UNIVERSIDAD DE SEVILLA

E.T.S. Ingeniería Informática
Departamento de Lenguajes y Sistemas Informáticos

Técnicas automáticas para la diagnosis de errores en software diseñado por contrato

Memoria de Tesis Doctorial presentada por

Rafael Ceballos Guerrero

Directores:
Dr. D. Rafael Martínez Gasca y Dr. D. Carmelo del Valle Sevillano

. Sevilla, Otoño de 2011

Este trabajo ha sido parcialmente financiado por la Comisión Europea (FEDER) y por el Gobierno de España bajo los Proyectos DPI2003-07146-C02-01, DPI2006-15476-C02-00 y TIN2009-13714, y por la Junta de Andalucía bajo el Proyecto de Excelencia P08-TIC-04095.

Primera publicación, noviembre de 2011 por el Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingeniería Informática. Avda. Reina Mercedes s/n. 41012 Sevilla, España.

http://www.lsi.us.es/
ceball@us.es

A Silvia, a mi hermano, a mis padres, a mis abuelos, y a mi familia, que tanto han respetado las innumerables horas de trabajo y esfuerzo que he tenido que dedicar a desarrollar esta tesis doctoral.

Resumen

Cada vez más la calidad en los productos software es primordial. En el desarrollo del software los costes asociados a la reparación de los errores representan un porcentaje importante de recursos, por tanto, mejorar el proceso de reparación de dichos errores permitiría obtener un ahorro significativo en el coste final del producto software. La reparación de errores no sólo está presente en el desarrollo, sino también en el mantenimiento, si tenemos en cuenta que muchos de los productos software desarrollados sufrirán cambios durante su vida útil para poder adaptarse a los nuevos requisitos que se presenten.

En esta tesis se propone una metodología para la identificación automática de defectos en el software, evitando el gasto de recursos que conllevaría la identificación de los defectos de manera manual. La metodología propuesta, denominada Diagnosis del Software Basada en Modelos de Restricciones, puede ser aplicada en el desarrollo del software y también durante el mantenimiento del software. Se basa en la combinación de diferentes paradigmas, tales como: la Diagnosis Basada en Modelos, la Programación con Restricciones, la generación de pruebas, y el Diseño por Contrato. Tiene por objetivo localizar e identificar de forma automática la causa de los errores de una aplicación software. Permite actuar sobre los asertos que forman el Diseño por Contrato y sobre las sentencias que forman el código fuente.

- Con respecto a los asertos, la metodología permite comprobar si la especificación contiene inconsistencias, y si las contiene, permite identificar qué defectos semánticos han dado lugar a dichas inconsistencias.

- Con respecto a las sentencias, la metodología permite identificar cuáles son los defectos semánticos del código fuente que han provocado que determinados casos de prueba no produzcan los resultados correctos.

Los defectos son semánticos cuando la especificación o el código fuente pueden ejecutarse, pero sin embargo no se producen los resultados correctos. Es decir, los defectos son debidos a un mal diseño de la especificación o del código fuente, y por tanto, los asertos de la especificación o las sentencias del código fuente deben cambiar para poder generar los resultados correctos.

La metodología se basa en la aplicación secuencial de tres pasos:

1. La generación automática del modelo basado en restricciones del sistema software a partir de la especificación o código fuente.

2. Detección automática de inconsistencias a partir del modelo y casos de prueba.

3. Identificación automática de posibles defectos a partir del modelo y casos de prueba.

Las pruebas realizadas en ejemplos reales, demuestran que hay una mejora significativa al aplicar la metodología, pues el número de sentencias o asertos a revisar se reduce de forma apreciable, y en muchos casos, la metodología es capaz de determinar exactamente cuál fue la sentencia o el aserto donde se introdujo el defecto semántico.

Para los casos en los que existen varias causas posibles, la metodología propuesta ofrece como resultado una relación con los posibles defectos siguiendo el principio de la parsimonia. La obtención de los posibles defectos de forma automática reduce el tiempo de desarrollo y mantenimiento, pues la solución a los errores encontrados se puede acometer antes.

Agradecimientos

Desarrollar una tesis es un trabajo largo en el que el investigador se desarrolla personal y profesionalmente. Cuando la tesis finaliza, surge la oportunidad de agradecer a quiénes han sido las personas que han jugado un papel importante en este camino.

En esta tesis han colaborado muchas personas, pero sin duda mis directores, los doctores Rafael Martínez Gasca y Carmelo del Valle Sevillano, son quienes más han marcado este trabajo con su ayuda y comprensión. En especial, me gustaría resaltar su siempre buena actitud personal y profesional en los momentos difíciles para un doctorando, ya que es en esos momentos cuando más se le exige a un director de tesis. Me gustaría agradecer las ideas y el tiempo dedicado por Miguel Toro para con este trabajo. Y también quisiera agradecer los consejos que Rafael M. Gasca, y en especial su esposa Carmen, me han dado en los trabajos publicados en inglés.

El trabajo en un grupo de investigación me ha proporcionado muchas ideas y experiencias enriquecedoras. Por eso me gustaría agradecer poder formar parte de este gran proyecto a Irene Barba, Diana Borrego, Pablo Neira, Luisa Parody, Ángel J. Varela, Andrés Jimenez, y en especial a Mayte Gomez, Sergio Pozo, y Fernando de la Rosa, ya que empezamos juntos en esta andadura, y se han convertido en amigos además de compañeros.

Respecto a los miembros del departamento me gustaría resaltar a compañeros como David Benavides, Juan Álvarez, Fernando de Salamanca, y en especial a Toñi Reina y Octavio Martín que siempre me han prestado su apoyo en todo momento, y me han guiado cuando más lo he necesitado.

Por último, y no menos importante, me gustaría agradecer a mis amigos y familia, y en especial a Silvia, a mis padres y a mi hermano, que tanta paciencia han tenido conmigo. Gracias por vuestro apoyo.

Índice general

I Prefacio **1**

1. Introducción **3**
1.1. Contexto . . . 3
1.1.1. Defecto, fallo y error . . . 5
1.2. Motivación . . . 6
1.3. Metodología propuesta . . . 8
1.3.1. Diagnosis de la especificación . . . 9
1.3.2. Diagnosis del código fuente . . . 11
1.4. Marco de la tesis y contribuciones . . . 13
1.4.1. Proyectos de investigación . . . 13
1.4.2. Contribuciones . . . 14
1.5. Estructura de la tesis . . . 18

II Antecedentes **21**

2. Técnicas de detección de errores y aislamiento de defectos en el software **23**
2.1. Introducción . . . 23
2.2. Técnicas basadas en la especificación . . . 24
2.2.1. Métodos formales . . . 25
2.2.2. Diseño por Contrato . . . 27
2.2.3. Generación automática de la especificación . . . 31
2.2.4. Verificación de la conformidad . . . 34
2.3. Técnicas basadas en el análisis estático . . . 37
2.3.1. Análisis de los grafos de control de flujo . . . 37
2.3.2. Análisis de las dependencias en el código fuente . . . 37
2.3.3. Slicing . . . 39
2.4. Técnicas basadas en el análisis dinámico . . . 41
2.4.1. Pruebas Unitarias . . . 42
2.4.2. Desarrollo guiado por pruebas . . . 43
2.4.3. Generación de pruebas . . . 44

2.4.4. Medidas de la calidad de las pruebas 46
2.5. Depuración del software 48
2.5.1. Depuración interactiva asistida 49
2.5.2. Depuración automática 50

3. Técnicas adaptadas en la propuesta 53
3.1. Diagnosis Basada en Modelos 53
3.1.1. Definiciones y conceptos de la metodología DX 55
3.1.2. Definiciones y conceptos de la metodología FDI 57
3.2. Problemas de satisfacción de restricciones 61
3.2.1. Conceptos básicos del modelado con restricciones 61
3.2.2. Consistencia 63
3.2.3. Algoritmos de búsqueda 66
3.2.4. Problemas sobrerrestringidos 68

III Propuesta 71

4. Metodología de diagnosis 73
4.1. Adaptaciones necesarias en la metodología DX para el diagnóstico del software 74
4.2. Metodología propuesta 75
4.3. Detección de errores 78
4.4. Localización de los defectos 79
4.4.1. Generación del problema de diagnosis 79
4.4.2. Diagnosis de defectos semánticos en el software 83
4.4.3. Diagnosis mínima 85
4.4.4. Utilización de varios casos de prueba 86
4.5. Conclusiones 87

5. Diagnóstico de la especificación 89
5.1. Introducción 89
5.2. Adaptación de la metodología de diagnosis 91
5.2.1. Detección de errores 93
5.2.2. Localización de los defectos 95
5.3. Diagnóstico de defectos en invariantes 97
5.3.1. Invariantes propios 98
5.3.2. Invariantes propios y heredados 99
5.4. Diagnóstico de la especificación de los métodos 101
5.4.1. Reglas para el refinamiento de la especificación de los métodos . 102
5.4.2. Precondición propia y heredada de cada método 103
5.4.3. Postcondición propia y heredada de cada método 106
5.4.4. Precondición, postcondición e invariantes de cada método 108

5.4.5. Diagnóstico utilizando casos de prueba concretos 112
5.5. Conclusiones 115

6. Diagnóstico del código fuente 117
6.1. Introducción 117
6.2. Adaptación de la metodología de diagnosis 118
6.3. Detección de errores 120
6.3.1. Características del código fuente a diagnosticar 120
6.3.2. Resultados de las pruebas 123
6.3.3. Sentencias y asertos ejecutados 126
6.3.4. Traza de Unidades Ejecutadas 127
6.3.5. Identificadores para los elementos del código fuente y asertos . . 128
6.3.6. Tipos de Unidades Ejecutadas 130
6.4. Localización de los defectos 140
6.4.1. Traza de Componentes Desplegados 140
6.4.2. Tipos de defectos 143
6.4.3. Funciones de transformación 146
6.4.4. Transformación de asignaciones, asertos y bloques de sentencias 147
6.4.5. Transformación de sentencias selectivas 157
6.4.6. Transformación de llamadas a métodos y constructores 168
6.4.7. Problema de diagnosis para defectos en el código fuente 177
6.5. Mejora de los resultados utilizando asertos en el código fuente 181
6.5.1. Uso del invariante de los bucles 183
6.6. Conclusiones 185

7. Experimentos realizados y mejora de la precisión en la identificación de los defectos 187
7.1. Pruebas sobre la metodología aplicada al código fuente 187
7.1.1. Implementación 188
7.1.2. Medidas y consideraciones previas 189
7.1.3. Defectos en bucles 191
7.1.4. Variables reales 193
7.1.5. Defectos en las expresiones enviadas a métodos 194
7.1.6. Defectos en las sentencias selectivas 195
7.1.7. Defectos multiples 195
7.1.8. Otros ejemplos 196
7.1.9. Conclusiones sobre los experimentos 197
7.2. Ampliaciones para mejorar el proceso de identificación de defectos . . . 199
7.2.1. Utilización de varios casos de prueba 200
7.2.2. Propagación de los fallos 204
7.3. Conclusiones 213

8. Conclusiones y trabajo futuro **215**
8.1. Contribuciones de esta tesis 215
8.2. Líneas de trabajo futuro 219

A. Algoritmos auxiliares **221**
A.1. Transformación de una TCD a un grafo de dependencias 221
A.1.1. Cálculo de las entradas y salidas de cada CD 222
A.1.2. Obtención del grafo de dependencias a partir de los CD 224
A.2. Algoritmo para la determinación de los cluster 230

B. Ejemplos utilizados en las pruebas **235**
B.1. ToyProgram 235
B.2. IndexOf 235
B.3. Raiz cuadrada 236
B.4. Polinomio 237
B.5. CuentaBancaria 237
B.6. SumDiferencia 240
B.7. Problema de la mochila 241
B.8. BusqBinaria 242
B.9. NTree 243
B.10. Problema de las monedas 245
B.11. SumaSubSecMaxima 247

Bibliografía **249**

Índice de figuras

1.1. Paradigmas que engloba la Diagnosis del Software Basada en Modelos . 4
1.2. Ejemplo para la definición de *defecto*, *fallo* y *error* 6
1.3. Especificación de la clase clase *Account* 10
1.4. Identificación de defectos en el código fuente 12

3.1. Sistema polybox básico (Toy Problem) 54
3.2. Modelo CSP para el coloreado de un mapa 63

4.1. Principales diferencias entre la Metodología DX y la Diagnosis del Software Basada en Modelos 75
4.2. Aplicaciones de la Diagnosis del Software Basada en Modelos 76
4.3. Fases de la metodología propuesta 77
4.4. Transformación de la clase Account 82
4.5. Problema de Diagnosis 86

5.1. Gramática permitida para los asertos 90
5.2. Ejemplo de defecto en la especificación 94
5.3. Problema de diagnosis para el caso de prueba vacío (TC = $\emptyset$) 97
5.4. Ejemplo de herencia de invariantes 99
5.5. Ejemplo de herencia entre clases 103
5.6. Herencia de la precondición. Casos posibles. 104
5.7. Ejemplo de herencia de precondición y postcondición 105
5.8. Herencia de la postcondición. Casos posibles. 106
5.9. Clase Cuenta Bancaria (Account) 109

6.1. Fases de la metodología de diagnóstico aplicada al código fuente 118
6.2. Sintaxis del lenguaje soportado por la metodología propuesta 120
6.3. Detección de errores usando casos de prueba 125
6.4. Tipos de UE que pueden aparecer en una traza 130
6.5. Orden de descomposición de las unidades diagnosticables de un programa 131
6.6. Representación visual de las UE 133
6.7. UE_Constructor obtenida de la ejecución del constructor de la clase IntBinOper 134

6.8. UE implicadas en la ejecución del método *dif_abs* de la clase *IntBinOper* 136
6.9. Transformación de las declaraciones locales de variables 147
6.10. Transformación de sentencias de asignación 148
6.11. Transformación de asignaciones 153
6.12. Transformación de los asertos 155
6.13. Transformación de bloques de sentencias 156
6.14. Sentencia selectiva con defecto en la condición 157
6.15. TUE original y TUE ampliada 161
6.16. Transformación de sentencias selectivas 164
6.17. TUE Ampliada 167
6.18. Transformación de las llamadas a contructores 169
6.19. UE obtenida de la asignación del resultado de un constructor a una variable 171
6.20. Transformación de las llamadas a métodos 173
6.21. Transformación de la sentencia de retorno 174
6.22. UE correspondiente a la asignación de la llamada al método *dif_abs* . . 175
6.23. Resultado de la transformación a CD de la UE mostrada en la figura 6.22 176
6.24. Problema de diagnosis para el método *dif_abs* 178
6.25. Traza y domino de las variables para la diagnosis mínima S_1 179
6.26. Traza y domino de las variables para la diagnosis mínima S_6 180
6.27. Problema de diagnosis 182
6.28. Resolución del problema Max-CSP 183
6.29. Algoritmo cálculo raíz cuadrada de un número entero 184

7.1. Relación entre el número de sentencias implicadas en las diagnosis mínimas (Dmin) y el número de sentencias del programa (STS) 198
7.2. Porcentaje de sentencias del código fuente incluidas en las sentencias implicadas en las diagnosis mínimas obtenidas 198
7.3. Nº de sentencias implicadas en las diagnosis mínimas (Dmin) en comparación con el nº de sentencias del programa (STS) y el nº de sentencias que forman la traza (LCT) 199
7.4. Obtención de la diagnosis mínima común a varios casos de prueba . . . 203
7.5. Propagación de los fallos en el código fuente 204
7.6. Algoritmo para convertir un grafo de componentes en un grafo de nodos de componentes 207
7.7. Nodos de CD obtenidos para el ejemplo propuesto 208
7.8. Ejemplo de resolución de un problema Max-CSP 211

8.1. Comparación de la metodología propuesta frente a otras técnicas 218

A.1. Grafo de dependencias obtenido para el ejemplo ToyProgram 222
A.2. Algoritmo para la conversion de un conjunto de CD a un grafo de dependencias 224
A.3. Método addVertex y addVertexs 225

A.4. Métodos addInputs y addOutputs 226
A.5. Método addInputsAux y addInputsSet 227
A.6. Método addOutputsAux y addOutputsSet 228
A.7. Método addEdges 229
A.8. Grafo de dependencias obtenido para la llamada al método dif_abs . . . 230
A.9. Algoritmo CD2Cluster encargado de enlazar cada componente con el conjunto de salidas del sistema software sobre las que influye 231
A.10. Clusters de CD obtenidos para el ejemplo propuesto 232

Índice de Tablas

3.1. Modelo DX para el ejemplo de la figura 3.1 56
3.2. Modelo FDI para el ejemplo de la figura 3.1 58
3.3. Matriz de firmas del sistema polybox para fallos simples 59
3.4. Matriz de firmas del sistema polybox para fallos múltiples 60

4.1. Tipos básicos del entorno de programación con restricciones 81

6.1. Ejemplo de generación de una TUE 128
6.2. Ejemplo de generación de una TUE para una sentencia iterativa 138
6.3. Relación entre las UE y los CD 142

7.1. Resultados obtenidos para el ejemplo ToyProgram 189
7.2. Resultados obtenidos para el ejemplo IndexOf 191
7.3. Resultados obtenidos para el ejemplo RaizCuadrada 192
7.4. Resultados obtenidos para el ejemplo Polinomio 193
7.5. Resultados obtenidos para el ejemplo CuentaBancaria 194
7.6. Resultados obtenidos para otros ejemplos 195
7.7. Resultados obtenidos para otros ejemplos 196
7.8. Resultados obtenidos para los casos de prueba propuestos 197
7.9. Resultados obtenidos para los casos de prueba propuestos 202
7.10. Resultados obtenidos para los casos de prueba propuestos 209
7.11. Resultados obtenidos para los casos de prueba propuestos 212

A.1. Entradas y salidas para cada una de las UE de un sistema software . . 223
A.2. Relación de las entradas y salidas entre las UE y los CD 223

Parte I

Prefacio

Capítulo 1

Introducción

En este primer capítulo se introducirán los objetivos perseguidos en esta tesis. Se expondrán por primera vez los conceptos en los que se sustenta el trabajo realizado, se mostrará la motivación y el contexto donde la tesis se ha desarrollado, y por último, se presentará una visión general de los contenidos de cada uno de los capítulos en los que se divide esta tesis.

1.1. Contexto

En los sistemas técnicos modernos basan su funcionamiento en la combinación de elementos físicos (hardware) y lógicos (software). El software permite una mayor flexibilidad y eficiencia en los sistemas. Por este motivo, cada vez más la calidad en los productos software es primordial. Los fallos en los sistemas pueden deberse a problemas en elementos físicos o a problemas en los elementos de tipo software.

Alcanzar un nivel aceptable de calidad en los productos software resulta cada vez más difícil, ya que existen varios factores que juegan en contra, como son:

- Incremento de la complejidad. La extensión y la complejidad de los productos software ha crecido exponencialmente en los últimos años, y todo apunta a que la tendencia es que continúe así.
- Aumento del número de personas implicadas. Debido a que generalmente los proyectos son desarrollados por varias entidades, o por varios grupos de desarrollo.
- Disminución del ciclo de vida de los productos. Cada vez se reutilizan más paquetes de código. Además las empresas requieren que los productos estén en el mercado antes que la competencia.
- Sistemas abiertos. Los productos software están cada vez más abiertos al mundo exterior y pueden surgir nuevos problemas de fiabilidad, seguridad, etc. Entre otros motivos, debido a que los proyectos son desarrollados usando código de terceras partes para añadir funcionalidades ya desarrolladas.

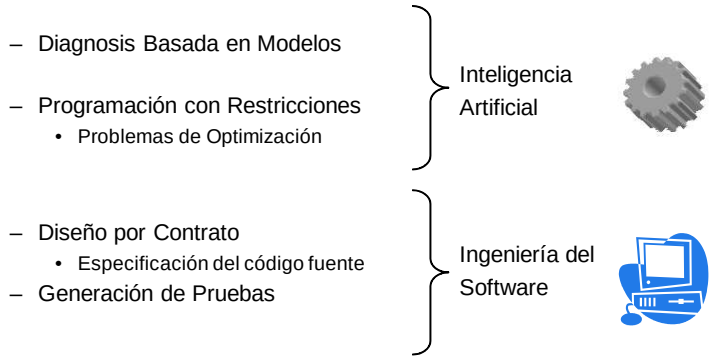


Figura 1.1: Paradigmas que engloba la Diagnosis del Software Basada en Modelos

Teniendo en cuenta que muchos de los productos desarrollados sufrirán cambios durante su vida útil, y tendrán que ser adaptados a nuevos requisitos, cada vez es más necesario que existan métodos rápidos y automáticos para diagnosticar defectos en el software y cumplir con la calidad deseada de dichos productos.

Para los elementos físicos que forman un sistema, parte de la comunidad científica ha dedicado sus esfuerzos al desarrollo de una metodología que permita su diagnóstico de forma automática. Una de estas metodologías se denomina Diagnosis Basada en Modelos (MBD, Model Based Diagnosis). Dicha metodología parte de un modelo explícito del sistema a diagnosticar, y a partir de él, y de los valores monitorizados a través de sensores, es capaz de identificar los elementos que causan los fallos.

Para los elementos de tipo software, durante los últimos años, la investigación en técnicas para generar y realizar pruebas sobre el software ha aumentado sustancialmente. El objetivo de este aumento de la inversión en investigación sobre las pruebas, es lograr metodologías que permitan determinar cuanto antes si existen defectos en el software desarrollado, y evitar que estos defectos se propaguen en el ciclo de vida.

Pero aunque los errores sean detectados, para su reparación normalmente es necesario parar el sistema, depurar el software, realizar los cambios necesarios, y volverlo a poner en funcionamiento. La depuración del software no es automática, y se debe realizar cada vez que se detecta un error. Por tanto, en el desarrollo del software, son necesarias metodologías que permitan identificar de forma automática cuál es el origen de los errores detectados por las pruebas, y que permitan realizar la depuración de manera más eficiente.

Las pruebas y depuración del software representan el 30 % de los recursos de un desarrollo de aplicaciones. Por tanto, es ya uno de los procesos más costosos en el desarrollo y mantenimiento del software. Un estudio del NIST [72] del año 2002, estima que el coste de las pruebas y depuración del software es cercano a los 60 mil millones de dólares anuales sólo en EEUU.

Un estudio del 2008 del IDC [5] realizado a 139 empresas americanas dedicadas

a la producción de software, estima que el coste de reparar los defectos del software producido puede suponer para las empresas estudiadas, de los 5 millones de dólares hasta los 22 millones de dólares anualmente, dependiendo del tamaño del proyecto. Además, 7 de cada 10 compañías estiman que la complejidad del código desarrollado ha incrementado su complejidad en los últimos 2 años. Otro dato interesante es que el 72 por ciento de las compañías sostienen que la depuración del software puede llegar a ser un verdadero problema.

El objetivo de esta tesis es diseñar una metodología que permita localizar e identificar de forma automática la causa de los errores de una aplicación software, en otras palabras, que permita diagnosticar el software. Esta metodología la llamaremos Diagnosis del Software Basada en Modelos, y se abordará, tal como aparece en la figura 1.1, a través de una combinación de diferentes paradigmas, tales como: la Diagnosis Basada en Modelos, la Programación con Restricciones, la generación de pruebas, y el Diseño por Contrato.

La Diagnosis del Software Basada en Modelos (en adelante Diagnosis del Software) permite a una organización dedicada a la producción del software, realizar la localización de los defectos de forma automática, evitando el gasto en tiempo y recursos que conllevaría la realización de la misma tarea de manera manual. Aunque en un principio la organización tendría que realizar una inversión inicial para implantar la metodología, esta inversión se vería recompensada por los recursos ahorrados en la localización de defectos que podrían ser dedicados a trabajo productivo. Lo que permitiría el retorno de la inversión en un corto plazo, teniendo en cuenta que en los estudios reseñados anteriormente [72][5], la depuración del software aparece como un problema cuya resolución encarece excesivamente el proceso de producción del software.

Antes de seguir, es conveniente remarcar las diferencias entre tres términos que se usarán con frecuencia en este trabajo. Estos tres términos son *defecto*, *fallo* y *error*.

1.1.1. Defecto, fallo y error

Estos términos se definen detalladamente en el estándar IEEE 610.12-1990 [84], Standard Glossary of Software Engineering Terminology.

- Un *defecto* (fault) en un programa es un paso, proceso, o definición de datos que es incorrecto e impide cumplir el resultado esperado. El termino 'bug' también se utilizará en este trabajo para hacer referencia a los defectos en el código fuente.

- Un *fallo* (failure) es el resultado incorrecto, la consecuencia o el efecto no deseado que produce un defecto. El fallo es debido a la incapacidad del sistema de lograr cumplir los requerimientos especificados. Los fallos en el software son sistemáticos y no aleatorios, ya que se deben a defectos en el diseño o en la implementación, no a fallos esporádicos en el sistema, y por tanto para solucionarlos debe cambiar el diseño o la implementación del software.

- Un *error* (error) es la discrepancia entre el resultado obtenido (real) y el resultado pretendido o teóricamente correcto (especificado). Los errores son los síntomas de que el comportamiento esperado no se ha cumplido en la ejecución del programa.

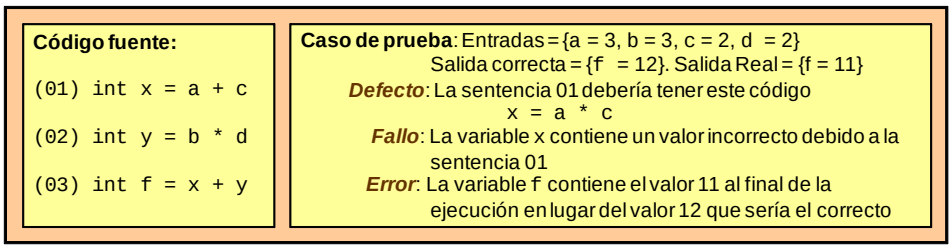


Figura 1.2: Ejemplo para la definición de *defecto*, *fallo* y *error*

Ejemplo 1.1. En la figura 1.2 se muestra un ejemplo en el que se ha introducido un defecto en la línea 1, en lugar de una multiplicación se ha escrito una suma. Como consecuencia de este defecto se producirá un fallo en el valor asignado a la variable x. De esta variable no se conoce cuál debe ser el valor correcto a asignar, por tanto, no es posible determinar si ese valor es un error. La propagación del fallo en la línea 1 provoca el error detectado en la salida f. El error es detectado al observar que la salida f no corresponde con la especificada como correcta. Si el encargado de probar este código conociese el valor correcto de la variable x, entonces el resultado de dicha variable sería a la vez fallo y error. Los defectos que provocan fallos en el software, son detectables gracias a los errores.

1.2. Motivación

En el ciclo de vida del software los efectos de cada una de las etapas son trasladados a las siguientes etapas. Cuando se detectan errores, es necesario volver atrás hasta identificar los defectos que dieron lugar a los errores, encontrar una solución y repasar todo aquello que tenía relación con los defectos. Como norma general, cuanto más se tarda en detectar los errores de un proyecto software, más costosa resulta su solución. Los defectos no corregidos se propagan en el ciclo de vida, y pueden condicionar el resto del desarrollo.

Las pruebas son tareas tradicionalmente realizadas por los desarrolladores, y no existen herramientas que permitan automatizar por completo este proceso. El uso de las pruebas permite detectar errores, pero no determinan cuál es el defecto que los origina. Suele ser el propio desarrollador el que debe deducir, conforme a la información disponible, cuál es la causa del error y los cambios necesarios en el software para alcanzar el resultado esperado. Además, la realización de pruebas permite detectar si existe o

no un error en una aplicación, pero no garantiza si un programa es correcto al cien por cien.

La implementación y generación del código fuente está lejos de ser un proceso estandarizado y automático. Por tanto no hay seguridad sobre la corrección del código generado. Una de las principales causas de esta falta de automatización está en no diferenciar entre la especificación del software y la implementación, es decir, no diferenciar entre el objetivo y la forma de conseguir el objetivo establecido. De esta forma, se podría dividir el esfuerzo en el desarrollo y mantenimiento del software, ya que sería factible verificar los resultados de ambos por separado, y evitar la propagación de defectos entre la especificación y la implementación. Esta es la idea en la que se fundamenta el Diseño por Contrato.

En el desarrollo de las aplicaciones software, el Diseño por Contrato permite especificar, a través de asertos, el comportamiento correcto del software. El código fuente de la aplicación a desarrollar está obligado a cumplir lo establecido en los asertos del Diseño por Contrato. Los contratos permiten además, automatizar el proceso de generación de pruebas, ya que pueden ser utilizados como oráculos para la generación automática de casos de prueba.

El Diseño por Contrato mejora el desarrollo del software y ofrece claras ventajas, pero aun así, no es posible garantizar en todos los casos que el producto software desarrollado de esta forma esté libre de defectos, por las siguientes razones:

- La especificación no es generada de forma automática, son los desarrolladores quienes establecen la especificación. Por tanto, es posible que introduzcan involuntariamente algún defecto, por ejemplo, por descuidos, por falta de entendimiento con el cliente, o por falta de coordinación entre los desarrolladores. A su vez, los defectos introducidos en los contratos se propagarán al código fuente, teniendo en cuenta que el código fuente debe cumplir lo establecido en los contratos.

- Normalmente el diseñador tiene flexibilidad a la hora de realizar la especificación. Queda en sus manos la completitud o el grado de dureza de la especificación. De esta forma es posible realizar una especificación muy restrictiva y fuerte en aquellas partes que así lo requieran, o débil y poco restrictiva si el desarrollo requiere flexibilidad. Lamentablemente, aunque la especificación no tenga defectos, si es poco restrictiva, el código fuente podría contener defectos aun cumpliendo lo establecido en la especificación. Es decir, si el desarrollador no diseña el contrato de forma completa, puede dejar escapar casos en los que el código fuente satisfaga el contrato pero no se comporte como debiera.

La especificación, por tanto, puede tener defectos introducidos por los ingenieros de requisitos, o, aunque no contenga defectos, puede no ser completa. Si la especificación contiene defectos, es posible que dichos defectos sean transmitidos al código fuente. Además de los defectos inducidos por la especificación, los desarrolladores pueden introducir más defectos al diseñar el código fuente.

Los defectos que se tratarán en esta tesis son de tipo semántico, no sintácticos. La especificación y el código fuente contienen defectos sintácticos cuando no es posible su ejecución debido a que el entorno no es capaz de interpretar los asertos o las sentencias que se deben ejecutar. Generalmente este tipo de defectos son resueltos en tiempo de compilación, ya que simplemente contradicen la sintaxis del lenguaje.

Los defectos son semánticos cuando la especificación o el código fuente pueden ejecutarse, pero sin embargo no produce los resultados establecidos como correctos. Es decir, los defectos son debidos a un mal diseño de la especificación o del código fuente, y por tanto, la especificación o el código fuente deben cambiar para poder generar el resultado especificado como correcto. Por ejemplo, pueden contener defectos semánticos los asertos que forman la especificación, las expresiones asignadas a variables o enviadas como parámetros, o las guardas de las sentencias selectivas. En esta tesis no se tendrán en cuenta defectos que provoquen errores dinámicos, tales como excepciones, violaciones de acceso a memoria, etc.

Los objetivos de la metodología propuesta en esta tesis son dos:

- En primer lugar para la especificación, comprobar si contiene inconsistencias, y si las hay, localizar los defectos semánticos que dan lugar a dichas inconsistencias.
- En segundo lugar, suponiendo correcta la especificación, para aquellos casos de prueba que aplicados sobre el código fuente no producen las salidas especificadas como correctas, localizar cuáles son los defectos semánticos que impiden obtener los resultados correctos.

En esta tesis se supondrá que la especificación se presentará a través de precondiciones, postcondiciones, invariantes, y otros tipos de asertos incluidos en el código fuente, escritos en lógica de primer orden. Para el caso del código fuente, éste debe estar escrito en un lenguaje de programación imperativo y orientado a objetos, y en el que sólo se permita un hilo de ejecución. Para poder aplicar la metodología se debe cumplir que los asertos que forman la especificación están bien formados, y que el código fuente no contiene errores sintácticos. Además, la evaluación de los asertos y la ejecución del código fuente deben poder realizarse en un tiempo finito.

1.3. Metodología propuesta

En la metodología propuesta en esta tesis, se supondrá que la especificación y código fuente que va a ser diagnosticado son casi correctos. Es decir, se supondrá que en la especificación y en el código fuente se han tenido en cuenta todos los requisitos que se pretenden alcanzar con el producto software, y que por tanto, el número de defectos debe ser pequeño pero sin embargo suficiente para que se produzcan errores en la ejecución.

