IDENTIFICACIÓN DE ERRORES EN SOFTWARE USANDO DIAGNOSIS BASADA EN MODELOS Y SATISFACCIÓN DE RESTRICCIONES

R. Ceballos[1] y R. M. Gasca[1]

1: Dpto. Lenguajes y Sistemas Informáticos
E.T.S. Ing. Informática
Universidad de Sevilla
41012 Sevilla

Palabras clave: Diagnosis basada en modelos, bug, testing, diseño por contrato, depuración

Resumen.

Las pruebas y depuración del software representan hoy en día una cantidad considerable de los recursos del desarrollo de aplicaciones. Para resolver los fallos en el software se han ido desarrollando técnicas automáticas para generar pruebas y depurar. La diagnosis basada en modelos permite determinar porque un sistema no funciona tal como esta especificado. Nuestro objetivo es adaptar las técnicas que se usan para la diagnosis de sistemas industriales a aplicaciones software, y generar una metodología que permita la diagnosis de software diseñado por contrato. El modelo usado para la diagnosis estará formado por un conjunto de restricciones generadas como una abstracción del código fuente y los asertos del diseño por contrato. Dicho modelo permite identificar de forma automática que parte de una aplicación contiene los fallos.

1. Introducción

La investigación en técnicas para generar pruebas y depurar software ha aumentado sustancialmente en los últimos años[9][6][5][3]. Los sistemas de información son cada vez más complejos, acercándose cada vez más a los límites de la capacidad humana y obligando a desarrollar técnicas para la identificación automática de fallos en el software. Las técnicas para generar pruebas permiten detectar errores en una aplicación, pero no garantizan si un programa es correcto al cien por cien. Las técnicas de depuración permiten aislar los errores pero de manera interactiva. Nuestro objetivo es la aplicación de las técnicas de diagnosis basada en modelos para detectar y aislar los errores de forma automática.

En las últimas décadas, la comunidad DX dedicada a la diagnosis basada en modelos (MBD) ha centrado sus esfuerzos en el perfeccionamiento de una metodología para la diagnosis de sistemas. Se parte de un modelo explícito del sistema, y a partir de él se

identifican los subsistemas que provocan los fallos basándose en la monitorización de las entradas y salidas del sistema. En la diagnosis del software el objetivo es aislar los errores debidos a la no concordancia entre los resultados especificados como correctos y los resultados reales recogidos al ejecutar un programa. Los programas que se diagnostican son aquéllos que aunque no producen errores en su ejecución y terminan correctamente, sin embargo no producen las salidas especificadas.

La diagnosis de programas se basa en técnicas de diagnosis basada en modelos (MBD, Model-Based Diagnosis), el diseño por contrato (DbC, Design by Contract), técnicas de generación de pruebas (testing) y la programación con restricciones (CP, Constraint Programming). Para una diagnosis más precisa que consiga un mejor aislamiento del error, es necesario conocer de la forma más exacta posible cuál es exactamente el comportamiento esperado. Por este motivo el diseño por contrato [7][8] es muy importante, ya que se especifica cuál debe ser el comportamiento de un programa.

Este trabajo se ha organizado en los siguientes apartados. En el apartado 2 se muestra el framework que permite la implementación de la metodología de diagnosis, y en el apartado 3 se muestra un ejemplo de diagnosis sobre un programa en Java^{TM}. Finalmente, se extraen las conclusiones y se describe el trabajo futuro en el apartado 4.

2. Framework para la diagnosis de errores en software

La figura 1 muestra los tres módulos que componen el proceso de diagnosis. El módulo de generación del modelo abstracto (AMG) recibe el código fuente y los contratos (asertos del diseño por contrato), para obtener el modelo abstracto del sistema. La principal idea es la transformación del código fuente y contratos en un modelo abstracto basado en restricciones. La obtención de las restricciones se basa en la división de las clases del sistema en bloques básicos que a su vez son divididos o transformados en restricciones.

