*parámetros*/){
    //sentencias del programa original
    for (int i = 100000;  i>0 ;  i--){
        b.F();
        b.H();
    }
    //sentencias del programa original
}
```

Figura 11. Múltiple envío de mensajes a métodos dentro de un sentencia for.

Al observarse el código de la Figura 11 , se puede ver que dentro de la iteración se realiza el envío de mensajes a los métodos F() y H() del objeto b. Si los métodos F y H han sido instrumentados en la clase b, en cada iteración se extrae información respecto al comportamiento de los mismos. El cuerpo de la sentencia *for* se ejecuta 10.000 veces, por lo tanto se extrae, esa misma cantidad de veces, información sobre el comportamiento de los métodos mencionados. Si se considera un programa de miles de líneas de código, con cientos de iteraciones con envío de mensajes dentro, la cantidad de información extraída de la IC será demasiada. Esto provoca que los datos sean dificiles de manejar y, aún más importante, de analizar.
Se debe buscar una manera de resolver el problema planteado en los párrafos anteriores, para poder hacer de la IC una técnica útil para el análisis de programas. Una posible solución consiste en fijar la cantidad de veces que se desea extraer información de las iteraciones. De esta manera, se restringe la extrac-

ción de datos a un número fijo y reducido de repeticiones en las iteraciones del programa. Esto permite mantener la posibilidad de extraer información desde el cuerpo de las iteraciones, pero llevando un control sobre la cantidad extraída.

Una posible implementación de la solución planteada previamente, consiste en almacenar el valor límite fijo establecido por el programador en una variable global. Cada vez que se ejecuta el cuerpo de la iteración se decrementa el valor almacenado en dicha variable. Cuando dicho valor llegue a 0, no se extrae más información de los pasajes de mensajes que se encuentran dentro del cuerpo de la iteración. Sin embargo, esta solución falla para el caso en que se tienen iteraciones anidadas, porque se deben llevar varias cantidades de repeticiones a la vez.

```
ModAcceso tipoRet nbreMet(/*parámetros*/){
    //sentencias del programa original
    for (int i = 100000;  i>0 ;  i--){
        b.F();
        int j = 0;
        while(j<2000){
             b.H(z);
             c.G();
             j++;
         }
    }
    //sentencias del programa original
}
```

Figura 12. Múltiple envío de mensajes a métodos dentro iteraciones anidadas.

En la Figura 12 se tiene una sentencia *while* dentro de un *for*. La cantidad de ejecuciones para los mensajes *b.H(z)* y *c.G()*, no es la misma que la de *b.F()*; por lo tanto se deben almacenar de manera separada. Podrían usarse múltiples variables globales para solucionar este inconveniente pero no es una buena práctica de programación.
Una mejor implementación a la solución del valor límite fijo consiste en el uso de un stack global, en el cual se mantienen los valores de cada iteración. A mayor profundidad de anidamiento de la iteración, más cerca del tope de la estructura se encuentra su valor. Aquella de mayor profundidad en el anidamiento mantiene su cantidad en el tope de la estructura. Una vez que se termina la ejecución de la iteración más profunda se elimina su valor del tope, dejando en el mismo el de la iteración que le sigue en orden de profundidad. Al aplicar la instrumentación de iteraciones sobre el código genérico de la Figura 12, se obtiene:

Sintetizando, el procedimiento de instrumentación realiza básicamente 3 pasos:

```
ModAcceso tipoRet nbreMet(/*parámetros*/){
    //sentencias del programa original
    IC.insertStack(n);// n = valor límite fijo
    for (int i = 100000;  i>0 ;  i--){
        b.F();
        int j = 0;
        IC.insertStack(n);
        while(j<2000){
             b.H(z);
             c.G();
             j++;
             IC.decrementar();
         }
         IC.delete();      //Elimina el tope del Stack
         IC.decrementar();
    }
    IC.delete();  //Elimina el tope del Stack
    //sentencias del programa original
}
```

Figura 13. Instrumentación de iteraciones con múltiple envío de mensajes a métodos dentro.

1. Antes de cada iteración, se inserta el valor fijo establecido en una nueva posición del stack de control.
2. Dentro del cuerpo de la iteración, se decrementa el valor de control en el tope de la pila.
3. Luego de cada iteración, se elimina el tope del stack, para descartar el valor de control no necesario.

Es importante notar que los *inspectores* realizan el control de la cantidad de información extraída, en base al valor que se encuentra en el tope del stack. El resultado de esta comparación, determina si los mismos extraen o no información del programa.

En esta sección, junto con las anteriores, se ha definido un esquema de instrumentación que permite la extracción dinámica de información de sistemas. Esta información es de suma utilidad para la re-construcción de la relación real existente entre el Dominio del Programa y el Dominio del Programa. En la siguiente sección se presenta una estrategia para facilitar la relación entre los dominios antes mencionados.

3. Aplicación del Esquema de Instrumentación para la Construcción de Estrategias de Interconexión de Dominios

En esta sección se describe la estrategia SVSi (*Simultaneous Visualization Strategy Improved*), que permite vincular el Dominio del Problema con el Do-

minio del Programa[7,13]. Esta técnica surge como combinación de otras dos estrategias conocidas con el nombre de BORS y SVS (Ambas descriptas en el apéndice de este artículo). SVSi utiliza el esquema de IC definido en las secciones previas para: i) Obtener información dinámica del programa y ii) Lograr ua relación más fuerte, que la alcanzada por BORS y SVS, entre el Dominio del Problema y el Dominio del Programa.

Para alcanzar su objetivo, SVSi realiza las siguientes tareas: i) Instrumenta el sistema; ii) Inserta primitivas que permitan la ejecución paralela del sistema bajo estudio y el módulo que implementa la funcionalidad de los inspectores (este módulo se conoce con el nombre de Monitor) y iii) Compila el sistema y el monitor formando un único sistema a partir de la unión de ambos.

El paralelismo aplicado permite que el sistema instrumentado se ejecute al mismo tiempo que lo hace el monitor. De esta manera, el monitor muestra la información que extrae, mientras el programa instrumentado es ejecutado. Esto permite al usuario establecer la relación entre los dominios de manera directa. Por otro lado, la información extraída por los inspectores es registrada en un archivo, que luego sirve para la construcción de árboles de ejecución de métodos (me-tree). Un me-tree está formado por nodos, etiquetados con los nombres de los métodos, y arcos que unen dichos nodos. Dados dos nodos A y B, existirá un arco j que los una si y sólo si, durante la ejecución del sistema, el método A realizó un pasaje de mensajes al método B. La raíz del árbol es siempre el primer método ejecutado (generalmente llamado main).

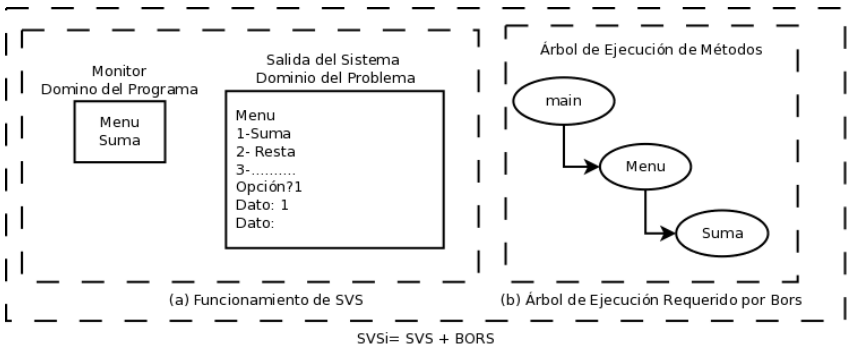

Figura 14. Representación de la información mediante IC y me-tree.

En la Figura 14 se observa el comportamiento hipotético de SVSi. En la parte (a) se muestra un sistema instrumentado en ejecución y la salida del monitor. Observe que para una salida específica del Dominio del Problema se muestran los métodos utilizados para producir dicha salida. En la Figura 14.b se observa el árbol de métodos del programa que deberá generar SVSi, para poder explicar el funcionamiento del sistema por medio de recorridos sobre esta estructura. SVSi aplica la misma técnica que BORS para explicar las funciones del sistema. El

lector interesado en conocer dicho proceso debe ver el apéndice adjunto a este artículo.

En base a lo expuesto en esta sección, se determina que a partir del esquema de IC definido, se pueden crear estrategias de interconexión de los dominios del problema y del programa. Un ejemplo de esto lo constituye SVSi, la cual no solo muestra la información durante la ejecución del sistema, sino que también crea me-tree's que permiten realizar explicaciones respecto al comportamiento del programa.

4. Conclusión y Trabajo Futuro

La Comprensión de Programas es un área fundamental dentro de la Ingeniería de Software debido a que agiliza el proceso de entendimiento de los sistemas. Para lograr esto se vale del desarrollo de métodos, técnicas, estrategias y herramientas que ayudan al proceso cognitivo humano. En este contexto, dicho proceso implica comprender, memorizar, razonar y conceptualizar aspectos que lleven a un entendimiento global de programas. El éxito de esto radica en lograr re-construir la relación existentes entre el Dominio del Problema y el Dominio del Programa, para lo cual es necesario obtener información de ambos dominios.

En este artículo se presentó una estrategia de extracción de la información dinámica de sistemas escritos en lenguaje Java. Dicho esquema se puede utilizar para implementar SVSi, una estrategia que vincula los dominios antes mencionados.