Cada una de las etapas en las que se divide el desarrollo de un producto software genera un tipo diferente de resultados. Para poder verificar los resultados obtenidos en

cada una de las etapas, es necesario adaptarse a las características de estos resultados. Así por ejemplo, las pruebas y verificaciones que se pueden realizar sobre un diagrama de clases, correspondiente a las primeras etapas del desarrollo, no tienen mucho que ver con las pruebas que se pueden realizar sobre el código fuente de un componente software que se supone es el producto final del desarrollo.

De la misma manera que las técnicas para generar pruebas deben adaptarse al tipo de producto que se da en cada una de las etapas del desarrollo del software, también las técnicas de diagnóstico que las acompañen deben adaptarse al tipo de producto generado. Por este motivo, aunque en esta tesis se propone una metodología general para el diagnóstico, dicha metodología tendrá que ser adaptada para ser aplicada, primero para la localización de defectos en la especificación, y luego para la localización de defectos en el código fuente, correspondiente a la implementación del producto software.

A continuación se muestran las líneas generales de la aplicación de la metodología a la especificación y al código fuente. La metodología propuesta permite localizar defectos en el diseño de la especificación o en el código fuente, y propone de forma automática los elementos a eliminar o sustituir de la especificación o del código fuente.

1.3.1. Diagnosis de la especificación

Propuesta

La especificación establece condiciones que debe cumplir el código fuente. Cuando en esta tesis se hace referencia a la especificación, se supondrá que dicha especificación es automatizable, en el sentido de que pueda ser directamente procesada por un sistema informático. Por ejemplo, si se tiene un código fuente con comentarios en lenguaje natural, dichos comentarios sólo tienen valor para el procesamiento por un humano, y no tendrían utilidad para una automatización de los diferentes procesos que se realicen sobre dicho código. Sin embargo, si dichos comentarios están por ejemplo expresados en lógica de primer orden, y contienen las relaciones entre las entradas y salidas de dicho código a modo de precondiciones y postcondiciones, esta información sí podría ser procesada de forma automática para servir de oráculo en las pruebas o el diagnóstico.

El primer objetivo de esta tesis es proponer una metodología que permita comprobar la consistencia de los asertos que forman la especificación, e identificar qué asertos incluyen defectos en el caso de detectar inconsistencias. Los defectos son de tipo semántico, y residen en el diseño de los asertos de la especificación. Para solucionarlos será necesario modificar los asertos o sustituirlos por otros.

Para comprobar la consistencia de la especificación, en esta tesis se propondrán diferentes pasos, que de forma secuencial permitirán validar los asertos que forman la especificación. Con estas comprobaciones se podrá determinar si existen inconsistencias entre los elementos que forman la especificación, y se podrán identificar los defectos que provocan dichas inconsistencias antes de comenzar la implementación del código fuente. Como consecuencia, se evitará transmitir dichos defectos al código fuente.

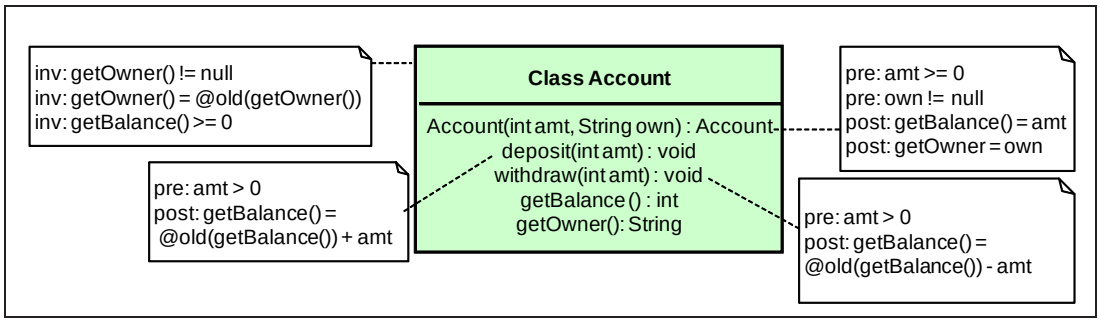


Figura 1.3: Especificación de la clase clase *Account*

Para realizar las comprobaciones de forma automática, la metodología propuesta utilizará un modelo basado en restricciones, que fácilmente puede ser resuelto con las técnicas de Programación con Restricciones. La metodología propuesta permite identificar qué asertos deberían cambiar para que la especificación sea correcta, pero es el diseñador de la especificación quien debe decidir los cambios. Es decir, la metodología propone dónde realizar los cambios, pero no cómo deben realizarse.

Por ejemplo, una de las verificaciones que se propondrán será comprobar si los asertos y la especificación de un método (precondición y postcondición) se pueden satisfacer en conjunto. A continuación se muestra un caso concreto de ejemplo.

Ejemplo 1.2. La figura 1.3 muestra los asertos correspondientes a la especificación de la clase *Account*. Esta clase simula una cuenta bancaria y las operaciones asociadas a la misma (depositar y retirar dinero por ejemplo). Se dispone del siguiente caso de prueba para la comprobación del código fuente del método *withdraw* (permite retirar dinero de la cuenta bancaria):

Entradas: `getBalance() = 0, amt = 100`
Salidas: `getBalance() = 0`

El caso de prueba establece que si el saldo es cero y se retiran 100 unidades de la cuenta, el saldo después de la ejecución del método debe ser 0, y que no debe lanzarse ninguna excepción, ya sea debido al código fuente, o por el incumplimiento de algún aserto (en este caso, precondición y postcondición). El método *getBalance* es un método observador que devuelve el valor del saldo de la cuenta bancaria.

Dado el caso de prueba mostrado anteriormente, y suponiendo que los invariantes deben cumplirse siempre antes y después de ejecutar cualquier método de una clase, la idea es comprobar si la secuencia formada por {invariantes antes del método + precondición + postcondición + invariantes después del método} se puede satisfacer para el método *withdraw*. Para realizar dicha comprobación, se utilizará un modelo basado en restricciones.

La metodología propuesta comprobará de forma automática para el caso de prueba

propuesto, que es imposible que todos los asertos que forman la secuencia se cumplan a la vez. La postcondición obliga a que el saldo al final del método deba ser igual al saldo inicial menos la cantidad retirada, es decir, una cantidad negativa. Por otra parte el invariante, tras la ejecución del método, obliga a que el saldo sea mayor o igual que cero, por tanto, no es posible satisfacer a la vez la postcondición y el invariante.

La propia metodología determinaría qué asertos de la secuencia deben cambiar para que la especificación sea correcta, suponiendo que el caso de prueba es correcto. Es decir, determinaría qué parte de la especificación contiene defectos. En este ejemplo, los defectos pueden estar en la postcondición del método *withdraw* o en el invariante de la clase.

Ventajas

Al aplicar la metodología propuesta se obtienen una serie de ventajas, que de forma sintetizada pueden resumirse en las dos siguientes:

- Se consigue automatizar la comprobación y localización de defectos en la especificación. La metodología propuesta sirve de ayuda al diseñador de la especificación, y ofrece de forma automática los defectos que explican las inconsistencias entre los asertos que forman la especificación.

- Se evita la propagación al código fuente de los defectos contenidos en la especificación. El código fuente debe cumplir la especificación, por tanto si la especificación tiene defectos, éstos se propagarían al código fuente. Detectar estos defectos en el diseño de la especificación, y antes de implementar el código fuente, disminuye el coste en recursos necesarios para el desarrollo del código fuente.

1.3.2. Diagnosis del código fuente

Propuesta

Los casos de prueba establecen, para unas entradas determinadas, cuáles son los resultados correctos que deben ser generados por el código fuente. Los errores son detectados al comprobar que los resultados obtenidos no corresponden con los establecidos en los casos de prueba. Las pruebas permiten detectar errores, pero no identifican el defecto o defectos que originan dichos errores. El segundo objetivo de la metodología propuesta en esta tesis es dar un paso más, e identificar defectos del código fuente a través de un proceso de diagnóstico. La idea es poner en funcionamiento el proceso de diagnóstico en el caso de no coincidir los resultados esperados con los generados en la ejecución del código fuente.

Para identificar los defectos se utilizará un modelo basado en restricciones, y obtenido de la transformación de la especificación y del código fuente. Se supondrá que la especificación es correcta, y que los defectos deben estar en las sentencias del código fuente. La información contenida en los casos de prueba servirá para determinar cuál es

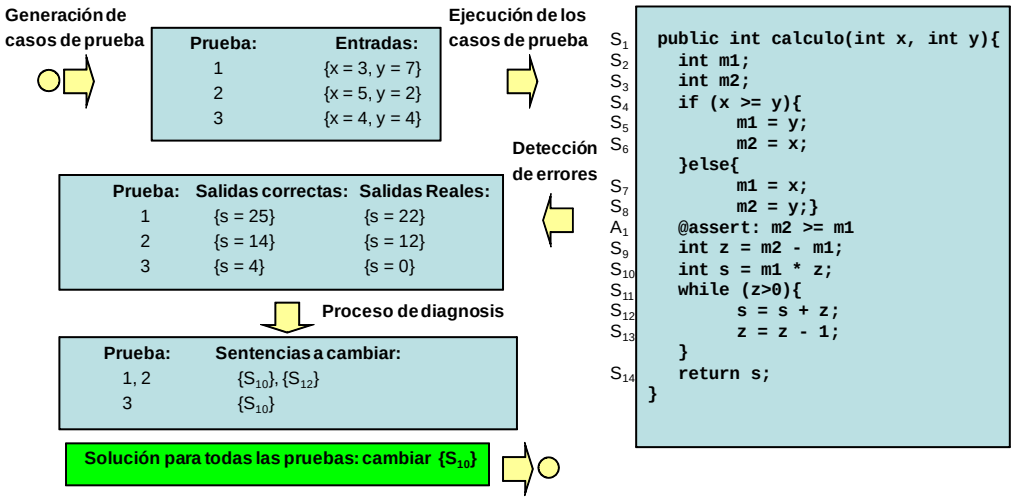


Figura 1.4: Identificación de defectos en el código fuente

el comportamiento correcto que se espera del código fuente. El problema de diagnosis se resolverá como un problema de satisfacción de restricciones, y de la forma más eficiente posible.

La metodología propuesta en esta tesis permite identificar qué partes del código fuente deben cambiar, pero no proporciona el código que debería reemplazar al original. La metodología propone dónde realizar los cambios, pero no cómo deben realizarse. A continuación se muestra un ejemplo de un código fuente y el resultado que la metodología propuesta en esta tesis obtendría.

Ejemplo 1.3. En la figura 1.4 se muestra el código fuente corresponde a un método escrito en el lenguaje Java^{TM}. Dicho método recibe dos parámetros de entrada y calcula un valor de salida en función de las entradas. Para comprobar la validez de dicho método se tienen tres casos de prueba. Cada caso de prueba proporciona las dos entradas requeridas y el valor correcto de salida. En ninguno de los tres casos se consigue el resultado correcto tras ejecutar el código fuente.

En la figura 1.4 se muestran cuáles serían las sentencias que la metodología propondría cambiar para que en cada caso de prueba se puedan alcanzar los resultados especificados como correctos. Por ejemplo para el caso de prueba primero, se propone modificar una única sentencia, que podría ser la sentencia S_{10} o la sentencia S_{12}. La metodología proporciona el mismo resultado para el segundo caso de prueba, y para el tercero propone una sentencia: S_{10}. La metodología también proporcionaría otras soluciones, pero todas obligarían a modificar más de una sentencia a la vez del método para cada caso de prueba. Por simplificar el ejemplo, no se han mostrado en la figura.

Como el objetivo final es poder solucionar todos los casos de prueba, la metodología propondría como solución para los tres casos de prueba a la vez, modificar la sentencia S_{10}. Es decir, cambiando dicha sentencia sería posible resolver el defecto de diseño que

tiene el método mostrado, y se podría obtener el resultado correcto en los tres casos de prueba.

Es importante reseñar que la metodología garantiza que si se cambia una única sentencia, y dicha sentencia no es S_{10}, sería imposible alcanzar el resultado correcto al mismo tiempo en los tres casos de prueba. La única solución que implique únicamente cambiar una sentencia, es la propuesta. Aunque puede haber otras combinaciones de dos sentencias que al ser modificadas a la vez permitan también generar el valor correcto de salida en los tres casos de prueba.

En este ejemplo, el error detectado en los tres casos de prueba se solucionaría colocando la sentencia *s = m1 * (z + 1)* en lugar de la sentencia S_{11}.

Ventajas

La aplicación de la metodología propuesta ofrece una serie de ventajas, que se pueden resumir en las siguientes:

- Es la propia metodología la que establece de forma automática qué es relevante para solucionar el error detectado, ya que de todo el código fuente, sólo es necesario revisar aquellas partes donde el proceso de diagnóstico determine que puede haber defectos.
- Reduce el tiempo de reparación de los defectos, ya que reduce la cantidad de información que el desarrollador debe revisar cuando se detecta un error.
- También reduce el coste del proyecto, al automatizar el proceso de identificación de defectos.
- Mejora la calidad del proyecto, ya que permite localizar defectos que podrían quedar ocultos.

1.4. Marco de la tesis y contribuciones

A continuación se describirán brevemente los proyectos de investigación en los que se desarrollado esta tesis y las contribuciones más relevantes que se han realizado.

1.4.1. Proyectos de investigación

Esta tesis doctoral se ha desarrollado en el marco de los siguientes proyectos de investigación financiados en convocatorias públicas:

- **Proyecto**: Automatización de la detección y diagnosis de fallos de sistemas estáticos y dinámicos usando conocimiento semicualitativo
  Investigador principal: Rafael Martínez Gasca

Identificador: DPI2003-07146-C02-01
Duración: 2003 - 2006

- **Proyecto**: Detección automática de fallos, diagnosis y tolerancia a fallos en sistemas con incertidumbre y distribuidos.
  Investigador principal: Rafael Martínez Gasca
  Identificador: DPI2006-15476-C02-00
  Duración: 2006 - 2009

- **Proyecto**: Mejora de la Calidad de Procesos de Negocios Mediante Tecnologías de Optimización y Tolerancia a Fallos.
  Investigador principal: Rafael Martínez Gasca
  Identificador: P08-TIC-04095
  Duración: 2009 - 2010

- **Proyecto**: Técnicas para la Diagnosis, Confiabilidad y Optimización en los Sistemas de Gestión de Procesos de Negocio.
  Investigador principal: Carmelo del Valle Sevillano
  Identificador: TIN2009-13714
  Duración: 2010 - 2012

1.4.2. Contribuciones

Para poder completar los objetivos marcados en la tesis, ha sido necesario abrir diferentes líneas de investigación que han dado sus frutos en forma de publicaciones. A continuación se presenta una relación, organizada por fechas, de las publicaciones más relevantes. El índice de aceptación aparece en aquellos congresos en que es conocido o se ha publicado en las actas.

- **Título:** A constraint-based methodology for software diagnosis.
  Autores: R. Ceballos, R. M. Gasca, C. del Valle, M. Toro.
  Publicación: Constraints in Formal Verification Workshop. Conference on Principles and Practice of Constraint Programming (CP), páginas 56-64, 2002.

- **Título:** Max-CSP approach for software diagnosis.
  Autores: R. Ceballos, R. M. Gasca, C. del Valle, M. Toro.
  Publicación: IBERAMIA, Lecture Notes in Artificial Intelligence. Vol. 2527, páginas 172-181, 2002.
  ISSN: 0302-9743.
  Índice de aceptación:30,5. %

- **Título:** Diagnosis software usando técnicas Max-CSP.
  Autores: R. Ceballos, R. M. Gasca, C. del Valle, M. Toro.
  Publicación: VII Jornadas de Ingeniería del Software y Bases de Datos (JISBD),

páginas 425-426, 2002.
ISBN: 84-6888-0206-9.
Índice de aceptación:50 %.

- **Título:** An integration of FDI and DX approaches to polynomial models.
  Autores: R. M. Gasca, C. del Valle, R. Ceballos y M. Toro.
  Publicación: 14th International Workshop on Principles of Diagnosis (DX), páginas 153-158, 2003.

- **Título:** A constraint programming approach for software diagnosis
  Autores: R. Ceballos, R. M. Gasca, C. del Valle y F. de la Rosa
  Publicación: Fifth International Workshop on Automated Debugging (AADEBUG), páginas 187-197, 2003.

- **Título:** CSP aplicados a la diagnosis basada en modelos
  Autores: R. Ceballos, C. del Valle, M. T. Gómez-López, R. M. Gasca
  Publicación: Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial, páginas 137-150, 2003.
  ISSN: 1137-3601.

- **Título:** Applying Constraint Databases in the Determination of Potential Minimal Conflicts to Polynomial Model-based Diagnosis.
  Autores: M. T. Gómez-López, R. Ceballos, R. M. Gasca y C. del Valle
  Publicación: Constraint Databases (CDB), Lecture Notes in Computer Science. Vol. 3074, páginas 74-87, 2004.
  ISSN: 0302-9743.
  Índice de aceptación:34 %.

- **Título:** An integration of FDI and DX approaches to polynomial models.
  Autores: R. Ceballos, M. T. Gómez-López, R. M. Gasca y Sergio Pozo.
  Publicación: 15th International Workshop on Principles of Diagnosis (DX), páginas 153-158, 2004.

- **Título:** Determination of Possible Minimal Conflict Sets Using Constraint Databases Technology and Clustering.
  Autores: M. T. Gómez-López, R. Ceballos, R. M. Gasca y Sergio Pozo.
  Publicación: IBERAMIA, Lecture Notes in Artificial Intelligence. Vol. 3315, páginas 942-952, 2004.
  ISSN: 0302-9743.
  Índice de aceptación:31 %.

- **Título:** Diagnosis de inconsistencia en contratos usando el diseño bajo contrato.
  Autores: R. Ceballos, F. de la Rosa, S. Pozo.
  Publicación: IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD),

páginas 25-36, 2004.
ISBN: 84-6888-8983-0.
Índice de aceptación:31 %.

- **Título:** Diagnosis of inconsistency in contracts using Design by Contract.
  Autores: R. Ceballos, F. de la Rosa y S. Pozo.
  Publicación: IEEE Latin America Transactions, Volume: 3, Issue: 1, 2005.
  ISSN: 1548-0992.

- **Título:** A model integration of DX and FDI techniques for automatic determination of minimal diagnosis.
  Autores: R. Ceballos, M. T. Gómez-López, R.M. Gasca and C. del Valle.
  Publicación: 2nd MONET Workshop on Model-Based Systems (IJCAI-05).

- **Título:** Constraint satisfaction techniques for diagnosing errors in Design by Contract software.
  Autores: R. Ceballos, R. M. Gasca and D. Borrego.
  Publicación: Specification and Verification of Component-Based Systems Workshop. ESEC/FSE'05.

- **Título:** An Integration of FDI and DX Techniques for Determining the Minimal Diagnosis in an Automatic Way.
  Autores: R. Ceballos, S. Pozo, C. del Valle y R. M. Gasca.
  Publicación: MICAI 2005. Advances in Artificial Intelligence. Lecture Notes in Computer Science. Vol. 3789, páginas 1082-1092.
  ISSN: 0302-9743.
  Índice de aceptación:41 %.

- **Título:** Constraint satisfaction techniques for diagnosing errors in Design by Contract software.
  Autores: R. Ceballos, R. M. Gasca y D. Borrego.
  Publicación: ACM SIGSOFT Software Engineering Notes (SEN). Volumen 31. Número 2, 2006.
  ISSN: 0163-5948.

- **Título:** Diagnosing Errors in DbC Programs Using Constraint Programming.
  Autores: R. Ceballos, R. M. Gasca, C. Del Valle y D. Borrego.
  Publicación: 11th CAEPIA. Selected Papers. Lecture Notes in Computer Science. Vol. 4177, páginas 200-210, 2006.
  ISSN: 0302-9743.

- **Título:** A Topological-Based Method for Allocating Sensors by Using CSP Techniques.
  Autores: R. Ceballos, V. Cejudo, R. M. Gasca y C. Del Valle.

Publicación: 11th CAEPIA. Selected Papers. Lecture Notes in Computer Science. Vol. 4177, páginas 62-68, 2006.
ISSN: 0302-9743.

- **Título:** The minimal diagnosis determination by using an integration of model-based techniques.
  Autores: R. Ceballos, M.T. Gómez-López, R. M. Gasca y C. Del Valle.
  Publicación: Inteligencia Artificial. Volumen 10. Número 31, páginas 41-52, 2006.
  ISSN: 1137-3601.

- **Título:** A compiled model for faults diagnosis based on different techniques.
  Autores: R. Ceballos, M. T. Gómez López, R. M. Gasca y C. del Valle.
  Publicación: Artificial Intelligence Communications. Volumen 20. Número 1, páginas 7-16, 2007.
  ISSN: 0921-7126.
  Índice de calidad: Indexado en ISI JCR 2007 0,59.

- **Título:** CSP-Based Firewall Rule Set Diagnosis using Security Policies.
  Autores: S. Pozo, R. Ceballos, R. M. Gasca.
  Publicación: ARES 2007, páginas 723-729, 2007.
  Índice de aceptación:28 %.

- **Título:** NMUS: Structural Analysis for Improving the Derivation of All MUSes in Overconstrained Numeric CSPs.
  Autores: R. M. Gasca, C. Del Valle, M. T. Gómez-López y R. Ceballos.
  Publicación: 12th CAEPIA. Selected Papers. Lecture Notes in Computer Science. Volumen 4788, páginas 160-169, 2008.
  ISSN: 0302-9743.

- **Título:** Developing a labelled object-relational constraint database architecture for the projection operator.
  Autores: M. T. Gómez López, R. Ceballos, R. M. Gasca y C. del Valle.
  Publicación: Data and Knowledge Engineering. Volume 68. Número 1, páginas 146-172, 2009.
  ISSN: 0169-023X.
  Índice de calidad: Indexado en ISI JCR 2009 1,75.

- **Título:** Model-Based Development of firewall rule sets: Diagnosing model inconsistencies.
  Autores: S. Pozo, R. Ceballos y R.M. Gasca.
  Publicación: Information and Software Technology. Volume 51. Número 5, páginas 894-915, 2009.
  ISSN: 0950-5849.
  Índice de calidad: Indexado en ISI JCR 2009 1,82.

- **Título:** On the Complexity of Program Debugging Using Constraints for Modeling the Programs Syntax and Semantics.
  Autores: F. Wotawa, J. Weber, M. Nica and R. Ceballos.
  Publicación: 13th CAEPIA. Selected Papers. Lecture Notes in Computer Science. Vol. 5988, páginas 22-31, 2010.
  ISSN: 0302-9743.

1.5. Estructura de la tesis

El documento consta de las siguientes partes:

- **Prefacio**. En la primera parte se introducirán los objetivos perseguidos en esta tesis. Se expondrán por primera vez los conceptos en los que se sustenta el trabajo realizado, se mostrará la motivación y el contexto donde la tesis se ha desarrollado. Consta de un único capítulo.

- **Antecedentes**. El propósito de la segunda parte es describir los conceptos que el lector necesitará para comprender mejor la propuesta que se presentará en la tercera parte. Consta de dos capítulos.

  - En el capítulo 2 se mostrarán las metodologías más utilizadas en la detección de errores y depuración del software. Las metodologías encargadas de la detección de errores se han agrupado en tres grupos: basadas en el uso de la especificación del código fuente, basadas en el análisis estático del código fuente, y por último las basadas en el análisis dinámico del código fuente.
  - En el capítulo 3 se realiza un repaso sobre las definiciones y conceptos más importantes de las metodologías de Diagnosis Basada en Modelos y la Programación con Restricciones. La adaptación de los conceptos de la Diagnosis Basada en Modelos será primordial para la formalización de la metodología propuesta en la tercera parte de esta tesis. La Programación con Restricciones se utilizará como base para la implementación y automatización de la metodología de diagnosis propuesta.

- **Propuesta**. En esta parte se detalla la metodología propuesta en la tesis. Se ha dividido en cuatro capítulos.

  - En el capítulo 4 se presentan las bases de la metodología. Primero se mostrarán las adaptaciones propuestas para la metodología DX con el fin de que los conceptos utilizados en dicha metodología puedan ser reutilizados en la metodología propuesta en esta tesis. Luego se introducirá la metodología propuesta, y las fases de las que consta. Para cada una de las fases se expondrán cuáles son las tareas a realizar y los objetivos a cumplir. Una vez fijadas las

bases de la metodología, y las fases de las que consta, en los siguientes dos capítulos, la metodología propuesta será ampliada y adaptada para cumplir con los dos objetivos marcados, el diagnóstico de la especificación, y del código fuente.

- En el capítulo 5 se adaptará la metodología de diagnosis propuesta en el capítulo anterior para comprobar la consistencia entre los asertos que forman la especificación, e identificar los asertos que deben cambiar si se detectan inconsistencias. En primer lugar se mostrarán cuáles son las adaptaciones necesarias con respecto a la metodología general de diagnosis, y en segundo lugar se mostrará de manera detallada como debe aplicarse la metodología para detectar e identificar defectos en los invariantes, precondiciones y postcondiciones que forman la especificación.
- En el capítulo 6 la metodología de diagnosis propuesta será adaptada con el objetivo de poder identificar defectos en el código fuente. En primer lugar se mostrarán cuáles son las adaptaciones necesarias con respecto a la metodología general de diagnosis, y en segundo lugar se mostrará de manera detallada cuáles son los pasos a seguir para la detección de errores e identificación de los defectos en el código fuente.
- En el capítulo 7 se persiguen dos objetivos principales. Por una parte presentar las pruebas que se han realizado para comprobar la eficiencia y eficacia de la metodología propuesta en el diagnóstico de programas. Y por otra parte, ampliar la metodología propuesta para mejorar la eficacia a la hora de reparar los defectos. Para ello se propondrán varias mejoras.
- Por último, en el capítulo 8 se mostrarán cuáles son las principales aportaciones de esta tesis, a como de conclusiones. Y se incluirán algunas líneas en las que se está trabajando para ampliar y mejorar la metodología propuesta.

- **Apendices**. En el apéndice A se desarrollan diferentes algoritmos auxiliares utilizados en el capítulo 6. En el apéndice B se muestra los ejemplos utilizados en las pruebas realizadas en el capítulo 7.

Parte II

Antecedentes

Capítulo 2

Técnicas de detección de errores y aislamiento de defectos en el software

2.1. Introducción

El principal objetivo a la hora de desarrollar software es satisfacer y cubrir las expectativas y necesidades de los usuarios finales. Para lograr este objetivo se han desarrollado diversas metodologías y técnicas encaminadas a controlar el proceso de producción del software.

Las técnicas de validación y verificación (V&V) [20] del software permiten controlar el proceso de producción del software. La verificación garantiza que el software se ha diseñado teniendo en cuenta todas las funcionalidades que el cliente requiere. Pretende dar respuesta a la pregunta: ¿se está construyendo el software correctamente? Está orientada a controlar y mejorar la calidad del proceso de producción del software.

La validación del software es la evaluación del grado de cumplimiento de los requisitos establecidos para el producto software. Pretende dar respuesta a la pregunta: ¿se está construyendo el software correcto? Está orientada a controlar el producto, es decir, el software. La validación es un proceso posterior a la verificación. La verificación está más ligada al razonamiento y a la lógica deductiva [34], mientras que la validación suele realizarse utilizando casos de prueba.

Resulta evidente que para poder validar el grado de corrección de un producto software es necesario saber qué es lo que se espera de él. El comportamiento correcto puede expresarse a través de la especificación del software. Dicha especificación puede hacerse de manera informal o siguiendo una metodología. El principal problema de la especificación informal es la ambigüedad, ya que muchas veces la especificación puede tener diversas interpretaciones. Otro problema añadido está en que si no se revisa adecuadamente la documentación, la especificación puede que sólo sea comprensible por el propio desarrollador del software [104].

Se dice que hay un error en la ejecución de un código fuente, cuando al ejecutarlo se produce un resultado diferente al esperado. Los defectos, también llamados 'bugs', son los fragmentos de código que generan los errores. Son los programadores los que introducen los defectos. Estos defectos pueden deberse a una mala especificación del software que se está desarrollando, o a fallos cometidos por los propios programadores de forma involuntaria al implementar los algoritmos.

Existen diferentes metodologías que permiten detectar errores en el software. En este capítulo estas técnicas se han agrupado en tres tipos. Un primer grupo de técnicas basadas en el uso de la especificación del código fuente, otro grupo de técnicas basadas en el análisis estático del código fuente, y por último las técnicas basadas en el análisis dinámico del código fuente.

- Técnicas basadas en la especificación. La especificación establece qué es lo que el software debe realizar, o que relaciones entre las E/S deben satisfacerse. La especificación permite comprobar si los resultados generados por la implementación son correctos. El código fuente o implementación del sistema software debe responder a todos los requisitos establecidos en la especificación.

- Técnicas basadas en el análisis estático. El análisis estático de un sistema software puede detectar defectos potenciales o reales, sin utilizar casos de prueba y sin ejecutar el código fuente. Los defectos que se identifican afectarían a cualquier traza de ejecución que los incluya.

- Técnicas basada en el análisis dinámico. En el análisis dinámico de un código fuente, en contraste con el análisis estático, se estudian casos de prueba concretos. Los casos de prueba tienen éxito cuando el sistema software produce al menos un error. El caso de prueba establece qué condiciones iniciales deben aplicarse al sistema software, y cuáles son los resultados que el sistema software debe producir. Si hay discrepancia entre el resultado real y el correcto, entonces debe existir un defecto en el sistema software.

La depuración del software tiene por objetivo localizar y reparar los defectos. Depurar un software implica formular una hipótesis sobre el comportamiento del software, y comprobar si dicha hipótesis permite solucionar los errores encontrados. En este capítulo del estado del arte se hará un repaso primero de las técnicas y metodologías presentadas anteriormente para la detección de errores, y en segundo lugar, de las técnicas y metodologías para la depuración del software.

2.2. Técnicas basadas en la especificación

En esta sección se hará un repaso de las técnicas formales, y dentro de las técnicas formales se profundizará en el diseño por contrato, por ser ésta la metodología que se utilizará en esta tesis para expresar la especificación del software. La especificación

puede ser generada de manera manual, o también de forma automática, por este motivo se hará un recorrido sobre los principales trabajos en el campo de la generación automática de la especificación. Como último punto, se mostrará un resumen de las técnicas que permiten la comprobación de la conformidad.

2.2.1. Métodos formales

Producir software de calidad es uno de los objetivos primordiales de la Ingeniería del Software. La calidad es un concepto complejo y con muchas facetas, que puede describirse desde diversas perspectivas, tales como el usuario, el productor o el propio producto. En los últimos años los métodos formales [99], y las metodologías derivadas de éstos, se han posicionado como el enfoque más apropiado para conseguir mejorar la calidad en la construcción de sistemas de información. Los métodos formales se basan en el empleo de lenguajes y técnicas definidos de forma matemática que facilitan el análisis y diseño de software. Su uso implica un aumento de recursos y tiempo, pero en contrapartida, garantizan un desarrollo de calidad ya que se evitan ambigüedades e inconsistencias en los desarrollos. Además permiten demostrar, en algunos casos de forma automática, propiedades o comportamientos de los programas especificados.

Las técnicas formales pueden clasificarse en:

- Axiomáticas. La semántica axiomática consiste en definir una serie de reglas que caracterizan las propiedades de los diferentes elementos del lenguaje. Estas reglas se expresan en lógica de primer orden y teoría de conjuntos. Se utilizan aserciones y ternas de Hoare [82] basadas en precondiciones y postcondiciones. Una aserción es una fórmula lógica de predicados que toma un valor booleano en un contexto. Una terna de Hoare es una expresión de la forma $\{P\}C\{Q\}$ donde P y Q son aserciones y C son sentencias. P se denomina precondición y Q se denomina postcondición. Este tipo de formalismo facilita la demostración de propiedades de los programas.

- Denotacionales. En la semántica denotacional el comportamiento del programa se describe modelizando los significados mediante entidades matemáticas básicas (generalmente funciones). El estudio denotacional de lenguajes de programación fue propuesto originalmente por C. Strachey en 1967 [160] utilizando λ-cálculo.

- Operacionales. En la semántica operacional [138] las transiciones elementales de un programa se especifican mediante reglas de inferencia definidas por inducción sobre su estructura.

- Algebraica. La semántica algebraica permite la especificación algebraica de tipos de datos [70]. Se basan en la descripción de estructuras de datos, y se establecen tipos y operaciones sobre esos tipos. Las operaciones de un tipo se definen a través de ecuaciones que especifican las restricciones que deben satisfacer las operaciones. Los métodos más conocidos son: Larch, OBJ, TADs.

