Diagnosis basada en modelos para la depuración de software mediante técnicas simbólicas.

R. Ceballos, R. M. Gasca, Carmelo Del Valle y Miguel Toro
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla
Escuela Superior de Ingeniería Informática,
Avenida Reina Mercedes s/n 41012 Sevilla(Spain)

Resumen

En la programación es esencial tener herramientas para la diagnosis del software que ayuden al programador y al ingeniero de desarrollo a localizar los errores. En este trabajo, proponemos una nueva aproximación que permite identificar los posibles errores en programas software usando técnicas simbólicas(Bases de Gröbner). Esta técnica permite generar, a partir la estructura y semántica del programa original, un modelo más simple del programa para llevar a cabo la diagnosis.
Se tienen en cuenta las especificaciones formales y el código fuente del programa, y es a partir de ellos como se detecta la sentencia o el conjunto de sentencias que contienen el error al emplear un caso de test determinado. La metodología comienza eliminando las variables no observables y obtiene nuevos asertos referidos a grupos de sentencias del programa. A partir de ahí construimos una red de contextos formada por estos grupos de sentencias y sus correspondientes asertos como nodos, y por último obtenemos la diagnosis mínima utilizando un algoritmo estándar.

1. Introducción

El desarrollo de programas introduce errores que son la causa de paradas y pérdidas de tiempo en la producción de software. La diagnosis permite identificar las partes del programa que no se comportan de la forma esperada. La mayoría de las aproximaciones aparecidas en la última década para realizar diagnosis de software se han basado en el uso de modelos (DBM).

La formalización de la diagnosis basada en modelos se presentó en [DeKleer&Williams87] y [Reiter87], donde se propuso una teoría general para el problema de explicar las discrepancias entre los comportamientos observados y correctos de los mecanismos. Apoyándose en ella, la mayoría de las aproximaciones de DBM para componentes caracterizan la diagnosis de un sistema como una colección de conjuntos mínimos de componentes fallando, que explican los comportamientos observados.

El Proyecto JADE lleva varios años investigando sobre la diagnosis basada en modelos aplicada al software, o más concretamente en la depuración basada en modelos (MB Debugging). Usa un modelo de dependencias basado en el código fuente del programa. Se trata de representar las sentencias y expresiones como si fuesen componentes y las variables como si fuesen las conexiones, realizando un mapeo de constructos Java^{TM} a componentes. Las asignaciones, condiciones, bucles, etc; tienen su correspondiente método de transformación. Por ejemplo los bucles son transformados en sentencias selectivas anidadas. Para mayor concreción puede consultarse [Mateis99] y [Mateis00].

Previamente a estos trabajos, se ha propuesto la técnica de *Slicing* en la diagnosis de software, que identifica los constructos del código fuente que puede influir en el valor de una variable en un punto dado del programa[Weiser82],[Weiser84]. Una extensión de esta técnica denominada *dicing* [Lyle&Weiser87] también se ha aplicado en la diagnosis de software. La diferencia entre dos *slices* es un *dice*, por ello se puede definir como el conjunto diferencia entre *slices* estáticos para un valor procesado incorrectamente y un valor procesado correctamente. La tareas previas son realizadas muchas veces de acuerdo con la experiencia de los expertos y por ello se han propuesto, en los últimos años, nuevos métodos para automatizar el proceso de diagnosis de software [Khalil98][Khalil99].

En este trabajo presentamos una aproximación diferente a las anteriores, con el objetivo de alcanzar la diagnosis del software. Para aplicar esta metodología se debe disponer de los recursos siguientes: Código Fuente, precondición y postcondición. La precondición será un predicado cuyas únicas variables libres deben ser los parámetros de entrada. De manera análoga, la postcondición será un predicado cuyas únicas variables libres deben ser parámetros de entrada y variables de salida. Si el código se ejecuta en alguno de los estados definidos por la precondición se garantiza que el código debería terminar en alguno de los estados definidos por la postcondición. No se garantiza nada si el código es ejecutado en un estado inicial que no satisface la precondición.

Las sentencias o bloques de sentencias del código fuente se transforman a restricciones polinómicas, que serán manipuladas simbólicamente, hasta reducir el modelo. Esta reducción se consigue utilizando las bases de Gröbner. A su vez, evitamos la construcción explícita del grafo de dependencias funcionales de las variables del programa.

El uso de las bases de Gröbner para la diagnosis basada en modelos es algo que ya se ha propuesto para dispositivos en un reciente trabajo [Gasca01]. Parte de la metodología presentada en ese artículo, se adapta aquí, para la diagnosis de software.

