Diagnosis de inconsistencia en contratos usando el diseño por contrato

R. Ceballos, F. de la Rosa T., S. Pozo

***Resumen*--El diseño por contrato permite desarrollar aplicaciones más fiables y robustas. Un software es fiable si realiza su trabajo conforme a una especificación, y es robusto si es capaz de controlar las situaciones anormales que puedan producirse. En este artículo se propone una metodología para la diagnosis de errores en el software, basada en la combinación del diseño por contrato, la diagnosis basada en modelos y la programación con restricciones. Los contratos establecidos en el diseño (en forma de asertos), junto con una abstracción del código fuente son transformados en restricciones, formando el modelo del sistema. Estableciendo una función objetivo adecuada a estas restricciones, se detectan qué asertos o qué bloques de código son incorrectos. A partir del contrato software y del código fuente, se plantea un problema típico de diagnosis, pero aplicado al software. La originalidad del trabajo radica en el mapeo del contrato y código a restricciones, y en la obtención de los asertos y bloques de código que hacen inviables los contratos establecidos. Para ello se utilizan técnicas de resolución de restricciones.**

***Índice de Términos*-- diagnosis basada en modelos, diseño por contrato, testing, restricciones.**

I. Introducción

El diseño de software por contrato (DbC)([2]) permite desarrollar aplicaciones más fiables y robustas. El mayor componente de calidad dentro del software es la fiabilidad, es decir, la exactitud con la que el sistema realiza su trabajo conforme a una especificación. Un sistema robusto ofrece garantías de poder controlar las situaciones anormales. La calidad del software toma mayor relevancia en la OO (Orientación a Objetos) debido a la reutilización del software. Otra ventaja del DbC consiste en permitir alcanzar, casi de forma automática, un framework para testar y depurar.

El uso de asertos en el DbC permite aproximarnos a un software potencialmente libre de errores. En [11] se realiza un estudio del impacto que conlleva el uso de los asertos en la programación. Para ello se definen principalmente dos métricas: La robustez y la diagnosticabilidad. La robustez se define como la capacidad del sistema para recuperarse de un fallo. La diagnosticabilidad expresa el esfuerzo necesario para localizar un fallo, así como la precisión obtenida al realizar el testing del sistema. Este estudio pone de relieve que la robustez crece rápidamente con pocos contratos, pero a su vez, resulta muy costoso alcanzar la robustez total. Del estudio también se desprende que la calidad de los contratos, más que la cantidad, influye positivamente en la diagnosticabilidad.

En [12] se demuestra que el uso de contratos permite un mayor aislamiento de los fallos de un sistema. En la OO, las diferentes funcionalidades se consiguen combinando los resultados de diferentes componentes, lo que implica una distribución mayor de las funcionalidades, y una mayor complejidad en las interacciones. La especificación de los contratos debe empezar en la etapa de análisis, por ejemplo creando contratos en OCL (Object Constraint Language). Este mayor esfuerzo en la creación de contratos mejora los resultados a la hora de detectar y localizar los errores.

En este artículo se propone una metodología para la diagnosis de errores en el software, permitiendo la detección de errores en los contratos y en el código fuente. La idea principal consiste en convertir los contratos y código fuente a restricciones, y posteriormente utilizar técnicas Max-CSP (Maximal Constraint Satisfaction Problem) para obtener la diagnosis mínima.

La programación con restricciones nos permite modelar y resolver problemas reales como un conjunto de restricciones entre variables. Un problema de satisfacción de restricciones se define basándose en un conjunto de variables $X=\{X_1,X_2,...,X_n\}$ asociadas a unos dominios, $D=\{D_1,D_2,...,D_n\}$, y un conjunto de restricciones $C=\{C_1,C_2,...,C_m\}$. Cada restricción C_i es una tupla (W_i, R_i), donde R_i es una relación $R_i \subseteq D_{i1} x ... x D_{ik}$ definida para el subconjunto de variables $Wi \subseteq X$.

Si tenemos un CSP, el objetivo del Max-CSP consiste en encontrar una asignación que satisfaga el mayor número de restricciones, y que minimice el número de restricciones violadas. El objetivo de la diagnosis consiste en encontrar qué sentencias al ser ejecutadas provocan el comportamiento incorrecto del programa.