En [99] se hace una comparación cualitativa de éstas y otras técnicas de especificación semántica, teniendo en cuenta diferentes características, como por ejemplo la modularidad, la reutilización, la posibilidad de realizar demostraciones o la flexibilidad.

En los últimos años se ha producido un acercamiento entre las comunidades de la Ingeniería del Software y la comunidad de los Métodos Formales. Los métodos formales se centran en la calidad, pero en muchos casos no se tiene en cuenta o no existe un entorno donde puedan aplicarse o herramientas que los sustenten. La falta de herramientas hace de los métodos formales recursos bastante difíciles de manejar. En la Ingeniería del Software el paradigma de la Orientación a Objetos se ha establecido como base para los desarrollos software. Un sistema software orientado a objetos está constituido por una colección de objetos que colaboran entre sí para conseguir un propósito

La automatización en la producción del software necesita de la formalización. Por este motivo muchos de los métodos formales han sido ampliados para recoger conceptos de orientación a objetos. Un Método Formal Orientado a Objetos (MFOO) permite especificar, desarrollar y validar sistemas software mediante el uso del lenguaje matemático y características orientadas a objetos. La orientación a objetos incluye conceptos como: objetos, ocultación, interacción, abstracción, clases, herencia, tipo o subtipos. En el trabajo de Galán Morillo [64] se hace un estudio de las diferentes metodologías que permiten los métodos formales Orientados a Objetos.

Dentro de los MFOO, y tomando como base la Lógica de Primer Orden, podríamos diferenciar entre los métodos semiformales y los formales.

- Los denominados métodos semiformales representan una transición más suave hacia los métodos formales. Dentro de éstos destaca, como método orientado al desarrollo de componentes, Catalysis [47].

- Los métodos formales basados en Lógica de Primer Orden más conocidos son Z [158] y VDM [88]. También existen extensiones orientadas a objetos: Object-Z [49], Z++ [173] y VDM++ [46], que presentan características como la identidad de objetos, la encapsulación, la herencia, la creación de objetos, agregaciones, colecciones, etc.

Una vez diseñada la especificación, es posible transformarla y derivar la implementación del sistema software, es decir, el código fuente. En [117] se hace un repaso de diferentes métodos para la generación del código fuente (para programas recursivos e iterativos), aplicando métodos de inducción y derivación sobre la especificación del sistema software. También es posible realizar demostraciones, a partir de la especificación, sobre el sistema software. En esta línea, es interesante reseñar la metodología presentada Beckert y Klebanov en [9], para la reutilización de las demostraciones asociadas a un sistema software cuando éste sufre pequeñas modificaciones.

Aunque estas técnicas han demostrado su valía en la generación de código de pequeños programas, resulta difícil aplicarlas a problemas de cierto tamaño, y si el proceso quiere realizarse de manera automática no existen herramientas que permitan completar la generación de código. Por tanto su aplicación práctica se ve muy limitada.

2.2.2. Diseño por Contrato

El Diseño por Contrato (Design by Contract, DbC) tiene su origen en los métodos formales, aunque mantiene una visión más práctica. El DbC aplicado al lenguaje de programación Eiffel fue propuesto por Bertrand Meyer [123]. El DbC permite desarrollar software más fiable y robusto. La fiabilidad, es decir, la exactitud con la que el sistema realiza su trabajo conforme a una especificación, es uno de los mayores componentes de calidad dentro del software. Un sistema robusto ofrece garantías de poder controlar las situaciones anormales. La calidad del software toma mayor relevancia en la Orientación a Objetos debido a la reutilización del software.

Para Meyer [125], y dentro del paradigma de la Diseño por Contrato, un componente software debe cumplir las siguientes condiciones:

1. Puede ser utilizado por otros elementos del software.
2. Puede ser utilizado por los clientes sin la intervención del desarrollador de los componentes.
3. Incluye la especificación de todas las dependencias.
4. Incluye la especificación de las funcionalidades ofrecidas.
5. Puede usarse sobre la base única de sus especificaciones.
6. Se puede formar parte de otros componentes.
7. Puede integrarse en otros sistemas de forma rápida y suave.

Cuando un componente cumple estas condiciones, es posible abstraerse de la implementación concreta y trabajar con el componente como si fuese una caja negra. Es en la especificación donde se establecen las características de las entradas suministradas al componente software, y de las salidas generadas.

El DbC permite especificar de forma automática las relaciones entre los elementos de un programa orientado a objetos. Un sistema software es tratado como un sistema de componentes con capacidad para comunicarse, y cuya interacción se basa en un conjunto definido y preciso de especificaciones a modo de contratos. La presencia de una especificación no garantiza completamente la corrección del software, pero es una buena base para evitar y localizar defectos. Los beneficios del DbC son entre otros [122]:

- Mayor facilidad para comprender el sistema software desarrollado, ya que los contratos pueden almacenar gran parte de la especificación.
- Se consigue un marco más eficaz para eliminar errores en los procesos y garantizar calidad en el software generado. Por tanto se generan sistemas software donde existen más posibilidades de detectar y aislar errores.

- Permite automatizar la generación de documentación para los componentes software desarrollados, usando para ello la información de los contratos.
- Mejor comprensión y control sobre los métodos y mecanismos de programación orientados a objetos, como por ejemplo la herencia. Mejores hábitos para desarrollar programas orientados a objetos.

Aunque el primer lenguaje de programación capaz de soportar el diseño por contrato fue Eiffel, actualmente existen otros lenguajes que soportan por completo o en parte el diseño por contrato. También existen entornos y bibliotecas que permiten extender determinados lenguajes, de forma que puedan incorporar el diseño por contrato, como por ejemplo el proyecto JML [106] para Java, o el proyecto SPEC# [7] para C#.

Elementos básicos

Los contratos controlan la interacción de los objetos con el resto del sistema. Especifican, de forma no ambigua, las relaciones entre los clientes de un servicio y los servidores. Cliente y servidor obtienen ventajas y obligaciones del contrato. El contrato protege al cliente, especificando qué debe quedar hecho, y al servidor, que sólo está obligado a realizar lo que está especificado. Los contratos deben ser explícitos, y a su vez deben formar parte de los elementos del software en sí mismos.

Los contratos se especifican mediante la utilización de asertos. Un aserto es una fórmula lógica de predicados que toma un valor booleano en un contexto. Existen diferentes tipos de asertos. En DbC se utilizan básicamente tres tipos de asertos: precondiciones, postcondiciones, e invariantes de las clases. Los asertos de tipo precondición establecen una condición que debe cumplirse antes de ejecutar un método concreto, mientras que los asertos de tipo postcondición establecen una condición que debe cumplirse después de ejecutar un método concreto. Los invariantes de clase son asertos que establecen condiciones que deben cumplirse antes y después de la ejecución de cualquier método de un objeto.

Ejemplo 2.1. Aunque existen muchos lenguajes que han incorporado el DbC en sus últimas versiones (o a través de bibliotecas o extensiones), Eiffel [124] sigue siendo el lenguaje mejor adaptado al DbC. A continuación se muestra un ejemplo de contrato utilizando el lenguaje Eiffel. En concreto se muestra la operación de inserción en un tipo abstracto de datos diccionario [162]:

```
put (e: ELEMENT; clave: STRING) is
 require
   contador <capacidad
   not clave.empty
 do
   ... Algoritmo de inserción ...
 ensure
```

```
    has (e)
    elemento (clave) = e
    contador = old contador + 1
end
```

La operación requiere que el diccionario no tenga completa su capacidad y que la clave recibida no sea nula. Si esta precondición (cláusula *require*) se cumple, el método garantiza, a través de la postcondición (cláusula *ensure*), que al terminar, el objeto recibido se encontrará en el diccionario asociado a la clave recibida, y que el contador de elementos se habrá incrementado en uno. El operador *old* permite hacer referencia al valor de una variable antes de entrar en un método.

Las precondiciones y postcondiciones se aplican sobre métodos. Existen otros tipos de asertos que almacenan características generales e invariables de una clase. Estos tipos de asertos son los *invariantes* de clase, y deben cumplirse en cualquier punto de una clase. Por ejemplo, un invariante para la anterior implementación del diccionario podría ser:

```
invariant
  0 <= contador
  contador <capacidad
```

El DbC plantea una solución que evita tener que aplicar la denominada programación defensiva [123]. La programación defensiva incita a realizar todas las comprobaciones posibles para evitar que la ejecución de un código pueda producir errores. Como consecuencia, se produce un aumento drástico de la complejidad del software. Tal como plantea Meyer [123], las comprobaciones redundantes agregan más código, que puede contener defectos.

Los principios básicos que una aplicación debe cumplir para poder incorporar de forma adecuada el DbC son los siguientes [126]:

- Separar las consultas de las órdenes. Las consultas deben devolver el estado de los objetos, pero no deben permitir la modificación de éste. Las órdenes pueden cambiar el estado de los objetos, pero no deben devolver resultados.
- Separar las consultas básicas de las consultas complejas o derivadas. Las consultas complejas o derivadas deben formularse en función de las consultas básicas.
- Para cada consulta compleja, escribir la postcondición en función de una o varias consultas básicas. De esta forma si se conocen los valores de las consultas básicas será posible conocer el efecto global de la consulta compleja.
- Para cada orden, escribir la postcondición en función de una o varias consultas básicas. De esta forma se puede conocer el efecto global de cada orden.

- Para cada consulta u orden, establecer una precondición. La precondición especifica qué condiciones deben cumplir los clientes para poder llamar a la consulta u orden.

- Escribir los invariantes necesarios para definir características constantes de objetos. Los invariantes deben concentrarse en las características que ayudan al lector de una clase a construir un modelo conceptual.

Herencia de contratos

Una de las características más importantes de la Programación Orientada a Objetos es la herencia. Al igual que el código fuente es heredado de la superclase a las subclases, la especificación en forma de contratos, debe ser heredada. Teniendo en cuenta que las instancias de una clase son también instancias de las superclases, cuando se modifican los métodos heredados en las subclases, es necesario respetar los asertos definidos y heredados de las superclases. Es lo que se conoce como el principio de sustitución de Liskov [112]. Aunque al sobrescribir un método puede cambiar totalmente el código, existe obligación de satisfacer el contrato heredado, lo que condiciona qué puede ser refinado en el contrato y código fuente de la clase hija, e indirectamente condiciona también qué puede incluir el contrato de la clase padre. Esto implica consecuencias para los invariantes, precondiciones y postcondiciones establecidos.

- Invariantes: Los nuevos invariantes introducidos en las subclases no pueden entrar en contradicción con los invariantes heredados de la superclase. Es decir, el invariante de una clase es la conjunción lógica de los nuevos asertos que aparecen en el invariante y de los asertos del invariante heredado de la superclase, si hay alguno. Esta obligación garantiza que los atributos heredados de la superclase cumplan las mismas condiciones establecidas en la superclase.

- Precondiciones y postcondiciones: En el caso de las precondiciones, éstas no pueden ser refinadas en una subclase para ser más duras, sólo pueden ser debilitadas. La razón es simple, toda llamada que se pueda realizar al método definido en la clase padre, debe poder realizarse también al método refinado en la clase hija. En el caso de las postcondiciones es justo lo contrario, no se pueden debilitar, sólo pueden ser endurecidas. Esta obligación garantiza que no se producirá una salida diferente al llamar a un método refinado en la clase hija de las que estaban establecidas como posibles salidas en el método definido en la clase padre.

Ventajas del DbC para la detección de defectos

En un sistema software donde se ha seguido la metodología de DbC, la información almacenada en los asertos puede utilizarse para validar dicho sistema software. Para ello se pueden utilizar lo que se denominan ternas de Hoare [82], basadas en precondiciones y postcondiciones. Una terna de Hoare, tal como se ha explicado anteriormente, es una

expresión de la forma $\{Pre\}C\{Post\}$ donde *Pre* y *Post* representan los asertos que forman la precondición y la postcondición respectivamente, y C son sentencias.

Con las ternas de Hoare se pueden establecer las condiciones de corrección de una clase. Una clase es correcta con respecto a sus aserciones si y sólo si se cumplen las siguientes condiciones [123]:

- Para cada constructor C: $\{Pre_C\}C\{Post_C\}\{Inv\}$
- Para cada método M: $\{Inv\}\{Pre_M\}M\{Post_M\}\{Inv\}$

Donde *Inv* representa el conjunto de invariantes de la clase, Pre_C y $Post_C$ representan la precondición y postcondición del constructor C, y Pre_M y $Post_M$ representan la precondición y postcondición del método M. Es importante reseñar que un invariante se debe cumplir una vez ejecutado el constructor, pero no antes, ya que es el constructor donde se establecerán los valores apropiados para que se pueda satisfacer. Para la validación de programas con varios hilos se puede consultar el trabajo de Jacobs [86]. En él se presenta una metodología para la validación de sistemas software con varios hilos de ejecución que incorporen invariantes. Esta metodología es una extensión de la metodología propuesta anteriormente para un único hilo de ejecución.

En [105], Yves Le Traon realizó un estudio del impacto que conlleva el uso de los asertos del DbC 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 pruebas sobre el 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 [22], Briand demuestra que el uso de contratos permite un mayor aislamiento de los defectos en el software. En la orientación a objetos, 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 tiene asociado un mayor esfuerzo en el desarrollo de software, pero mejora los resultados a la hora de detectar y localizar los errores.

2.2.3. Generación automática de la especificación

Esta especificación suele realizarse antes del desarrollo de la implementación del sistema software. Existen casos en los que la especificación no es accesible o no existe. También puede suceder, que aunque se tenga acceso, éste sea parcial, o sólo contemple parte del sistema software.

En el caso de que no exista la especificación, o que ésta sea parcial, existen metodologías y herramientas que permiten inferir de manera automática dicha especificación.

Esta inferencia puede realizarse de dos formas diferentes, estática y dinámicamente. En el primer caso en el proceso de inferencia se utiliza el código fuente del sistema software, y no es necesario ningún caso de prueba para ejercitar el código fuente. En el segundo caso, la inferencia se basa en el uso de diferentes casos de prueba sobre el mismo sistema software.

A continuación se muestran los trabajos más relevantes en la generación automática de la especificación, tanto de forma estática, como dinámica. Aunque estas metodologías han demostrado su valía en muchos casos, su eficacia depende mucho del tipo de problema al que se enfrentan. Existen trabajos en los que se defiende la imposibilidad de generar de forma totalmente automática la especificación, como es el caso de [154], donde, con diversos ejemplos, se pone de manifiesto que en muchos casos es necesaria la intervención humana para complementar la especificación generada de forma automática.

Generación estática de la especificación

En la generación estática de la especificación, la idea es inferir la especificación en forma de asertos utilizando como entrada solamente el código fuente del sistema software. Para ello será necesario abstraerse de los detalles de implementación y extraer el modelo del comportamiento del sistema software. Este modelo se representará usando asertos que contendrán las condiciones del comportamiento esperado del sistema software. En este campo son especialmente significativos los avances en la obtención de los invariantes para bucles. Un invariante de un bucle es una expresión que debe cumplirse siempre al final de cada iteración del bucle.

En [90] Deepak Kapur propone una metodología para la generación automática de los invariantes de bucles basada en la eliminación de los cuantificadores. Para generar los invariantes se proponen unas plantillas en forma de fórmulas parametrizadas con cuantificadores. Cada una de estas fórmulas con parámetros son hipótesis, es decir, posibles invariantes de los que se debe comprobar su validez. Para comprobar su validez y dar valores concretos a los parámetros, la idea es realizar iteraciones, y comprobar qué valores deben tomar estos parámetros para dichas iteraciones. Si las restricciones asociadas a una formula parametrizada no tienen solución, entonces el invariante asociado no tiene sentido para el bucle. En [90] se proponen diferentes heurísticas para tratar las diferentes trazas que se pueden producir en función del número de iteraciones. La metodología se puede utilizar para refinar los invariantes, las precondiciones y postcondiciones asociadas al código del bucle. Fruto de la colaboración entre Deepak Kapur y Enric Rodríguez-Carbonell, en los trabajos [150] y [151] se proponen nuevas mejoras, como por ejemplo el uso de las Bases de Gröbner. Además se propone una cota para el número de iteraciones necesarias para lograr generar los invariantes de los bucles. Por último, son también muy interesantes los trabajos de Kovács sobre la generación de invariantes en bucles. Por ejemplo en [97] se presenta una metodología para generar invariantes de tipo polinómico, y se acompaña de una implementación

denominada Aligator que puede ejecutarse en la herramienta Mathematica.

También se han propuesto trabajos en los que se consigue obtener la especificación de todo un programa completo. Por ejemplo, en el trabajo de Pleszkoch y Linger [137] se propone una metodología para extraer la funcionalidad de un programa, es decir, para generar de forma automática y en forma de funciones matemáticas cuál es el comportamiento de un programa. Para abstraerse de la implementación concreta, las estructuras de control y las sentencias de un programa se transforman a funciones y estructuras matemáticas.

Generación dinámica de la especificación

Cuando la especificación se genera de forma dinámica debe existir una fase de entrenamiento. En esta fase de entrenamiento se audita el comportamiento dinámico del sistema software ante diferentes casos de prueba. Con los datos recogidos se crearían nuevos asertos, que se irían adaptando a medida que en la fase de entrenamiento se introducen más casos de prueba. Una vez acabada la fase de entrenamiento, los asertos contienen el comportamiento esperado del sistema software. La precisión de los asertos generados dependerá de la calidad de los casos de prueba utilizados en la fase de entrenamiento. Una vez completada la fase de entrenamiento, los asertos guardarán las condiciones que el sistema software debe cumplir en su ejecución.

Si la especificación generada de forma automática no se satisface durante la ejecución del sistema software, esta inconsistencia puede deberse a dos causas. En general, si la especificación es violada se deduce que debe existir un problema en la implementación del sistema software, ya que el sistema software ha violado lo que se supone es su comportamiento normal y correcto. Pero también puede ocurrir que la especificación no sea correcta del todo, y se haya detectado un problema que no existe en realidad, lo que se denomina un falso positivo. También puede ocurrir que algunos errores no violen la especificación generada, cuando en realidad sí existe un problema, lo que se denomina un falso negativo.

Existen varias metodologías y herramientas que permiten la generación dinámica de la especificación [76][51][52][146]. En este apartado se hará un repaso de las más extendidas.

DIDUCE [76] es un detector de errores basado en la generación dinámica de invariantes, desarrollado por Sudheendra Hangal (Sun Microsystems) y Monica Lam (Universidad de Stanford). DIDUCE incluye un generador y verificador de invariantes para programas escritos en el lenguaje Java. Concretamente, la herramienta está desarrollada a nivel de bytecode, por lo que en realidad no es necesario tener el código fuente del sistema software, bastaría con tener los bytecodes. Las expresiones que son auditadas por la herramienta para generar los invariantes en la fase de entrenamiento, son las sentencias de lectura y escritura en las variables, parámetros de entrada, y valores devueltos por las llamadas a métodos.

DIDUCE ha sido probado en diversas aplicaciones reales, reportando buenos resul-

tados. La especificación generada por DIDUCE ofrece buenos resultados cuando el tipo de error a detectar es una asignación de un valor que no se encuentra dentro del rango de valores que han sido asignados a dicha variable en la fase de entrenamiento. En [148] se hace una comparativa de las diferentes mejoras que se pueden realizar para auditar las expresiones en la fase de entrenamiento, y de esta forma conseguir reducir los falsos positivos y negativos debidos a defectos en los invariantes.

En la herramienta DIDUCE y en las mejoras presentadas en [148], los invariantes generados hacen referencia a una variable o como mucho a una expresión, pero no relacionan variables o expresiones que aparezcan en diferentes posiciones del código fuente. Son, por así decirlo, invariantes unarios. Por tanto, no se podrían detectar errores que impliquen a varias variables o expresiones a la vez. Las relaciones entre parejas o tripletas, o conjuntos de más variables pueden llegar a ser intratables, debido al número combinatorio de relaciones que se pueden dar, pero existen herramientas, como Daikon [51][53], que permiten llegar a una aproximación.

La herramienta Daikon [51][53] persigue la idea de generar invariantes que relacionen varias variables o expresiones contenidas en el código fuente de programas escritos en C, C++, Java o Perl. Es capaz de chequear unos 75 tipos de diferentes invariantes, aunque cuantos más tipos de invariantes se tengan en cuenta, más lento será el proceso de generación. La herramienta genera invariantes que relacionan los parámetros de entrada de las llamadas, los valores devueltos por las llamadas, las variables y atributos. Daikon permite generar precondiciones y postcondiciones de métodos, además de invariantes de clase y bucles. La herramienta Carrot [146] es similar a Daikon, pero utiliza un conjunto menor de posibles invariantes.

El framework ClearView [136] se basa en una variante de Daikon, y permite generar invariantes para el codigo binario ejecutable en una sistema Windows x86, sin necesidad del código fuente. Primero observa el comportamiento normal del programa para poder generar invariantes que caractericen su comportamiento, y luego monitoriza la ejecución para poder detectar violaciones de los invariantes. Además genera las sentencias adecuadas para que el comportamiento del programa sea el adecuado y que el estado permita satisfacer el invariante violado. Permite detectar los cambios en el comportamiento normal del código detectando a través de los invariantes desbordamientos de buffer o sentencias que transfieran el control de flujo, y de esta forma se previenen los daños que la inyección de un código malicioso puede producir.

2.2.4. Verificación de la conformidad

Cuando se implementa una aplicación en base a una especificación, la implementación debe ser compatible con las condiciones establecidas en la especificación, es decir, la ejecución de la implementación debe satisfacer las condiciones impuestas en la especificación. El objetivo de la verificación de la conformidad es comprobar que la especificación es una consecuencia de la implementación, y que por tanto la ejecución de la implementación satisfará siempre las condiciones impuestas en la especificación.

En esta línea han surgido varias metodologías y herramientas que permiten verificar la conformidad. En [29][30], Collavizza propone una metodología que permite realizar la verificación de programas donde ciertas propiedades de las entradas están fijadas, usando como base la programación con restricciones. Estas propiedades son, el valor de las variables y el tamaño de los arrays recibidos como entrada. El framework propuesto, utiliza restricciones para representar la especificación y el código fuente, y explora los posibles caminos de ejecución de forma no determinista. Las entradas son correctas si aplicadas sobre las restricciones satisfacen la postcondición establecida para el programa. Si las entradas llevan a un estado que no cumple la postcondición, entonces el programa debe ser modificado para ser conforme con la especificación.

Para la verificación de la conformidad también han aparecido herramientas como: ESC/Java [54], CBMC [25], Blast [19] y EUREKA [55] que se basan en las técnicas de Model Checking. Las técnicas de Model Checking permiten la verificación de una especificación formal del sistema, derivada de un sistema hardware o software. Para realizar la verificación, el sistema debe estar descrito mediante un modelo, que debe satisfacer una especificación formal expresada mediante fórmulas lógico-temporales. De manera formal, para verificar una propiedad p, expresada como una fórmula lógico-temporal, y un modelo M con un estado inicial s, se debe cumplir que $M, s \models p$. La propiedad a comprobar puede ser por ejemplo verificar si se puede producir un bloqueo y que por tanto el sistema no pueda terminar. Si M es finito, como sucede en el caso del hardware, el modelo suele estar expresado como un sistema de transiciones, es decir, un grafo dirigido, y la verificación de la propiedad puede realizarse como una búsqueda en un grafo. También pueden utilizarse algoritmos simbólicos para conseguir verificar si el modelo es capaz de soportar la especificación del sistema.

En [157], Schuele realiza un estudio comparativo de las técnicas a utilizar dependiendo de si el modelo M es finito o no. Para la verificación de modelos finitos, se pueden utilizar, generalmente y de forma indistinta, técnicas locales o globales de Model Checking. Sin embargo para modelos no finitos, la elección de una u otra técnica puede llevar a que el proceso de verificación no acabe. Para una revisión de los primeros trabajos en Model Checking puede consultarse los autores E. M. Clarke y E. A. Emerson [27], y J. P. Queille and J. Sifakis [147].

Son importantes los trabajos de Andreas Podelski. En [42] se propone el uso de la programación lógica con restricciones (Constraint Logic Programming, CLP) como implementación para la resolución de verificaciones basadas en Model Checking sobre modelos no finitos. En este trabajo aparecen dos contribuciones; por una parte una metodología para la transformación del sistema concurrente a un modelo basado en un programa CLP; y por otra parte una metodología para la verificación de propiedades en el modelo generado. En el trabajo se utilizan ejemplos con número de estados no finitos y sobre variables enteras. En [139] se propone la herramienta ARMC (Abstraction Refinement Model Checking), para la verificación de software industrial. Esta herramienta se basa en el uso del lenguaje Prolog con algunas extensiones basadas en CLP. También son importantes sus trabajos en modelos finitos, como por ejemplo [165]. En

este trabajo se propone una mejora para la evaluación de las transiciones de estados. Concretamente se presenta un esquema con una función de distancia como parámetro, para la evaluación de las transiciones.

Cuando se aplican las técnicas de Model Checking a programas software, existen dos objetivos principalmente: por una parte verificar si el programa es correcto, y por otra parte descubrir los errores del programa. Existen herramientas basadas en Model Checking, que al ser aplicadas a programas software, debido a que se basan en técnicas combinatorias para explorar el espacio de estados posibles, resultan poco eficaces por el problema de la explosión de estados. Para intentar evitarlo, se han desarrollado técnicas basadas en algoritmos simbólicos, abstracción, o de reducción del orden parcial. Por ejemplo, en [74], se presentan varias heurísticas para Model Checking con la idea de combatir el problema de la explosión de estados en la búsqueda de errores en programas software. En esta misma línea, el grupo de Sistemas de Software Robusto (The Robust Software Systems group) del centro de investigación de la NASA, viene desarrollando una herramienta para verificar programas escritos en el lenguaje Java. La herramienta Java PathFinder [111] (JPF), desarrollada por este grupo, permite explorar las diferentes trazas de un programa escrito en Java, con la intención de verificar si el programa es correcto o no. Relacionado con la herramienta Java PathFinder, en [73] se propone una técnica híbrida entre el Model Checking y las técnicas de generación de pruebas para encontrar errores. El fundamento de este trabajo es usar una heurística basada en las métricas para la cobertura de pruebas y en otras métricas sobre la estructura de los programas, tales como la interdependencia entre hilos de ejecución. En dicho trabajo también aparecen resultados experimentales que demuestran la validez de las heurísticas propuestas.

Otra herramienta de uso muy extendido para la verificación de software es SPIN [164]. Se trata de una herramienta de uso general para la verificar software distribuido de manera automática. Fue desarrollado por Gerard J. Holzmann y otros colaboradores en un grupo de desarrollo en Unix del Centro de Investigaciones Informáticas de los laboratorios Bell a principios de los ochenta. El software está disponible de forma libre desde 1991. Los sistemas a verificar deben estar descritos en Promela (Process Meta Language), que soporta el modelado de algoritmos distribuidos y asíncronos (en realidad el nombre SPIN proviene de "Simple Promela Interpreter"). Las propiedades que deben ser verificadas deben expresarse en fórmulas de lógica temporal lineal. Además de Model Checking, la herramienta permite hacer simulaciones sobre una traza concreta. SPIN ofrece diversas opciones para adaptar la herramienta al problema concreto, y así mejorar la velocidad del proceso de verificación. Gracias a su versatilidad, esta herramienta se ha aplicado a diversos problemas. Por ejemplo, se ha aplicado en la generación de pruebas para la composición de servicios web. En concreto, García-Fanjul [66] utiliza la herramienta SPIN como soporte para la generación de forma automática de pruebas para la composición de servicios web expresados en BPEL.

2.3. Técnicas basadas en el análisis estático

El análisis estático de un sistema software puede detectar defectos potenciales o reales, sin ejecutar el código fuente. Existen diferentes entornos de desarrollo (como por ejemplo Eclipse [163]), que son capaces de realizar un análisis estático, y en base a este análisis comprobar si existen defectos en el código fuente.

2.3.1. Análisis de los grafos de control de flujo

Para poder analizar un programa de forma automática, es necesario transformar el programa escrito en un lenguaje de programación concreto, a una representación independiente de la sintaxis y las peculiaridades de cada lenguaje. En la Ingeniería del Software se suele utilizar como representación intermedia los grafos de control de flujo. Los grafos de control de flujo muestran las acciones asociadas a un programa y la secuencialidad de éstas. Estos diagramas pueden aparecer también como parte de la documentación asociada a la implementación. Pueden tener diferentes niveles de abstracción, aunque generalmente suelen utilizarse a nivel de sentencias del código fuente, donde cada nodo representa una sentencia.

El análisis de los grafos de control de flujo, obtenidos de forma automática a partir del código fuente, permite detectar defectos en los programas [80]. Por ejemplo es posible detectar si determinadas partes de un programa no llegarán nunca a ejecutarse. Teniendo en cuenta que debe existir un punto inicial, si ninguna de las secuencias posibles de ejecución mostradas en el diagrama alcanza a un nodo concreto, las sentencias asociadas a dicho nodo nunca se ejecutarían.

Los diagramas pueden contener sólo información sobre las transferencias de control entre los nodos que forman un programa, o también pueden contener cierta información semántica, como por ejemplo información asociada a las variables utilizadas en el código fuente. De esta forma se podrían detectar defectos que dependen de las variables y de las transferencias de control, como por ejemplo condiciones en los bucles que siempre se cumplan, y que por tanto lleven a un bucle infinito.

Dentro del análisis de estos diagramas, es usual hablar del concepto de camino (Path). Dados dos nodos de un grafo de control de flujo un camino sería una secuencia permitida de nodos adyacentes del grafo.

2.3.2. Análisis de las dependencias en el código fuente

Como ya se ha visto anteriormente, con el análisis de los grafos de control de flujo es posible detectar defectos en los programas. Con el análisis de las dependencias de un programa, se puede lograr la detección de un mayor número de defectos. Las dependencias que se estudiarán serán de dos tipos: dependencias de datos, entre las variables de un programa; y de control, que vendrán dadas por el grafo de control de flujo.

El análisis estático de las dependencias persigue tres objetivos:

- Detectar defectos o anomalías potenciales o reales en un programa.
- Proporcionar una explicación de las relaciones y dependencias existentes en un programa.
- Determinar qué partes del código deben ser ejercitadas a través de pruebas (análisis dinámico).

El análisis de las dependencias puede detectar anomalías como la doble definición o sentencias redundantes. La doble definición hace referencia al caso en el que tras una primera asignación de una variable, ésta es de nuevo asignada por segunda vez, sin que haya habido ninguna lectura o utilización entre ambas asignaciones; y si en la segunda asignación no se ha utilizado el valor almacenado en la variable tras la primera asignación. Obviamente, la primera asignación no tiene sentido, ya que no tendrá ningún efecto en el resultado final del programa. Las sentencias redundantes son aquellas sentencias que no contribuyen en nada a los resultados generados por un programa. Es el típico caso de variables que son iniciadas, pero que nunca son utilizadas.

El análisis de las dependencias [59], tanto de datos como de control, puede generar información importante sobre el comportamiento de un programa, como por ejemplo:

- Cuáles son las variables que influyen en cada salida generada por un programa.
- Cuáles de las sentencias son responsables de los valores tomados por una variable.
- Si existe algún acceso a una variable que no haya sido inicializada.
- Qué sentencias deben ejecutarse antes de llegar a ejecutar una sentencia concreta.
- Si una sentencia es modificada, qué otras partes de un programa se ven afectadas.

Dentro del análisis de las dependencias de control, es importante el concepto de dominancia. Dados dos nodos v y w pertenecientes a un grafo de control de flujo, si v aparece en todos y cada uno de los caminos que desde el inicio del programa llevan a w, entonces v domina a w. En [107], Lengauer propone un algoritmo para obtener los dominadores de un grafo de control de forma eficiente. El algoritmo es de orden $\log(n)$, siendo n el número de nodos del grafo.

En general, el análisis estático debe aplicarse como primera técnica en la comprobación de un código fuente [103]. Una vez verificado el software de forma estática, y corregidos los defectos y anomalías encontrados, el siguiente paso es realizar el análisis dinámico, usando casos de prueba. El análisis de forma dinámica permitirá detectar defectos que sólo se producen en determinadas condiciones, es decir, para casos concretos de prueba.

2.3.3. Slicing

Las técnicas de Slicing [168] se basan en realizar un seguimiento de las dependencias que existen entre las asignaciones de variables incluidas en el código fuente. De esta forma se pueden deducir, o bien el conjunto de sentencias que influyen en un determinado punto del código fuente, o bien el conjunto de sentencias que dependen de una determinada sentencia. El termino Slice se puede traducir como una "rebanada" o conjunto de sentencias del código fuente.