Para seleccionar que observaciones son las más significativas y por tanto nos van a dar más información debemos usar técnicas de Testing Orientadas a Objetos. En [Binder00] aparecen reflejados los objetivos y complicaciones que un buen Testing conlleva. Debemos ser consciente de los límites del Testing. Las combinaciones de entradas y salidas de los programas (incluso de los más triviales) son demasiado amplias.

Para centrar el objetivo de este trabajo podemos decir que los programas que están en el ámbito de este artículo son:

1. Aquellos que pueden ser compilados para después depurarlos pero incumplen las especificaciones de precondición y postcondición.
2. Aquellos que son una variación pequeña del programa correcto, pero que son erróneos.

El resto del artículo se estructura como sigue. Primero presentamos la gramática del lenguaje tratado. Mas tarde, expondremos las definiciones necesarias para explicar la metodología, y mostraremos la forma de obtener el modelo. Finalizaremos mostrando los resultados obtenidos en tres ejemplos diferentes y las conclusiones y futuros trabajos en esta línea de investigación.

2. Especificación de la gramática y ejemplos

En este artículo se recoge un subconjunto de toda la gramática del lenguaje Java^{TM}.

```
Programa : Sentencias
Sentencias : Sentencia Sentencias | Ø
Sentencia : DecVar | Asignacion | Selectiva | Bucle
DecVar : Tipo Ident
Tipo : int | float
Asignacion : Ident = Exp
Exp : Ident | Constante | (Exp) | Exp Op Exp
Selectiva : if (Exp) Sentencias Alternativa
Alternativa : else Sentencias | Ø
Bucle : while (Exp) Sentencias
```

Ejemplo 1:

```
{Pre: a,b,c,d,e>0}
(--) int x,y,z,f,g;
```



```
(S1) x=a*c;
(S2) y=b*d;
(S3) z=c*e;
(S4) f=x+y;
(S5) g=y+z;
{Post:f=a*c+b*d ∧ g=b*d+c*e}
```

El ejemplo anterior es correcto cuando los parámetros de entrada satisfacen las restricciones impuestas por la precondición. Con este ejemplo cubrimos la parte de la gramática que incluye las declaraciones y asignaciones. Nos permitirá comprobar como la metodología es capaz de detectar las dependencias entre instrucciones.

Para probar la validez de la metodología, realizaremos cambios en el código fuente. La diagnosis debe detectar estos cambios, y deducir el conjunto de sentencias que causan el incumplimiento de la postcondición. Si cambiamos la sentencia S_5 por g=y-z, tendremos un nuevo programa (llamado **Ejemplo 1a**) que no cumplirá la postcondición.

Ejemplo 2: Con este ejemplo se intenta cubrir las sentencias selectivas. Para comprobar la validez de la metodología, cambiaremos la sentencia S_4 por x=2*x+3. Por tanto tendremos un nuevo programa (llamado **ejemplo 2a**) que no cumplirá la satisfacción de la postcondición.

```
{Pre: a>0 ∧ b>0}
(--) int x,y;
(S1) x=a+b;
(S2) y=2*b+3;
(S3) if (x>y)
(S4)     x=2*x;
(S5) else
(S4)     x=3*x;
{Post:(a+b>2*b+3 ∧ x=2a+2b)
∨(a+b<=2*b+3 ∧ x=3a+3*b)}
```

Ejemplo 3: Este último ejemplo nos permitirá validar la metodología en la diagnosis de bucles. Para comprobar la validez de la metodología cambiaremos la sentencia S_7 por s=2*s+p. Entonces tendremos un nuevo programa (llamado **ejemplo 3a**) que no cumplirá la postcondición.

```
{Pre: n>0}
(S1) int i=0;
(S2) int p=1;
(S3) int s=1;
```

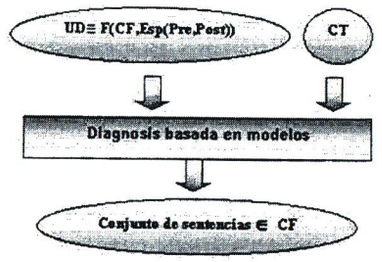

Figura 1: Proceso de diagnosis

```
(S4) while (i<n){
(S5)     i=i+1;
(S6)     p=2*p;
(S7)     s=s+p;}
{Post:s = ∑φ:0≤φ≤n:2^φ ∧ p=2^n}
```

3. Definiciones y notación

Para formalizar el proceso de diagnosis es necesario previamente exponer algunas definiciones.

Definición 1. Especificación de la Unidad de Diagnosis: Se define como la tupla que contempla los siguientes elementos: Código fuente (CF) que cumple la gramática presentada, especificación de la precondición (Pre) y postcondición (Post). A esta unidad de diagnosis se le aplicará la metodología propuesta, que mediante un conjunto de casos de tests, obtendrá como resultado la sentencia o sentencias del programa que son erróneas. Si el conjunto resultado es vacío, concluiremos que el proceso de diagnosis no ha encontrado ninguna anomalía.