Para llevar a cabo el proceso de diagnosis de una forma eficiente, se utilizan técnicas de testing para seleccionar qué observaciones son las más significativas y aportan más información para detectar errores en los programas. En [1]

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia y Tecnología de España (DPI2003-07146-C02-01) y el European Regional Development Fund (ERDF/FEDER).

R. Ceballos, F. de la Rosa T. y S. Pozo pertenecen al Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Escuela Técnica Superior de Ingeniería Informática, Avenida Reina Mercedes s/n 41012 Sevilla (Spain) (e-mail: ceballos,ffrosat,sergio@lsi.us.es).

aparecen los objetivos y las complicaciones que implica un buen testing.

La mayoría de las aproximaciones desarrolladas en diagnosis software han basado su metodología en el uso de modelos. El proyecto JADE ([8]) investiga la diagnosis software desde la perspectiva de la Depuración Basada en Modelos (Model Based Debugging, MBD). En los trabajos relativos a este proyecto, se usa un modelo de dependencias basado en el código fuente. El modelo representa las sentencias y las expresiones como si fuesen componentes, y las variables como si fuesen las conexiones entre componentes. Se trata de transformar constructos Java™ a componentes. Las asignaciones, sentencias condicionales, bucles, etc. tienen su correspondiente método de transformación.

También para diagnosis de software, y previamente a estos trabajos, fue propuesta la técnica *Slicing*. Esta técnica identifica los constructos del código fuente que pueden influenciar el valor de las variables en un punto determinado del programa ([10]). *Dicing* ([7]) es una extensión de esta técnica. En los últimos años, nuevos métodos ([5]) han sido desarrollados para automatizar el proceso de diagnosis de software.

El trabajo que presentamos mejora y extiende artículos anteriores como [3] y [4] en varios aspectos. Como principales aportaciones aparecen la capacidad de evaluar si un contrato es viable o no, y la capacidad de detectar las incompatibilidades entre el código fuente y los contratos.

El artículo está dividido en las siguientes partes. Primero se exponen algunas definiciones necesarias y previas a la explicación de la metodología, y a continuación se presenta un ejemplo sencillo donde aplicaremos la metodología. El proceso de diagnosis se realiza en dos fases: Una primera en la que sólo se tienen en cuenta los contratos (asertos), y una segunda en la que se utilizan además las sentencias de los programas. Por último se presentan las conclusiones y trabajos futuros.

II. Notación y definiciones

Definición 1. Aserto inviable (AI): Un contrato software está formado por un conjunto de asertos que deben cumplirse dentro de la traza de un programa. Un aserto inviable será aquel que no puede cumplirse, porque entra en contradicción con asertos o instrucciones que forman parte de la traza antes o después de dicho aserto.

Definición 2. Error en el código fuente (bug): Un *bug* será una o varias sentencias del código fuente que impiden alcanzar los resultados especificados como correctos. Los errores que contemplaremos serán aquellos que surgen como una pequeña variación del programa correcto, como por ejemplo errores en las guardas (de bucles o sentencias selectivas) o errores en la asignación a variables. Este artículo no tiene en cuenta lo que serían errores en tiempo de compilación (como errores en la sintaxis o semántica del lenguaje), ni los errores dinámicos (tales como excepciones, violaciones de acceso a memoria, bucles infinitos, etc).

Definición 3. Caso de Test (CT)}: Será un conjunto de valores para las variables o parámetros de entrada del código, obtenidos a través de técnicas de testing. Las salidas correctas del código vendrán dadas por los asertos si el diseño por contrato es lo suficientemente fuerte; si esto no sucede, un experto es quien debe decidir qué salidas son las correctas.

Definición 4. Traza: La ejecución de un programa depende del CT escogido. Las sentencias selectivas, bucles y llamadas a métodos, permiten incorporar diferentes instrucciones a la traza del programa en función de las condiciones iniciales. Por tanto, para diferentes entradas es posible que la secuencia de sentencias ejecutadas sea diferente. La secuencia de instrucciones que se ejecuten formarán lo que denominaremos *traza*.

Definición 5. Modelo de restricciones del programa (MRP): El MRP está formado por una red de restricciones C y un conjunto de variables V con sus dominios. La red de restricciones se obtiene a partir de los asertos del contrato software y del código fuente. El MRP permite tener un modelo del comportamiento que tendrá el programa. El número de restricciones que se incorporan al MRP depende de la *traza*.