El módulo de detección de errores (ED) permite detectar los errores (salidas incorrectas) producidos en la ejecución de un programa. Cuando un programa se ejecuta puede producirse un fallo en algún aserto, fallos en las salidas o la parada del programa por alguna excepción no manejada. El módulo ED recibe la división en bloques básicos obtenida en el módulo AMG. De esta forma, cuando el programa es ejecutado de forma supervisada, se almacena la traza de bloques básicos que forman la ejecución del código. Esta información se usará en el último módulo para el diagnostico del sistema.

El módulo de generación de la diagnosis (DG) obtiene la diagnosis del sistema para los fallos encontrados en el módulo ED. En la metodología propuesta se supone que el código fuente es casi correcto, y que la diferencia entre el programa actual y el correcto es pequeña. El objetivo es buscar esa pequeña diferencia. Para ello se identifican las incompatibilidades en los contratos debido a *asertos no factible* (infeasible assertion), y *errores en el código fuente* (bugs), como por ejemplo errores en las asignaciones o errores en las condiciones de sentencias selectivas, bucles o de lanzamiento y captura de excepciones. En los siguientes apartados se describirá de forma más precisa cada uno de los módulos enumerados.

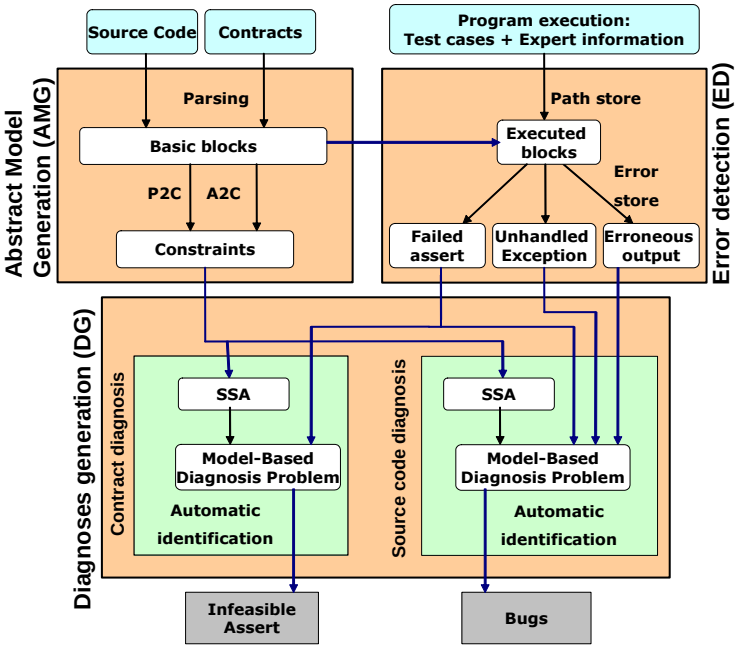

Figura 1: Framework para la diagnosis

2.1. Generación del modelo abstracto (AMG)

En la diagnosis basada en modelos[4] es necesario un modelo del sistema. El modelo del sistema para la metodología propuesta simula los componentes y sus interconexiones, usando para ello restricciones donde las variables representan las entradas y salidas de los componentes (sentencias del código). La generación de este modelo abstracto fue propuesta en trabajos anteriores [2][1]. Se basa en dos pasos secuenciales:

- División del sistema en bloques básicos: Cada una de las clases que forman un programa será transformada en un conjunto de bloques básicos. Estos bloques pueden ser: bloques de invariantes, bloques de atributos estáticos de la clase, bloques de atributos del objeto, o bloques de métodos de una clase. A su vez cada bloque de tipo método será transformado en subbloques de tipo condicional, bucles, llamadas a métodos, lanzamiento de excepciones, captura de excepciones o asignaciones.