El esquema de instrumentación descripto básicamente consiste en la inserción de métodos de inspección conocidos con el nombre de inspectores. Los inspectores extraen información del comportamiento del programa a nivel métodos (es importante destacar que los mismos pueden ser especializados para obtener otro tipo de información más precisa). Cuando el esquema de instrumentación antes mencionado se utiliza con estrategias del tipo de SVSi se alcanza una relación directa entre el Dominio del Problema y el Dominio del Programa. Esta relación puede ser explicada de forma automática utilizando mecanismos similares a la estrategia BORS descripta en el apéndice adjunto a este artículo. Esta relación es relevante porque provee una solución a uno de los principales desafíos en Comprensión de Programas, el cual es: *Relacionar los Dominios del Problema y Programa.*

Como trabajo futuro a corto plazo, se espera: i) Definir una herramienta que implemente el esquema de instrumentación descripto en este artículo y ii) Implementar SVSi.

APÉNDICE

En este apéndice se describen: i) En la sección A, dos estrategias de interconexión de los dominios del problema y del programa, sobre las cuales se basa SVSi; ii) En la sección B, se muestra un código Java de ejemplo instrumentado. El objetivo de las tareas antes mencionadas, es proveer las bases necesarias para una mejor comprensión de la temática abordada en el artículo.

A. Estrategias para Relacionar los Dominios del Problema y Programa

El principal objetivo de este apéndice es describir dos estrategias que permiten relacionar el Dominio del Problema con el Dominio del Programa, ellas son: SVS (*Simultaneous Visualization Strategy*) y BORS (*Behavioral-Operational Relation Strategy*). Para alcanzar este objetivo, SVS y BORS *utilizan información estática y dinámica*, *organizan esa información* y *aplican un proceso de vinculación que visualiza la relación entre el Dominio del Problema y el Dominio del Programa* [16]. Esas estrategias se explican en las siguientes subsecciones.

Es importante notar que las estrategias antes mencionadas han sido implementadas para sistemas escritos en lenguaje C, por consiguiente se explican utilizando dicho lenguaje. Ambas estrategias forman la base de SVSi (*Simultaneous Visualization Strategy Improved*) descripta en la sección 3 para el lenguaje Java, la cual representa una contribución importante de este artículo.

A.1. Estrategia de Visualización Simultánea - SVS

Una propuesta interesante para recuperar y, luego si es necesario, visualizar las componentes usadas en tiempo de ejecución consiste en instrumentar el sistema. Para llevar a cabo esta tarea, es importante decidir:

- ¿Qué componentes del programa y qué atributos de ellas se desean visualizar ? y
- ¿Cómo se debe llevar a cabo la instrumentación para producir el comportamiento deseado?

Teniendo presente el primer ítem, el sistema se puede conceptualizar como una gran 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 ítem 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 (También conocidas como *inspectores*) que procesan y muestran esta información. Los inspectores se implementan en otro sistema independiente, denominado *Monitor*, para simplificar su reuso.

Ahora, suponga que un programa y un monitor, como los que se muestran en la Figura 15, se compilan formando un único sistema cuya característica es permitir la ejecución paralela de las funciones del sistema de estudio y de los inspectores. Si las primitivas de comunicación están correctamente implementadas, la salida de este nuevo sistema será como se ilustra en la Figura 16. El lector puede observar la relación directa entre el Dominio del Problema (Calculadora básica) y el Dominio del Programa (Funciones usadas para computar las operaciones básicas).

Requerimientos de Implementación. La implementación de este esquema requiere:

- La definición de un esquema de Instrumentación de Código.
- La elaboración de mecanismos que combinen información estática y dinámica, con el propósito de recuperar información más precisa acerca de los objetos del programa usados en tiempo de ejecución. Por ejemplo: *Para una función sus parámetros y tipo de retorno*, *Para una variable su tipo*; etc.
- Uso de un API estándar para acceder a las componentes del programa.

La implementación de esta técnica requiere la definición e implementación de un esquema de instrumentación de código. Además, es importante la combinación de este esquema con métodos de extracción de información estática, para la recuperación de información más precisa acerca de los objetos del programa usados en tiempo de ejecución. Por ejemplo: *para una función sus parámetros y tipo de retorno*, *para una variable su tipo*, etc. En este caso, es aconsejable el uso de un API estándar para acceder a las componentes del programa.

A.2. BORS: una Estrategia de Relación Comportamental-Operacional

Para relacionar las vistas comportamental y operacional se debe usar la información de los objetos del Dominio del Problema y el flujo de ejecución del programa (Dominio del Programa). El primer tipo de información se obtiene a través de la observación de la salida del sistema y la aplicación de algunas estrategias para recuperar los atributos de los objetos. El segundo se extrae usando una aproximación similar a la descripta en la sección A.1. Mezclando el conocimiento acerca de los Dominios del Problema y Programa, se desarrolló una estrategia que alcanza una estrecha relación entre el comportamiento del sistema y su operación interna.

Antes de describir la nueva estrategia, es importante reflexionar sobre algunos aspectos relacionados con la representación de la información dinámica, dado que ésta característica es muy importante para la reconstrucción de la relación entre los Dominios del Problema y Programa. La estrategia SVS (Definida en la sección A.1), registra las funciones de tiempo de ejecución usando una lista. Si bien esta estructura de datos es adecuada para el propósito de SVS, no sucede lo mismo cuando el objetivo es la elaboración de otras estrategias. Esto se debe a que,

SYSTEM