Definición 2. Caso de test (CT): Es una tupla que asigna valores a las variables observables. Para el ejemplo 1a un caso de test puede ser: CT≡{a=2,b=2,c=3,d=3,e=2,f=10,g=12} Para encontrar que *CTs* son los que nos pueden reportar una diagnosis más precisa podemos utilizar técnicas de Testing. En nuestro caso usaremos un conjunto de casos de test obtenido mediante técnicas de caja blanca.

Definición 3. Modelo Polinómico del Programa (MPP): Se define como un conjunto finito de restricciones de igualdad polinómicas P que determinan el comportamiento del programa, mediante las relaciones entre las vari-

ables no observables del sistema (*Vnobs*) y las variables observables (*Vobs*). En nuestro modelo las variables observables serán aquellas de las que podemos deducir su valor correcto en la postcondición, además, debemos incluir en este conjunto los parámetros de entrada del código. El resto de las variables serán no observables: **MPP(P,Vobs,Vnobs)**

Definición 4. Contexto: Es una tupla formada por dos conjuntos: Un conjunto de sentencias y su conjunto de restricciones polinómicas correspondientes. El conjunto de contextos posibles será 2^{sent}. Donde *sent* es el número de sentencias que tendremos al aplicar la normalización al programa.

Definición 5. Red de contextos: Es un grafo formado por todos los contextos del modelo, de acuerdo con la forma que fue propuesta por los sistemas de mantenimiento de la razón(ATMS) [DeKleer&Williams87]. La figura 2 representa la red de contextos para el problema presentado en el ejemplo 1a. **Definición 6. Conjunto conflictivo posible(CCP)**: Será aquel conjunto de sentencias de un contexto de la red de contextos en los que se puede determinar a partir de los casos de test un error en el programa.

Definición 7. Conjunto conflictivo posible mínimo (CCPM): Será un conjunto conflictivo posible donde todos sus subcontextos son consistentes. Además el contexto conflictivo mínimo contiene las restricciones mínimas y no repetidas en otro contexto cuando se hace un recorrido en anchura y bottom-up de la red de contextos.

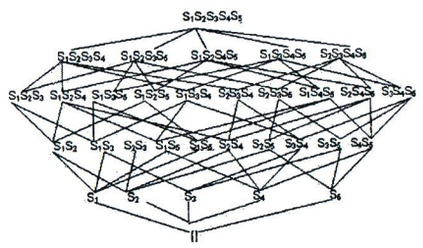

Figura 2: Red de contextos para el ejemplo 1a

4. Metodología de diagnosis

La idea principal es localizar los posibles conflictos a través de una aproximación simbólica. Será un proceso que transformará un programa en un conjunto de restricciones polinómicas. Más tarde simplificamos los contextos obtenidos, para finalmente obtener la diagnosis mínima. Para realizar el proceso se deben seguir los siguientes pasos:

1. Normalización.
2. Obtención de las restricciones del contexto y simplificación de la red de contextos.
3. Determinación de la diagnosis mínima.

5. Normalización

La normalización es el proceso por el cual obtenemos las restricciones del contexto. Se trata de realizar un mapeo desde el programa original a restricciones polinómicas. El proceso de trasformación puede perder cierta información que el programa original incorpore, pero esa información no será relevante para la diagnosis, ya que lo que nos permitirá obtener la diagnosis son las relaciones explícitas e implícitas entre las sentencias, y estas relaciones serán conservadas en el modelo. Proponemos los siguientes pasos a seguir:

Renombramiento de variables: De esta forma pretendemos que nunca existan, entre la precondición y la postcondición, dos instrucciones que asignen un valor a la misma variable.

Por ejemplo:

```
x=a*c;
...
x=x+3;
...
{Post:x=...}
```

Sería normalizado a:

```
x1=a*c;
...
x2=x1+3;
...
{Post:x2=...}
```

Normalización de las declaraciones y asignaciones: Las asignaciones del código fuente serán mapeadas a igualdades polinómicas. En nuestro ejemplo 1a las líneas (2), (3), (4), (5) y (6) serán mapeadas a: S_1={x=a*c}, S_2={y=b*d}, S_3={z=c*e}, S_4={f=x+y} y S_5={g=y-z}. Podemos observar el resultado en la tabla 1.