Definición 6. Restricciones de satisfacción concretada (reified constraints): Las restricciones de satisfacción concretada poseen un valor de verdad asociado a la satisfacción o no de cada restricción.

Definición 7. Problema de restricciones de satisfacción concretada (RCSP, Reified Constraint Satisfaction Problem): Se define como un problema de satisfacción de restricciones (CSP) en el cual existen restricciones de satisfacción concretada.

Definición 8. Problema de maximización de restricciones (Max-CSP, Maximal Constraint Satisfaction Problem): El objetivo de un Max-CSP consiste en encontrar una asignación que satisfaga el mayor número de restricciones, y que minimice el número de restricciones violadas.

Definición 9. Diagnosis: Será el conjunto de asertos o sentencias que provocan el comportamiento anómalo del programa, y que por tanto deben ser modificados.

Definición 10. Diagnosis mínima: Será la diagnosis que implica modificar el menor número de asertos o sentencias de un programa para lograr el comportamiento definido como correcto.

III. Ejemplo

Para clarificar las explicaciones se usará la clase *CuentaBancoImp* que implementa la interfaz *CuentaBanco*. Esta interfaz modela una cuenta bancaria en la cual podemos realizar ingresos y retirar capital. El código fuente, incluyendo la especificación de los contratos, se muestra en la figura 1. El método *ingresar* se ha alterado, de forma intencionada, para que decremente el saldo en lugar de

```
/**
 *  @inv getSaldo()>= 0
 */
interface CuentaBanco {

  /**
   *  @pre ingreso > 0
   *  @post getSaldo() >= 0
   */
  public void ingresar (double ingreso);

  /**
   *  @pre cantidad > 0
   *  @post getSaldo() ==
   *           getSaldo()@pre - cantidad
   */
  public void retirar (double cantidad);

  public double getSaldo ();

}
```

```
class CuentaBancoImp implements CuentaBanco {

    private double interes;

    private double saldo;

    public CuentaBancoImp()  {
      this.saldo = 0;
    }

    public void ingresar (double ingreso) {
      this.saldo = this.saldo - ingreso;
    }

    public void retirar (double cantidad) {
      this.saldo = this.saldo - cantidad;
    }

    public double getSaldo() {
      return this.saldo;
    }
}
```

Fig. 1. Código fuente de la clase CuentaBancoImp y de la interfaz CuentaBanco

de incrementarlo. Este error será diagnosticado más adelante.

IV. Determinación de los bloques básicos

La diagnosis basada en modelos ([9]) necesita generar un modelo de la realidad para detectar inconsistencias. El modelo del sistema representa los componentes interconectados entre sí. El sistema ofrece una funcionalidad en conjunto debido a la forma en que están conectados los componentes y debido al comportamiento especificado de cada uno de los componentes.

En nuestro caso pretendemos realizar la diagnosis de un programa. Para alcanzar una determina funcionalidad con un programa OO debemos interconectar las llamadas entre los diferentes métodos que forman las clases o componentes software. Cada método se puede considerar como un subsistema que genera una funcionalidad determinada, debido a las entradas que recibe y a los atributos del objeto (que definen el estado del objeto). Al ejecutar un programa estamos interconectando diferentes sentencias, logrando obtener una determinada funcionalidad, y formando lo que se llama la traza del programa.

Para realizar la diagnosis es necesario reconocer del programa cuáles son los bloques de sentencias que dependiendo de la traza se interconectarán de una u otra forma. Cualquier aplicación OO se compone de un conjunto de clases, por tanto el primer paso consiste en localizar cuáles son las clases implicadas en la ejecución de un programa. A partir de cada una de las clases se obtendrá una serie de bloques de los tipos:

- Bloque de invariantes de la clase: Compuesto por los asertos que deben cumplirse en cualquier momento dentro de un objeto.
- Bloque de atributos de la clase: Incluyendo las declaraciones de los atributos estáticos de la clase.
- Bloque de atributos de los objetos: Incluirá las declaraciones de los atributos no estáticos.
- Bloque método o constructor: Formado por las declaraciones y sentencias incluidas en el método o constructor, así como los asertos asociados al bloque (precondición y postcondición por ejemplo).

A su vez, cada uno de los bloques método o constructor contendrá bloques de los tipos:

- Bloques condicionales: Compuestos por un bloque para cada una de las alternativas posibles seguido de las sentencias a ejecutar en cada condición.
- Bloques tipo bucle: Compuestos por un bloque para cada una de las iteraciones que se produzcan. El número de iteraciones sólo es conocido en tiempo de ejecución.