```
int add (int x, int y)
 {
   int r;
     INPUT_INSPECTOR("add");
     r=x+y;
     OUTPUT_INSPECTOR("add");

     return r;
 }

int subtract(int x, int y)
 {
   int r;
    INPUT_INSPECTOR("substract");
    r=x-y;
    OUTPUT_INSPECTOR("substract");
    return x-y;
 }

....
int main()
 {
  int op;
   INPUT_INSPECTOR("main");
   do {
    printf("1. Add\n");
    printf("2. Substract\n");
    printf("Option:\");
    scanf("%d",&op);
    printf("X:");
    scanf("%d",&x);
    printf("Y:");
    scanf("%d",&y);
    switch (op) {
     case 1: add(x,y);
     case 2: substract(x,y);
     ......
    }
   } while (op!=0)
   OUTPUT_INSPECTOR("main");
   return;
 }
```

MONITOR

```
void INPUT_INSPECTOR(char *s)
{
     printf("Begin: %s",s);
}

void OUTPUT_INSPECTOR(char *s)

{
     printf("End: %s",s);
}
```

Figura 15. Sistema Instrumentado y el Monitor

```
SYSTEM OUTPUT                          MONITOR OUTPUT

1)                                       Begin main
  1. Add
  2. Substract
  Option:

2) /* op=1 */
  X: 1
  Y: 2
3)                                        Begin: add
  3
                                          End add
4)
  1. Add
  2. Substract
  Option:
5) /* op=2 */
  X: 1
  Y: 2
6)                                        Begin: substract
  -1
                                          End: substract
7)
  1. Add
  2. Substraction
  Option:
8) /* op=0 */                            End: main
```

Figura 16. Salida de la Ejecución Paralela

desde una lista de funciones, no se puede extraer más información que posibilite la obtención de nuevos conocimientos vinculados con las funciones usadas por el sistema. Por esta razón, surge la necesidad de cambiar la forma de representar la información dinámica con la finalidad de proporcionar datos con más semántica. En los próximos párrafos se justifica este requisito.

Las funciones de tiempo de ejecución se pueden representar como un árbol que refleja la relación llamador-llamado. Esta forma de ver a las funciones es mejor que una simple lista, porque permite identificar la funcionalidad de cada una de las funciones. Por ejemplo, si F es una función del sistema, entonces su funcionalidad se puede inferir a través del análisis de las funciones llamadas directa o indirectamente por ella. Es importante notar que esta información se almacena en un subárbol cuya raíz es la función F.

Asuma que A,B,C,D,E,F,G son funciones usadas por un sistema hipotético para producir su salida; y observando la Figura 17 de la izquierda, no es posible identificar la descripción de las funciones.

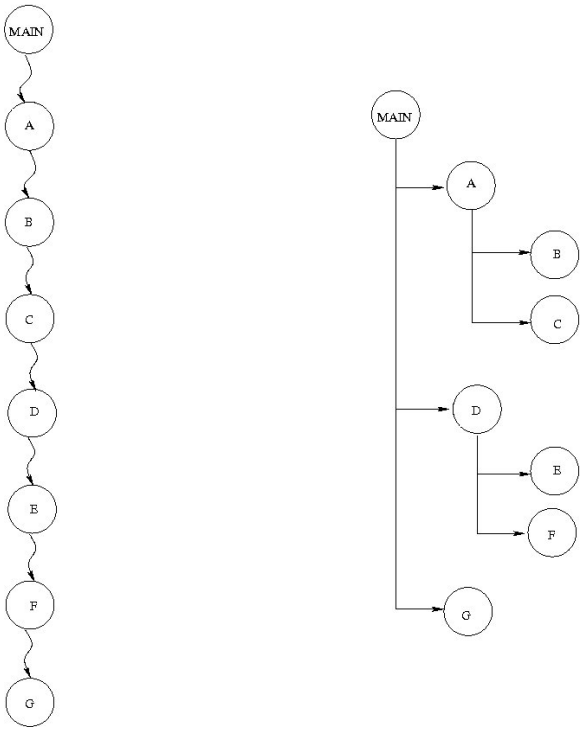

Figura 17. Lista de Funciones vs. fe-Tree

En cambio, si se usa un árbol, como el que se muestra sobre la derecha en la Figura 17, para representar la misma información, el problema se resuelve exi-

tosamente. Observe la Figura 17 de la derecha donde se puede entender claramente como se obtienen las explicaciones para cada función. Por ejemplo, la descripción de la función A consiste de las funciones B y C, y la explicación de la función D está compuesta por las funciones E y F. Finalmente, la función G no se puede expresar en términos de otra función.

En este ejemplo los nombres de las funciones no tienen significado. Por esta razón, se pierde mucha semántica. No obstante, las reglas de programación mencionan la necesidad de colocar nombres con significado a las componentes del programa, y generalmente esos nombres están directamente relacionados con los objetos del Dominio del Problema. De este modo, la información provista contiene más semántica porque:

- La estructura de datos árbol proporciona información de la relación llamador-llamado.
- Los nombres semánticos proporcionan pistas de la tareas realizadas por las funciones.
- La combinación de la información mencionada en los ítems anteriores, proveen una explicación parcial de la funcionalidad del sistema de estudio.

Esta característica justifica la preferencia de representar a las funciones de tiempo de ejecución como un árbol en lugar de una lista. Este árbol se denomina *fe-Tree* (Function Execution Tree) y es similar al me-tree, definido en la sección 3. El lector interesado en estudiar con más detalles las características y operaciones de fe-tree puede leer [3].