Normalización de las sentencias selectivas: Dependiendo de cómo se desarrolle la traza del programa, se ejecutará un grupo de sentencias u otro, o incluso ninguna instrucción. El paso de una sentencia selectiva a un polinomio que sea capaz de representar el contenido de carácter lógico que contiene no es sencillo. Si comparásemos la diagnosis de software con la diagnosis de componentes sería como incorporar un componente u otro dependiendo de la evolución del sistema; algo que ha sido poco tratado en la teoría de diagnosis de componentes. La sentencia selectiva nos da la posibilidad de incorporar instrucciones en función de la traza del programa. Parece lógico pensar que la solución que propongamos debe depender de la traza del programa, o más concretamente, del CT. Generalmente nos encontraremos ante un modelo parecido a:

if (cond){code_if} else {code_else}

Dependiendo del caso de test se ejecutará *code_if* o bien *code_else*, es decir escogeremos un grupo u otro de sentencias que se traducirán a polinomios y que formarán parte del *MPP*. Así, para el ejemplo 2a, si las entradas son a=6 y b=2, se deduce que x>y; por tanto se ejecutará S_4. Por tanto la traducción de la sentencia selectiva de nuestro ejemplo sería la correspondiente traducción de la sentencia S_4 que en esta ocasión es una asignación.

Tabla 1: Modelo del Ejemplo 1a

Restricciones Polinómicas	Variables observables	Variables no observables
S_1: x=a*c	a,b,c,d,e	x,y,z
S_2: y=b*d	f,g	
S_3: z=c*e		
S_4: f=x+y		
S_5: g=y-z		

El problema de realizar este tipo de transformación es que perdemos información que intrínsecamente esta contenida en el código. De esta forma no podremos diagnosticar que el error sea debido a la condición de la sentencia, pero si que sea debido a cualquiera de las instrucciones que se deriven en cualquiera de los dos casos posibles.

Si nos fijamos en el ejemplo 2a, el valor final de x depende de si se cumple o no la condición en S_3. El valor final de la variable x dependerá de los valores que tomen las variables x e y que son las implicadas en la condición de la selectiva. Por tanto el valor final de x no solo depende del código incluido en la selectiva y de la asignación en S_1; sino que también depende de S_2 ya que en esta instrucción se asigna el valor a la variable. Esta relación implícita en el código debe ser reflejada en nuestro modelo basado en polinomios. Para ello proponemos los siguientes pasos:

1. Localizar el conjunto de variables que están implicadas en la condición de la sentencia selectiva. Lo llamaremos Var_{Cond}. En el ejemplo 2a dicho conjunto sería:

$$Var_{Cond} = \{x,y\}$$

2. Localizar el conjunto de variables que toman un valor nuevo (debido a una asignación) en el grupo de sentencias code_if o en grupo de sentencias code_else. Lo llamaremos Var_{Code}. En el ejemplo 2a dicho conjunto sería:

$$Var_{Code}=\{x\}$$

3. Crearemos una nueva sentencia de asignación que relacione cada una de las variables que cambien de valor dentro del código comprendido en la sentencia selectiva (Var_{Code}). Esta relación será con cada una de las variables que forman parte de la condición de la selectiva (Var_{Cond}). Para el ejemplo 2a sería de la forma:

$$x_{i+1}= x_i +f(x,y)-f(x_i,y_i);$$

La función f es una función polinómica genérica, en principio no nos interesa como este implementada, lo importante es que estamos creando una relación entre las variables. Cuando x tome el valor x_i e y tome el valor y_i, entonces $f(x,y)-f(x_i,y_i)$ se anulará, por tanto no estamos influenciando en el resultado final del programa; es decir, estaríamos sumando y restando el mismo valor, lo que no alterará el resultado final. Lo importante es que tendremos en el modelo un polinomio que relaciona las variables que forman parte de la sentencia selectiva y las que forman parte de la condición. En el penúltimo apartado de este artículo se muestra varios ejemplos de definición de esta función.

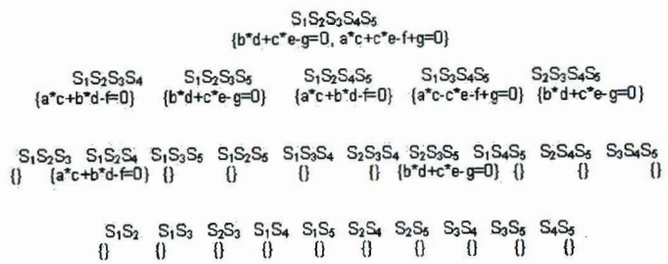

Figura 3: Red de contextos con restricciones simbólicas

4. Añadiremos estas nuevas instrucciones justo delante de la sentencia selectiva, que se deben colocar aún cuando no se ejecute ninguna instrucción de la sentencia selectiva.