Cada uno de los bloques puede contener asignaciones, que serán consideradas como bloques indivisibles. Las llamadas a métodos y creación de objetos permite deducir cuál es el orden en que los diferentes bloques se enlazan.

V. Diagnosis basada en asertos

El proceso de diagnosis se realiza en dos fases: una primera en la que sólo se tienen en cuenta los contratos, y una segunda en la que entran en juego las sentencias de los programas más los asertos de los contratos. En la primera fase podemos distinguir pruebas en las que se utilizan casos de test y pruebas en las que no.

Para comprobar que los asertos del contrato son viables, se irán formando grupos de restricciones de satisfacción concretada que formarán un RCSP, donde cada aserto estará asociado a una variable booleana, denominada variable de satisfacción. El objetivo será que el máximo número de asertos alcancen un valor verdadero en la variable de satisfacción, es decir, intentar que todos los asertos sean viables secuencialmente. Para maximizar el número de asertos satisfactibles la búsqueda se concreta en la denominada función objetivo, que junto al RCSP formará un problema Max-CSP. Utilizando un resolutor de restricciones

se obtendrán las diferentes soluciones. Si todas las variables de satisfacción alcanzan el valor verdadero implica que todas las restricciones son viables en conjunto. Si no, el propio Max-CSP ofrecerá como resultado el número mínimo de restricciones que deben modificarse (el paso a los asertos asociados a esas restricciones es directo).

Las restricciones generadas pueden ser evaluadas por el resolutor del CSP en cualquier orden, por tanto será necesario transformar las restricciones derivadas de los asertos realizando un renombrando de las variables. Con este paso se consigue simular la evaluación secuencial de los asertos, ya que cada restricción sólo contiene variables en común con las que aparecen en las restricciones anteriores.

A través de la programación con restricciones podemos detectar si un conjunto de asertos es viable o no, en función de si existe o no una posible solución para el sistema.

A. Diagnosis basada en asertos y sin casos de test

Las comprobaciones a realizar serán:

- Comprobación de todos los invariantes en conjunto de una clase: Los invariantes de una clase deben cumplirse siempre, por tanto se generará un Max-CSP con los invariantes de cada clase.
- Comprobación de los asertos de los métodos: Cada uno de los asertos de un método (precondición o postcondición por ejemplo) debe ser compatible con el conjunto de invariantes de la clase que contiene el método. Por cada aserto más el conjunto de invariantes podremos generar un Max-CSP.

Cada uno de las comprobaciones realizadas, nos aporta unos dominios posibles para los atributos y variables del sistema. Esos dominios son valiosos para generar casos de test en los siguientes apartados. El objetivo en esta primera fase es doble. Por una parte diagnosticar la viabilidad de los asertos que forman el contrato software. Y por otra parte, para la diagnosis basada en casos de test, encontrar cuáles son los dominios posibles para dichos casos de test.

TABLA I
DIAGNOSIS DEL MÉTODO *RETIRAR* CON CASO DE TEST

Entradas del CT	saldo@pre=0, cantidad>0
Invariante	(saldo@pre>=0)=SA1
Precondición	(cantidad>0)=SA2
Postcondición	(saldo=saldo@pre-cantidad)=SA3
Invariante	(saldo>=0)=SA4
Salidas del CT	saldo=0

B. Diagnosis basada en asertos y con casos de test

Aplicando diferentes casos de test a la secuencia formada por {invariantes + precondición + postcondición + invariantes} en cada uno de los métodos, podremos obtener más información sobre la viabilidad de los métodos. Por cada precondición y postcondición más el conjunto de invariantes se podrá generar un Max-CSP. Las variables de satisfacción asociadas a los asertos las nombraremos empezando por *SA*.

El objetivo es intentar que todos los asertos se cumplan, y si no es posible, intentar determinar qué aserto es inviable. En la tabla 1 aparece la transformación a restricciones de la especificación del contrato del método *retirar*. Aparecen cuatro restricciones asociadas a cuatro variables de satisfacción. Las cuatro restricciones corresponden al invariante antes de la precondición, la precondición, la postcondición y el invariante tras la postcondición. El caso de test utilizado corresponde con un saldo inicial de 0 unidades y un intento de retirar una cantidad positiva, dando como resultado que el saldo debe seguir siendo 0.