Las técnicas de Slicing pueden aplicarse directamente sobre el código fuente, lo que se conoce como Slicing Estático [167], o sobre una traza concreta obtenida de la ejecución de un caso de prueba, lo que se conoce como Slicing Dinámico [1].

El conjunto de sentencias seleccionadas (Slice) dependerá del objetivo marcado:

- Se habla de *Forwared Slices* ("rebanamos" hacia delante) si se siguen las dependencias desde una sentencia A, hasta todas las sentencias que, o bien leen variables modificadas en A, o bien su ejecución puede estar influenciada por A. Formalmente se define como $\mathrm{S}^F(\mathrm{A}) = \{\mathrm{B}|\mathrm{A}\rightarrow {}^{+}\mathrm{B}\}$. Las sentencias no incluidas en $\mathrm{S}^F(\mathrm{A})$ no pueden ser influidas por A.

- Se habla de *Backward Slices* ("rebanamos" hacia atrás) si se siguen todas las dependencias hacia atrás, para encontrar qué sentencias pueden influir en una sentencia B. Formalmente se define como $\mathrm{S}^B(\mathrm{B}) = \{\mathrm{A}|\mathrm{A}\rightarrow {}^{*}\mathrm{B}\}$. Las sentencias no incluidas en $\mathrm{S}^B(\mathrm{B})$ no pueden influir en B.

Sobre los diferentes conjuntos de sentencias dependientes (Slices) se pueden realizar operaciones de combinación:

- Se denomina *Chop* a la intersección entre un Forward Slice y un Backward Slice. Permite conocer cómo una sentencia A (Forward Slice) influye sobre otra B (Backward Slice).

- Se denomina *Backbone Slice* a la intersección entre dos Backward Slices. Permite conocer qué parte del código influye en el cálculo de dos valores diferentes del código fuente. Cuando varios errores son detectados, esta operación permite determinar cuál sería el origen común de los errores detectados.

- Se denomina *Dice* a la diferencia entre dos Backward Slices. Permite conocer qué parte del código influye sólo en el cálculo del valor que ha dado lugar a un Backward Slice, pero no influye en el cálculo del valor que ha dado lugar al otro Backward Slice. Cuando sólo se ha detectado un error, este tipo de operación permite determinar qué sentencias han influido en dicho error, pero no en el resto de los resultados correctos. Este tipo de operación ha dado lugar a la metodología denominada en la bibliografía como Dicing [113]. En el trabajo [92], de M. Khalil, se muestra una comparativa de la metodología Dicing con otras técnicas.

Las técnicas de Slicing también pueden formularse como un problema de búsqueda, tal como se propone en el trabajo de Jiang [87], en el que el objetivo es localizar las dependencias estructurales de un programa. Tradicionalmente el análisis del flujo de información de un programa se ha basado en el análisis del flujo de control. Debido a la herencia, al polimorfismo, y otras características, estudiar el flujo de información en un programa orientado a objetos resulta aún más complejo. Por este motivo, para aplicar las técnicas de Slicing a la Programación Orientada a Objetos, resulta necesaria una adaptación. En [108] se proponen las pautas necesarias para esta adaptación. Además, se discuten diferentes coeficientes de correlación que permiten un manejo más eficiente del flujo de información de un programa. En el trabajo [26] se presenta una aproximación para adaptar las técnicas de Slicing a las dependencias que se pueden generar en un programa escrito en Java^{tm}. Esta aproximación permite incorporar a las técnicas de Slicing la posibilidad de distinguir entre datos de diferentes objetos, de manejar el polimorfismo, y de ser capaz de controlar enlaces dinámicos (constructores y llamadas a métodos).

En general las técnicas de Slicing Dinámico son más precisas que las de Slicing Estático, ya que en una traza concreta, se conocen todos los valores de las variables, y por tanto, se puede reducir más el tamaño de las slices. Pero existen dos desventajas inherentes al Slicing Dinámico, que no se dan en el Slicing Estático. La primera es la necesidad de conocer la secuencia de sentencias ejecutadas en la traza. La otra desventaja se debe a que al centrarse en una única traza, todos los cálculos realizados son válidos exclusivamente para dicha traza, y no se pueden generalizar para el resto de las trazas, tal como ocurre con el Slicing Estático, o en el análisis de las dependencias del código fuente.

Apoyándose en el Slicing Dinámico, la técnica de Dicing puede ser mejorada, tal como se propone en el trabajo de Wong [169]. Cuando se tienen dos casos de prueba que ejercitan un mismo código, pero uno provoca un error y el otro no, es lógico pensar que las sentencias que provocan el error deben estar en la traza donde se ha producido el error, pero no en la traza donde el comportamiento ha sido correcto. Utilizando la técnica de Dicing, se puede determinar la parte del código fuente que ha influido en el error, y que a su vez no ha influido en el caso de prueba donde el comportamiento ha sido correcto.

La técnica de Dicing se basa en suponer que siempre que exista un defecto, éste debe manifestarse en todas las trazas donde aparezca. Sin embargo esta suposición no siempre es cierta. Por ejemplo supongamos que una sentencia de suma de enteros ha sido sustituida por una resta de enteros. Ambas operaciones darían el mismo resultado si el elemento sumado o restado fuera el cero. Habrá por tanto, casos de prueba para los que el defecto no influye en el resultado final. Para dar respuesta a estos casos, en [169] se propone un método para aumentar el conjunto de sentencias (Dice) que pueden incluir el defecto buscado, en el caso de que las sentencias que proporciona la técnica de Dicing no lo incluyan. Estas nuevas sentencias estarían relacionadas con las sentencias que proporcionó la técnica de Dicing en un principio. Además, se propone otro método

para reducir el número de sentencias a revisar, en el caso de que la técnica de Dicing genere un conjunto excesivamente grande de sentencias a revisar.

En la línea del Dicing Dinámico, pero utilizando el análisis N-gram, es interesante el trabajo de Nessa [130]. En muchos casos, los errores detectados son debidos a una secuencia específica de sentencias. Esta secuencia será común a diferentes casos de prueba donde se hayan detectado errores, pero aparecerá en los casos de prueba donde no se hayan detectado errores. La metodología propuesta en [130] trata de encontrar estas secuencias, analizando a la vez las trazas de los casos de prueba en los que han aparecido errores y en los que no.

2.4. Técnicas basadas en el análisis dinámico

La generación de pruebas tiene como objetivo descubrir los defectos ocultos que se han producido durante el diseño y desarrollo del software [129]. Las pruebas del software requieren ya el 30 % de los recursos [79], siendo por tanto uno de los procesos más importantes en el desarrollo y mantenimiento del software. Como la mayoría de los defectos se introducen durante las fases iniciales de programación, es esencial evitar su propagación en la vida del producto [50].

Durante los años 80 aparecieron los primeros estándares sobre pruebas, como el IEEE/ANSI Standard 829-1983, y el Software Test Documentation y el IEEE/ANSI Standard 1008-1987, Software Unit Testing. Otros estándares relacionados son IEEE/-ANSI Standard 1012-1986, Software Verification and Validation Plans y el IEEE/ANSI Standard 730-1989, Software Quality Assurance Plans.

La validación a través de pruebas se suele dividir en diferentes fases [140][141]. En primer lugar se aplican las pruebas unitarias, luego las pruebas de integración de los diferentes subsistemas, las pruebas de validación de requisitos, las pruebas sobre el sistema en condiciones reales, y finalmente las pruebas de aceptación.

- **Pruebas unitarias**. Una prueba unitaria permite comprobar el correcto funcionamiento de un módulo de código. La idea es asegurar que cada uno de los módulos funciona correctamente por separado. Las pruebas unitarias tienen el fin de que el código fuente escrito cumpla el diseño especificado como correcto.

- **Pruebas de integración**. Tras las pruebas unitarias, con las pruebas de integración se asegura el correcto funcionamiento de los subsistemas. En estas pruebas se verifican en conjunto todos los elementos unitarios que componen una funcionalidad concreta o un proceso. De esta forma se puede comprobar que diferentes partes del software pueden funcionar en grupo.

- **Pruebas de validación de requisitos**. Se trata de validar las acciones y salidas del sistema que son visibles por los usuarios. Dichas acciones y salidas deben estar definidas en las especificaciones de los requisitos del software. Existen dos grandes

grupos de tipos de requisitos a validar. Por una parte los requisitos funcionales, y por otra parte los no funcionales.

- **Pruebas del sistema**. En este tipo de pruebas se trata de comprobar el funcionamiento del software en las condiciones reales, suponiendo que el software desarrollado forma parte de un sistema en funcionamiento, y las pruebas anteriores se han realizado en un entorno de simulación del sistema.

- **Pruebas de aceptación**. Por último, son los usuarios, ayudados por los desarrolladores o el equipo de pruebas, los que deben realizar las pruebas para comprobar si se cumplen los requisitos de los usuarios.

Este trabajo se centrará fundamentalmente en las pruebas unitarias. La metodología propuesta en esta tesis es un complemento a las pruebas unitarias que permite localizar los defectos que las pruebas unitarias detectan.

2.4.1. Pruebas Unitarias

Una prueba unitaria [134] es un procedimiento utilizado para comprobar que un módulo de un código fuente funciona correctamente en determinadas circunstancias. Un módulo es la unidad mínima de código fuente que se puede comprobar. Las pruebas unitarias pueden estar escritas por los propios desarrolladores. Por definición, este tipo de pruebas no tiene por objetivo descubrir errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto.

Las pruebas unitarias deben cumplir las siguientes características:

- Deben ser repetibles y automatizables, es decir, no deben requerir de la intervención humana. De esta forma si se introducen nuevos cambios en el código fuente, el propio entorno puede comprobar de forma automática si el sistema software sigue cumpliendo las pruebas establecidas. Por tanto, son también un medio eficaz de implementar y controlar pruebas de regresión. Las pruebas de regresión permiten comprobar si tras las modificaciones realizadas en el código fuente de un sistema software, el nuevo código fuente sigue cumpliendo los requisitos ya validados anteriormente. Para que el proceso sea óptimo se deben volver a comprobar las pruebas que tengan relación con los cambios realizados.

- Deben ser independientes y completas, es decir, cada prueba unitaria está desarrollada para verificar un módulo y su ejecución no debe afectar a la ejecución de otras pruebas.

Las pruebas unitarias permiten desarrollar un código de mejor calidad gracias a las ventajas que aporta:

- Fuerzan a que el desarrollo sea modular e independiente del resto, siendo el código fuente más robusto y resistente a cambios. Facilitan los cambios en el código y las pruebas de regresión, ya que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.

- Facilitan la separación de la interfaz y la implementación, ya que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas. Además, proporcionan un contrato escrito (las propias pruebas unitarias) que el módulo de código debe satisfacer y por tanto facilitan la documentación del código.

- Permiten llegar a la fase de integración con la confianza de que las unidades probadas funcionan correctamente. Las pruebas unitarias permiten aislar aquellos defectos que sólo afectan a determinadas unidades, y facilitan posteriormente las pruebas de integración.

Un buen conjunto de pruebas unitarias debe recoger la mayor parte de los posibles comportamientos de un sistema software. Cuantos más casos especiales de entradas sean tenidos en cuenta en las pruebas unitarias, mayor será la calidad de las pruebas unitarias y mayor será su efectividad. El objetivo es que las pruebas unitarias sean capaces de descubrir el mayor espectro posible de defectos que puedan presentarse en el código fuente.

Existen diferentes entornos dedicados a las pruebas unitarias. Para el lenguaje Java, el más utilizado y extendido es JUnit [118]. Fue creado por Erich Gamma y Kent Beck, basandose en SUnit, un entorno para realizar pruebas unitarias en el lenguaje Smalltalk. JUnit permite comprobar el funcionamiento de cada uno de los métodos de una clase. Para realizar las comprobaciones, el usuario debe establecer las entradas necesarias para la ejecución de un método y el resultado esperado, y el entorno comprobará si el resultado generado al ejecutar el método, es igual o diferente al esperado. Si es diferente, existe un defecto en el código del método.

El framework Korat [21] permite generar pruebas para programas escritos en el lenguaje de programación Java. Dada una precondición de un método, Korat genera las pruebas necesarias de forma automática. Después ejecuta cada uno de las pruebas y utiliza la postcondición como oráculo para verificar si la prueba es correcta o no. Los contratos deben ser compatibles con el lenguaje JML [106]. Para generar las pruebas, Korat explora de forma eficiente los valores posibles para cada una de las entradas, eliminando gran parte del espacio de búsqueda.

2.4.2. Desarrollo guiado por pruebas

En la metodología de desarrollo guiado por pruebas (Test Driven Development, TDD [4]), el desarrollo de un sistema software comienza por concretar las pruebas que dicho sistema software debe cumplir. Una vez establecidas las pruebas es cuando se continúa

con la generación del código fuente. El propósito del desarrollo guiado por pruebas es lograr un código que funcione y cumpla las pruebas establecidas. Si los requisitos son transformados a pruebas, y dichas pruebas se satisfacen, se puede concluir que los requisitos han sido implementados correctamente.

La metodología TDD está muy relacionada con las pruebas unitarias, ya que la idea es que las pruebas que deben dirigir el desarrollo puedan ser aplicadas de forma automática una vez generado el código fuente. Las pruebas unitarias suelen ser sencillas, y existen diversas herramientas (como por ejemplo JUnit [118]) que permiten la verificación del software de forma automática.

El ciclo de desarrollo dirigido por pruebas se basa en los siguientes pasos [8]:

1. Elegir un requisito, preferiblemente un requisito que no dependa de otros.

2. Escribir una prueba para dicho requisito. Para ello el programador debe tener claro la especificación del requisito a implementar. Cada prueba escrita debe poder ser ejecutada de forma automática.

3. Verificar si no se satisface la prueba. Si la prueba se satisface es porque el requisito ya está implementado o porque la prueba es defectuosa. Si la prueba es defectuosa es necesario volver a escribir la prueba.

4. Escribir el código fuente necesario para que la prueba funcione.

5. Comprobar que todo el conjunto de pruebas desarrollado se pude satisfacer, una vez introducido el nuevo código fuente.

6. Refactorización del código fuente para eliminar, por ejemplo, duplicidades. Tras la refactorización será necesario verificar que todas las pruebas son satisfechas (de forma automática).

7. Actualización de la lista de requisitos si han aparecido nuevos requisitos necesarios.

Como se puede observar, la metodología TDD impone que las pruebas desarrolladas puedan ser ejecutadas de forma automática. En muchos casos se utilizan pruebas unitarias, ya que resulta más sencillo automatizar su ejecución.

2.4.3. Generación de pruebas

La generación de pruebas puede hacerse de forma manual o automática. La generación de pruebas es un proceso costoso [11][58][131], de ahí que muchas de las líneas de investigación estén encaminadas a la automatización de estas técnicas. En cualquiera de los dos casos, la estrategia a seguir para seleccionar el conjunto de pruebas a realizar debe tener en cuenta tres factores: que permitan detectar un amplio espectro de

tipos de defectos, que el número de pruebas y su tamaño sean razonables, y que las funcionalidades del software a probar queden cubiertas con las pruebas seleccionadas.

Tradicionalmente las técnicas de generación de pruebas se han clasificado en: 1) técnicas de caja blanca (o pruebas estructurales), basadas en examinar la implementación, más concretamente la estructura de los programas, y en establecer criterios para la cubrir determinadas condiciones; y 2) técnicas de caja negra, que generan las pruebas basándose sólo en las especificaciones definidas para los datos de entrada y salida.

- *Técnicas de caja blanca*. Se basan en el conocimiento de la estructura de los programas. Por ejemplo, en [43], la generación de las pruebas se basa en la ejecución simbólica del programa, es decir, en sustituir variables de entrada por valores simbólicos (variables con dominios libres), y de forma estática ir evaluando los posibles dominios para las variables a lo largo de una trayectoria del flujo del control. Los valores que resulten en los dominios de las variables serían los utilizados como casos de prueba.

  Otro ejemplo es la metodología propuesta en [96], que se basa en la idea de que las diversas partes de un programa se pueden tratar como funciones que deben ser evaluadas al ejecutar el programa. Los valores de la función serán mínimos para esas entradas. Por tanto, el problema de generar pruebas se reduce a un problema de la minimización de una función.

  En [75] se propone la generación de los casos de prueba utilizando Model Checking. La novedad en este trabajo con respecto a otros trabajos anteriores radica en que las pruebas generadas permiten una mejor cobertura del código fuente, eliminando pruebas redundantes.

- *Técnicas de caja negra*. Pueden ser técnicas al azar, donde los datos para las pruebas se generan aleatoriamente de forma que permitan cubrir al máximo los dominios posibles de las variables de entrada [152]. Cuanto mayor es la complejidad del programa, más difícil es encontrar las pruebas apropiadas con las técnicas al azar.

  También pueden basarse en la especificación formal del tipo de entradas y salidas de un programa. En varios trabajos se utiliza la ejecución simbólica y la programación lógica con restricciones (CLP) [121] para generar de forma automática los datos de las pruebas en programas basados en la especificación con Z y VMDL-SL.

Cuando se realizan pruebas de caja negra, no se conoce la estructura del sistema software, y por tanto, resulta difícil saber las relaciones que existen entre las entradas suministradas y las salidas generadas por un sistema software. En el trabajo [116] se propone una metodología que permite inferir las relaciones entre las entradas suministradas y las salidas producidas por el sistema. La idea es generar una tabla de decisiones basada en el estudio de los diferentes casos de pruebas de caja negra disponibles. La

información contenida en esta tabla permite conocer qué entradas influyen en cada salida del sistema software. De esta forma, cuando se detecten errores en un conjunto de salidas, se podrán determinar cuáles son las entradas que han influido sobre dichas salidas. Una vez conocidas las entradas que han influido sobre los errores detectados, el proceso de depuración debería centrarse en las sentencias relacionadas directamente con dichas entradas.

La dificultad para generar pruebas de forma automática depende en gran medida de la posibilidad de disponer de un Oráculo que determine para las entradas generadas, cuáles son los resultados correctos. En el trabajo [3] por ejemplo, se generan pruebas de forma automática para localizar defectos en páginas PHP, utilizando como oráculo un navegador web. El resultado de la prueba es correcto si el navegador es capaz de interpretar correctamente lo que contiene la página PHP.

La Programación Orientada a Objetos introduce nuevas posibilidades a la hora de desarrollar software, como por ejemplo la herencia, el polimorfismo, y las interfaces. Como consecuencia, se añaden nuevos tipos de defectos que los programadores pueden introducir en el código fuente. Antes de la orientación a objetos cuando se comprobaba una función f sólo existía un comportamiento posible, pero en un programa orientado a objetos ese comportamiento puede variar, dependiendo de si se trata de un método perteneciente a una clase u otra, o una subclase. Es más, si por ejemplo se trata de una subclase, se tendrá que decidir si se debe comprobar de nuevo, aunque ya se haya comprobado su comportamiento en la clase padre. Para una sola llamada, es necesario explorar la unión de todos los posibles comportamientos. Por tanto, se hace necesario realizar adaptaciones a las técnicas para la generación de casos de prueba, comúnmente llamadas clásicas. R. V. Binder describe en su libro Testing Object-Oriented Systems [16] las adaptaciones que son necesarias para obtener un buen grupo de pruebas para software orientado a objetos.

2.4.4. Medidas de la calidad de las pruebas

Aunque no es posible garantizar con el uso de las pruebas que un software no tiene defectos, en general se considera que un software ha alcanzado un grado de calidad aceptable si se ha seguido una metodología de desarrollo adecuada y se han superado las pruebas establecidas. La generación de pruebas estructurales o de caja blanca suele estar asociada al concepto de cobertura de código. La cobertura de código es una medida que describe el grado en el cual determinadas áreas de una aplicación han sido ejercitadas por un conjunto de pruebas. El análisis de la cobertura garantiza que los elementos donde residen los defectos han pasado un determinado número de pruebas. Se basa en pruebas de caja blanca ya que requiere tener acceso al código fuente de la aplicación. Existen diferentes formas de medir la cobertura de un código, las más usadas son: la cobertura de sentencias, cobertura de decisiones, cobertura de condiciones, y cobertura de caminos o trazas.

La *cobertura de sentencias* informa si cada línea del código fuente de una aplica-

ción ha sido probada. Es también conocido como cobertura de bloques básicos. En la cobertura de bloques básicos la unidad mínima a probar es cada secuencia de sentencias sin ninguna sentencia condicional. Esta medida de cobertura intenta cubrir todas las sentencias, pero no tiene en cuenta si las condiciones establecidas en las sentencias condicionales y bucles son ejercitadas en todo el espectro de posibilidades. La *cobertura de decisiones* informa de cuando las expresiones de las sentencias de control (sentencias condicionales y bucles) son evaluadas. Cada una de las expresiones booleanas son consideradas como un predicado que puede tomar el valor verdadero o falso. La cobertura de decisiones incluye implícitamente la cobertura de sentencias ya que todas las sentencias son probadas en ambos casos. La *cobertura de condiciones* recoge el valor verdadero o falso de cada una de las subexpresiones que forman una condición, separadas por los operadores lógicos ("Y" u "O"). La cobertura de condiciones permite un análisis más fino, y por tanto garantiza un conjunto de pruebas más fuertes que en el caso de la cobertura de decisiones.

La *cobertura de caminos o trazas*, recoge si cada uno de los posibles caminos o trazas en cada uno de los métodos ha sido probado. Un camino o traza es una secuencia de sentencias ejecutadas y condiciones evaluadas desde la entrada al método hasta la salida. La cobertura de caminos o trazas incluye los resultados que se podrían obtener con la cobertura de decisiones. Para generar los casos de prueba que cubran cada uno de los posibles caminos, han surgido diferentes propuestas. Por ejemplo, Zhang [175] propuso una metodología para la generación automática de pruebas orientadas a la cobertura de caminos. La metodología se basa en la combinación de la ejecución simbólica del sistema software, y en la resolución de problemas de satisfacción con restricciones. El sistema software de entrada es transformado a una máquina finita de estados donde se realizará una búsqueda que cumpla las condiciones impuestas, y se ejecute el camino deseado. El objetivo es encontrar valores para las variables de entrada del sistema software que permitan alcanzar el estado final que se persigue comprobar.

El análisis de la cobertura es una medida de la calidad del conjunto de casos de pruebas, pero no permite garantizar la ausencia de defectos en el software probado. Es decir, es una medida de la calidad del conjunto de pruebas pero no del software probado. El hecho de que un programa funcione para un caso de prueba no garantiza que el código sea el correcto, es correcto para ese caso de prueba, pero no tiene por que ser correcto en otras condiciones. Por ejemplo, si tenemos un programa que realiza la multiplicación de dos números, y le enviamos como entradas uno y uno, dará como resultado uno. Si cambiamos la multiplicación por una división, el resultado sería también uno, pero evidentemente el código no sería el correcto. Para otros casos de prueba el defecto sería detectable, pero para el caso de prueba propuesto (entradas uno y uno) el defecto no produce un resultado erróneo. Por tanto, aunque un código es correcto para ciertas pruebas, no se puede garantizar la ausencia de defectos.

La herramienta Tarantula utiliza la información sobre la cobertura, pruebas fallidas y pruebas pasadas, y el código fuente, para proponer de forma automática donde se encuentran los defectos. Tal como explica Jones en su trabajo [89], la herramienta

supone que los defectos deben encontrarse en aquellas sentencias que se han ejecutado antes, y sobre todo en aquellas sentencias que participan más en los casos de prueba donde se han detectado errores. A diferencia de la técnica de Dicing [113], cuando una sentencia participa en una prueba donde no se han detectado errores, dicha sentencia no es exonerada, y puede identificarse como defectuosa si los resultados en otros casos de prueba inducen a ello.

Otra forma de medir la calidad de un conjunto de pruebas es utilizar las técnicas de mutación [78]. Las técnicas de mutación se basan en introducir defectos simples (pequeñas desviaciones del programa correcto) en un sistema software, con la intención de cubrir los diferentes fallos que podrían aparecer. Estos cambios vendrán dados por los operadores de mutación. Dependiendo del lenguaje tendremos diferentes operadores. Por ejemplo, para Fortran [94] se dispone de 22 operaciones de mutación, y para C [93] son 77 operaciones de mutación. Cuantos más defectos introducidos por las técnicas de mutación sea capaz de detectar un conjunto de casos de prueba, mejor será ese conjunto de casos de prueba, pues a priori es capaz de detectar un mayor espectro de tipos de defectos.

2.5. Depuración del software

Cuando se detectan errores en un sistema software, es necesario hacer una depuración del código fuente. La depuración tiene como objetivo aislar y localizar el defecto o defectos (usualmente llamados bugs) que impiden obtener el comportamiento esperado para un programa. La depuración es una actividad costosa en tiempo y recursos que suele hacerse de manera manual. En el proceso de depuración se suelen seguir tres pasos secuenciales:

- Primero se analiza la especificación y documentación asociada, intentando comprender cuál es el objetivo del sistema software, y luego se analiza el código fuente para conocer la implementación.

- En segundo lugar, tomando como base el comportamiento anómalo detectado y el análisis anterior sobre el sistema software, se lanza una hipótesis sobre la causa o causas del comportamiento anómalo.

- En tercer lugar, se realizan los cambios oportunos sobre las partes del código que se creen defectuosas, conforme a la hipótesis planteada en el paso anterior, y se vuelve a probar el sistema software en las mismas condiciones en las que fue detectado el mal comportamiento que llevo a la depuración.

Una vez completados los tres pasos, si el sistema no genera el resultado esperado, será necesario buscar otra hipótesis que permita explicar el comportamiento anómalo. Si el sistema genera el resultado esperado, para comprobar que no se han introducido

nuevos defectos con los cambios efectuados, será necesario volver a ejecutar los casos de prueba que afecten a las modificaciones realizadas.

Cuanto más precisa es la especificación y documentación de un sistema software, más eficaz podrá ser el proceso de depuración. Una buena especificación y documentación, ayuda a comprender al responsable de hacer la depuración cuál es el objetivo perseguido por el sistema software, y le permitirá formular hipótesis más acertadas sobre qué cambios deben realizarse para producir el comportamiento esperado del sistema software.

Se han propuesto diferentes metodologías para reducir el coste y mejorar el proceso de depuración del software. Estas metodologías se pueden dividir en dos grandes grupos. Como primer bloque estarían aquellas metodologías que proponen mejoras para que el proceso de diagnóstico sea más efectivo, pero que siguen defendiendo que el proceso debe ser interactivo. Y como segundo bloque, estarían aquellas metodologías que proponen la mejora en la eficiencia en base a la automatización de la depuración.

2.5.1. Depuración interactiva asistida

La idea del primer bloque de metodologías, basadas en la interacción, es mejorar la eficiencia en el proceso de depuración utilizando mejoras visuales o nuevas heurísticas. Por ejemplo, se puede ofrecer la información del proceso de depuración, de forma que la asimilación por el programador sea más rápida y eficaz, a través de una presentación visual (con gráficos) más intuitiva para el ser humano. La información presentada de esta forma permite tomar las decisiones más rápido.

Dentro de este primer grupo, por ejemplo, en el trabajo de Liang [109] se propone una metodología para monitorizar y depurar un sistema software a través de vistas sobre los comportamientos. Estas vistas, especifican el comportamiento esperado para determinados escenarios, y deben ser introducidas de por el desarrollador. De esta forma, al depurar el sistema software, se dispone de mas información sobre los efectos que provoca cada una de las rutinas o tareas que forman el sistema, y se puden aislar los defectos de forma más eficiente.

También dentro de este primer grupo, es interesante el trabajo de Caballero [23]. En él se propone, como apoyo en la depuración de un programa, la presentación visual del árbol de llamadas, donde el desarrollador puede ir añadiendo, de manera interactiva mientras se depura, si el valor generado por las llamadas es el esperado, y por tanto se descarta que el problema esté en esa llamada. Además, en este trabajo se propone una heurística, basada en la cobertura de las pruebas, que permite disminuir el número de preguntas que el desarrollador debe responder para llegar a localizar el defecto del sistema software.

Aunque los trabajos englobados en este primer bloque representan una mejora en el proceso de depuración, no dejan de ser interactivos. Existen otras propuestas que tratan de realizar el proceso de depuración de forma automática, y son los que se engloban en el segundo bloque de metodologías de depuración.

2.5.2. Depuración automática

Existen metodologías que tratan de automatizar la depuración de un sistema software. La mayoría de estas metodologías se basan en suponer que el comportamiento anómalo se debe a uno o varios elementos del sistema software, y proponen diferentes estrategias de búsqueda con la intención de localizar y aislar estos elementos defectuosos. Estas metodologías pueden ser un apoyo en el proceso de depuración, ayudando a la localización de los defectos; o pueden directamente producir una hipótesis sobre cuáles son los defectos.

Como metodologías más destacadas, podemos citar Delta Debugging y Model Based Debugging. La metodología Delta Debugging fue propuesta por Andreas Zeller [174]. El objetivo es el aislamiento de las causas de errores a través de reducir las diferencias (Deltas) entre diferentes trazas o ejecuciones. Más concretamente, Delta Debugging podría considerarse como una instancia del Testing Adaptativo, en el que la generación de cada prueba depende de los resultados obtenidos en las pruebas anteriores.

La idea es ir simplificando los casos de prueba eliminando los elementos irrelevantes, a través de pruebas sistemáticas sobre el código. Un elemento es irrelevante si el error detectado ocurre estando o no presente dicho elemento. Para poder aplicar la metodología de forma automática, es necesario definir una herramienta que permita verificar de forma automática las pruebas propuestas, y una estrategia para determinar qué elementos son relevantes o no. La estrategia más utilizada es la basada en la búsqueda binaria, en la que los elementos que forman la traza son divididos en dos subconjuntos (sin elementos en común).

La metodología ha sido aplicada a diversos sistemas y lenguajes. En [28] por ejemplo, es aplicada a programas escritos en el lenguaje C. En [36] se propone una herramienta que permite aislar las causas de errores en pruebas unitarias sobre programas escritos en el lenguaje Java.

Otras aproximaciones han intentado aprovechar las técnicas de diagnosis basada en modelos para realizar la depuración del software de forma automática. Es el caso del proyecto JADE (Java Diagnosis Experiments), de depuración de código basado en modelos (MBD, Model-Based Debugging). En los trabajos relativos a este proyecto [119][172], de los autores Mateis, Stumptner y Wotawa, se utiliza un modelo basado en el código fuente y sus conexiones. El modelo representa las sentencias y expresiones como si fuesen componentes de un sistema físico. En [171] Wotawa presenta una comparativa entre Slicing y MBD.

En el trabajo [120], Mayer y Stumptner presentan un estudio sobre diferentes tipos de técnicas para obtener el modelo del sistema software y poder aplicar MBD. Además se muestra una comparación empírica de dichas técnicas, utilizando diferentes ejemplos.

En el campo de la depuración automática en los últimos años han aparecido metodologías que permiten proponer soluciones a los defectos encontrados. No se trata de metodologías que permitan la reparación automática, ya que se limitan a proponer una serie de cambios, y es el programador el que debe elegir cúal es el cambio adecuado. Un

buen ejemplo de ellas es la propuesta de Wei y colaboradores [166] aplicada al lenguaje Eiffel. En dicho trabajo se propone la herramienta AutoFix-E que genera de manera automática posibles soluciones a los errores encontrados. Estas soluciones se basan en analizar los diagramas de estado y los contratos establecidos.

Capítulo 3

Técnicas adaptadas en la propuesta

En el anterior capítulo del estado del arte se han introducido los conceptos fundamentales del Diseño por Contrato y de las técnicas más utilizadas para comprobar y depuración el código fuente de un sistema software. El objetivo de esta tesis es diseñar una metodología que permita identificar de forma automática defectos semánticos en la especificación (cuando se detectan inconsistencias en los asertos que forman la especificación) y en el código fuente (cuando se detectan errores utilizando casos de prueba).

Para conseguir este objetivo, se adaptarán las técnicas utilizadas en las metodologías de Diagnosis Basada en Modelos (MBD, Model Based Diagnosis), de forma que puedan ser aplicadas al software. La Diagnosis Basada en Modelos permite encontrar de forma automática qué elementos están fallando cuando se produce un error en un sistema. Como soporte para la implementación y automatización de la metodología de diagnosis propuesta, se utilizará la Programación con Restricciones, por su eficiencia y facilidades a la hora de modelar problemas.

En este capítulo se hará un repaso de los principios en los que se sustenta la Diagnosis Basada en Modelos y la Programación con Restricciones, y se explicarán algunos conceptos y técnicas que serán utilizados o adaptados en los capítulos donde se explique con detalle la metodología propuesta en esta tesis.