Los valores parciales de las variables y de la función f añadida deben ser obtenidos gracias a la monitorización del programa cuando se le aplica el caso de test propuesto. Este proceso es perfectamente automatizable. Esta técnica la llamamos **recuperación de relaciones implícitas**. Ayudándonos del renombramiento de variables obtendríamos las restricciones que aparecen en la tabla 2.

Tabla 2: Modelo del Ejemplo 2a

Restricciones Polinómicas	Variables observables	Variables no observables
S_1: x0=a+b S_2: y0=2*b+3 S_3: x1=x0+ f(x,y)-f(x0,y0) S_4: x2=2*x1+3	a,b,x2 f(x0,y0)	x0,x1 f(x,y)

Si el proceso de diagnosis ofrece como solución que el error puede estar en la sentencia S_3 nos indicará que el error del código podría encontrarse en la condición de la sentencia selectiva.

Normalización de los bucles: Un bucle es un conjunto de instrucciones que se repiten un número determinado de veces. Igual que en el caso de las sentencias selectivas, si comparásemos la diagnosis de software con la diagnosis de componentes, sería como incorporar un componente o varios, dependiendo de la evolución del sistema. Para reducir el modelo de n iteraciones a una sola, perdiendo el mínimo de información, proponemos añadir una instrucción por cada variable que cambie de valor en el bucle, que añada la cantidad necesaria (positiva o negativa) para alcanzar el valor al que llegaríamos en el paso *n-1*.

En el ejemplo 3a las variables i, p y s cambian su valor. Si i_0 es el valor de i antes de entrar en el bucle y i_{n-1} el valor de i en el paso n-1, llamemos β_i a la diferencia entre i_{n-1} y i_0. Entonces la instrucción $i_{n-1}=i_0 + \beta_i$ colocada antes del bucle nos permitiría conservar la dependencia del valor i_n con respecto a sus anteriores valores y nos ahorraría los *n-1* pasos anteriores. Esta técnica la denominaremos **reducción de bucles**. Ayudándonos del renombramiento de variables y de la recuperación de relaciones implícitas, obtendríamos la tabla 3. El proceso de diagnosis nos podrá ofrecer como resultado las sentencias S_7, S_8 o S_9,que no serán tenidas en cuenta en la interpretación de la diagnosis, ya que su utilidad es solamente simplificar el tratamiento de los bucles.

Tabla 3: Modelo del Ejemplo 3a

Restricciones Polinómicas	Variables observables	Variables no observa.
S1:i0=0	n,s3,p3,	s0,s1,s2,p0,
S2:p0=1	β_i,β_p,β_s,	p1,p2,i0,i1,
S3:s0=1	f(i0,n)	i2,i3,f(i,n)
S4:i1=i0+f(i,n)-f(i0,n)		
S5:p1=p0+f(i,n)-f(i0,n)		
S6:s1=s0+f(i,n)-f(i0,n)		
S7:i2=i1+ β_i		
S8:p2=p1+ β_p		
S9:s2=s1+ β_s		
S10:i3=i2+1		
S11:p3=2*p2		
S12:s3=2*s2+p3		

6. Obtención de las restricciones del contexto

6.1. Las bases de Gröbner

La teoría de las Bases de Gröbner es el origen de muchos algoritmos simbólicos que se usan para manipular polinomios de variables múltiples [Helzer95]. Para una introducción a las bases de Gröbner se pueden consultar [Buchberger65], [Buchberger85], [Hoffman89] y [Kapur&Laksman92].

Dado el conjunto de restricciones polinómicas de igualdad de la forma *P=0*, las bases de Gröbner nos proporcionan un sistema equivalente *G=0*, el cual tiene la misma solución que el original, pero generalmente es más sencillo de resolver. Respecto a las ventajas que supone el uso de las bases de Gröbner para los modelos de sistemas sujetos a diagnosis, tenemos que:

- Si el modelo es sobrerrestringido y contiene ecuaciones redundantes, al calcular una base de Gröbner reducida, estas redundancias se eliminan.
- Si el modelo es sobrerrestringido e inconsistente, una de las restricciones que proporciona el algoritmo es 1=0, que es inconsistente obviamente.
- Si el modelo es infrarrestringido, el nuevo modelo también aporta información útil para su posterior tratamiento.

En este trabajo se ha implementado una función denominada *BaseDeGrobner*, que calcula las bases de Gröbner de un determinado contexto a partir de las restricciones de las sentencias que lo constituyen, las variables observables en dichas restricciones y las variables no observables. Sea por ejemplo el contexto representado por las sentencias $S_1S_2S_3S_4S_5$, entonces la función *BaseDeGrobner* recibe los tres conjuntos anteriores. Así tenemos que para el primer ejemplo la función BaseDeGrobner({x=a*c,y=b*d, z= c*e, f=x+y, g=y-z},{a,b,c,d,e,f,g},{x,y,z}) daría como resultado el siguiente sistema reducido de restricciones polinómicas:

{b*d+c*e-g=0, a*c+c*e-f+g=0}

La aplicación de esta función a los diferentes contextos de un modelo particular, permitirá la construcción de la red de contextos.