En este ejemplo resulta imposible que todos los asertos se cumplan a la vez, debido a la especificación de la postcondición. Si observamos, el saldo tiene que ser mayor o igual que cero al final del método, pero cumplir el invariante implica que la precondición y la postcondición no pueden cumplirse a la vez. La postcondición obliga a cumplir *saldo=saldo@pre-cantidad*, o lo que es lo mismo si se cumplen el resto de restricciones es equivalente a suponer que *0-cantidad>0*, algo no satisfactible si *cantidad* es positivo.

El problema radica en que la precondición no es lo suficientemente fuerte como para impedir llegar a la postcondición, es decir, no basta con que la cantidad a retirar sea positiva, es necesario además que sea menor al saldo que en ese momento tiene la cuenta.

La solución general para estos casos será reforzar la precondición, o bien debilitar la postcondición. Otra opción sería modificar el invariante, permitiendo que el saldo de la cuenta admitiera valores negativos. Dependiendo del caso una u otra solución será la correcta.

VI. DIAGNOSIS USANDO ASERTOS Y CÓDIGO FUENTE

A. Generación del caso de test y auditoria de la traza

Los casos de test que se usarán se basarán en los dominios obtenidos en el apartado anterior. Las técnicas de testing permiten seleccionar qué observaciones son las más significativas y aportan más información para detectar los errores en los programas, por tanto, su uso, permitirá llevar a cabo el proceso de diagnosis de una forma más certera y eficiente.

Una vez generado el caso de test, se ejecuta el programa de forma controlada, almacenando qué bloques básicos se han ejecutado y en qué orden. Se trata de auditar la traza que ha seguido el programa, incluyendo los asertos por los que se ha pasado y se han verificado. Las llamadas entre métodos permiten enlazar bloques básicos de código que en su conjunto forman la traza del programa. La traza seguida dependerá del CT, llamadas entre métodos, y de las condiciones de bucles y sentencias selectivas. Cada vez que se haga referencia a una clase por primera vez se debe ejecutar el bloque de atributos de la clase. Igualmente cada vez que se haga referencia al constructor de una clase se debe ejecutar el bloque de atributos del objeto asociado a dicha clase.

El siguiente paso consiste en transformar la secuencia de sentencias, de tal forma que no se permitan dos sentencias que asignen un valor a la misma variable. Este paso es previo y necesario para generar las restricciones ya que permite almacenar todos los valores por los que ha pasado una variable en la ejecución del código a diagnosticar, y su vez, permite simular la evaluación secuencial del código fuente. Será necesario distinguir entre los que son atributos de clase, de objeto y variables locales, ya que el tratamiento debe tener en cuenta hasta dónde llega la visibilidad de estas variables.

B. Obtención de las restricciones

Las restricciones que se obtengan de los asertos del contrato y del código fuente formarán las restricciones del programa. Las restricciones del DbC se obtendrán directamente de los asertos. Para obtener las restricciones del código fuente, transformaremos las sentencias que forman los bloques básicos a restricciones.

El paso de sentencias del código fuente a restricciones es algo que ya se ha presentado en anteriores trabajos, como [3] y [4], que pueden ser revisados para una mayor concreción. A continuación mostramos un resumen de los pasos a seguir.

- Bloques básicos indivisibles:
  Asignaciones: { *Ident:=Exp* }
  A partir de la sentencia de asignación obtendremos la restricción: {SS_{SAsig}=(Ident=Exp)}, donde SS_{SAsig} representa la satisfacción de la restricción *Ident=Exp*. Con esta restricción indicamos que si la sentencia S_{Asig} no forma parte de la diagnosis mínima, tras la ejecución de S_{Asig} debe cumplirse la igualdad entre la variable asignada y la expresión que se asigna.
  Llamadas a métodos: Para cada llamada a un método, añadiremos las restricciones definidas en la precondición y la postcondición del método. Cuando nos encontremos ante una método recursivo, las llamadas internas al método recursivo deben suponerse válidas para poder llevar a cabo la verificación formal de las llamadas recursivas.
- Bloques condicionales: {*if* (*cond*) {$Bloque_{If}$} *else* {$Bloque_{Else}$}}
  Las trazas posibles son para una sentencia condicional dos, según se cumpla o no la condición de la guarda. Dependiendo del CT, iremos por una traza u otra. Las restricciones que se obtendrán de una sentencia condicional incluirán la condición de la guarda que se cumpla más las restricciones que se obtengan de las sentencias del bloque que se ejecute según la traza seguida.
- Bloques tipo bucle: {*while* (*cond*) {$Sentencias_{Bucle}$}}
  El principal problema viene dado en la imposibilidad de establecer una cota para el número de veces que debe iterar un bucle, ya que dependerá del CT. La estructura de un bucle puede ser simulada a través de varias sentencias selectivas anidadas. Cada una de las iteraciones sería transformada a una sentencia selectiva, en la que además de la condición de finalización del bucle, tendríamos que tener en cuenta que previamente se han ejecutado las iteraciones anteriores de forma secuencial y en el orden previsto. En [3] se propone una técnica para reducir el modelo a menos de *n* iteraciones, y obtener una mayor eficiencia en el proceso de diagnosis.