3.1. Diagnosis Basada en Modelos

La mayoría de las aproximaciones aparecidas en la última década para diagnosticar sistemas se han basado en el uso de modelos. Estos modelos se apoyan en el conocimiento del sistema a diagnosticar, que puede estar bien estructurado formalmente y de acuerdo con teorías establecidas, o bien, puede conocerse a través de los conocimientos de un experto y datos del sistema o proceso. También se presentan algunas veces combinaciones de ambos tipos de información.

La localización de la causa de los errores se realiza a través del examen de un modelo. Los modelos están centrados en los componentes, y recogen el comportamiento correcto de cada uno de los componentes que forman el sistema, y las relaciones de entrada/salida

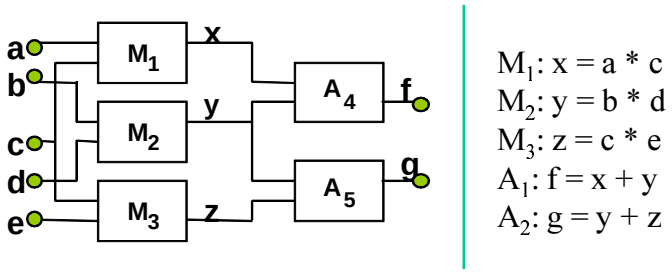


Figura 3.1: Sistema polybox básico (Toy Problem)

entre los componentes y el exterior. Además del modelo, es necesario colocar una serie de sensores para recoger medidas sobre el funcionamiento real del sistema.

La detección de los errores se basa en comparar los resultados esperados, proporcionados por el modelo, y los reales, proporcionados por los sensores. Si no existe discrepancia el sistema funciona correctamente, en caso contrario, estas discrepancias son el síntoma de un funcionamiento anómalo. La diagnosis es una hipótesis que explica la discrepancia entre el modelo y el comportamiento observado, y que propone los componentes que deben reemplazarse. Es decir, la diagnosis es una colección de conjuntos mínimos de componentes que fallan, y que explican los síntomas. La diagnosis basada en modelos tiene como objetivo explicar la causa que origina que un sistema bien diseñado no se comporta de la forma esperada.

Además del modelo del comportamiento correcto para el sistema, existen métodos de diagnóstico que requieren desarrollar modelos de fallos junto con el modelo de operación normal. Construir modelos de fallos en un sistema resulta apropiado cuando los fallos son bien conocidos y fáciles de modelar, pero también limita el sistema de diagnosis a fallos conocidos.

Ejemplo 3.1. En la figura 3.1 se muestra un sistema polybox que se puede considerar como el ejemplo estándar en la comunidad de diagnosis [38][68][39][32]. El sistema consiste en cinco componentes: tres multiplicadores y dos sumadores. Las restricciones que rigen cada uno de los cinco componentes aparecen en la parte derecha de la figura. En la parte izquierda se muestran las conexiones entre los componentes y los sensores. Los sensores se han representado usando círculos sombreados sobre la figura. Este sistema será utilizado en las siguientes secciones para aclarar conceptos referentes a las metodologías DX y FDI.

La diagnosis basada en modelos es un campo muy activo de investigación. Existen dos comunidades que trabajan en paralelo en este campo: la comunidad FDI (Fault Detection and Isolation, Localización y Detección de Fallos) proveniente del área del control automático, y la comunidad DX (Acrónimo inglés de la palabra Diagnosis) enmarcada en el área de la Inteligencia Artificial. Aunque ambas comunidades trabajan de forma normalmente aislada, se han propuesto diversos trabajos para la integración

de las técnicas utilizadas en ambas comunidades. La integración de las teorías de FDI con DX y las pruebas de sus equivalencias se han mostrado para varios supuestos en [32][33][144][18].

Una revisión de las aproximaciones acerca de la automatización de las tareas de diagnosis se puede encontrar en [45] y para una discusión de las aplicaciones de la diagnosis basada en modelos se puede consultar [31]. La generalización de la diagnosis basada en la consistencia para cubrir sistemas dinámicos se propuso en [81].

En los dos siguientes apartados se hará una revisión de los conceptos básicos de las metodologías DX y FDI.

3.1.1. Definiciones y conceptos de la metodología DX

En el área DX, el trabajo pionero de Davis [38] presenta una aproximación que permite diagnosticar sistemas de componentes usando su estructura y comportamiento. Las primeras implementaciones para diagnosis fueron DART [68] y GDE [95], que utilizaban diferentes formas de inferencia para detectar los posibles fallos. La formalización de la diagnosis se concretó en los trabajos de Reiter [149] y deKleer [39], donde se propone una teoría general para el problema de explicar las discrepancias entre los comportamientos observados y los definidos como correctos para los componentes.

El primer paso para aplicar la metodología DX es definir y concretar el modelo del sistema.

Definición 3.1. Modelo del Sistema. Se llama Modelo del Sistema (MS) al par (DS, COMPS) donde: COMPS (Componentes) será el conjunto finito de componentes; y DS (Descripción del Sistema) será un conjunto de ecuaciones lógicas que definen el comportamiento correcto de los componentes. Las interconexiones entre los diferentes componentes se pueden deducir a partir del modelo del sistema.

La definición de diagnosis se sustenta en el concepto de comportamiento anormal. Básicamente si un componente tiene un comportamiento no anormal, entonces su funcionamiento es correcto. En la descripción del sistema se hará uso del predicado AB para identificar si un componente tiene un comportamiento anormal o no. Dado un componente c perteneciente a COMPS, si el predicado $\neg$AB(c) es verdadero, entonces el componente funciona correctamente. En caso contrario se dice que tiene un comportamiento anormal.

En la tabla 3.1 aparece el modelo del sistema para el ejemplo de la figura 3.1. El conjunto de ecuaciones lógicas de igualdad que forman el SD, describen el comportamiento de los diferentes componentes (sumadores y multiplicadores), y establece las interconexiones entre ellos. Por ejemplo, la ecuación lógica asociada a los sumadores expresa que si un componente es un sumador y no tiene un comportamiento anormal, entonces la salida del sumador es la suma de las entradas. Sin embargo, no se establece cómo se comportará el sumador si su comportamiento es anormal.

COMPS:	A1, A2, M1, M2, M3		
SD:	ADD(c) $\wedge\neg$AB(c) $\Rightarrow$ out(c) = in1(c) + in2(c) MULT(c) $\wedge\neg$AB(c) $\Rightarrow$ out(c) = in1(c) + in2(c) ADD(A_1), ADD(A_2), ADD(A_3), MULT(M_1), MULT(M_2) out(M_1) = in(A_1), out(M_2) = in(A_1), out(M_2) = in(A_2), out(M_3) = in(A_2), in(M_1) = in(M_3)	$\Longrightarrow$	$\neg$AB(A_1) $\Rightarrow$ f = x + y $\neg$AB(A_2) $\Rightarrow$ g = y + z $\neg$AB(M_1) $\Rightarrow$ x = a * c $\neg$AB(M_2) $\Rightarrow$ y = b * d $\neg$AB(M_3) $\Rightarrow$ z = c * e
OBS:	MO_1={a=3, b=2, c=2, d=3, e=3, f=10, g=12} MO_2={a=3, b=2, c=2, d=3, e=3, f=10, g=10}		

Tabla 3.1: Modelo DX para el ejemplo de la figura 3.1

Nombrando las entradas como *a*,*b*,*c*,*d*, y *e*, y las salidas como *f* y *g* tal como aparece en la figura 3.1, es posible reducir la descripción del sistema a su equivalente que aparece en la misma tabla 3.1 a la derecha. Los sensores son los encargados de monitorizar el valor que toman determinadas conexiones o puntos del sistema. Estos sensores están asociados a variables del modelo del sistema. Las variables que tienen asociado un sensor se denominan variables observables, mientras que las que no tienen ningún sensor asociado se denominan no observables.

Para completar el problema de diagnosis, el siguiente paso es recoger, a través de los sensores, los datos reales del sistema. El conjunto de datos recogidos es lo que se denomina Modelo Observacional (OBS). En la tabla 3.1 aparecen ejemplos de OBS para el sistema propuesto.

Definición 3.2. Problema de Diagnosis. Se denomina Problema de Diagnosis a la tripleta (DS, COMPS, OBS).

Para detectar si existen errores, basta poner en marcha el sistema para unas determinadas entradas (recogidas en el modelo observacional), y comparar los resultados esperados, proporcionados por la descripción del sistema (DS), y los reales, proporcionados por los sensores (OBS). El objetivo, una vez detectado que existe un problema, es determinar la diagnosis mínima, es decir, qué componentes deben ser cambiados para que el sistema vuelva a generar los resultados esperados.

Definición 3.3. Diagnosis. La diagnosis para (DS, COMPS, OBS) será un conjunto de componentes $\mathcal{D} \subseteq$ COMPS tal que SD $\cup$ OBS $\cup$ $\{$AB(c) | c $\in \mathcal{D}\}$ $\cup$ $\{\neg$AB(c) | c $\in$ COMPS-$\mathcal{D}\}$ se puede satisfacer.

El número de posibles diagnosis es exponencial, concretamente 2^{COMPS}. El objetivo es refinar el número de posibles diagnosis, y seleccionar un número menor de posibilidades sin perder información. Para lograrlo de manera eficiente, la metodología DX localizará los conjuntos mínimos de componentes que fallan, y que explican los síntomas. En concreto, la diagnosis de componentes se basa en detectar cuáles son los conflictos, es decir, los conjuntos de componentes que no funcionan bien conjuntamente.

Definición 3.4. Conjunto Conflictivo. Será un conjunto de componentes C = {c_1, ..., c_k} ⊆ COMPS tal que SD ∪ OBS ∪ {¬AB(c) | c ∈ C} es inconsistente.

Definición 3.5. Conjunto Conflictivo Mínimo (CCM). Es aquel conjunto conflictivo que no incluye en ningún otro conjunto conflictivo.

Para el problema de diagnosis mostrado en la tabla 3.1 y el modelo observacional MO_1, los conjuntos conflictivos mínimos serían {M_1, M_2, A_1} y {M_1, M_3, A_1, A_2}; y para MO_2 serían {M_1, M_2, A_1} y {M_2, M_3, A_2}.

Para obtener los conjunto conflictivos se suele utilizar un motor de inferencia lógica [149], y luego seleccionar cuáles son los CCM. Otros autores proponen utilizar la programación con restricciones, como en [161] y [153]. En [37] ser propone el análisis de la estructura de componentes que forman el modelo.

Definición 3.6. Conjunto Mínimo Afectado o Hitting Set. Es un conjunto de componentes intersección de los conjuntos conflictivos mínimos. El conjunto mínimo afectado (*Minimal Hitting Set*) será un conjunto de componentes que incluirá un componente de cada uno de los conjuntos conflictivos mínimos.

Definición 3.7. Diagnosis Mínima. Es aquella diagnosis $\mathcal{D}$ que cumple que $\forall\ \mathcal{D}' \subset \mathcal{D}$, $\mathcal{D}'$ no es una diagnosis. Para calcular las diagnosis mínimas para un problema de diagnosis (SD, COMPS, OBS), se hará uso de la siguiente propiedad: $\mathcal{D}$ es una diagnosis mínima si y sólo si, es un conjunto mínimo afectado de la colección de conjuntos conflictivos mínimos para (SD, COMPS, OBS).

Para el cálculo de los Hitting Sets se han propuesto diversos algoritmos. Para una revisión de los más importantes pueden consultarse los trabajos [110] y [170].

Para el modelo observacional MO_1 aplicado al ejemplo de la tabla 3.1, se obtendrían las siguientes diagnosis mínimas: {M1},{A1},{M2, M3}, y {M2, A2}. Estas cuatro opciones posibles son el conjunto de diagnosis mínimas. Cada una contiene un conjunto de componentes que deben modificarse, ya que se supone tienen un comportamiento anormal. Cualquiera de estas cuatro opciones puede explicar la discrepancia entre el modelo y el comportamiento observado. La diagnosis mínima explica la causa por la cual un sistema bien diseñado no se comporta de la forma esperada. En el caso del modelo observacional MO_2, las diagnosis mínimas serían: {M2}, {M1, M3}, {M3, A1}, {M1, A2}, y {A1, A2}.

3.1.2. Definiciones y conceptos de la metodología FDI

La metodología FDI (Fault Detection and Isolation) es una teoría madura que se ha ido formalizando en diferentes trabajos; los más representativos son los trabajos [60][85][135], que son ampliamente citados por el resto de investigadores del campo. Se basa en teorías y técnicas provenientes de la comunidad de Control.

De forma análoga a la metodología DX, el modelo del sistema se obtiene directamente del comportamiento esperado del sistema. Dicho modelo tiene en cuenta cuáles son

Relaciones:	Sensores:	OBS:
M_1: x = a * c	S_a: a = a_{obs}	$a_{obs} = 2$
M_2: y = b * d	S_b: b = b_{obs}	$b_{obs} = 2$
M_3: z = c * e	S_c: c = c_{obs}	$c_{obs} = 3$
A_1: f = x + y	S_d: d = d_{obs}	$d_{obs} = 3$
A_2: g = y + z	S_e: e = e_{obs}	$e_{obs} = 2$
	S_f: f = f_{obs}	$f_{obs} = 10$
	S_g: g = g_{obs}	$g_{obs} = 12$

Tabla 3.2: Modelo FDI para el ejemplo de la figura 3.1

las conexiones entre componentes (modelo estructural) y el comportamiento de cada componente.

Definición 3.8. Modelo del Sistema. Es el conjunto de ecuaciones que definen la conducta del sistema en función de relaciones. Las ecuaciones del sistema contienen información sobre el comportamiento esperado (correcto) de los componentes, y sobre las relaciones entre las variables del sistema y los sensores del sistema.

En la tabla 3.2 aparece el modelo del sistema para el sistema polybox de la figura 3.1. Está formado por cada una de las ecuaciones que modelan los componentes M_1, M_2, M_3, A_1 y A_2, y las relaciones entre el juego de sensores establecido y las variables del sistema.

Definición 3.9. Problema de Diagnosis. Es la unión del modelo del sistema (MS), un conjunto de observaciones (OBS) recogidas por los sensores, y un conjunto de fallos (F).

Un fallo es un conjunto de componentes defectuosos. Los fallos se clasifican en simples y múltiples, los simples incluyen sólo un componente y los múltiples incluyen más de un componente. Suponiendo que los sensores no pueden fallar, el conjunto de posibles fallos simples será n, el número de componentes del sistema. El número de posibles fallos (simples y múltiples) será 2^n, ya que potencialmente cualquier conjunto de subconjunto de componentes puede fallar.

La base para la detección y la localización del origen de los errores en la metodología FDI son las relaciones de redundancia analítica.

Definición 3.10. Relación de Redundancia Analítica (Analytical Redundancy Relation, ARR). Será una relación obtenida del modelo del sistema de tal forma que sólo contiene variables asociadas a sensores.

Para detectar si existen fallos, la idea es sustituir los valores recogidos por los sensores en las ARRs. El resultado de la evaluación de una ARR se denomina *residuo*. Si el residuo es igual a cero se dice que las observaciones satisfacen la relación ARR. La obtención de las ARRs se basa en la búsqueda de los sistemas sobre-determinados

ARR	F_{A1}	F_{A2}	F_{M1}	F_{M2}	F_{M3}
1	1	0	1	1	0
2	0	1	0	1	1
3	1	1	1	0	1

Tabla 3.3: Matriz de firmas del sistema polybox para fallos simples

de ecuaciones del modelo, en donde es posible eliminar las variables desconocidas (sin sensor asociado). Este problema suele formalizarse a través de un grafo bipartito donde se localizan emparejamientos completos entre las variables desconocidas que se quieren eliminar.

Los trabajos [24][159] presentan la formalización del análisis estructural, el proceso de obtención de las relaciones de redundancia analítica de un sistema. Dentro de la metodología DX, los trabajos de Krysander [98], Pulido [145][143], y Huang [83], ofrecen otras técnicas diferentes al análisis estructural, que permiten calcular las relaciones de redundancia de forma off-line (fuera de línea). En [57], Fattah propone una aproximación a la metodología FDI pero utilizando un modelo de restricciones lógicas para la resolución de la diagnosis.

Las relaciones ARR obtenidas para un modelo del sistema son válidas para todos los conjuntos de observaciones posibles. Es decir, fijado un sistema, sólo es necesario calcular una vez las relaciones ARR. Este trabajo puede hacerse off-line y previamente al proceso de diagnóstico. Esta es una de las mayores diferencias con respecto a la metodología DX, donde casi todo el trabajo debe realizarse on-line (en línea), y una vez detectado el problema.

Para el ejemplo del sistema polybox de la figura 3.1, suponiendo que los sensores funcionan correctamente, las ARRs que se obtendrían serían:

ARR_1: $r_1 = f_{obs}$ - a_{obs} c_{obs} - b_{obs} d_{obs} (Componentes: A_1, M_1, M_2)
ARR_2: $r_2 = g_{obs}$ - b_{obs} d_{obs} - c_{obs} e_{obs} (Componentes: A_2, M_2, M_3)

Además de estas ARRs elementales, es posible conseguir nuevas relaciones de redundancia combinando las anteriores relaciones. Estas nuevas relaciones permiten discernir mejor los fallos posibles y conseguir así una diagnosis más precisa. Combinando las relaciones anteriores es posible obtener una tercera relación en base a los componentes A_1, A_2, M_1, M_3:

ARR_3: $r_3 = f_{obs}$ - g_{obs} - a_{obs} c_{obs} + c_{obs} e_{obs} (Componentes: A_1, A_2, M_1, M_3)

Es muy importante resaltar que esta nueva relación no necesita la unión de todos los componentes de las dos ARRs que la generan, que serían ARR_1=$\{A_1, M_1, M_2\}$ y ARR_2=$\{A_2, M_2, M_3\}$.

ARR	F_{A1A2}	F_{A1M1}	F_{A1M2}	F_{A1M3}	F_{A2M1}	F_{A2M2}	F_{A2M3}	F_{M1M2}	F_{M1M3}
1	1	1	1	1	1	1	0	1	1
2	1	0	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1

Tabla 3.4: Matriz de firmas del sistema polybox para fallos múltiples

Definición 3.11. Firma de fallo. Dado un conjunto de n relaciones ARR, ARR = {ARR_1, ARR_2, ..., ARR_n}, y un conjunto de m fallos F = {F_1,...,F_m}, la firma de fallo F_j viene dada por $FS_j = [s_{1j},..., s_{nj}]^T$ en la cual $s_{ij} = 1$ si el conjunto de componentes que genera la restricción ARR_i contiene los componentes incluidos en el fallo F_j, en otro caso $s_{ij} = 0$.

Definición 3.12. Matriz de Firmas. Será la unión de todas las firmas para el conjunto de posibles fallos.

La tabla 3.3 muestra la matriz de firmas para fallos simples en el sistema polybox de la figura 3.1, y la tabla 3.4 muestra la matriz de firmas para fallos múltiples. La parte correspondiente a los fallos simples responde a los fallos en los componentes M_1, M_2, M_3, A_1 y A_2. Cada fallo múltiple F_j, que incluye n componentes, será obtenido usando el fallo F_k, que incluye $n-1$ componentes, y un fallo simple F_s cuyo componente asociado no esté incluido en los componentes asociados al fallo F_k. Las firmas de fallos múltiples F_j vendrán dadas por $FS_j = [s_{1j},..., s_{nj}]^T$ en donde $s_{ij} = 0$ si $s_{ik} = s_{is} = 0$, y $s_{ij} = 1$ en otro caso. La generación de la matriz de firmas se detendrá cuando se alcancen las 2^{m-1} columnas, donde m es el número de posibles fallos simples.

Definición 3.13. Firma de un Modelo Observacional. Vendrá dada por OS=[OS_1, ..., OS_n] donde OS_i=0 si cada restricción ARR_i es satisfecha, y OS_i=1 en otro caso.

Definición 3.14. Diagnosis. Será el conjunto de fallos cuyas firmas son consistentes con la firma del modelo observacional. Lo usual en la metodología FDI es suponer que una firma de un modelo observacional OS es consistente con otra firma FS_j si $OS_i = s_{ij}$ para todo i.

Para el ejemplo del sistema polybox de la figura 3.1, en función de las firmas de los modelos observacionales, las diagnosis serían:

Si OS = [1, 0, 1] entonces $\mathcal{D}$ = {A_1 o M_1}}
Si OS = [0, 1, 1] entonces $\mathcal{D}$ = {A_2 o M_3}}
Si OS = [1, 1, 0] entonces $\mathcal{D}$ = {M_2}
Si OS = [1, 1, 1] entonces $\mathcal{D}$ = {Cualquier fallo múltiple excepto {A_1, M_1} o {A_2, M_3}}

En la metodología FDI, el proceso de diagnosis se puede hacer en su mayoría fuera de línea, sólo la interpretación de la matriz de firmas debe hacerse en línea (on-line),

justo cuando se dispone del modelo observacional. La generación de la matriz de fallos y los pasos anteriores no dependen del modelo observacional, a diferencia de la metodología DX, que desde un principio usa el modelo observacional para activar el motor de inferencia lógica y de esta forma detectar los conflictos.

Otra diferencia notable con DX es la no utilización de un motor de inferencia lógica para calcular la diagnosis. Para calcular la diagnosis, FDI no necesita el motor de inferencia, ya que ese trabajo es sustituido por el análisis y generación de las relaciones tipo ARR.

3.2. Problemas de satisfacción de restricciones

Un problema de satisfacción de restricciones se basa en la especificación de un conjunto de relaciones que debe cumplir cualquier solución al problema planteado. En este paradigma, en lugar de especificar los pasos para obtener la solución, lo que se especifica son los datos iniciales y las condiciones que la solución debe cumplir. La programación con restricciones permiten tratar un amplio abanico de problemas reales [6], como por ejemplo problemas de planificación, scheduling, razonamiento temporal, de diseño de sistemas, etc. Para su implementación se han extendido técnicas provenientes de la inteligencia artificial, la lógica, y la investigación operativa.

La resolución de un problema de satisfacción de restricciones consta de dos fases:

1. Modelar el problema mediante un conjunto de variables, dominios y restricciones.
2. Procesar el problema de satisfacción de restricciones mediante técnicas de consistencia y/o de búsqueda. Las técnicas de consistencia se basan en la eliminación de valores inconsistentes reduciendo el espacio de posibles soluciones. Los algoritmos de búsqueda se basan en la exploración del espacio de posibles soluciones.

En los siguientes apartados se introducirán los conceptos y técnicas necesarias para la modelización, el mantenimiento de la consistencia y la búsqueda de soluciones.

3.2.1. Conceptos básicos del modelado con restricciones

La programación con restricciones permite modelar y resolver problemas reales como un conjunto de restricciones entre variables.

Definición 3.15. Problema de Satisfacción de Restricciones (Constraint Satisfaction Problems, CSP). Se define como la tripleta <X, D, C> donde X es un conjunto de variables X=$\{x_1, x_2..., x_n\}$ asociadas a unos dominios, D=$\{dx_1, dx_2,..., dx_n\}$, y a un conjunto de restricciones C=$\{C_1, C_2,..., C_m\}$.

Definición 3.16. Restricción. Una restricción C_i es una tupla (W_i,R_i), donde R_i es una relación $R_i \subseteq D_{i1}\times...\times D_{ik}$ definida para el subconjunto de variables $W_i \subseteq X$, que restringe los valores que las variables pueden tomar.

La aridad de una restricción es el número de variables que componen dicha restricción. Una restricción unaria es una restricción que consta de una sola variable, cuando tiene dos variables se denomina binaria.

La búsqueda de soluciones para un CSP se basa en el concepto de *instanciación*.

Definición 3.17. Instanciación de una variable. Es la *asignación* de un valor a una variable. La instanciación de un valor v a una variable x se suele representar como un par variable-valor (v, x).

El concepto de instanciación o asignación hace referencia a una única variable. Cuando la asignación es sobre varias variables se utiliza el término tupla. Una *tupla* $((x_1, v_1), ..., (x_i, v_i))$ de una restricción es una asignación sobre las variables que forman una restricción.

Definición 3.18. Consistencia. Una tupla $((x_1, a_1), ..., (x_i, a_i))$ es *consistente* con una restricción si satisface dicha restricción.

La consistencia puede tener ámbito local o global. Aunque estos conceptos se explicarán con más detalle más adelante, a continuación se adelanta su definición:

Definición 3.19. Consistencia local. Una tupla $((x_1, v_1), ..., (x_i, v_i))$ es *localmente consistente* si satisface todas las restricciones que incluyen variables de la tupla.

Definición 3.20. Consistencia global. Una tupla $((x_1, v_1), ..., (x_i, v_i))$ es *globalmente consistente* si para todo x_i, v_i se cumple que $x_i = a_i$ forma parte de la solución del problema CSP.

Definición 3.21. Solución a un CSP. Es una asignación de valores que permite satisfacer todas las restricciones del problema, es decir, una solución es una tupla consistente que contiene todas las variables del problema. Una solución parcial es una tupla consistente que contiene algunas de las variables del problema.

Cuando se plantea un CSP, puede haber diferentes tipos de objetivos a demostrar o comprobar. Por ejemplo, el objetivo puede ser comprobar si un CSP se puede satisfacer para unos valores concretos de ciertas variables o de forma general; es decir, determinar si existe al menos una solución que cumple todas las restricciones planteadas. Un problema es consistente, si existe al menos una solución, es decir una tupla consistente. Una vez encontrada la solución, se podrían obtener los valores concretos asignados a las variables en la solución. En otras ocasiones el objetivo puede ser recoger todas las soluciones posibles, o una solución óptima en base a un determinado criterio de optimización (generalmente expresado por una función objetivo que se debe maximizar o minimizar).

Ejemplo 3.2. Uno de los problemas típicos en la programación con restricciones es la coloración de un mapa. En la figura 3.2 se muestra un mapa de Andalucía con sus

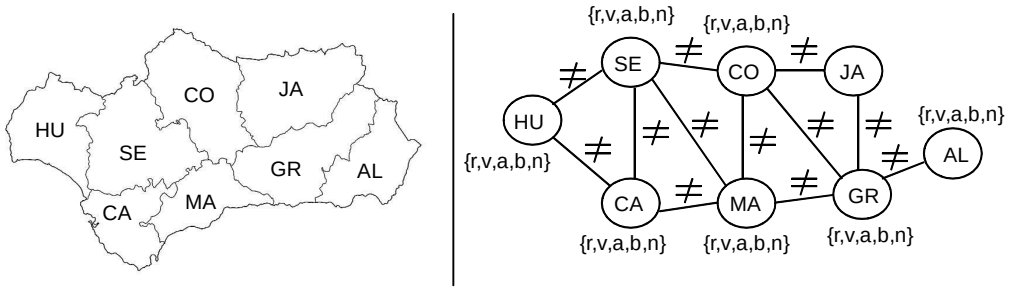


Figura 3.2: Modelo CSP para el coloreado de un mapa

diferentes provincias. En este problema hay un conjunto de colores que deben colorear cada provincia del plano de tal manera que las provincias adyacentes tengan distintos colores. Para formular el CSP podemos declarar una variable por cada provincia del mapa, siendo el dominio de cada variable el conjunto de colores disponible. Es este caso se han escogido 5 colores, rojo, verde, azul, blanco, y negro (r, v, a, b, n). Para cada par de provincias adyacentes se establecerá como restricción que los colores asociados a las provincias deben ser diferentes, es decir, que las variables asociadas a dos provincias colindantes no deben contener el mismo valor en la solución. En este problema son necesarios al menos 5 colores para obtener una solución. Si el número de colores se reduce a 4 o menos, el problema no tendría solución.

3.2.2. Consistencia

Encontrar una asignación a las variables de un CSP de forma que todas las restricciones se satisfagan simultáneamente es, en el caso general, un problema NP-Completo. En la resolución de un CSP se utilizan técnicas de consistencia y de búsqueda. Los algoritmos de búsqueda se basan en la exploración del espacio de posibles soluciones, y tienen un coste exponencial en el caso peor.

Con la idea de reducir el espacio de posibles soluciones que debe explorar un algoritmo de búsqueda, las técnicas de consistencia se deben utilizar previamente a la aplicación de los algoritmos de búsqueda, y durante la ejecución de dichos algoritmos. Las técnicas de consistencia se basan en eliminar valores concretos de las variables, o combinaciones de éstos, que no pueden participar en la solución porque no satisfacen alguna restricción. Se denominan inconsistencias locales a aquellas que sólo tienen en cuenta un subconjunto de las restricciones del problema completo. Si un algoritmo de búsqueda no tiene en cuenta las inconsistencias locales que se generan al combinar las restricciones de un CSP, se perdería tiempo al explorar posibilidades que no pueden llevar a una solución.

Por ejemplo, para cumplir la restricción que obliga a que la variable y debe ser mayor a la variable z, se pueden eliminar del dominio de la variable y todos aquellos valores

que sean menores o iguales, al menor de los valores del dominio de la variable z. Estos valores eliminados del dominio de la variable y nunca podrían formar parte una solución del problema, ya que incumplen la restricción establecida para los valores permitidos de la variable z. Habrá valores, que aunque no hayan sido eliminados del dominio de la variable y, serán inconsistentes con respecto a otras restricciones. A diferencia de la consistencia local, en la consistencia global, todos los valores que no pueden participar en una solución serían eliminados.

Consistencia local

Existen diferentes técnicas de consistencia local para eliminar valores de las variables. Freuder presentó una noción genérica de consistencia llamada (i, j) - consistencia [62]. Un problema es (i, j)-consistente si cualquier solución a un subproblema con i variables puede ser extendido a una solución incluyendo j variables adicionales. La mayoría de las formas de consistencia se pueden ver como especificaciones basadas en la (i, j)-consistencia. La k-consistencia hace referencia al caso en el cual i es k-1 y j vale 1. En [40] aparecen definiciones formales de algunas de las consistencias locales que más utilizadas. La implementación de estas consistencias locales en algoritmos puede verse en [100].

Como casos particulares de la k-consistencia local [6], pueden definirse: la consistencia de nodo (1-consistencia), la consistencia de arco (2-consistencia), la consistencia de caminos (3-consistencia).

- La nodo-consistencia obliga a que todos los valores en el dominio de una variable deben satisfacer todas las restricciones unarias sobre esa variable. Es decir, un problema es nodo-consistente, si y sólo si, todas sus variables son nodo-consistentes: $\forall\ x_i \in X$, $\forall\ C_i$, $\exists a \in D_i$: a satisface C_i.

- La arco-consistente obliga a que dado un par cualquiera de variables restringidas x_i y x_j, para cada valor a en D_i debe haber al menos un valor b en D_j tal que las asignaciones (x_i, a) y (x_j, b) satisfacen la restricciones entre x_i y x_j. Es decir, un problema es arco-consistente si y sólo si todos sus arcos son arco-consistentes: $\forall C_{ij} \in C, \forall a \in D_i, \exists b \in D_j$ tal que b es un soporte para a en C_{ij}, es decir los valores a y b hacen consistente la restricción C_{ij}. Cualquier valor en el dominio D_i de la variable x_i que no es arco-consistente puede ser eliminado de D_i ya que no puede formar parte de ninguna solución que cumpla las restricciones asociadas a x_i. Los algoritmos de arco-consistencia son de orden polinómico. Los más importantes son:

  - AC-3. Se trata del algoritmo más conocido para arco-consistencia y fue propuesto por Mackworth en [114]. El algoritmo principal es un bucle en el que se revisan los arcos mientras se reducen los dominios de las variables para mantener la consistencia entre las restricciones. Cuando no hay más cambios el algoritmo se detiene. Para evitar realizar revisiones innecesarias en

arcos donde ya se mantiene la arco-consistencia, algo que ocurría en los algoritmos AC-1 y AC-2 anteriores, se añadió una lista que mantiene los pares variable-restricción para los cuales no está garantizada la arco-consistencia.

- AC-4. Fue propuesto por Mohr y Henderson [127][128] con la idea de mejorar la complejidad y el tiempo de ejecución del algoritmo AC-3. El algoritmo AC-4 realiza un procesado previo, y almacena bastante más información que el algoritmo AC-3. De esta forma se consigue evitar realizar muchos chequeos sobre las restricciones, que el algoritmo AC-3 se ve obligado a realizar. Aunque AC-4 mejora teóricamente el tiempo de ejecución de AC-3, aunque esto no es así en todos los problemas, debido principalmente a que en algunos problemas no es posible realizar la etapa de procesado previo en un tiempo asumible.
- AC-6. Fue propuesto por Bessiere y Cordier [13][12] con la intención de ser un algoritmo intermedio entre AC-3 y AC-4. La idea es tener un algoritmo que mantenga una estructura de datos más ligera que AC-4, y que pueda ser aplicado a más problemas, como en el caso de AC-3.
- AC-2001. Los algoritmos AC-4 y AC-6 mantienen la consistencia mediante la propagación de valores, AC-2001 [14][15] mantiene la consistencia mediante la propagación de los cambios; y utilizando una lista con los pares variable-restricción. Aunque el algoritmo es menos preciso en cuanto a la propagación, presenta como ventaja frente a los anteriores algoritmos AC-4 y AC-6, que es más fácil de implementar con respecto a las herramientas que existen para la resolución de restricciones.