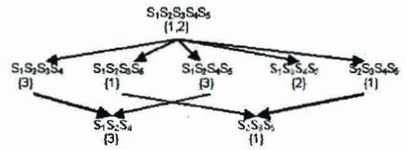

Figura 4: Red de contextos para la búsqueda de CCPM

6.2. Reducción de la red de contextos

Para el ejemplo 1a y mediante la aplicación de la correspondiente función *BaseDeGrobner* a cada contexto de la red de contextos presentada en la figura 2; se obtiene la red de contextos completada con las restricciones y que se ha representado en la figura 3. En esta figura los números representan las restricciones que están ligadas al contexto y que aparecen recogidas en la tabla 4.

Tabla 4: Número identificativo de las restricciones de los contextos

Identificador de restricción	Restricción
1	b*d+c*e-g=0
2	a*c+c*e-f+g=0
3	a*c+b*d-f=0

Este esquema de restricciones del contexto, nos indica en un primer lugar, las restricciones que deben satisfacerse para cada nodo de la red de contextos y en segundo lugar muestra que la mayoría de los nodos de tres componentes y todos los nodos de dos componentes de la red de contextos no podrán catalogarse como conjuntos conflictivos, por tanto el algoritmo para la diagnosis evitará el tratamiento computacional de los mismos. Ello favorecerá la eficiencia del proceso de búsqueda.

7. Diagnosis de software

7.1. Determinación de conjuntos conflictivos posible mínimos

La búsqueda de los conjuntos conflictivos mínimos se realiza partiendo de la estructura de datos que representa la red de contextos que se muestra en la figura 4. Para determinar los posibles conjuntos conflictivos mínimos solamente tenemos que recorrer el grafo partiendo de los nodos inferiores de tal forma que si alguno de los contextos superiores contiene los números de restricciones de los anteriores estos no se consideran como conjuntos conflictivos mínimos.

Así tenemos que se puede observar que al recorrer el grafo a partir del contexto $S_1S_2S_4$ los contextos que le anteceden $S_1S_2S_3S_4$ y $S_1S_2S_4S_5$ no tendrán que considerarse, pues contienen el mismo número de restricción y su evaluación por tanto no tiene sentido realizarla en el proceso de diagnosis. Por tanto el contexto $S_1S_2S_4$ es un conjunto conflictivo posible mínimo.

Igualmente ocurre con el contexto $S_2S_3S_5$. Por último tenemos que $S_1S_3S_4S_5$ contiene la restricción 2 que ya está recogida en el nodo anterior y por tanto este contexto también es un conjunto conflictivo posible mínimo. Por todo ello la diagnosis se basará sólo en la consideración de este conjunto reducido de contextos que se han denominado *CCPM*, de un total de 8 *CCP* se ha pasado a un conjunto de 3 *CCPM*, que son para el ejemplo 1: $\{S_1S_2S_4\}$, $\{S_1S_3S_4S_5\}$, $\{S_2S_3S_5\}$. Hemos conseguido reducir la cardinalidad del espacio de búsqueda a tan sólo tres contextos.

Para estudiar un caso particular de diagnosis se tiene el siguiente caso de test:

$$CT1 \equiv \{a_{obs}=3, b_{obs},=2, c_{obs}=2, d_{obs}=3, e_{obs}=3, f_{post}=12, g_{post}=12\}$$

Con los anteriores *CCPM* y cada *CT* vamos a encontrar la diagnosis mínima. Se comprobará si el programa está respondiendo al comportamiento previsto, mediante la evaluación de las restricciones de los citados posibles conjuntos conflictivos mínimos. Si las restricciones de dicho contexto se evalúan a falso, entonces este conjunto conflictivo mínimo recoge el conjunto de posibles instrucciones que no se comporta correctamente. En el **ejemplo 1a** cuando se aplica el caso de test *CT1* se tiene los siguientes *CCPM* $\{S_1S_2S_4\}$,$\{S_1S_3S_4S_5\}$ y $\{S_2S_3S_5\}$ con restricciones asociadas falsas.

7.2. Determinación de la diagnosis mínima a partir de los conjuntos conflictivos posibles mínimos

A partir de estos *CCPM* se utiliza un método estándar para obtener la diagnosis mínima que consiste en encontrar los *hitting set* para el conjunto de restricciones correspondientes a los *CCPM*. Del caso de test presentado anteriormente se obtiene como diagnosis los siguientes conjuntos de sentencias $\{S_3\}$,$\{S_5\}$,$\{S_2S_4\}$,$\{S_1S_2\}$. Estos conjuntos nos indican que modificando las sentencias S_3 o S_5 podemos conseguir que se cumpla la postcondición para el CT usado.