C. Obtención del MRP

Una vez transformados los asertos y sentencias a restricciones, el siguiente paso consiste en obtener qué parte del código o qué asertos impiden obtener el resultado esperado. A partir del caso de test se ha auditado la traza del programa (epígrafe VI, punto A), obteniendo la secuencia de sentencias ejecutadas y asertos verificados. Con la traza es posible crear un RCSP con variables de satisfacción asociadas a las sentencias del código fuente. Este tipo de variables las nombraremos empezando por *SS*.

TABLA II
CASO DE TEST PARA LA CLASE *CUENTABANCOIMP*

Entradas	saldo@pre=300, cantidad=300, ingreso=300
Salidas	saldo=300
Código prueba	cuenta.retirar(cantidad) cuenta.ingresar(ingreso)

D. Obtención de la diagnosis mínima

El siguiente paso es encontrar el conjunto de sentencias que forman la diagnosis. Para que dicho conjunto tenga la cardinalidad mínima, se debe intentar maximizar el número de predicados *SS* que tomen el valor verdadero. Para lograrlo se establece un Max-CSP que intente lograr dicho objetivo, obteniendo como resultado los valores de los predicados *SS* para cada una de las sentencias de la traza seguida. Estos valores definirán el grupo sentencias que forman la diagnosis.

Las restricciones obtenidas de los asertos aportan información que debe considerarse valida ya que la precisión de la diagnosis se basa en esta información. El proceso de búsqueda dependerá de los resultados obtenidos con el caso de test, y de la situación en la que termine el programa:

- Si se produce la parada del programa por un aserto no válido y no se alcanza el final establecido en el CT del programa, puede deberse a que el aserto es demasiado estricto o bien que el programa no está bien diseñado. Para saber cuál es el origen del problema, se debe ejecutar de nuevo el programa sin el aserto. De esta forma se podrá deducir si el programa es capaz de llegar al resultado esperado, aunque el aserto no sea satisfecho. Si esto sucede nos encontramos ante un aserto muy restrictivo que debe ser modificado. Si no se alcanza el final y el programa vuelve a interrumpirse, el problema se debe al código que está hasta el aserto.
- Si el programa acaba sin incumplir ningún aserto, pero el resultado no es el especificado por el caso de

test, puede deberse a que alguna sentencia impide llegar al resultado esperado, o bien, algún aserto es poco restrictivo y deja realizar operaciones que no permiten llegar al resultado correcto. Con las restricciones del MRP se planteará un problema Max-CSP. El proceso determinará qué sentencias son erróneas.

E. Ejemplo

Para clarificar la presentación de la metodología, nos vamos a centrar en el caso de test propuesto en la tabla 2. Se trata de una cuenta con un saldo inicial de 300 unidades, sobre ella realizamos dos operaciones secuenciales. Primero una retirada de capital de 300 unidades y segundo un ingreso por la misma cantidad. Como resultado el saldo de la cuenta debe quedar con 300 unidades.