En los siguientes párrafos se describe, BORS (*Behavioral-Operational Relation Strategy*), una estrategia que relaciona las vistas comportamental (Lo que resulta de la ejecución del programa) y operacional (Cómo funciona el programa). Esta estrategia se basa en la observación de los objetos del Dominio del Problema y 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 interfaces. 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 de detección de tipos de datos abstractos. Todas las funciones seleccionadas en este paso se almacenan en una lista.

Construir un fe-Tree con las funciones usadas en tiempo de ejecución: Usa la información de tiempo de ejecución para construir el fe-Tree. Esta información se recupera empleando una aproximación similar a la explicada en la sección A.1.

Explicar las funciones encontradas en el paso 1 usando el árbol construído en el paso 2: Este paso se implementa aplicando un recorrido por niveles sobre el *fe-Tree*. Cuando el nombre del nodo visitado concuerda con algún nombre en la lista de funciones construída en el paso 1, BORS reporta el camino desde la raíz hasta el nodo corriente y el subárbol correspondiente.

La Figura 18 ilustra este procedimiento. Sobre la izquierda está el *fe-Tree* correspondiente a la ejecución de un sistema hipotético. Sobre la derecha, se encuentra la lista que contiene las funciones seleccionadas por el programador. En dicha figura, el lector puede observar la explicación y el contexto para cada función.

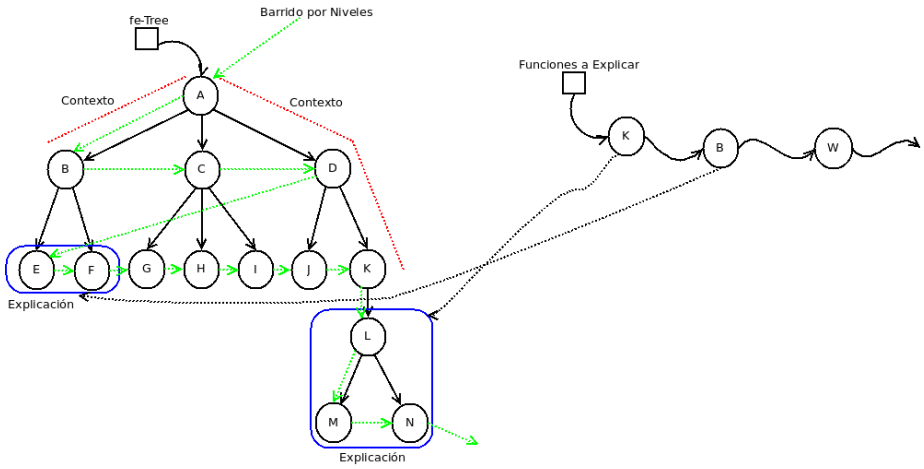

Figura 18. Estrategias para Explicar los Aspectos del Sistema

Requerimientos de Implementación. Para implementar BORS se necesita:

Definir estrategias que asocien los objetos del Dominio del Problema con funciones: Esta tarea tiene dos pasos importantes:

- Seleccionar los objetos del Dominio del Problema.
- Detectar sus interfaces.

En el primer ítem, el programador decide cuáles objetos del Dominio del Problema quiere estudiar. Este paso es obviamente observacional. En el segundo, se asocian las funciones del sistema con cada objeto seleccionado en el primer paso. En este caso, se pueden utilizar estrategias tales como: *Técnicas de búsqueda*, *Detección de tipos de datos abstractos*, *Agrupación de funciones*, etc.

Crear un Esquema de Instrumentación de Código: Este ítem es importante porque permite recuperar información dinámica del sistema.

Proponer técnicas para resumir información: La información recuperada por las técnicas de análisis dinámico suele ser enorme. Para facilitar el procesamiento de información, se necesita la aplicación de estrategias que posibiliten la extracción de la información más relevante sin perder demasiada semántica.

Implementar un procedimiento para construir un fe-Tree: El *fe-Tree* es una estructura de datos fundamental para el funcionamiento de BORS. El esquema usado para extraer la información dinámica debe proveer todos los datos necesarios para construir esta estructura.

B. Ejemplos de Instrumentación de Programas Java

En esta sección, se muestra el código de un programa ejemplo en lenguaje Java sin instrumentar (sección B.1) e instrumentado (sección B.2). Para realizar dicha Instrumentación de Código se utiliza el esquema definido en la sección 2.

B.1. Código Ejemplo de un Programa en Java