8. Proceso de diagnosis en los ejemplos presentados

Ejemplo 1a: Para este ejemplo (ver el punto 7.2) se ha obtenido como diagnosis que la sentencia S_5 puede ser la que contenga el error, esta sentencia es precisamente la que hemos cambiado. Otra posible solución podría ser la sentencia S_3. Si se produce un cambio en S_3 este no influirá en S_4 pero si en S_5 que es donde realizamos el cambio y por tanto podremos volver correcto el resultado sin modificar S_5 y cambiando S_3. Hay que resaltar que S_5 también depende de S_2, pero un cambio en S_2 podría implicar un resultado erróneo en S_4.

Ejemplo 2a: Para el código del ejemplo 2a se obtienen los resultados que aparecen en la tabla 5. *Al no usar la recuperación de relaciones implícitas*, la sentencia S_2 no forma parte de la diagnosis mínima ya que el modelo que usamos es más pobre y ofrece menos información.

Si utilizamos la recuperación de relaciones implícitas en el ejemplo 2a obtendremos el resultado recogido en la tabla 6. En este caso la función escogida para aplicar la metodología es *f(x,y)=x+y*, que es la forma mas simple de relacionar las variables x e y en un polinomio

Tabla 5: Diagnosis mínima para el ejemplo 2a

Restricciones polinómicas	S1= x0-a-b S2= y0-2*b-3 S3= x2-2*x1-3
CT	{a=7,b=2,x2=18}
CCPM	$\{S_1S_3\}$
Diagnosis mínima	$\{S_1\},\{S_3\}$

y que implicará una menor complejidad de los cálculos. Por tanto, usaremos el sumatorio de las variables implicadas para calcular la función f en el resto de los ejemplos.

La instrucción S_4 es la instrucción que hemos modificado. La sentencia S_3 nos indica que el error puede estar en la condición de la sentencia selectiva. La ejecución de S_4 depende de la satisfacción de la condición en la sentencia selectiva. La sentencia S_1 que asigna el valor inicial a la variable x puede ser incorrecta. La sentencia S_2 es parte de la diagnosis mínima, por que en dicha sentencia asignamos el valor inicial a una de las variables que influyen en condición de la sentencia selectiva (esta sentencia selectiva permite modificar el valor de la variable x).

Tabla 6: Diagnosis mínima para el ejemplo 2a

Restricciones polinómicas	S1= x0-a-b S2= y0-2*b-3 S3= x1-x0-f(x,y)+f(x0,y0) S4= x2-2*x1-3
f(x,y)	x+y
CT	{a=7,b=2,x2=18,f(x0,y0)=16}
CCPM	$\{S_1S_2S_3S_4\}$
Diagnosis mínima	$\{S_1\},\{S_2\},\{S_3\},\{S_4\}$

Otra modificación interesante es añadir en la postcondición cual debería ser el valor correcto de la variable y. Cambiando la postcondición a:$\{Post: y0=2b+3 \wedge ...\}$. El resultado de la diagnosis aparece en la tabla 7. La sentencia S_2 ya no forma parte de la diagnosis mínima, por que al tener el valor final de la variable la metodología es capaz de detectar que dicha instrucción no debe variar, ya que de por si se asigna el valor correcto a dicha variable. Por tanto, cuanto más completa sea la postcondición más precisa será nuestra diagnosis.

Ejemplo 3a: Para el código del ejemplo 3a se obtienen los siguientes nodos que aparecen en la tabla 3. Siguiendo el mismo procedimiento que para el ejemplo 1 y 2 se obtienen los resultados que se reflejan en la tabla 8. La sentencia S_{12} es justamente la que hemos cambiado. La sentencia S_6 nos indica que el error puede que se encuentre en la condición del bucle. La ejecución de la sentencia S_{12} más o menos veces depende de la satisfacción o no de la condición del bucle. El modelo no nos ofrece S_{11} como diagnosis mínima ya que, aunque el valor de la variable s depende del valor de p, llegamos a un valor correcto de p. El problema está en el valor de s, que no coincide con el esperado.