- Generaración del modelo de restricciones: Para transformar los asertos a restricciones se ha implementado la función $A2C(Asserts\ to\ Constraints)$. La transformación del código fuente se basa en la trasformación de cada uno de los bloques básicos detectados de forma recursiva. El caso más simple es el de las asignaciones, que son trasformadas a restricciones de igualdad entre la expresión asignada y la variable o atributo correspondiente. Para automatizar este proceso de transformación se ha implementado la función $P2C(Program\ to\ Constraints)$.

CT	CP		
Código del test: Metodo sum Entradas: {x = 5, y = 6} Salidas correctas: {sum^2=11 $\wedge$ z^2 = 0} Salidas reales: {sum^2=6 $\wedge$ z^2 = 0}	Pre:{x > 0 $\wedge$ y > 0}	(S_8)	Assert:{max $\geq$ min}
	(S_0) public int sum(int x, int y){	(S_9)	z = max - min;
	(S_1) if (x >= y){	(S_{10})	sum = min * z;
	(S_2) min = y;	(S_{11})	while (z >0){
	(S_3) max = x;	(S_{12})	sum = sum + z;
	(S_4) }else{	(S_{13})	z = z - 1;
	(S_5) min = x;	(S_{14})	}
	(S_6) max = y;	(S_{15})	}
	(S_7) }	Post:{z = 0 $\wedge$ (1+|y-x|)*min(x,y) $\leq$ sum $\leq$ (1+|y-x|)*max(x,y)}	

Tabla 1: Ejemplo de caso de test (TC) aplicado al método *sum*

2.2. Detección de errores (ED)

Este módulo es el encargado de detectar cuando los resultados son diferentes a los especificados por el contrato, casos de test o la información de un experto. Cuando un programa se ejecuta, este módulo almacena información concerniente a los bloques básicos ejecutados en la traza seguida. Información como: secuencia de bloques básicos ejecutados, métodos llamados, valores de los atributos y variables en determinados puntos del programa, resultados de las evaluaciones de los asertos, valores de las condiciones de bucles o sentencias selectivas, excepciones lanzadas y capturadas, etc. Esta información es necesaria para el módulo de generación de la diagnosis, ya que el modelo que se usará para la diagnosis de un programa, estará basado en las restricciones obtenidas de los bloques básicos que forman la traza ejecutada.

2.3. Generación de la diagnosis (DG)

El módulo DG recibe los errores detectados en el módulo ED. Si el error es un aserto no satisfecho la diagnosis podría ser un aserto no factible o un bug en el código fuente. Si el error se debe a una salida incorrecta o a una excepción no capturada, el problema debe tener origen en un bug dentro del código fuente. La diagnosis de un programa es una hipótesis sobre cómo debería cambiar el programa diagnosticado para poder llegar a obtener el resultado esperado. El modelo utilizado en la metodología de diagnosis basada en modelos se construye a través del concepto de comportamiento anormal [4]. El predicado $AB(c)$ almacenará si un componente c del sistema tiene un comportamiento anormal. Los componentes con un comportamiento anormal compondrán la diagnosis del sistema.

Antes de seguir vamos a adaptar dos definiciones de la diagnosis basada en modelos:

Definición 1. *Modelo del sistema*: El modelo del sistema será una tupla formada por {DP(CP), CT} donde: CP es el conjunto de componentes del programa, es decir, el conjunto de instrucciones y asertos; DP es el problema de diagnosis, es decir, el modelo abstracto basado en restricciones; y CT es un caso de test.

Definición 2. *Diagnosis*: Sea $\mathcal{D} \subseteq$ CP, entonces el subconjunto $\mathcal{D}$ será una diagnosis si el conjunto de restricciones DP$'$ $\cup$ CT es satisfactible, donde DP$'$ = DP(CP $-$ $\mathcal{D}$).