- La consistencia de caminos obliga a por cada par de valores a y b de dos variables x_i y x_j, que la asignación de a a x_i y de b a x_j satisfaga la restricción entre x_i y x_j, y que además exista un valor para cada variable a lo largo del camino entre x_i y x_j de forma que todas las restricciones a lo largo del camino se satisfagan. Es decir, un problema satisface la consistencia de caminos si y sólo si todo par de variables (x_i, x_j) es camino-consistente: $\forall(a, b) \in C_{ij}$, $\forall x_k \in X$, $\exists c \in D_k$ tal que c es un soporte para a en c_{ik} y para b en c_{jk}.

Consistencia global

Los algoritmos de consistencia global tratan de mantener solamente aquellas combinaciones de valores que formen parte de al menos una solución, por tanto implican una relación más fuerte que la consistencia local. Para ello filtran combinaciones de valores inconsistentes. Si se mantiene la consistencia global, sólo se almacenan valores que pueden llevar a una solución, y por tanto se evita la búsqueda de las soluciones inconsistentes (reduciendo la complejidad, y por consiguiente el tiempo de la búsqueda). Mantener la consistencia global suele ser un proceso exponencial en el peor caso.

Para clases especiales de problemas, es posible mantener la consistencia global usando algoritmos polinónicos. Por ejemplo [6]:

- La Arco-consistencia es equivalente a la consistencia global cuando la red de restricciones es un árbol [61].
- La consistencia de caminos es equivalente a la consistencia global cuando el CSP es convexo y binario [41][10].

3.2.3. Algoritmos de búsqueda

Los métodos de búsqueda de soluciones CSP permiten instanciar las variables dentro de sus dominios y así cumplir las restricciones del problema. La búsqueda consiste en explorar el espacio de estados del problema hasta encontrar una solución, o bien demostrar que no existe solución. La búsqueda puede ser completa si el recorrido del espacio de estados es sistemático, o incompleta si se utilizan otras estrategias como por ejemplo la búsqueda local. Los algoritmos de búsqueda se pueden completar con diferentes heurísticas, que en determinados problemas, pueden mejorar los tiempos de búsqueda de forma significativa.

Algoritmos de búsqueda sistemática

Este tipo de búsquedas se caracteriza por el recorrido del espacio de estados de forma sistemática y completa, generalmente siguiendo un algoritmo basado en backtracking. El espacio de estados de un problema se suele representar mediante un árbol de búsqueda. Cada camino que une un nodo con la raíz representa un conjunto de variables que han sido asignadas. El árbol puede ser recorrido de diferentes formas, aunque lo más habitual es su recorrido en profundidad ya que su complejidad espacial es lineal y acotada por el número de variables, tal como hace el algoritmo de backtracking.

El algoritmo de backtracking, en cada nodo del árbol de búsqueda comprueba si se satisfacen las restricciones cuyo conjunto de variables ha sido asignado. Si las restricciones se satisfacen, el algoritmo sigue la búsqueda en profundidad. Pero si no se satisfacen, el algoritmo realiza una vuelta atrás sobre la última variable asignada. Si en el recorrido del árbol son asignadas todas las variables entonces se considera que se ha alcanzado una solución.

El algoritmo deja abierto el orden en el que deben instanciarse las variables y el orden en el que los valores de los dominios de las variables deben ser escogidos. Es a través de heurísticas donde se definen estos órdenes. En cada clase de problema tendrá más sentido un tipo u otro de heurística.

El algoritmo de backtracking tiene diferentes variantes entre las que se pueden destacar los que a continuación se detallan.

- En el *Backtracking Cronológico* cuando se detecta una inconsistencia en la rama actual del árbol de búsqueda, se vuelve sobre la última variable asignada para cambiar su valor. Es el método más sencillo, pero el menos optimizado.

- Los algoritmos *Look-Back* tratan de reforzar el comportamiento del backtracking cronológico mediante un comportamiento más inteligente cuando dan vuelta atrás. Son algoritmos de este tipo: *Backjumping* [67], *Conflict-directed Backtracking* [142] y *Backtracking Dinámico* [69].

    - El algoritmo *Backjumping* en lugar de retroceder a la variable anteriormente instanciada, cuando da vuelta atrás salta a la variable más profunda x_j que está en conflicto con la variable actual x_i donde $j < i$. Una variable x_j está en conflicto con una variable x_i si la instanciación de x_j evita uno de los valores de x_i debido a una restricción entre x_i y x_j.
    - El algoritmo *Conflict-Directed Backtracking* tiene un comportamiento de salto hacia atrás más sofisticado que Backjumping, que se basa en guardar el conjunto de conflictos que cada variable ha tenido con otras variables en el pasado.

- Los algoritmos Look-Ahead llevan a cabo comprobaciones para detectar las posibles inconsistencias con las variables que se van a instanciar más adelante. La idea es identificar antes las situaciones en las que es necesario dar vuelta atrás. Son algoritmos de este tipo: *Forward Checking* [77] y *Minimal Forward Checking* [44].

    - El algoritmo *Forward Checking* en cada etapa de la búsqueda comprueba hacia adelante la asignación actual con todos los valores de las futuras variables que están restringidas con la variable actual. Los valores de las futuras variables que son inconsistentes con la asignación actual son temporalmente eliminados de sus dominios. Si el dominio de una variable futura se queda vacío, la instanciación de la variable actual se deshace y se prueba con otro valor aplicando bactracking cronológico. En cada etapa las asignaciones realizadas son consistentes con al menos un valor de cada variable futura.
    - El algoritmo *Minimal Forward Checking* es una version mejorada de la anterior, que reduce el número de comprobaciones a realizar, ya que sólo comprueba la asignación actual con los valores de cada variable futura hasta que se encuentra una que es consistente. En general este algoritmo mejora al anterior, pero la diferencia de tiempo sólo es significativa en algunos problemas.

Para una revisión de estos algoritmos puede consultarse [115].

Algoritmos de búsqueda local

Los algoritmos de búsqueda local intentan optimizar una función objetivo. Trabajan con asignaciones totales, que incluyen a todas las variables. Dada una asignación total inicial, el proceso itera para obtener en cada paso una mejor asignación, hasta llegar al óptimo. Al ser algoritmos incompletos no garantizan encontrar una solución, y puede darse el caso de que un estado puede ser visitado varias veces. Tampoco garantizan que no exista una solución en el caso de que no la encuentren. Aún con sus limitaciones, son muy utilizados en problemas de optimización, debido a su mayor eficiencia y el alto coste que requiere una búsqueda completa.

Existen diversos algoritmos de búsqueda local, como por ejemplo los algoritmos genéticos [71] y la búsqueda tabú [65]. En general todos tienen en común una serie de elementos. Debe existir una función de coste que asigne a cada asignación total un valor numérico, en base a lo óptima que sea la solución. También la propiedad de vecindad, que dado un estado determina a que conjunto de estados se puede mover la siguiente iteración. Y por último un criterio de selección, que dada una vecindad y una función de coste, elige el estado siguiente al actual entre su vecindad.

3.2.4. Problemas sobrerrestringidos

Cuando no hay soluciones para un CSP, una opción puede ser eliminar ciertas restricciones, y tratar de resolver el nuevo CSP. Estás restricciones, que son prescindibles si no existen soluciones, almacenan condiciones que para el usuario son recomendables pero no obligatorias. Para poder automatizar el proceso en el que el usuario iría eliminando restricciones hasta encontrar una solución, se han propuesto extensiones que permiten trabajar con dos tipos de restricciones: restricciones *duras*, y *blandas*. Las restricciones blandas denotan preferencias, no obligación, que pueden tener diferentes semánticas, tales como prioridades, preferencias, costes, o probabilidades. Mientras que las restricciones duras, son condiciones de obligado cumplimiento.

Las restricciones blandas también pueden ayudar a seleccionar la mejor solución. Cuando se resuelve un CSP, el usuario puede escoger entre detenerse cuando se encuentre una solución o un número concreto de soluciones, o recoger todas las soluciones. Una vez recogidas las soluciones el usuario utilizara aquéllas que se adapten mejor a lo que busca. El problema está cuando el número de soluciones es alto, y resulta tedioso y lento hacer un procesamiento manual de las soluciones recogidas para evaluar cuáles son las más adecuadas. Para resolver este problema, se pueden utilizar restricciones blandas a modo de preferencias, y de esta forma priorizar unas soluciones frente a otras.

Para poder expresar las restricciones blandas con diferentes objetivos, se han propuesto diferentes extensiones que permiten expresar prioridades [156], grados de preferencia [48], costes [63] o probabilidades [56]. La resolución de un CSP con restricciones blandas transforma un CSP en un problema de optimización de tipo NP-Hard. En los problemas de optimización con restricciones (Constraint Optimization Problem, COP), el objetivo es encontrar la mejor solución, y el criterio de preferencia entre unas solu-

ciones y otras viene especificado por las restricciones blandas y una función que debe ser optimizada.

Existen varios modelos que permiten razonar sobre restricciones blandas. Podemos destacar los Valued CSP (VCSP) [156] y los Semiring CSP (SCSP) [17]. El modelo SCSP es algo más general porque permite definir problemas en los que el conjunto de soluciones tan solo esté parcialmente ordenado. El modelo VCSP es conceptualmente más simple, y es el tipo de modelo que se usará en la implementación de la metodología que se presenta en esta tesis. Por este motivo, esta sección en adelante se centrará en los VCSP.

En los VCSP [101] las restricciones son funciones de coste que expresan el grado de satisfacción de las posibles asignaciones parciales. El conjunto de costes se denomina E y debe estar totalmente ordenado. También se tiene una operación $\oplus$ sobre E, que debe ser conmutativa y asociativa. La operación $\oplus$, llamada suma o agregación, permite combinar costes locales provenientes de diferentes restricciones, y de esta forma obtener costes globales con los que poder ordenar el conjunto de soluciones. Los elementos máximo y mínimo de E se denominan $\top$ y $\bot$, respectivamente. El modelo VCSP trabaja con problemas de minimización, por tanto los valores bajos de E son preferibles a valores altos. El valor $\bot$ se usa para representar la satisfacción máxima, mientras que $\top$ representa la satisfacción mínima.

Un CSP con restricciones blandas se define como la tripleta <X, D, C> donde X es un conjunto de variables X=$\{x_1, x_2..., x_n\}$ asociadas a unos dominios, D=$\{dx_1, dx_2,..., dx_n\}$, y a un conjunto de restricciones (blandas y duras) C=$\{C_1, C_2,..., C_m\}$. Cada restricción C_i es una función sobre un subconjunto de las variables $var(c) \subseteq X$. Para cada tupla t que asigne valores a las variables de $var(c)$, $c(t)$ devuelve un elemento de E que indica el grado de satisfacción de la tupla t. La restricción será dura si para toda tupla t, $c(t) \in \{\top, \bot\}$, y por lo tanto se satisface o no, pero no hay grados intermedios. Resolver un VCSP es encontrar la mejor solución, es decir una solución con valuación mínima. Los elementos del conjunto E deben poder ordenarse, y de esta forma poder comparar los diferentes niveles de preferencia. La operación $\oplus$ debe ser commutativa y asociativa, de esta forma se podrán realizar operaciones sobre los valores de las funciones (valuaciones) sin importar el orden.

Dependiendo del conjunto E y de la operación $\oplus$ se tendrá un modelo diferente de CSP con restricciones blandas [101]:

- El CSP Clásico se da cuando las restricciones son funciones booleanas que tan sólo distinguen entre tuplas permitidas o no. En este caso E = {cierto, falso} (siendo cierto < falso) y u $\oplus$ v = u $\wedge$ v.
- En el CSP Posibilístico [155] las restricciones devuelven valores en el intervalo [0, 1]. En este caso E = [0, 1] y u $\oplus$ v = max{u, v}.
- En los CSP con Peso (WCSPs, [156]) E es el conjunto de los naturales y $\oplus$ es la suma. Un caso especial del WCSP es aquel en el que todas las restricciones de-

vuelven costes en {0,1}, que corresponde con el modelo Max-CSP que se usará en la implementación de la metodología propuesta en esta tesis.

- En los CSP Probabilísticos [56] $E = [0, 1]$ y u $\oplus$ v = 1 - (1 - u)(1 -v).

Al igual que en los CSP clásicos, existen dos formas de implementar las búsquedas de las soluciones: la búsqueda sistemática o completa, y la búsqueda local. En el caso de la búsqueda completa, se puede utilizar una mejora sobre la búsqueda backtracking, la técnica de ramificación y poda [63]. Durante la búsqueda, el algoritmo mantiene el coste de la mejor solución encontrada hasta el momento, y usa este valor como cota. En cada avance, el algoritmo comprueba si el camino seguido es capaz de mejorar la cota, y da vuelta atrás si no es posible mejorar dicha cota.

Para mejorar la eficiencia en la búsqueda de soluciones para un Max-CSP, muchos investigadores han realizado diferentes propuestas. Como ejemplo de estas mejoras se pueden consultar las referencias de Larrossa [102] y Kask [91].

Parte III

Propuesta

Capítulo 4

Metodología de diagnosis

Tal como se expuso en la introducción, en esta tesis se propone una metodología para la identificación de defectos semánticos. Estos defectos estarán localizados en los asertos que forman la especificación, y en las sentencias que forman el código fuente:

- En primer lugar, y sólo para los asertos que forman la especificación, se comprobará si la especificación es consistente. Si no es consistente se identificarán, de manera automática, cuáles son los defectos de la especificación que lo impiden.

- En segundo lugar, suponiendo que la especificación es correcta, se comprobará si la ejecución del código fuente produce resultados diferentes a los establecidos como correctos. Si es así, se identificarán cuáles son los defectos que impiden alcanzar los resultados correctos. Al igual que en el caso de la especificación, la identificación de los defectos en el código fuente también será de manera automática.

En este capítulo se presentarán las bases de la metodología comunes a ambos objetivos. En la metodología se adaptarán conceptos de la metodología DX de Diagnosis Basada en Modelos. El capítulo se ha estructurado de la siguiente manera. Primero se mostrarán las adaptaciones propuestas para la metodología DX con el fin de que los conceptos utilizados en dicha metodología puedan ser reutilizados en la metodología propuesta en esta tesis. Luego se introducirá la metodología propuesta (Diagnosis del Software Basada en Modelos), y las fases de las que consta. Para cada una de las fases se expondrán cuáles son las tareas a realizar y los objetivos a cumplir.

Una vez fijadas las bases de la metodología, y las fases de las que consta, en los siguientes dos capítulos, la metodología propuesta será ampliada y adaptada para cumplir con los dos objetivos marcados, el diagnóstico de la especificación, y del código fuente.

4.1. Adaptaciones necesarias en la metodología DX para el diagnóstico del software

Para poder aplicar la metodología DX es necesario disponer de un modelo que recoja el funcionamiento correcto del sistema. Los errores se detectan al comparar, para un conjunto de entradas, el comportamiento real con el esperado. El comportamiento real viene definido por los valores tomados por los sensores, mientras que el comportamiento esperado vendrá dado por los valores generados por el modelo.

Para adaptar la metodología DX al software, es necesario determinar qué se utilizará como modelo. Para el caso del software, los elementos que pueden ser utilizados como modelo son: el código fuente, la especificación y los casos de prueba. Tal como se ha dicho anteriormente, la metodología propuesta en esta tesis tiene dos objetivos diferentes, por tanto, el modelo a utilizar dependerá del objetivo a cubrir:

- En primer lugar debe ser capaz de diagnosticar los defectos semánticos en la especificación, si se detectan inconsistencias. En este caso, los asertos de la especificación no pueden considerarse como parte del modelo, pues son los elementos a diagnosticar. Los asertos de la especificación establecen condiciones que deben cumplirse. Sin embargo, al ser diseñados por humanos, dichas condiciones pueden contener defectos.

  La especificación debe cumplir ciertas propiedades para ser consistente. Por ejemplo, todos los invariantes de una clase deben poder cumplirse a la vez, y no deben ser excluyentes. Estas propiedades sí forman parte del comportamiento esperado y pueden ser utilizadas como modelo. También los casos de prueba que se utilicen en la comprobación de la especificación pueden ser utilizados como modelo del comportamiento esperado.

- En segundo lugar, la metodología propuesta debe pasar a localizar los defectos semánticos en el código fuente. El código fuente es un modelo del comportamiento real, pero no del comportamiento correcto, ya que precisamente es en el código fuente donde pueden estar localizados los defectos.

  Los casos de prueba que se utilicen para validar el código fuente proporcionan los resultados esperados para ciertas entradas, por tanto, sí permiten tener una visión parcial (sólo para las entradas establecidas) del modelo del funcionamiento correcto del sistema. La especificación también puede ser utilizada como modelo del comportamiento esperado, ya que en ella se establecen limitaciones que deben cumplirse por parte de los resultados generados por el código fuente.

Además de las diferencias a la hora de obtener el modelo, también hay diferencias con respecto al tipo de defectos. En la metodología DX, son los componentes defectuosos los que, al no funcionar correctamente, provocan los errores. Para eliminar los errores, los componentes defectuosos deben ser sustituidos por otros iguales o equivalentes.

Característica	Metodología DX	Diagnosis de los asertos de la especificación	Diagnosis de las sentencias del código fuente
¿Cómo detectar los errores?	Comparando los valores tomados de los sensores (comportamiento real) con los generados por el modelo (comportamiento esperado)	Comprobando que los asertos que forman la especificación son consistentes entre sí (comportamiento real) para determinados casos (comportamiento esperado)	Comparando los valores producidos por el código fuente (comportamiento real) con los establecidos en el caso de prueba (comportamiento esperado)
¿Qué origina el error?	Defectos de funcionamiento, uno o varios componentes defectuosos no se comportan de la forma esperada	Defectos de diseño, uno o varios asertos de la especificación, no son los adecuados	Defectos de diseño, una o varias sentencias del código fuente, no son las adecuadas
¿Cuál es la solución al error?	Cambiar los componentes por otros iguales, pero que funcionen correctamente	Cambiar los asertos por otros diferentes para obtener el comportamiento esperado	Cambiar las sentencias por otras diferentes para obtener el comportamiento esperado

Figura 4.1: Principales diferencias entre la Metodología DX y la Diagnosis del Software Basada en Modelos

En la Diagnosis del Software Basada en Modelos, los defectos a diagnosticar son de tipo semántico. Se deben a un mal diseño, es decir, los elementos a cambiar funcionan correctamente, pero no son los adecuados y deben ser modificados o cambiados por otros diferentes para alcanzar el resultado esperado. Por ejemplo, si la ejecución del código fuente no genera los resultados esperados para un caso de prueba, será debido a que el diseño del código fuente tiene defectos, y por tanto, alguna de las sentencias debe modificarse.

En la figura 4.1 se muestran, de forma resumida, las características y diferencias reseñadas anteriormente para la metodología DX y la Diagnosis del Software sobre la especificación, y sobre el código fuente.

4.2. Metodología propuesta

La metodología de Diagnosis Basada en Modelos DX se basa en utilizar un modelo denominado Descripción del Sistema. El proceso de diagnóstico razona en base a este modelo, que recoge el diseño correcto del sistema, y que dadas unas entradas concretas, es capaz de generar las salidas correctas que del sistema se deben obtener.

El sistema sobre el que se aplicará la Diagnosis del Software se denominará Sistema Software.

Definición 4.1. Sistema Software. Se denomina Sistema Software a la tupla (STS, SPEC), donde STS es el conjunto finito de sentencias que forman el código fuente, y

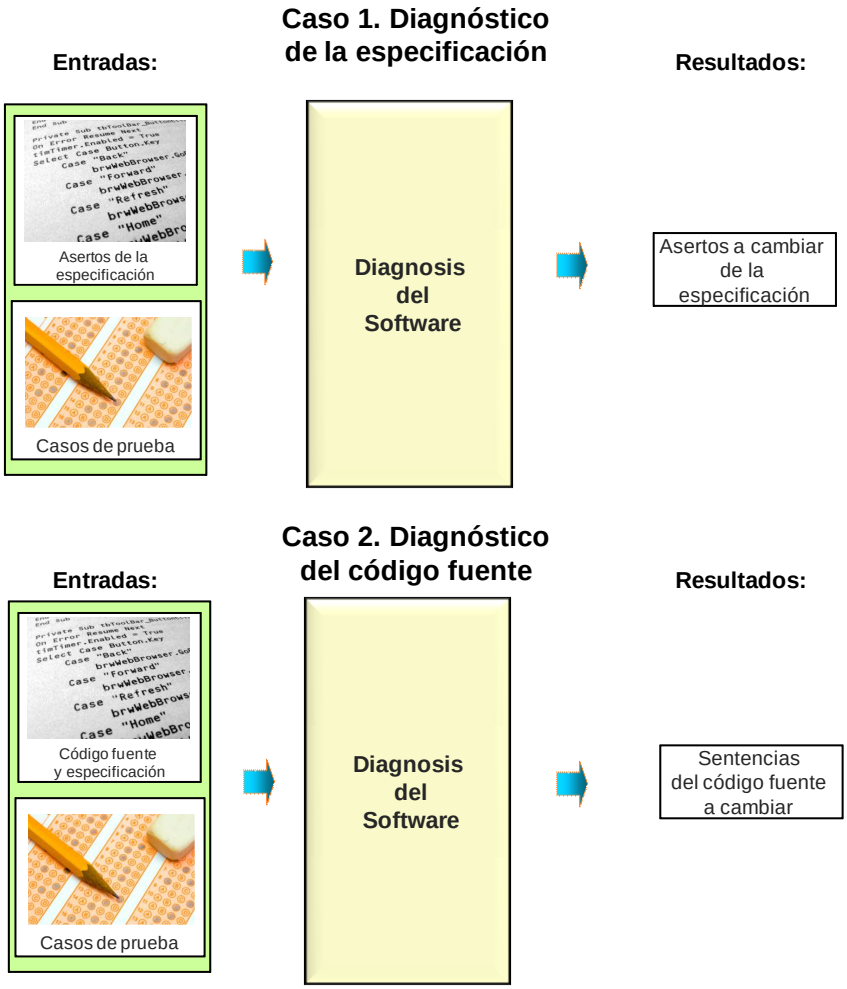


Figura 4.2: Aplicaciones de la Diagnosis del Software Basada en Modelos

SPEC es el conjunto de asertos que forman la especificación. El conjunto SPEC establece el comportamiento esperado de los elementos que forman el conjunto STS.

Cada uno de los elementos que forman los conjuntos STS y SPEC, estarán unívocamente identificados dentro del sistema. La metodología de diagnosis propuesta permite diagnosticar defectos semánticos en dos tipos de elementos: en los asertos que forman la especificación, y en las sentencias que forman el código fuente. Tal como aparece en la figura 4.2, para cada tipo de elemento a diagnosticar, la metodología recibe unas entradas concretas, y genera unos resultados específicos:

- Caso 1. En el diagnóstico de la especificación la metodología recibe como entradas los asertos que forman la especificación y los casos de prueba disponibles para la comprobación de la especificación. En este caso los elementos a diagnosticar, es

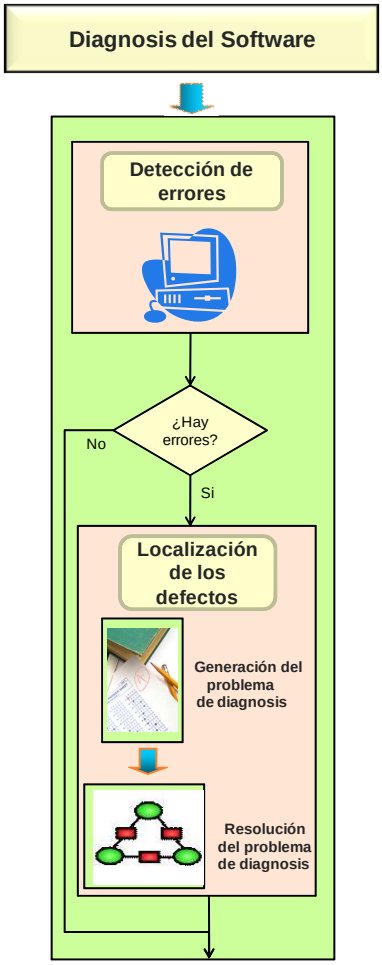


Figura 4.3: Fases de la metodología propuesta

decir, los elementos que pueden contener defectos, serán cada uno de los elementos que forman parte del conjunto SPEC. La metodología genera como resultado, aquellos asertos que deben cambiar para poder cumplir las propiedades y casos de prueba establecidos.

- Caso 2. En el diagnóstico del código fuente la metodología recibe como entradas las sentencias que forman el código fuente, los asertos que forman la especificación y los casos de prueba disponibles para la verificación del código fuente. En este caso los elementos a diagnosticar, es decir, los elementos que pueden contener defectos, serán cada uno de los elementos que forman parte del conjunto STS. Los elementos que forman el conjunto SPEC se supondrán correctos. La metodología genera como resultado, aquellas sentencias que deben cambiar para poder cumplir las propiedades y casos de prueba establecidos.

Para resolver ambos casos, la metodología propuesta sigue siempre dos fases secuenciales que se muestran en la figura 4.3. Las dos fases son: detección de errores y localización de los defectos. En los siguientes apartados se explicarán con más detalle cada una de estas fases. A continuación, a modo de resumen, se exponen los objetivos de cada una de las fases:

- Detección de los errores. En esta fase se comprobará de forma automática si aparecen errores en el sistema software. Existen errores si hay discrepancias entre el comportamiento real, que viene dado por el sistema software, y el comportamiento esperado, que viene dado por los casos de prueba o por propiedades establecidas.
- Localización de los defectos. Para aquellos casos en los que se han detectado errores, se generará un problema de diagnosis que permitirá localizar los defectos que han originado los errores detectados. El problema de diagnosis permite modelar a través de restricciones el comportamiento real y esperado del sistema software. Para modelar el comportamiento real las sentencias y asertos serán transformadas a restricciones, y los casos de prueba a restricciones sobre el dominio de las variables. La resolución del problema de diagnosis permitirá localizar cuáles son los defectos del sistema software. La resolución de dicho problema será a través de un problema de optimización de restricciones (COP, Constraint Optimization Problem).

En las siguientes secciones, se mostrarán con más detalle cada una de las fases presentadas.

4.3. Detección de errores

Para detectar si existen errores se comparará el comportamiento real, que viene dado por el sistema software, y el comportamiento correcto, que viene dado por los casos de prueba o por propiedades conocidas. Las discrepancias entre el comportamiento real y el correcto se considerarán errores. En este trabajo se proponen dos tipos de casos de prueba: los casos de prueba concretos, y los casos de prueba vacíos.

- Los casos de prueba concretos establecen condiciones iniciales para las que el caso de prueba es válido. Si se cumplen dichas condiciones iniciales el caso de prueba establece cuáles son los resultados correctos que el sistema software debe generar. Existe un error cuando los resultados son diferentes a los establecidos en el caso de prueba. Por ejemplo, supóngase que se quiere comprobar un método de una clase que recibe un valor entero y devuelve un valor entero. Un caso de prueba concreto establecería para una entrada concreta, cuál es el valor correcto que dicho método debe generar. Si aplicando la entrada establecida en el caso de prueba se obtiene un valor diferente del correcto, se considerará un error que tendrá su origen en algún defecto en el código fuente de dicho método.

- Se dice que un caso de prueba es vacío si no establece ninguna condición inicial para el sistema software. En los casos de prueba vacíos, el objetivo es comprobar si existen inconsistencias entre los elementos que forman el sistema software sin establecer condiciones iniciales. Este tipo de pruebas tiene sentido al comprobar los elementos que forman la especificación (asertos).

  Por ejemplo, supóngase que se quiere comprobar que todos los invariantes de una clase pueden cumplirse a la vez, sin establecer condiciones iniciales. Para ello se verificaría, con un caso de prueba vacío, la validez de los invariantes de una clase en conjunto. Supóngase que en uno de los invariantes de dicha clase se establece que una variable debe ser positiva, y más adelante en otro invariante, se establece que dicha variable debe ser menor o igual que cero. Ambos invariantes no podrán satisfacerse a la vez, aún cuando el caso de prueba vacío no ha establecido ninguna condición inicial, y por tanto, debe existir un defecto en el diseño de dichos invariantes. Los invariantes forman parte de la especificación de una clase, y deben cumplirse antes y después de ejecutar cada método.

Ambos tipos de casos de prueba tienen sus ventajas y desventajas. Cuando se usan casos de prueba concretos, los elementos defectuosos se manifiestan en unas condiciones concretas (las correspondientes a cada caso de prueba), y no tienen por qué manifestarse en otras condiciones. Hay defectos que no pueden ser detectados utilizando un caso de prueba vacío, ya que sólo aparecen en determinadas condiciones. Por otra parte, si la detección de los errores se realiza con un caso de prueba concreto, dicha verificación sólo es válida para dicho caso de prueba. Además, si el resultado generado fuera correcto, sólo se garantiza que el diseño del sistema es correcto para ese caso de prueba, no necesariamente para cualquier otro caso de prueba concreto.

Aunque no es posible garantizar que un software no tiene defectos con el uso de los casos de pruebas, en general, se considera que un software tiene un grado de calidad aceptable si se ha seguido una metodología de desarrollo y se han superado las pruebas necesarias. Así por ejemplo, el análisis de la cobertura, visto en el apartado 2.4.4, garantiza que los elementos susceptibles de tener defectos han pasado un determinado número de pruebas. En esta tesis se supondrá que los casos de prueba concretos serán seleccionados siguiendo una metodología que permita un grado cobertura adecuada.

4.4. Localización de los defectos

4.4.1. Generación del problema de diagnosis

Una vez detectados los errores, el siguiente paso es generar el problema de diagnosis. La resolución del problema de diagnosis permitirá determinar qué elementos del sistema software, es decir qué parte del código fuente, o de la especificación, contienen defectos.

Definición 4.2. Problema de Diagnosis. Se define como la tupla formada por (SD, STS, SPEC, TC), donde SD es la descripción del sistema, formada por un conjunto de restricciones; STS es el conjunto de sentencias que forma el código fuente; SPEC es el conjunto de asertos que forman la especificación; y TC representa un caso de prueba.

Los defectos semánticos se localizarán en los asertos que forman la especificación o en las sentencias que forman el código fuente.

- En el primer caso, el conjunto STS se supondrá vacío, ya que sólo en los asertos que forman la especificación podrán estar localizados los defectos.

- En el segundo caso, se supondrá que el conjunto de asertos SPEC son correctos, y los defectos semánticos sólo se buscarán en las sentencias que forman el código fuente, es decir, en el conjunto STS.

Para cada caso de prueba, la descripción del sistema debe proporcionar los mismos resultados que generaría el sistema software. Este modelo estará basado en un conjunto de variables y un conjunto de restricciones que acotarán los dominios de dichas variables. Es decir, la descripción del sistema debe ser un modelo del comportamiento real de los asertos que forman la especificación y de las sentencias del código fuente. Las restricciones serán obtenidas de la transformación de la especificación y del código fuente.

En la programación Orientado a Objetos la información se almacena y se transmite a través de los parámetros de los métodos, variables locales, o atributos de los objetos. Esta información puede ser de un tipo objeto o de un tipo básico, como enteros, reales, etc. Las sentencias del código fuente pueden modificar la información almacenada, y los asertos de la especificación establecen condiciones que dicha información debe cumplir. Para conseguir el modelo del comportamiento real de las sentencias y los asertos basado en restricciones, se seguirá un proceso en el que se realizarán básicamente dos tipos de transformaciones:

- Cada uno de los atributos de los objetos, parámetros de los métodos, y variables locales, serán transformados a variables del tipo adecuado para ser utilizadas en las restricciones que formarán la descripción del sistema.

- Cada sentencia o aserto será transformado a restricciones de la descripción del sistema que permitirán acotar el dominio de las variables que correspondan con los atributos, parámetros o variables locales sobre las que dicha sentencia o aserto influye. Una sentencia influye sobre un atributo, parámetro o variable local cuando le asigna un nuevo valor. Un aserto influye sobre un atributo, parámetro o variable local cuando establece sobre él una condición que debe cumplirse.