Tabla 7: Diagnosis mínima para el ejemplo 2a

Restricciones polinómicas	Igual que la tabla 6
f(x,y)	x+y
CT	{a=7,b=2,y0=7,x2=18,f(x0,y0)=16}
CCPM	$\{S_1S_3S_4\}$
Diagnosis mínima	$\{S_1\},\{S_3\},\{S_4\}$

Tabla 8: Diagnosis mínima para el ejemplo 3a

Restricciones polinómicas	Ver tabla 3
f(i,n)	i+n
CT	{n=5,p3=32,s3=63,f(i0,n)=5, β_i=4,β_p=15,β_s=79}
CCPM	$\{S_1S_3S_6S_9S_{12}\}$ $\{S_2S_3S_5S_6S_8S_9S_{11}S_{12}\}$
Diagnosis mínima	$\{S_3\},\{S_6\},\{S_{12}\}$

9. Conclusiones y futuros trabajos

En este trabajo se han aplicado las bases de Gröbner para diagnosticar el comportamiento del software. La metodología presentada realiza una gran reducción de la red de contextos, además realiza el proceso de forma simbólica y evita en gran medida la construcción del grafo de dependencias simbólicas, que se proponen en otras metodologías.

En los ejemplos mostrados hemos usado solo un *CT* para realizar la diagnosis, pero esta claro que cuantos mayor número de *CTs* utilicemos mejor será la diagnosis. La investigación sigue en ese camino, en buscar la forma de incorporar el resultado de varios *CTs* (lo que formaría un IUT) a la diagnosis de un mismo programa. A

medida que el cubrimiento de los diferentes *CTs* utilizados para un mismo programa se complete, la diagnosis será más precisa. Para investigaciones futuras estamos trabajando en ampliar la gramática que soporta la metodología. Una investigación en curso permitirá incorporar restricciones lógicas. La extensión de todo esto a la gramática completa de Java^{TM} se perfila como el objetivo final de esta investigación.

Referencias

[Binder00] Robert V. Binder. *Testing Object-Oriented Systems : Models, Patterms, and Tools.* Addison Wesley.

[Buchberger85] Buchberger B. Gröbner bases. *An algorithmic method in polynomial ideal theory.* Multidimensional Systems Theory, N. K. Bose, ed., D. Reidel Publishing Co., pag 184-232, 1985.

[DeKleer&Williams87] De Kleer J., and Williams, B.C. 1987. Diagnosis multiple faults. Atificial Intelligence 32(1):pag 97-130, 1987.

[Gasca01] R.M. Gasca, J.A. Ortega, M.Toro y F. De la Rosa T. *Diagnosis dirigida por restricciones simbólicas para modelos polinómicos.* Terceras Jornadas de trabajo sobre Metodologías Cualitativas Aplicadas a los Sistemas Socioeconomicos, Valladolid Julio 2001.

[Helzer95] Helzer G. *Gröbner Bases.* The Mathematica Journal Vol 5 Issue 1, 1995.

[Hoffman89] Hoffman C. M. *Geometric and solid modeling: An Introduction.* Morgan Kaufmann, 1989.

[Hollman93] Hollman J. y Langemyr. *Algorithms for Non-linear algebraic constraints.* Constraint Logic Programming Selected Research pag 113-131.

[Khalil98] Khalil, M. *Automated strategies for software diagnosis.* The Ninth International Sympsosium on Software Reliability Engineering, Paderborn, Germany, Nov. 1998.

[Khalil99] Khalil, M. *An Experimental Comparason of Software Diagnosis Methods.* 25^{th} Euromicro Conference 1999.

[Kapur&Laksman92] Kapur D. y Lakshman Y. N., *Elimination Methods: An Introduction.* Symbolic and Numerical Computation for Artificial Intelligence, D. Kapur and Mundy (eds.), Academic Press, 1992.

[Kapur95] Kapur, D. *An Approach for Solving Systems of parametric Polynomial Equations.* En Principles and Practice of Constraint Programming, pag 218-243,1995.

[Lyle&Weiser87] Lyle J. R. and Weiser, M. *Automatic bug location by program slicing.* Second International Conference on Computers and Applications, Beijing, China, pag. 877-883,June 1987.

[Mateis99] Cristinel Mateis, Markus Stumptner, Dominik Wieland and Franz Wotawa. *Debugging of Java programs using a model-based approach.* DX-99 Work-Shop,Loch Awe, Scotland (1999).

[Mateis00] Cristinel Mateis, Markus Stumptner, Dominik Wieland and Franz Wotawa. *Extended Abstract - Model-Based Debugging of Java Programs.* AADEBUG, August 2000,Munich.

[Reiter87] Reiter R. *A theory of diagnosis from first principles.* Artificial Intelligence, 32(1), pag 57-96, 1987.

[Weiser82] Weiser, M. *Programmers Use Slices When Debugging.* Communications of the ACM, Vol. 25, No. 7, pp.446-452,1982.

[Weiser84] Weiser, M. *Program Slicing.* IEEE Transactions on Software Engineering SE-10, 4, pp. 352-357, 1984