Para obtener la diagnosis mínima se genera un Max-CSP (Maximal Constraint Satisfac-

	DP
Pre. & CT:	$P(S_1) = \text{true} \wedge x = 5 \wedge y = 7$
S_1:	$P(S_1) \wedge \neg AB(S_1) \Rightarrow P(S_2) = (x \geq y)$
	$P(S_1) \Rightarrow P(S_2) \neq P(S_5)$
	$\neg P(S_1) \Rightarrow P(S_2) = \text{false} \wedge P(S_5) = \text{false} \wedge P(S_{50}) = \text{false}$
	$(P(S_2) \wedge P(S_4)) \vee (P(S_5) \wedge P(S_7)) \Rightarrow P(S_8) = \text{true}$
	$(P(S_2) \wedge \neg P(S_4)) \vee (P(S_5) \wedge \neg P(S_7)) \Rightarrow P(S_8) = \text{false}$
S_2:	$(P(S_2) \wedge \neg AB(S_2) \Rightarrow \min = y) \wedge P(S_3) = P(S_2)$
S_3:	$(P(S_3) \wedge \neg AB(S_3) \Rightarrow \max = x) \wedge P(S_4) = P(S_3)$
S_5:	$(P(S_5) \wedge \neg AB(S_5) \Rightarrow \min = x) \wedge P(S_6) = P(S_5)$
S_6:	$(P(S_6) \wedge \neg AB(S_6) \Rightarrow \max = y) \wedge P(S_7) = P(S_6)$
S_8:	$\max \geq \min$
S_9:	$(P(S_8) \wedge \neg AB(S_8) \Rightarrow z = \text{max-min}) \wedge P(S_9) = P(S_8)$
S_{10}:	$(P(S_9) \wedge \neg AB(S_9) \Rightarrow \text{sum} = \min^*(z+1)) \wedge P(S_{10}) = P(S_9)$
S^1_{11}:	$P(S^1_{11}) \wedge \neg AB(S^1_{11}) \Rightarrow P(S^1_{12}) = (z^1 > 0)$
	$P(S^1_{11}) \wedge \neg P(S^1_{12}) \Rightarrow P(S_{15}) = \text{true} \wedge \text{sum}^2 = \text{sum} \wedge z^2 = z$
	$\neg P(S^1_{11}) \Rightarrow P(S^1_{12}) = \text{false} \wedge P(S_{15}) = \text{false}$
	$P(S^1_{12}) \wedge P(S^1_{14}) \Rightarrow P(S_{15}) = \text{true}$
	$P(S^1_{12}) \wedge \neg P(S^1_{14}) \Rightarrow P(S_{15}) = \text{false}$
S^1_{12}:	$(P(S^1_{12}) \wedge \neg AB(S^1_{12}) \Rightarrow \text{sum}^1 = \text{sum} + z) \wedge P(S^1_{13}) = P(S^1_{12})$
S^1_{13}:	$(P(S^1_{13}) \wedge \neg AB(S^1_{13}) \Rightarrow z^1 = z - 1) \wedge P(S^2_{11}) = P(S^1_{13})$
S^2_{11}:	$P(S^2_{11}) \wedge \neg AB(S^2_{11}) \Rightarrow P(S^2_{12}) = (z^2 > 0)$
	$P(S^2_{11}) \wedge \neg P(S^2_{12}) \Rightarrow P(S^1_{14}) = \text{true} \wedge \text{sum}^2 = \text{sum}^1 \wedge z^2 = z^1$
	$\neg P(S^2_{11}) \Rightarrow P(S^2_{12}) = \text{false} \wedge P(S^1_{14}) = \text{false}$
	$P(S^2_{12}) \wedge P(S^2_{14}) \Rightarrow P(S^1_{14}) = \text{true}$
	$P(S^2_{12}) \wedge \neg P(S^2_{14}) \Rightarrow P(S^1_{14}) = \text{false}$
S^2_{12}:	$(P(S^2_{12}) \wedge \neg AB(S^2_{12}) \Rightarrow \text{sum}^2 = \text{sum}^2 + z^1) \wedge P(S^2_{13}) = P(S^2_{12})$
S^2_{13}:	$(P(S^2_{13}) \wedge \neg AB(S^2_{13}) \Rightarrow z^2 = z^1 - 1) \wedge P(S^2_{14}) = P(S^2_{13})$
CT:	$P(S_{15}) \wedge \text{sum}^2 = 18 \wedge z^2 = 0$
Post:	$z = 0 \wedge (1 + \lvert y - x \rvert) * \min(x, y) \leq \text{sum}^2 \leq (1 + \lvert y - x \rvert) * \max(x, y)$