Dominio del tipo	Nombre del tipo	Dominio inicial
Enteros	IntTypeVar	{INTEGER_MIN_VALUE..INTEGER_MAX_VALUE}
Reales	RealTypeVar	{REAL_MIN_VALUE..REAL_MAX_VALUE}
Lógicos	BooleanTypeVar	{true, false}
Objectos	ObjectTypeVar	{Object$_1$, Object$_2$,..., Object$_n$}

Tabla 4.1: Tipos básicos del entorno de programación con restricciones

Obtención de las variables del problema

Por cada una de las clases que intervienen en el sistema software, se creará una clase equivalente, en la que los atributos de la clase, y los parámetros y variables locales de los métodos, serán transformados a los tipos disponibles en el entorno de programación con restricciones. Para los tipos básicos tales como enteros, reales o valores lógicos, es relativamente sencillo realizar su transformación, ya que los dominios posibles se utilizan normalmente en la programación con restricciones.

Para modelar los tipos definidos por el usuario, es decir, las clases y los objetos que son instancias de éstas, el entorno de programación con restricciones debe permitir la utilización de un tipo básico que sea capaz de gestionar como dominio diferentes objetos que sean instancias de una clase. Cuando se tiene una variable tipo instancia de una clase, los valores que puede tomar dicha variable serán los objetos que puedan ser asignados a dicha variable, análogamente a como los valores enteros pueden ser asignados a las variables definidas con dominio de tipo entero.

Con el fin de que la metodología propuesta en este trabajo sea aplicable a diferentes entornos de programación con restricciones, de aquí en adelante se hará uso de los tipos de variables definidos en la tabla 4.1. Estos tipos son IntTypeVar, RealTypeVar, BooleanTypeVar y ObjectTypeVar; y permiten almacenar respectivamente dominios de tipo entero, real, lógico, u objetos. En una implementación concreta, estos tipos serán transformados a los tipos correspondientes al entorno de programación con restricciones. Cada atributo, parámetro o variable local será transformada a una variable del tipo adecuado para que pueda ser manejada por el entorno de programación con restricciones. Por ejemplo, para transformar las referencias a objetos se usará el tipo ObjectTypeVar, que permite almacenar un conjunto de objetos como domino de una variable.

Ejemplo 4.1. En la figura 4.4 se muestra un ejemplo sencillo de transformación a tipos del entorno de programación con restricciones. Como se puede apreciar en la figura, la clase original, clase *Account* (cuenta bancaria), se transforma a una clase en la que los tipos han sido reemplazados por los apropiados para el entorno de programación con restricciones. Así por ejemplo, el tipo entero se transforma al tipo IntTypeVar en el caso del atributo *balance*, y del método *getBalance*. Para el atributo *owner*, que es de tipo *String*, la transformación es al tipo ObjectTypeVar.

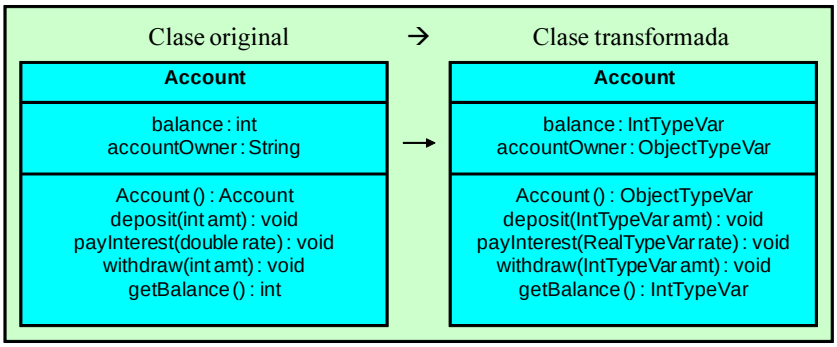


Figura 4.4: Transformación de la clase Account

Obtención de las restricciones del problema de diagnosis

Tal como se ha dicho anteriormente, las sentencias y asertos serán transformados a restricciones para ser añadidos a la descripción del sistema. En la metodología DX, un componente tiene un defecto cuando su comportamiento es anormal. El concepto de comportamiento anormal [39] está asociado al predicado AB(c). El predicado AB(c) se cumple cuando el componente c del sistema tiene un comportamiento anormal, es decir, cuando su comportamiento es defectuoso y genera fallos.

El uso del predicado AB(c) permite a la metodología DX determinar qué componentes tienen un comportamiento anormal, e inferir una explicación para los errores detectados. En la Diagnosis del Software también se utilizará el predicado AB(s), pero en este caso aplicado a sentencias del código fuente o a asertos de la especificación. Por este motivo se hace necesario adaptar su definición:

Definición 4.3. Predicado AB. Se define el predicado AB(s) de una sentencia o aserto s, como aquél que se cumple cuando la sentencia o el aserto s tiene un comportamiento anormal, y no se cumple cuando tiene un comportamiento normal. Se entiende que una sentencia o un aserto tienen un comportamiento anormal si contiene algún defecto en su diseño, y no genera el resultado correcto al ejecutarse.

La metodología de diagnosis propuesta se puede utilizar para diagnosticar qué asertos de la especificación son defectuosos, o para diagnosticar qué sentencias del código fuente contienen defectos. En función de qué elementos se quieran diagnosticar, asertos o sentencias, se hará uso del predicado AB de una forma u otra. Si el objetivo es localizar los defectos en la especificación, para cada aserto sp que forme parte del conjunto SPEC, el predicado AB(sp) permitirá conocer si el aserto es defectuoso o no. De manera análoga, si el objetivo es localizar los defectos en el código fuente, para cada sentencia st que forme parte del conjunto STS, el predicado AB(st) permitirá conocer si la sentencia es defectuosa o no.

Las transformaciones de los asertos y sentencias a restricciones se explicarán con más detalle en los capítulos 5 y 6 respectivamente. Pero como adelanto, y para ayudar

a comprender la metodología propuesta, a continuación se muestra un pequeño ejemplo en el que el objetivo es diagnosticar las sentencias del código fuente.

Ejemplo 4.2. A continuación se muestra cuál sería la descripción del sistema que se obtendría para una adaptación de un ejemplo ampliamente usado en la Metodología DX, el sistema polybox [39]. En este ejemplo sólo aparecen sentencias de asignación.

Código Fuente	Descripción del Sistema
`int x = a * c;`	$\neg AB(S_1) \implies x = a * c$
`int y = b * d;`	$\neg AB(S_2) \implies y = b * d$
`int z = c * e;`	$\neg AB(S_3) \implies z = c * e$
`int f = x + y;`	$\neg AB(S_4) \implies f = x + y$
`int g = y * z;`	$\neg AB(S_5) \implies g = y * z$

En este ejemplo, cada una de las sentencias del sistema software ha sido transformada a una restricción que modela el comportamiento real de cada sentencia. En este caso, al ser todas sentencias de asignación, tal como se verá más adelante en el capítulo 6, deben ser transformadas a restricciones de implicación. Cada restricción de implicación puede cumplirse de dos formas: o bien la sentencia tiene un comportamiento anormal, es decir, se cumple el predicado AB(s) aplicado a la sentencia s; o bien, el comportamiento esperado para dicha sentencia, que corresponde con la parte derecha de la implicación, se cumple. El comportamiento esperado para cada sentencia de asignación es cumplir que la variable asignada debe ser igual al resultado de la expresión a asignar en la sentencia original.

Para cualquier conjunto de valores concretos de las variables {*a, b, c, d, e*}, el sistema software y las restricciones obtenidas, generarían los mismos resultados en las variables f y g, si el comportamiento de los componentes es no anormal. El dominio de cada variable vendrá determinado por el tipo declarado. En este caso se trata de variables enteras.

4.4.2. Diagnosis de defectos semánticos en el software

La diagnosis es una hipótesis que permite hacer consistente el problema de diagnosis. Obtener la diagnosis es un problema equivalente a localizar los defectos semánticos en el software. Tal como se ha venido explicando anteriormente, los defectos semánticos se localizarán, en un primer paso en los asertos que forman la especificación, y en un segundo paso en las sentencias que forman el código fuente, suponiendo correcta la especificación. De una manera más formal, para el primer caso la definición de diagnosis será la siguiente:

Definición 4.4. Diagnosis de defectos semánticos en la especificación. Para un problema (SD, SPEC, STS, TC), se define como el conjunto $\mathcal{D}$ que permite satisfacer el problema de diagnosis, es decir, que hace consistente: SD $\cup$ TC $\cup$ $\{AB(s) \mid s \in \mathcal{D}\}$ $\cup$ $\{\neg$ AB(s) $\mid$ s $\in$ SPEC - $\mathcal{D}\}$.

Como se puede observar, si la metodología de diagnosis se utiliza para diagnosticar qué asertos de la especificación son defectuosos, entonces se cumple que $\mathcal{D} \subseteq SPEC$. Para el segundo caso, cuando el objetivo es diagnosticar el código fuente, se debe cumplir que $\mathcal{D} \subseteq STS$, tal como se puede deducir de la siguiente definición:

Definición 4.5. Diagnosis de defectos semánticos en el código fuente. Para un problema (SD, SPEC, STS, TC), se define como el conjunto $\mathcal{D}$ que permite satisfacer el problema de diagnosis, es decir, que hace consistente: SD $\cup$ TC $\cup$ $\{AB(s) \mid s \in \mathcal{D}\}$ $\cup$ $\{\neg$ AB(s) $\mid$ s $\in$ STS - $\mathcal{D}\}$.

La diagnosis se determinará a través de la resolución de un Problema de Satisfacción de Restricciones. La búsqueda de la solución a dicho problema será equivalente a la determinación de la diagnosis. Para realizar este proceso de forma automática, la clave está en la implementación de esta búsqueda está en las restricciones incluidas en la descripción del sistema, y más concretamente en el predicado AB(s), que establece para cada sentencia o aserto si tiene o no un comportamiento anormal. Los defectos de diseño se encuentran en aquellas sentencias o asertos (dependiendo de si se está diagnosticando la especificación o el código fuente) en los que se determine que existe un comportamiento anormal.

La diagnosis no suele ser única, pues suele ocurrir que diferentes diagnosis pueden explicar un mismo problema. Las diagnosis se obtendrán como soluciones de un problema de optimización, más concretamente como soluciones de un Max-CSP (Maximization Constraint Satisfaction Problem). En un Max-CSP se debe definir una función objetivo que el resolutor de restricciones tratará de mejorar en cada solución propuesta. Para encontrar las diagnosis del problema planteado, el objetivo del Max-CSP será maximizar el número de predicados $\neg$ AB(s_i) que tomen el valor cierto, de esta forma se minimiza el número de sentencias y asertos que tienen un comportamiento anormal, es decir: Max(N AB(s_i) : $\neg$ AB(s_i) $\mid$ $s_i \in$ (SPEC $\cup$ STS)).

Cualquiera de las asignaciones del conjunto de predicados AB(s_i) debe ser compatible con las restricciones establecidas en el problema de diagnosis. Aunque los predicados AB(s_i) partirán inicialmente como variables libres del CSP, las restricciones de la descripción del sistema restringen su dominio, y dichas restricciones deben cumplirse para alcanzar una solución en el Max-CSP. En esta tesis, una variable perteneciente a un CSP se dice que tiene un dominio libre cuando su dominio inicial incluye todo el espectro de valores disponibles para el tipo al que corresponde dicha variable.

La resolución del problema Max-CSP implica establecer un valor para los predicados AB(s_i) asociados a las sentencias o asertos. Si todos los predicados AB toman el valor falso, implicaría que todas las restricciones del problema de diagnosis pueden cumplirse a la vez, y por tanto no hay defectos. Una sentencia o un aserto debe cambiar, si tiene un comportamiento anormal, es decir, si el predicado AB(s_i) asociado toma el valor positivo en la solución al problema Max-CSP.

4.4.3. Diagnosis mínima

El Principio de la Parsimonia [32] recomienda seleccionar siempre la hipótesis que se base en un menor número de supuestos, siempre que los otros aspectos sean equivalentes. En el caso de la diagnosis, siempre será preferida aquella que explique el problema suponiendo un menor número de elementos defectuosos (asertos o sentencias). Las diagnosis simples son aquellas que implican modificar o eliminar solo una sentencia o un aserto defectuoso. La diagnosis será múltiple si implica modificar varias sentencias o asertos a la vez.

Cuanto menor sea el número de elementos la diagnosis, menor será el número de elementos a revisar o sustituir, y de forma proporcional, el esfuerzo y el tiempo necesario para realizar la revisión será menor. En la resolución del Max-CSP, se dará preferencia a aquellas soluciones en las que el conjunto de elementos que tienen comportamiento anormal sea el menor. Es decir, dará preferencia a aquellas soluciones en las que el mayor número posible de elementos tengan un comportamiento normal.

Más concretamente, de todas las diagnosis posibles, interesa quedarse con las que cumplan la definición de diagnosis mínima.

Definición 4.6. Diagnosis Mínima. Se dice que $\mathcal{D}$ es una diagnosis mínima si se cumple que $\forall\ \mathcal{D}' \subset \mathcal{D}$, $\mathcal{D}'$ no es una diagnosis.

La metodología de diagnosis presentada en este trabajo sólo genera diagnosis que cumplen la propiedad de ser mínimas. Una vez generadas todas las diagnosis mínimas, estás se proporcionarán al usuario de forma ordenada, de menor a mayor, por el número de elementos que incluyan (ya sean sentencias o asertos).

De este conjunto de diagnosis mínimas se pueden establecer dos propiedades:

1. Es completo. Si la resolución del problema Max-CSP asociado se realiza de manera exhaustiva, todas las posibles soluciones al problema serían generadas, y por consiguiente todas las posibles diagnosis mínimas.

2. Es mínimo. El sistema software generará los resultados correctos para el caso de prueba solamente en el caso de que se modifiquen todos los elementos que forman alguna de las diagnosis mínimas propuestas. Es decir, si se modifica sólo un subconjunto de los elementos que forman alguna de las diagnosis mínimas, no será posible hacer consistente el sistema software (formado por las sentencias del conjunto STS o por asertos del conjunto SPEC) con el caso de prueba TC.

Ejemplo 4.3. En la figura 4.5 se muestra el problema de diagnosis obtenido de la adaptación del sistema polybox [39], visto en el apartado anterior. Para cualquier conjunto de valores concretos para las variables $\{a,\ b,\ c,\ d,\ e\}$, el código fuente y las restricciones obtenidas (parte derecha de la figura 4.5), generarían los mismos resultados para las variables f y g, si el comportamiento de las sentencias es no anormal. El dominio de cada variable vendrá determinado por el tipo declarado. En este caso se trata de variables enteras.

Código Fuente	STS	SD
`int x = a * c`	S_1	$\neg AB(S_1) \Rightarrow (x = a * c)$
`int y = b * d`	S_2	$\neg AB(S_2) \Rightarrow (y = b * d)$
`int z = c * e`	S_3	$\neg AB(S_3) \Rightarrow (z = c * e)$
`int f = x + y`	S_4	$\neg AB(S_4) \Rightarrow (f = x + y)$
`int g = y * z`	S_5	$\neg AB(S_5) \Rightarrow (g = y * z)$

F. Objetivo: $\text{Max}(N\ s_i : s_i \in \{S_1, S_2, S_3, S_4, S_5\} : \neg AB(s_i) = \text{cierto})$

$TC = \{a = 3, b = 2, c = 2, d = 3, e = 3, f = 12, g = 12\}$

Diagnosis simples: $\{S_3\}$, $\{S_5\}$
Diagnosis múltiples: $\{S_2, S_4\}$, $\{S_1, S_2\}$

Figura 4.5: Problema de Diagnosis

El conjunto de restricciones SD ∪ TC ∪ SPEC ∪ {¬ AB(s) | s ∈ STS } que forma el problema no se puede satisfacer para el caso de prueba mostrado en dicha tabla. Concretamente, al aplicar las entradas establecidas por el caso de prueba propuesto al conjunto de restricciones que forman la descripción del sistema, se generaría un valor para la variable g diferente al establecido como correcto en el caso de prueba. Como consecuencia, no es posible encontrar una solución al problema de satisfacción con restricciones que mantenga todos los predicados AB como falsos, es decir, que mantenga que todas las sentencias tienen un comportamiento no anormal.

Para este problema se obtendrían en total cuatro diagnosis mínimas, las dos primeras diagnosis mínimas serían simples: $\{S_3\}$, $\{S_5\}$, y las otras dos serían múltiples: $\{S_2, S_4\}$ y $\{S_1, S_2\}$.

4.4.4. Utilización de varios casos de prueba

Para comprobar si un código fuente está bien diseñado, se pueden realizar diferentes pruebas encaminadas a analizar las diferentes trazas que pueden darse al ejecutar. Una buena selección de los casos de prueba permite un mejor cubrimiento de las posibles trazas.

Diferentes casos de prueba pueden generar diferentes trazas de ejecución, por tanto el uso de diferentes casos de prueba en principio generaría diferentes problemas de diagnosis. La aplicación de la metodología de diagnosis a cada uno de los casos de prueba por separado permitirá disponer del conjunto de elementos que forma la diagnosis mínima para cada caso de prueba.

Para que el proceso de diagnóstico también pueda sacar ventaja de la disponibilidad de varios casos de prueba, en este trabajo se propone procesar las diagnosis mínimas obtenidas en cada caso de prueba, y ofrecer al usuario una diagnosis mínima que sea capaz

de explicar todos los casos de prueba donde se hayan detectado resultados diferentes a los especificados como correctos.

La obtención de la diagnosis mínima para varios casos de prueba se verá con más detenimiento en el capítulo dedicado a las pruebas y mejoras (Capítulo 7). El objetivo es revisar primero aquellas sentencias que expliquen el comportamiento anómalo de un mayor número de casos de prueba. De esta forma se reduciría el número de sentencias a revisar, y como consecuencia también aumentaría la eficacia del proceso de reparación, ya que para un mismo defecto se tendrían agrupados los casos de prueba afectados.

4.5. Conclusiones

Como se ha podido comprobar, el entorno propuesto para el diagnóstico de la especificación, o del código fuente, se basa en modelar los problemas de diagnosis como problemas de optimización. Plantear un problema de diagnosis de esta forma permite incorporar las técnicas y avances que se han logrado en el campo de la búsqueda de soluciones para problemas Max-CSP.

En este capítulo se han presentado las bases de la metodología. En los siguientes dos capítulos, la metodología propuesta será ampliada y adaptada para cumplir con los dos objetivos marcados, el diagnóstico de la especificación, y del código fuente.

Capítulo 5

Diagnóstico de la especificación

5.1. Introducción

Tal como se explicó en el capítulo dedicado al estado del arte, los métodos formales [64] representan el enfoque más apropiado para conseguir mejorar la calidad en la construcción de sistemas de información. Dentro de los métodos formales, han sido los basados en la semántica axiomática, y en particular el Diseño por Contrato, los que mejor aceptación han tenido, existiendo diferentes entornos y herramientas que permiten su uso.

El Diseño por Contrato se basa en definir una serie de asertos que caracterizan las propiedades de los diferentes elementos del lenguaje. Estos asertos se expresan en lógica de primer orden y teoría de conjuntos. Su uso aumenta la claridad y evita ambigüedades e inconsistencias en los desarrollos. En general, el Diseño por Contrato añade una serie de ventajas que facilitan el desarrollo del software. Pero queda un problema sin resolver, las especificaciones son desarrolladas por personas, y por tanto, no están libres de incluir defectos en su diseño.

El Diseño por Contrato presenta una serie de ventajas con respecto al resto de métodos formales:

- En la formulación de los asertos se suele utilizar un lenguaje muy parecido al utilizado en la implementación. De esta manera se consigue que la redacción de la especificación sea más cercana y sencilla para el usuario. Además, resulta más sencillo aprender a especificar, ya que para el usuario es casi inmediata la transformación de las condiciones que se quieren establecer a asertos.

- Permite un mayor grado de flexibilidad a la hora de realizar la especificación. Queda en manos de quien diseña la especificación la completitud o el grado de dureza de dicha especificación, lo que permite un desarrollo de la especificación adaptable a las necesidades del software que se está desarrollando. De esta forma es posible realizar una especificación muy restrictiva y fuerte en aquellas partes que así lo requieran, o débil y poco restrictiva si el desarrollo requiere flexibilidad.

Assert	::= AssertExp	Aserto
AssertExp	::= Implies-expr	Expresion
Implies-expr	::= Logical-expr ('=>' Implies-expr)?	Implicación
Logical-expr	::= Equality-expr (LogBin-op Equality-expr)*	Expresión lógica
LogBin-op	::= '||' | '&&'	Op. binario lógico
Equality-expr	::= Relational-expr ('=' Relational-expr)*	Igualdad
	| Relational-expr ('!=' Relational-expr)*	Desigualdad
Relational-expr	::= AritBin-expr RelBin-op AritBin-expr	Exp. de comparación
RelBin-op	::= '<' | '>' | '<=' | '>='	Comparadores
AritBin-expr	::= Unary-expr (AritBin-op Unary-expr)*	Expresión arit. binaria
AritBin-op	::= '+' | '-' | '*' | '/'	Operadores
Unary-expr	::= Unary-op Unary-expr	Expresión unaria
	| Unary-expr-not-arit	
Unary-op	::= '+' | '-'	Operadores unarios
Unary-expr-not-arit	::= '!' Unary-expr	Negación unaria
	| Primary-expr Primary-suffix*	
Primary-suffix	::= '.' ident	Acceso a un atributo
	| '(' Expression-list? ')'	Llamada a método
Expression-list	::= AssertExp (, AssertExp)*	Lista de expresiones
Primary-expr	::= ident | constant	Variable local, constante
	| this | super	Propio objeto, clase padre
	| true | false	Literal cierto y falso
	| null	Literal nulo
	| '(' AssertExp ')'	Expresión entre paréntesis
	| Primary-spec	Expresión primaria
Constant	::= Literal	Literal
Primary-spec	::= @result	Retorno de un método
	| @old '(' AssertExp ')'	Valor previo a un método

Figura 5.1: Gramática permitida para los asertos

- Hay entornos que permiten comprobar en tiempo de ejecución las propiedades que deben cumplirse en el software especificado. Por tanto, la especificación no sólo es útil para el diseño de la implementación, sino que también permite en tiempo de ejecución, de forma dinámica, seguir comprobando que se cumplen las condiciones impuestas por la especificación.

- La propia especificación puede ser utilizada como documentación sobre el comportamiento esperado del sistema software desarrollado, ofreciendo un valor añadido, y facilitando la adaptación y reutilización futura.

El código fuente debe cumplir los asertos establecidos por la especificación, por tanto, para evitar propagar defectos a la implementación, es necesario establecer mecanismos que permitan detectar e identificar los defectos contenidos en la especificación. En este capítulo se propondrá una metodología para comprobar la consistencia entre los asertos que forman la especificación, e identificar los asertos que deben cambiar si se detectan inconsistencias. Es decir, una metodología para comprobar y diagnosticar los asertos que forman la especificación. Para lograr este objetivo, se adaptará la metodología de diagnosis propuesta en el capítulo 4 y se propondrán una serie de comprobaciones, con el objetivo de poder detectar errores en la especificación, e identificar de forma automática los defectos que dieron lugar a los errores detectados.

En la metodología propuesta en este trabajo se utilizará como base la gramática mostrada en la figura 5.1, descrita utilizando la notación EBNF (Extended Backus Normal Form).

Una especificación en la forma BNF es un sistema de reglas de derivación, escritas como sigue:

`simbolo` ::= `expresión_con_símbolos`

Donde '`símbolo`' es un elemento no terminal, y la '`expresión_con_símbolos`' es una posible substitución para el símbolo de la izquierda. Cuando hay varias opciones de sustitución, las opciones aparecen separadas por la barra vertical '|'. En la notación EBNF (Extended Backus Normal Form) se pueden utilizar algunos comodines, tales como el comodin '$*$', que especifica que un símbolo puede aparecer cero o más veces; el comodín '+', que especifica que un símbolo puede aparecer 1 o más veces; o el comodín '?', que especifica que un símbolo puede aparecer una vez o ninguna.

La gramática mostrada contiene los suficientes tipos de asertos y expresiones como para llegar al nivel de expresividad que tienen las especificaciones axiomáticas más extendidas, como por ejemplo OCL [132][133] o JML [106]. Los ejemplos que se mostrarán en esta tesis se basarán en esta gramática.

Una especificación está compuesta por una serie de asertos que deben cumplirse cuando sean evaluados. Los asertos se expresarán en lógica de primer orden, usando expresiones y ternas de Hoare [82]. En concreto, los asertos pueden ser del tipo:

- Invariantes de clases. Son expresiones lógicas que deben ser evaluadas a cierto siempre que se complete la ejecución del constructor o de cualquiera de los métodos públicos que posea una clase. Cuando un invariante es evaluado a falso implica que el sistema está fallando con respecto al modelo establecido. Todas las instancias de la clase deben cumplir los invariantes declarados en la clase, antes y después de la ejecución de un método.

- Precondiciones y postcondiciones. Son expresiones lógicas que deben evaluarse a cierto, antes y después respectivamente, de la ejecución de un método. En las postcondiciones pueden usarse dos tipos especiales de predicados: *@result* y *@old*(). El primero hace referencia al resultado devuelto por el método, y el segundo hace referencia al valor del atributo o método de consulta antes de ejecutarse el método especificado.

5.2. Adaptación de la metodología de diagnosis

En este capítulo se adaptará y utilizará la metodología de diagnosis propuesta en el capítulo anterior, para el diagnóstico de defectos semánticos en la especificación. En esta adaptación se incluirán diferentes tipos de comprobaciones sobre los asertos

que forman la especificación. Las comprobaciones permitirán a la metodología detectar errores en los resultados obtenidos al evaluar los asertos que forman la especificación. En concreto, las comprobaciones se aplicarán a los invariantes de las clases, que actúan sobre sus atributos; y sobre las especificaciones de precondiciones y postcondiciones, que actúan sobre los métodos de las clases. Se supondrá que puede existir la herencia de la especificación entre clases, y por tanto, en las comprobaciones que se realicen se tendrán en cuenta las implicaciones que sobre la especificación tiene la herencia.

La herencia permite compartir entre objetos el comportamiento (métodos) y la representación (atributos), es decir, permite a los diseñadores construir nuevo software reutilizando software ya construido y probado. La superclase o clase padre se define cómo la clase más abstracta, y la subclase o clase hija es la que hereda las propiedades y comportamiento establecidos en la clase padre. La sobrecarga de métodos permite que operaciones heredadas de una superclase cambien su implementación en la clase hija.

La semántica comúnmente aceptada para la herencia establece que en cada lugar donde aparezca una instancia de una clase es siempre posible sustituirla por una instancia de una subclase, ya que ésta debe comportarse de la misma forma que cualquiera de las instancias de la superclase. Es lo que se conoce como el principio de sustitución de Liskov [112]. Por tanto, aunque al sobrescribir un método se pueden añadir más asertos y se puede cambiar el código fuente, existe obligación de satisfacer el contrato heredado. Esto implica consecuencias en el desarrollo de los invariantes, precondiciones y postcondiciones, de las subclases.

En las comprobaciones sobre la especificación será necesario tener en cuenta el principio de sustitución de Liskov [112]. De forma resumida las comprobaciones que se proponen en esta tesis se pueden agrupar en dos tipos.

- Comprobaciones sobre los invariantes. El objetivo es comprobar si los invariantes de una clase (propios y heredados) pueden satisfacerse en conjunto.

- Comprobaciones sobre los métodos. El objetivo es comprobar si las precondiciones y postcondiciones asociadas a los métodos son consistentes. Estas comprobaciones se realizarán en dos fases. Primero únicamente sobre los asertos definidos en los métodos (propios y heredados), y luego sobre los asertos del método en conjunto con los invariantes de la clase (definidos y heredados).

Cada una de las comprobaciones se realizará a través de un problema de satisfacción de restricciones. Los asertos que forman la especificación serán transformados a restricciones. Si alguna de las comprobaciones no puede satisfacerse será debido a que existe al menos un defecto en alguno de los asertos. En las secciones 5.3 y 5.4 se explicarán con más detalle las comprobaciones propuestas.

Si alguna de las comprobaciones no se satisface, la metodología de diagnosis permitirá determinar cuáles son los defectos, es decir, cuáles son los asertos que deben ser

modificados. Sobre cada una de las fases de la metodología de diagnosis propuesta en el capítulo 4 será necesario realizar una serie de adaptaciones.

En los siguientes apartados se explica con más detalle cuáles serían las adaptaciones necesarias para las fases de detección de los errores y de localización de los defectos; y posteriormente, se expondrá cómo realizar las comprobaciones propuestas en combinación con la metodología de diagnosis descrita.

5.2.1. Detección de errores

La detección de los errores se realizará de forma automática a través de la resolución de un CSP. Para generar el CSP se transformarán a restricciones la secuencia de asertos y el caso de prueba propuesto. Resolver el CSP implica encontrar una asignación para las variables del problema que permita satisfacer todas las restricciones.

Como las restricciones fueron obtenidas de la especificación y del caso de prueba, si no se encuentra una solución para el CSP, implicará que la especificación no puede satisfacer la comprobación propuesta. Por tanto, debe existir al menos un defecto en el diseño de los asertos que forman la especificación, que impide que puedan cumplirse en conjunto para el caso de prueba propuesto. Si existen errores, el siguiente paso será generar un problema de diagnosis, adaptando las restricciones obtenidas de la especificación.

En los siguientes apartados se explicará con más detalle cómo obtener las variables y las restricciones del CSP.

Obtención de las variables del problema

Para transformar la especificación a restricciones, previamente será necesario realizar la transformación de los tipos, y de esta forma quedarán definidos en función de los tipos disponibles en el entorno de programación con restricciones. Por ejemplo, si se tiene una variable entera como atributo de una clase, y dicha variable interviene en la especificación, entonces esta variable tendrá su correspondiente en el modelo basado en restricciones. Los tipos de datos están divididos entre tipos básicos y los tipos definidos por el usuario. Los tipos predefinidos son los enteros, reales y lógicos. Los tipos definidos por el usuario corresponden con las clases que se utilicen en la especificación.

Cuando se realice la transformación a restricciones de la especificación, las variables tendrán un dominio libre, que se restringirá en función de las condiciones establecidas en las condiciones obtenidas de la especificación y de los casos de prueba. En esta tesis, una variable perteneciente a un CSP se dice que tiene un dominio libre cuando su dominio inicial incluye todo el espectro de valores disponibles para el tipo al que corresponde dicha variable. Por ejemplo, si se tiene un atributo de tipo entero, inicialmente el dominio de dicha variable será cualquier valor entero. Si más adelante el invariante obliga a que esta variable deba ser positiva, entonces para satisfacer dicha condición el dominio de dicha variable se reducirá a sólo valores positivos.

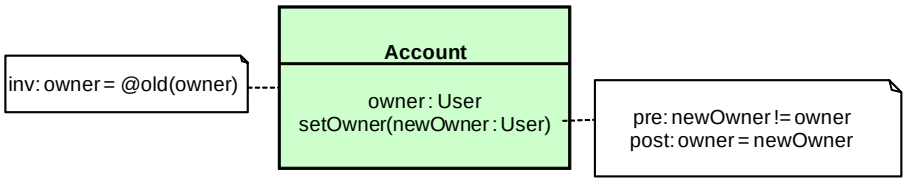


Figura 5.2: Ejemplo de defecto en la especificación

Obtención de las restricciones

Los asertos serán transformados a restricciones, formando un CSP. Las restricciones obtenidas acotarán el dominio de las variables que correspondan con los atributos y parámetros de entrada sobre los que cada aserto influye. Un aserto influye sobre un atributo o un parámetro de entrada cuando establece sobre él una condición que debe cumplirse.

Teniendo en cuenta la gramática presentada en la figura 5.1, se pueden distinguir entre las expresiones debidas al uso de una especificación en lógica de primer orden, y las expresiones debidas al uso de la orientación a objetos. El primer tipo de expresiones englobaría las expresiones conectivas (implicación, operaciones lógicas), expresiones binarias y unarias de tipo aritmético, aplicación de funciones a constantes y variables, aplicación de predicados a parámetros, y las variables, constantes y literales. Este tipo de expresiones pueden ser transformadas a restricciones de forma sencilla.

El segundo tipo de expresiones englobaría las operaciones de accesos a atributos y métodos. Para transformar este tipo de expresiones a restricciones, en esta tesis se supondrá que la implementación del entorno de programación con restricciones permite el manejo de objetos y las operaciones asociadas.

Al realizar la transformación de los asertos a restricciones se deben cumplir además las siguientes reglas:

- Desde los asertos que forman la especificación es posible acceder a los atributos de los objetos y realizar llamadas a métodos de tipo consulta, es decir, que no tengan efectos sobre los atributos de la clase. Cada llamada a estos métodos retornará una variable del tipo establecido en el método de consulta, pero con el dominio libre, ya que no se conoce la implementación de dicho método, y se supondrá que dicho método puede devolver cualquier valor. Será la postcondición de dicho método de consulta la que pueda establecer restricciones sobre el dominio del resultado generado. En las postcondiciones se podrá hacer uso de la función *@result* que hará referencia al resultado devuelto por el método.