```
public void F(char caracter){
   int suma;
   suma = caracter + 10;
   System.out.println("El resultado es: " + suma);
}

public int G(int X){
   return(2*X);
}

public int H(int X){
    return(3* G(X));
}

public static void main(String[] args) {
     Main ejemplo = new Main();
     for(int i=1000000; i>0 ; i--){
         ejemplo.F('A');
         for(int i=1000000; i>0 ; i--){
             ejemplo.H(2);
         }
     }
}
```

B.2. Programa Java de la Sección B.1 Instrumentado

```
public void F(char caracter){
    IC.inspectorDeEntrada("F");
    int suma;
    suma = caracter + 10;
    System.out.println("El resultado es: " + suma);
    IC.inspectorDeSalida("F");
}

public int G(int X){
    IC.inspectorDeEntrada("G");
    {
       int retorno = 2*X;
       IC.inspectorDeSalida("G");
       return(retorno);
    }
}

public int H(int X){
    IC.inspectorDeEntrada("H");
    {
       int retorno;
       retorno = 3* G(X);
       IC.inspectorDeSalida("H");
       return(retorno);
    }
}

public static void main(String[] args){
    IC.inspectorDeEntrada("Main");
    Main ejemplo = new Main();
    IC.insertStack(n);
    for(int i=1000000; i>0 ; i--){
         ejemplo.F('A');
         IC.insertStack(n);
         for(int i=1000000; i>0 ; i--){
             ejemplo.H(2);
             IC.decrementar();
         }
         IC.delete();
         IC.decrementar();
    }
    IC.delete();
    IC.inspectorDeSalida("Main");
}
```

Referencias

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. B. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. PhD thesis, Delft University of Technology, 2009.
3. M. Beron, P. Henriques, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Ph.D Thesis Dissertation at University of Minho. Braga. Portugal*, 2010.
4. K. Bertels, P. Vanneste, and C. De Backer. A cognitive approach to program understanding. *Reverse Engineering, 1993., Proceedings of Working Conference on Program Comprehension*, pages 1–7, 1993.
5. B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. Deurse, and J. Wijk. Execution Trace Analysis through Massive Sequence and Circular Bundle Views. *Report TUD-SERG-2008-008*, 2008.
6. T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM'01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, Washington, DC, USA, 2001.
7. C. Exton. Constructivism and program comprehension strategies. *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, 2002.
8. A. Hamou-Lhadj D. Rothlisberger A. Zaidman O. Greevy. 5rd International Workshop on Program Comprehension through Dynamic Analysis. *Report TUD-SERG-2010-039*, 2010.
9. S. Haiduc and A. Marcus. On the Use of Domain Terms in Source Code. *International Conference on Program Comprehension (ICPC08)*, 2008.
10. Thomas D. Latoza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *In Proceedings of Foundations of Software Engineering*, 2007.
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. M. P. O'Brien. Sofware Comprehension - A Review and Research Direction. *Technical Report*, 2003.
13. V. Rajlich. Program Comprehension as Learning Process. *International Conference on Cognitive Informatics.*, 2002.
14. A. Rohatgi, Hamou.Lhadjm A., and J. Rilling. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. *International Conference on Program Comprehension (ICPC08)*.
15. M. A. Storey. Theories, methods and tools in program comprehension: past, present and future. *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.
16. M. A. Storey, F. D. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mentalmodel during software visualization. *IWPC'97. Fifth International Workshop on Program Comprehension*, 1997.
17. A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
18. A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, Washington, DC, USA, 2002. IEEE Computer Society.
19. A. Hamou-Lhadj O. Greevy A. Zaidman. 3rd International Workshop on Program Comprehension through Dynamic Analysis. *Report TUD-SERG-2007-022*, 2007.