Tabla 2: Modelo abstracto del ejemplo de la tabla 1

tion Problem), un framework preparado para satisfacer el máximo número de restricciones posible. El problema Max-CSP estará guiado por una función objetivo: encontrar la asignación del conjunto de variables *AB* que satisfaga el mayor número de restricciones del DP. Si todas los predicados *AB* toman el valor falso implica que el programa es correcto. En otro caso, el problema Max-CSP obtendrá como solución cada una de las diagnosis.

3. Ejemplo de diagnosis automática

La tabla 2 muestra el DP obtenido para el programa de la tabla 1. En el ejemplo mostrado nos centraremos en fallos sólo en el código fuente. El proceso de diagnosis ofrecerá como solución la traza que se debería ejecutar y que sentencias fallan en esa traza. El predicado *P(S_i)* almacenará si una sentencia S_i del programa pertenece o no a la traza ejecutada en la solución. De esta forma las trazas posibles, dependientes de los valores que toman las condiciones, están recogidas en el modelo. Las sentencias incluidas en bucles aparecen con un superíndice que indica el número de iteración a la que pertenecen. Las restricciones obtenidas del CT y de los asertos del contrato se toman como correctas y en la solución se obliga su satisfacción.

Para el caso mostrado en la tabla 1, las diagnosis mínimas son S_{10} y S_{12}. Cambiando S_{12} se soluciona el problema pero sólo para el caso de test propuesto. Si se cambia S_{10} por *sum = min * (z + 1)*, el programa funcionaría correctamente en todos los casos posibles.

4. Conclusiones y trabajo futuro

La metodología de diagnosis propuesta integra diferentes técnicas provenientes de la Inteligencia Artificial y la Ingeniería del Software. Aunque el ejemplo presentado es sencillo, la metodología desarrollada permite ya detectar errores en los contratos, en asignaciones, en las condiciones de guardas, y en el lanzamiento y captura de excepciones. La investigación se centra ahora en extender la metodología a características propias de un lenguaje orientado a objetos, como por ejemplo la herencia. Otro importante objetivo, es aplicar la metodología a ejemplos más complejos, donde el mapeo a restricciones no sea tan directo.

5. Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología de España (Proyecto DPI2003-07146-C02-01, y la Red REPRIS TIN2005-24792-E).

REFERENCIAS

[1] R. Ceballos, F. de la Rosa, and S. Pozo. Diagnosis de inconsistencia en contratos usando el diseño bajo contrato. In *IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 25–36, Malaga, Spain, 2004.

[2] R. Ceballos, R. M. Gasca, C. D. Valle, and F. D. L. Rosa. A constraint programming approach for software diagnosis. In *AADEBUG Fith International Symposium on Automated and Analysis-Driven Debugging*, pages 187–196, Ghent, September 2003.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005.

[4] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 2-3(56):197–222, 1992.

[5] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21, Roma, Italy, 2002.

[6] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of java programs. In *AADEBUG*, August 2000.

[7] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[8] Y. L. Traon, F. Ouabdesselam, C. Robach, and B. Baudry. From diagnosis to diagnosability: Axiomatization, measurement and application. *The Journal of Systems and Software*, 1(65):31–50, January 2003.

[9] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4(10):352–357, 1984.