TABLA III
MRP DE LA CLASE *CUENTABANCOIMP* CON EL CT DE LA TABLA II

cuenta.retirar(cantidad)	Invariante : saldo0>=0 Precondición : cantidad>0 Código : (saldo1=saldo0-cantidad)=SS1 Postcondición : saldo1=saldo0-cantidad Invariante : saldo1>=0
cuenta.ingresar(ingreso)	Invariante : saldo1>=0 Precondición : ingreso>0 Código : (saldo2=saldo1-ingreso)=SS2 Postcondición : saldo2>=0 Invariante : saldo2>=0

Una vez generadas las restricciones del problema de diagnosis (tabla 3), el motor de inferencias para solucionar el CSP obtendrá que la sentencia incluida en el método *ingresar* es la que provoca el error. Si observamos la sentencia, se ha realizado una resta en lugar de una suma. La postcondición de dicho método es demasiado débil, y no detecta dicho problema. Cambiando la sentencia indicada, el problema se resuelve. Es necesario resaltar que la sentencia incluida en el método *retirar* no es ofrecida como posible error, aunque influye en el resultado final del valor del saldo de la cuenta. La razón está en su postcondición, que es lo suficientemente fuerte como para garantizar el valor que debe tomar el saldo al finalizar dicho método. Por tanto, contratos más fuertes permiten localizar mejor el origen de errores.

VII. CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo hemos propuesto una integración de técnicas de programación con restricciones, de diagnosis basada en modelos, y diseño por contrato, para automatizar la diagnosis del software y de los contratos. Con respecto a anteriores trabajos. se han ampliado aspectos del diseño por contrato, y se ha incorporado la capacidad de evaluar si un contrato es viable. De los ejemplos estudiados se deduce que cuanto más fuerte es el DbC y más casos de test se utilicen, más certera es la diagnosis, ya que la metodología dispone de más información del comportamiento real y del que se espera.

Nuestra investigación se centra ahora en hacer aún más precisa la diagnosis y extender la metodología a otras características de un lenguaje orientado a objetos, como pueden ser la herencia, gestión de excepciones, concurrencia, etc. Otro importante objetivo futuro es aplicar la metodología a ejemplos más complejos y reales, donde el mapeo a restricciones no sea tan directo.

VIII. REFERENCIAS

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

[2] Meyer, B.: Applying 'Design by Contract'. IEEE Computer, vol. 25, no. 10, October 1992, pp. 40-51.

[3] R. Ceballos, R. M. Gasca, Carmelo Del Valle and Miguel Toro: Max-CSP approach for software diagnosis. LNAI 2527 pp. 172-181, VIII Conferencia Iberoamericana de Inteligencia Artificial, Sevilla, November 2002.

[4] R. Ceballos, R. M. Gasca, Carmelo Del Valle and F. De La Rosa: A constraint programming approach for software diagnosis. AADEBUG 2003, September, Ghent, Belgium.

[5] Khalil, M.: An Experimental Comparason of Software Diagnosis Methods. 25th Euromicro Conference 1999.

[6] ILOG: ILOG Solver 5 User's Manual. ILOG 2003.

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

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

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

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

[11] Y. Le Traon, F. Ouabdesselam, C. Robach and B. Baudry: From Diagnosis to Diagnosability: Axiomatization, Measurement and Application", The Journal of Systems and Software, 65(1), pp. 31-50, January 2003.

[12] Briand L. C., Labiche Y., Sun H.: Investigating the use of analysis contracts to support fault isolation in object oriented code. International Symposium on Software Testing and Analysis 2002, Roma, Italy.

IX. BIOGRAFÍAS

Rafael Ceballos. Nacido en Sanlúcar de Barrameda, Cádiz.. Ingeniero Informático por la Universidad de Sevilla. Profesor en el departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla. Sus lineas de investigación se centran en la diagnosis basada en modelos (DX), diseño software por contrato, testing, programación con restricciones, y seguridad de sistemas informáticos. Actualmente desarrolla su tesis doctoral en el campo de la diagnosis software.

Fernando de la Rosa. Ingeniero en Informática por la Universidad de Sevilla. Actualmente profesor en el departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla. Sus investigaciones están fundamentalmente orientadas al resumen, el análisis y la visualización de información extraída de Internet (iK-SAV), vigilancia tecnológica e inteligencia competitiva (VTIC), informetría, mapas conceptuales y análisis de redes sociales (ARS).

Sergio Pozo. Ingeniero en Informática e Ingeniero Técnico en Informática de Sistemas por la Universidad de Sevilla. Actualmente es profesor del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla. Su línea de investigación se centra en Seguridad Informática: encontrar formas de detectar y recuperar sistemas ante fallos de seguridad, perfilado de intrusos utilizando técnicas de engaño y control, así como el modelado de su entorno social.