- Los atributos de una clase pueden variar tras la ejecución de un método, por tanto es necesario diferenciar entre el valor inicial del atributo, antes de la ejecución del método, y el valor final del atributo, una vez ejecutado el método. Para diferenciar entre uno y otro se utilizará la función *@old*() que hace referencia al valor del

atributo antes de ejecutar el método. En aquellos métodos que tengan precondición, al transformar la precondición a restricciones, cada acceso a un atributo se hará sobre el valor previo a la ejecución del método, es decir sobre *@old*(*atributo*).

Ejemplo 5.1. En la figura 5.2 se muestra la especificación del método *setOwner* que modifica el atributo *owner* de la clase mostrada. El método *setOwner* recibe el parámetro de entrada *newOwner*. Llamaremos *@old*(*owner*) al valor de dicho atributo antes de entrar en el método.

El método mostrado debe cumplir la siguiente secuencia de asertos (precondición, postcondición e invariante de la clase) en cada llamada que se realice:

Pre: newOwner ! = owner
Post: owner = newOwner
Inv: owner = @old(owner)

Transformando la secuencia de asertos a restricciones, y siguiendo las reglas expuestas anteriormente, se obtendría el siguiente CSP:

Aserto:	**Restricciones**:	**Dominio**:
Pre	newOwner ! = @old(owner)	newOwner, @old(owner), owner = libre
Post	owner = newOwner	
Inv	owner = @old(owner)	

Como se puede observar, en la transformación de la precondición, se ha utilizado *@old*(*owner*), ya que el valor del atributo que debe cumplir la precondición, es el que estuviera antes de la ejecución del método. Para comprobar la secuencia de asertos se utilizará el caso de prueba vacío (este concepto se explicó en la sección 4.3), por tanto, los dominios de las variables quedarían libres. Al intentar resolver el CSP planteado con las restricciones resultantes de transformar los asertos que forman la secuencia, se puede comprobar que no hay ninguna solución posible, ya que no hay ninguna asignación para las variables (*newOwner*, @*old*(*owner*) y *owner*), que permita satisfacer las tres restricciones a la vez. Los tres asertos no se pueden cumplir a la vez debido a que en el diseño del método no se ha tenido en cuenta la especificación del invariante, que impide cambios en el atributo *owner*.

Si se implementa el código fuente basándose en esta especificación, los defectos que tiene la especificación serían transmitidos a la implementación, provocando nuevos defectos en la implementación. De ahí la importancia de detectar estos defectos antes de pasar a la implementación.

5.2.2. Localización de los defectos

Una vez detectado el error, el siguiente paso es generar el problema de diagnosis. Tal como se explicó en el capítulo dedicado a la metodología (capítulo 4) un problema de diagnosis está formado por (SD, STS, SPEC, TC), donde SD es la descripción del

sistema, formada por un conjunto de restricciones; STS es el conjunto de sentencias que forma el código fuente; SPEC es el conjunto de asertos que forman la especificación; y TC representa un caso de prueba. En los problemas que se resolverán en este capítulo, el conjunto STS estará vacío.

Para obtener las restricciones asociadas a la descripción del sistema será necesario adaptar las restricciones que se utilizaron en el paso anterior para la detección de errores. En concreto será necesario añadir el predicado AB, para tener en cuenta la posibilidad de que los asertos tengan comportamiento anormal. Se entiende que un aserto tiene un comportamiento anormal, si contiene algún defecto en su diseño y debe modificarse. La metodología de diagnosis tratará de concretar cuáles de los asertos contienen defectos.

La unidad mínima donde se podrán detectar defectos será cada uno de los asertos que forman la especificación. Por tanto, el predicado AB será aplicado a cada aserto. El predicado AB(a) de un aserto a toma el valor verdadero cuando el aserto a tiene un comportamiento anormal, y el valor falso cuando tiene un comportamiento normal. Cada aserto será transformado a una restricción de implicación que puede cumplirse de dos formas: o bien el aserto tiene un comportamiento anormal, es decir, se cumple el predicado AB(a) aplicado al aserto a; o bien, se cumple el comportamiento esperado para dicho aserto, que corresponde con la parte derecha de la implicación. El comportamiento esperado para cada aserto es cumplir la condición establecida en el propio aserto.

La localización de los defectos en la especificación se logrará resolviendo el problema de diagnosis. La resolución se basará en encontrar una diagnosis que permita hacer consistente el problema (SD, SPEC, STS, TC). En este capítulo, los defectos a localizar pertenecen a la especificación, y el conjunto STS de sentencias del código fuente estará vacío. La diagnosis $\mathcal{D}$ debe cumplir que SD $\cup$ TC $\cup$ $\{AB(a) \mid a \in \mathcal{D}\} \cup \{\neg$ AB(a) $\mid$ a $\in$ SPEC - $\mathcal{D}\}$, siendo $\mathcal{D} \subseteq$ SPEC.

La localización de los defectos se implementará a través de un Max-CSP. Tal como se explicó en el capítulo 4, la metodología de diagnosis presentada en este trabajo sólo genera diagnosis que cumplen la propiedad de ser mínimas. Una vez generadas todas las diagnosis mínimas, están se proporcionarán al usuario de forma ordenada, de menor a mayor, por el número de asertos que incluyan.

Por cada una de las comprobaciones donde se hayan detectado errores, se obtendrá la diagnosis mínima. Los defectos de diseño residen en los asertos que se incluyen en la diagnosis mínima, y para solucionarlos será necesario modificar o eliminar dichos asertos.

Ejemplo 5.2. Reutilizando las restricciones que se obtuvieron en el ejemplo mostrado en el apartado anterior, y transformándolas, tal como se ha indicado en este apartado, a restricciones de implicación, se obtendría el problema de diagnosis mostrado en la figura 5.3.

Las restricciones que aparecen establecen el comportamiento de cada uno de los asertos, utilizando el predicado AB. Si el predicado AB(a_i) toma el valor falso, entonces

Especificación	SPEC	SD
Pre: newOwner ! = @old(owner)	a_1	$\neg AB(a_1) \Rightarrow$ (newOwner ! = @old(owner))
Post: owner = newOwner	a_2	$\neg AB(a_2) \Rightarrow$ (owner = newOwner)
Inv: owner = @old(owner)	a_3	$\neg AB(a_3) \Rightarrow$ (owner = @old(owner))

Figura 5.3: Problema de diagnosis para el caso de prueba vacío (TC = $\emptyset$)

el aserto a_i tiene un comportamiento normal (no contiene defectos), y en caso contrario se supondrá que el comportamiento es anormal (contiene defectos, y debe modificarse).

Añadiendo la función objetivo al problema de diagnosis mostrado en la tabla 5.3, se obtendría el siguiente problema Max-CSP:

Restricciones:
$\neg AB(a_1) \Rightarrow$ newOwner ! = @old(owner)
$\neg AB(a_2) \Rightarrow$ owner = newOwner
$\neg AB(a_3) \Rightarrow$ owner = @old(owner)

Dominio:
newOwner, @old(owner), owner = libre

F. Objetivo: Max(N $a_i : a_i \in \{a_1, a_2, a_3\} : \neg AB(a_i)$ = cierto)

Los predicados $AB(a_1)$, $AB(a_2)$ y $AB(a_3)$ son los encargados de almacenar si los asertos a_1, a_2 y a_3 tienen o no un comportamiento anormal. La función objetivo busca conseguir que el mayor número de restricciones se cumplan. La modificación de cualquiera de los tres asertos permitiría poder satisfacer el conjunto de restricciones. En este ejemplo, las posibles diagnosis mínimas son tres, que corresponden con tres defectos simples, uno por cada aserto.

5.3. Diagnóstico de defectos en invariantes

Los invariantes de una clase deben cumplirse tras la ejecución de un constructor o de cualquier método. Por tanto, todos los invariantes deben ser consistentes entre sí. El objetivo en este apartado es detectar y localizar si existen inconsistencias entre los invariantes. La revisión y comprobación de los invariantes es útil sobre todo cuando la clase que se quiere comprobar hereda de otras, y por tanto, se heredan y deben cumplir los invariantes establecidos en las superclases. En este caso, el número de invariantes a tener en cuenta puede ser elevado, y con la metodología propuesta en esta tesis, la consistencia entre los invariantes se podría comprobar de forma rápida y automática.

En este tipo de comprobaciones se utilizará el caso de prueba vacío, ya que la idea es determinar si existen inconsistencias entre los asertos, pero sin establecer unas condiciones iniciales concretas. En la sección 4.3 se mostraron las diferencias entre los casos de prueba vacíos y los casos de prueba concretos. Si los invariantes no son consistentes, es decir, si no se pueden cumplir a la vez, se considerará un error, y este error será debido a un mal diseño en los invariantes. Una vez comprobado que los

invariantes no son consistentes, el siguiente paso será identificar qué invariantes deben cambiar para lograr que se puedan satisfacer a la vez.

La comprobación sobre los invariantes de una clase se realizará en dos fases:

- Comprobación sobre invariantes definidos en la propia clase. Todos los invariantes propios de una clase deben ser consistentes entre sí. El objetivo es comprobar si existe alguna inconsistencia entre los invariantes, y si existe, identificar los defectos que hacen que los asertos sean inconsistentes. Este tipo de defectos podrían deberse a la participación de varias personas en el diseño de la especificación, o al diseño en diferentes instantes de tiempo.

- Comprobación sobre invariantes definidos en la propia clase y heredados. Para poder mantener el principio de sustitución de Liskov [112], los nuevos invariantes introducidos en las subclases no pueden entrar en contradicción con los invariantes heredados de la superclase, es decir, deben poder satisfacerse a la vez. Por tanto una subclase puede añadir invariantes siempre que sean consistentes con los heredados de las superclases. El objetivo es detectar si existen inconsistencias entre los invariantes definidos en la clase y los heredados, e identificar los defectos que dan lugar a las inconsistencias.

5.3.1. Invariantes propios

Siguiendo la metodología propuesta en el apartado 5.2, para comprobar si existen inconsistencias en los invariantes definidos en la propia clase, dichos invariantes deben ser transformados a restricciones. A partir de ahí los pasos serían:

- **Detección de errores**. Si se comprueba que no se pueden satisfacer todas las restricciones a la vez, utilizando el caso de prueba vacío, el problema debe estar en un mal diseño de los invariantes que han dado lugar a las restricciones, ya que dichas restricciones producen el mismo comportamiento que hay establecido en los invariantes.

- **Localización de los defectos**. Si no se pueden satisfacer todas las restricciones a la vez, el siguiente paso será localizar qué invariantes deben cambiar. Para ello se creará un problema de diagnosis, y resolviendo dicho problema, se localizarán los invariantes que impiden que el conjunto total de invariantes se pueda satisfacer.

Ejemplo 5.3. En la figura 5.4 se muestra un ejemplo con tres clases: A, B y C. Las clases B y C heredan de la clase A. Las clases B y C heredan los invariantes de la clase A, y además añaden los nuevos invariantes que se muestran en la figura. Este ejemplo se utilizará también en otros apartados de este capítulo.

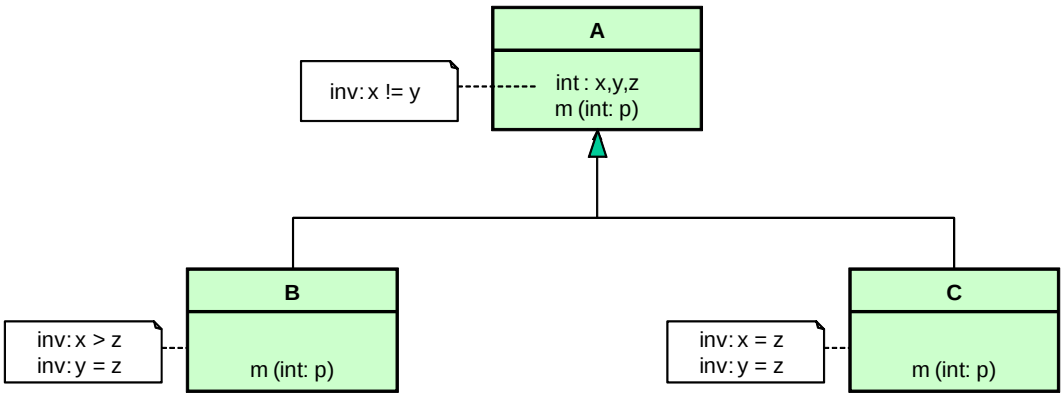


Figura 5.4: Ejemplo de herencia de invariantes

- **Detección de errores**. Siguiendo la metodología propuesta, para garantizar que los invariantes definidos en la clase C son consistentes entre sí, habría que comprobar que las restricciones obtenidas de dichos invariantes pueden satisfacerse a la vez. Por tanto el siguiente CSP se debe poder satisfacer.

  Restricciones:
  $\neg$AB(inv$_1$) $\Rightarrow$ x = z
  $\neg$AB(inv$_2$) $\Rightarrow$ y = z
  $\forall$ inv$_i$: inv$_i$ $\in$ {inv$_1$, inv$_2$} : $\neg$AB(inv$_i$) = cierto

  Dominio:
  x, y, z = libre

  En este caso, el problema tiene solución, por tanto los invariantes definidos en la clase C son consistentes. Igual sucede con las clases A y B, cuando se aplica la metodología propuesta. Tal como se ha explicado anteriormente una variable perteneciente a un CSP tiene un dominio libre cuando su dominio inicial incluye todo el espectro de valores disponibles para el tipo al que corresponda dicha variable.

- **Localización de los defectos**. Si el CSP anterior no tuviera solución, para identificar qué invariantes son los inconsistentes, se resolvería el problema Max-CSP asociado, tal como se explicó en el apartado 5.2.

5.3.2. Invariantes propios y heredados

Los invariantes de una superclase se deben poder satisfacer en las subclases, por tanto, es muy importante no establecer un comportamiento en un invariante de una subclase que entre en contradicción con los invariantes establecidos en las superclases. Por ejemplo, si en una clase *Account* que representa una cuenta bancaria se establece que el saldo no puede ser negativo, en una subclase de *Account* podría endurecerse dicha condición para que el saldo fuera obligatoriamente positivo, pero no se podría

permitir que los saldos fueran negativos, ya que se produciría un contradicción entre los invariantes de la superclase y la subclase. Para comprobar de forma automática que se cumple esta propiedad, se aplicará la metodología propuesta en el apartado 5.2:

- **Detección de errores**. Sean Inv y Inv′, los invariantes respectivamente de la superclase y de la subclase, para comprobar que los asertos que forman los invariantes son consistentes, se debe cumplir la siguiente regla: Inv $\wedge$ Inv′. Transformando los invariantes a restricciones será posible comprobar si se pueden satisfacer todos los invariantes a la vez (propios y heredados), o si por el contrario existen invariantes inconsistentes entre sí. Si se comprueba que no se pueden satisfacer todas las restricciones a la vez, utilizando el caso de prueba vacío, el problema debe estar en un mal diseño de los invariantes que han dado lugar a las restricciones.
- **Localización de los defectos**. Si no se pueden satisfacer todas las restricciones a la vez, el siguiente paso será identificar qué invariantes deben cambiar. Para ello se creará un problema de diagnosis, y resolviendo el problema Max-CSP asociado, se localizarán los invariantes defectuosos.

Como la única diferencia entre las restricciones que se utilizan para detectar los errores y las que aparecen en el problema de diagnosis es la utilización del predicado AB, para evitar tener que generar las restricciones en la detección de los errores y luego tener que adaptarlas para el problema de diagnosis, en los ejemplos que se mostrarán en este capítulo se generarán directamente las restricciones del problema de diagnosis. Y para poder comprobar si existen errores utilizando las restricciones del problema de diagnosis bastará con añadir la siguiente restricción:

$\forall\ a_i : a_i \in \{a_1, ..., a_n\} : \neg AB(a_i) = \text{cierto}$

Esta regla obliga a que todos los asertos tengan un comportamiento no anormal. De esta forma es posible comprobar si todos los asertos son consistentes entre sí, o si por contrario no pueden satisfacerse a la vez.

Ejemplo 5.4. Siguiendo con el ejemplo que se mostró en la figura 5.4, a continuación se realizará la comprobación de los invariantes de las clases B y C.

- **Detección de errores**. Siguiendo la metodología propuesta, para garantizar que todos los invariantes (propios y heredados) de la clase B son consistentes entre sí, se deben cumplir las siguientes restricciones:

Restricciones:	**Dominio**:
$\neg AB(inv_1) \Rightarrow x\ !=\ y$	x, y , z = libre
$\neg AB(inv_2) \Rightarrow x > z$	
$\neg AB(inv_3) \Rightarrow y = z$	
$\forall\ inv_i : inv_i \in \{inv_1, inv_2, inv_3\} : \neg AB(inv_i) = \text{cierto}$	

El problema tiene solución, y por tanto los invariantes definidos en la clase B son consistentes con los heredados de la clase A. En el caso de la clase C, para garantizar que todos los invariantes son consistentes (propios y heredados), se deben cumplir las siguientes restricciones:

Restricciones:	**Dominio**:
$\neg AB(inv_1) \Rightarrow x \,!= y$	x, y , z = libre
$\neg AB(inv_2) \Rightarrow x = z$	
$\neg AB(inv_3) \Rightarrow y = z$	
$\forall\ inv_i : inv_i \in \{inv_1, inv_2, inv_3\} : \neg AB(inv_i) = cierto$	

El CSP formado por las restricciones anteriores no tiene solución, y por tanto los invariantes definidos en la clase C no son consistentes con los heredados de la clase A.

- **Localización de los defectos**. Para identificar qué invariantes son los inconsistentes se debe añadir una función objetivo, y resolver el problema como un Max-CSP:

Restricciones:	**Dominio**:
$\neg AB(inv_1) \Rightarrow x \,!= y$	x, y , z = libre
$\neg AB(inv_2) \Rightarrow x = z$	
$\neg AB(inv_3) \Rightarrow y = z$	

F. Objetivo: $\text{Max}(N\ inv_i : inv_i \in \{inv_1, inv_2, inv_3\} : \neg AB(inv_i) = cierto)$

En este ejemplo es imposible satisfacer los tres asertos a la vez, al menos uno de los tres asertos debe ser modificado.

5.4. Diagnóstico de la especificación de los métodos

La comprobación sobre las precondiciones y postcondiciones de cada método se realizará en dos fases:

- En un primer momento sobre las precondiciones y postcondiciones de los propios métodos.
- Y en segundo lugar, sobre las precondiciones y postcondiciones de los métodos en conjunto con los invariantes.

La primera comprobación se realiza cuando la clase a diagnosticar hereda de otra, y por tanto debe cumplir las especificaciones establecidas en los métodos heredados.

5.4.1. Reglas para el refinamiento de la especificación de los métodos

Cuando una clase hereda de otra recibe todos los métodos de la clase padre, y puede añadir asertos a la precondición y postcondición, y/o refinar el código fuente incluido en el método. Esta posibilidad obliga a tener en cuenta una serie de reglas a la hora de ampliar o modificar la precondición y postcondición en los métodos heredados. La especificación que esté establecida en la superclase para los métodos, debe respetarse en las clases hijas (principio de sustitución de Liskov [112]), aunque éstas cambien o amplíen parte del código fuente, de otra forma se podrían dar incongruencias al ejecutar el sistema software, como la que a continuación se expone.

En la programación orientada a objetos, es posible tener una referencia *ref* del tipo A, y asignar a dicha referencia un objeto del tipo B que sea subclase de A. Cuando el usuario llame a un método del objeto almacenado en la referencia *ref*, espera el mismo comportamiento establecido en la especificación de la clase A. Será necesario que el objeto B mantenga como precondición y postcondición condiciones que no entren en contradicción con las establecidas en la clase A. En caso contrario, el usuario puede recibir un comportamiento diferente al que se estableció en la clase A, o puede que los parámetros que envíe no sean admitidos como válidos.

Para cumplir la coherencia de los métodos refinados con respecto a los métodos heredados, cuando se añadan nuevos asertos a la precondición y postcondición de un método heredado se deben cumplir las siguientes reglas:

- En el caso de las precondiciones, éstas no pueden ser refinadas en una subclase para ser más duras, sólo pueden ser debilitadas, es decir, cualquier entrada que fuera válida para el método definido en la superclase, debe ser también válida para el método refinado en la subclase. Es decir, $\mathrm{Pre(m)} \subseteq \mathrm{Pre'(m)}$, siendo m el método llamado, $\mathrm{Pre'}$ la precondición en la subclase, y Pre la precondición en la superclase. De esta forma se garantiza que toda llamada que se pueda realizar al método definido en la clase padre, debe poder realizarse también al método refinado en la clase hija.

- En el caso de las postcondiciones es justo lo contrario, no se pueden debilitar, sólo pueden ser endurecidas, es decir, sólo las salidas que fueran válidas para el método definido en la superclase, pueden ser también validas para el método refinado en la subclase. Es decir, $\mathrm{Post'(m)} \subseteq \mathrm{Post(m)}$, siendo m el método llamado, $\mathrm{Post'}$ la postcondición en la subclase, y Post la postcondición en la superclase. Esta obligación garantiza que no se generará como resultado al llamar a un método refinado en la clase hija, un resultado que no fuera posible generar llamando al método definido de la clase padre.

El objetivo de los siguientes apartados (5.4.2 y 5.4.3) será comprobar de forma automática si se cumplen o no estas reglas. Para ello se transformarán a restricciones los elementos que forman la especificación.

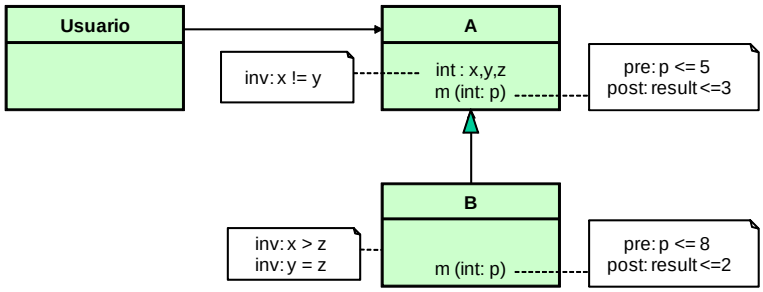


Figura 5.5: Ejemplo de herencia entre clases

Ejemplo 5.5. En la figura 5.5 se muestra la clase Usuario que tiene una referencia de tipo A. El objeto usuario conoce el contrato establecido en A para el método m, y espera que se cumpla la postcondición si los datos enviados cumplen la precondición. Si el usuario almacena en la referencia un objeto de tipo B, en lugar de un objeto de tipo A, él no notaría la diferencia, ya que en B se han seguido las reglas enunciadas en este apartado.

- La precondición del método m en la clase A establece que son válidos, para el parámetro p de entrada, los valores menores o iguales a 5. En la precondición del método m en la clase B, se admiten valores para el parámetro p que sean menores o iguales a 8. Por tanto, la precondición en la clase B es más permisiva, admite todo el espectro de valores permitido en la clase A para el parámetro p, y además pueden ser admitidos los valores 6, 7, y 8.

- La postcondición del método m en la clase A establece que son válidos como salida los valores menores o iguales a 3. En la postcondición del método m en la clase B, se admiten como valores de salida correctos aquellos que sean menores o iguales a 2. Por tanto, la postcondición de la clase B es más restrictiva, ya que permite un espectro de valores menor al permitido en la clase A. En concreto, ya no se producirá como salida el valor 3. Esto no es un problema para la clase Usuario, pero si sería un problema que se generarán valores que no estuvieran contemplados en la postcondición establecida en el método m en la clase A.

5.4.2. Precondición propia y heredada de cada método

Cuando se añaden nuevos asertos a la precondición de un método heredado, la precondición resultante, que llamaremos Pre', debe ser igual o más débil que la precondición establecida en el método heredado, que llamaremos Pre. De esta forma se garantiza que se podrá recibir al menos el mismo rango de entradas que en el método definido en la clase padre, y que nunca una entrada permitida en la precondición del método definido en la clase padre será admitida por la precondición del método definido en la clase hija.

En la figura 5.6 se muestran los cuatro casos posibles que pueden ocurrir. Para que la nueva precondición Pre′ sea correcta, debe cumplirse el caso 1, en el cuál todo el dominio de valores que satisfacen la precondición Pre, están incluidos en el dominio de valores que satisfacen la nueva precondición Pre′.

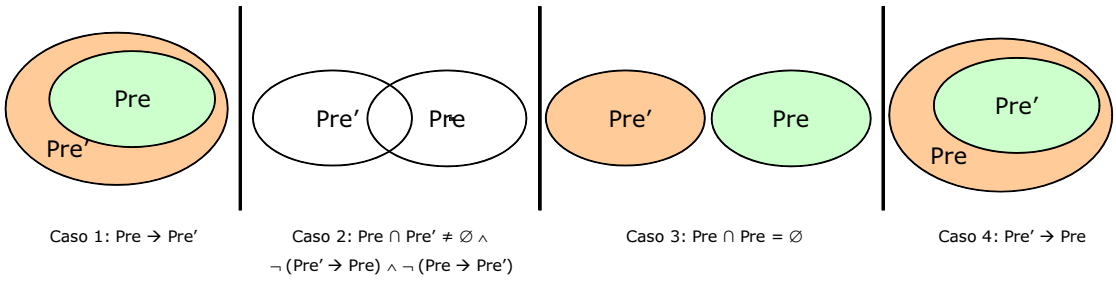


Figura 5.6: Herencia de la precondición. Casos posibles.

Para comprobar de forma automática si se cumple o no esta regla, se reutilizará la metodología propuesta en el apartado 5.2. Los pasos son:

- **Detección de errores**. La nueva precondición será válida si se cumple que la precondición definida en la clase padre, implica el cumplimiento de la precondición definida en la clase hija, es decir: Pre $\Rightarrow$ Pre′. Esta regla debe cumplirse siempre. Para garantizar que no existe un caso en el que no se cumpla dicha implicación, la idea es buscar soluciones al inverso de la regla planteada. Si se encuentran soluciones, cada una de estas soluciones representará un caso en el que no se cumple que la precondición heredada implica la precondición nueva. Por tanto el problema CSP a resolver sería:

  CSP: $\neg(\text{Pre} \Rightarrow \text{Pre}')$ que es equivalente al CSP: $\text{Pre} \wedge \neg\text{Pre}'$

  Si este CSP no tiene solución, será debido a que la nueva precondición es correcta. Si existe alguna solución, dicha solución será un contraejemplo que demostrará que dicha precondición no está bien establecida. Plantear el CSP de esta forma tiene como ventaja que las soluciones obtenidas son casos de prueba en los que la nueva precondición no es consistente con la heredada.

- **Localización de los defectos**. En caso de haber encontrado un contraejemplo que demuestre que la nueva precondición contiene defectos, el siguiente paso será conocer cuál es el motivo, es decir, encontrar el defecto que impide que la nueva precondición no sea consistente con la heredada. Para ello se planteará un problema de diagnosis donde se aplicará como caso de prueba el contraejemplo encontrado al comprobar la precondición. Las restricciones del problema serán obtenidas de transformar cada uno de los asertos que forman la nueva precondición: Pre′.

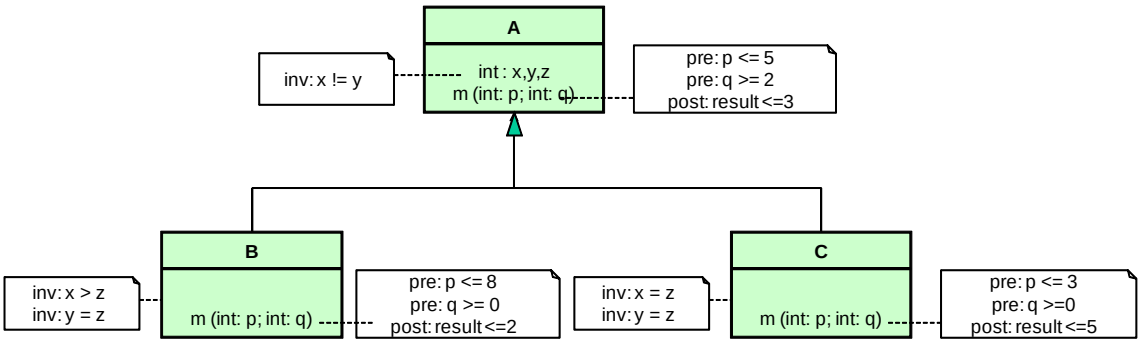


Figura 5.7: Ejemplo de herencia de precondición y postcondición

Además, para tener en cuenta la posibilidad de que los asertos tengan comportamiento anormal, será necesario añadir el predicado AB. El predicado AB será aplicado a cada aserto de los que forman la nueva precondición Pre′. Al resolver el problema de diagnosis se determinará cuáles de los asertos de la nueva precondición contienen defectos, y por tanto, deben ser modificados. Para ello se aplicará la metodología propuesta y se resolverá un problema Max-CSP, tal como se ha explicado en los ejemplos anteriores.

Ejemplo 5.6. En la figura 5.7 se muestra un ejemplo con tres clases: A, B y C. Las clases B y C añaden nuevas precondiciones y postcondiciones para el método *m*. Para comprobar que las nuevas precondiciones son correctas se realizarán las siguientes comprobaciones.

- **Detección de errores**. Para garantizar si el cumplimiento de la precondición heredada (*Pre*) implica siempre el cumplimiento de la nueva precondición (Pre′), se debe comprobar que no hay solución para el CSP: Pre $\wedge \neg$ Pre′.

  Esto se traduce en la clase B por el CSP: p <= 5 $\wedge$ q >= 2 $\wedge$ (p > 8 $\vee$ q < 0).

  No hay solución para dicho problema, ya que *p* no puede ser a la vez menor o igual que 5, y mayor que 8; y *q* no puede ser a la vez mayor o igual que 2, y menor que 0. Por tanto el contrato está bien establecido.

  En el caso de la clase C el CSP a resolver será: p <= 5 $\wedge$ q >= 2 $\wedge$ (p > 3 $\vee$ q < 0)

  Este problema si tiene solución si la variable *p* es igual a 4, o igual a 5. Cada uno de estos valores son soluciones al CSP y contraejemplos que demuestran que la precondición de la clase C debe modificarse, suponiendo correcta la precondición de A.

- **Localización de los defectos**. Para localizar qué asertos de la precondición establecida en el método *m* de la clase C deben modificarse, será necesario plantear

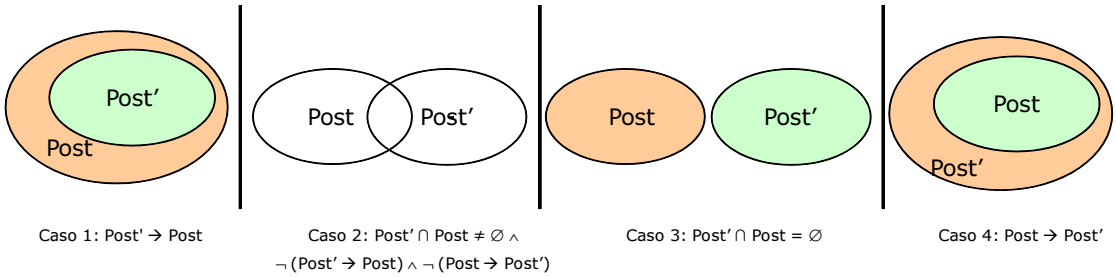


Figura 5.8: Herencia de la postcondición. Casos posibles.

un problema de diagnosis, transformando a restricciones cada uno de los asertos que forman la nueva precondición, y utilizando como caso de prueba uno de los contraejemplos encontrados, por ejemplo p = 4. Las restricciones del problema de diagnosis serían:

Restricciones:	**Dominio**:
$\neg$AB(pre_1) $\Rightarrow$ p <= 3	p = 4
$\neg$AB(pre_2) $\Rightarrow$ q >= 0	q = libre
$\forall$ pre_i : pre_i $\in$ {pre_1, pre_2} : $\neg$AB(pre_i) = cierto	

La resolución del Max-CSP permitirá identificar qué asertos deben modificarse. En este problema la diagnosis mínima es única, y consiste en modificar el aserto p <= 3 de la precondición propuesta en el método m de la clase C.

5.4.3. Postcondición propia y heredada de cada método

Cuando se añaden nuevos asertos a la postcondición de un método heredado, la postcondición resultante (Post′), debe ser igual o más fuerte que la postcondición establecida en el método heredado (Post). De esta forma las salidas que se generen siempre estarán incluidas en el espectro de salidas que el método definido en la clase padre podría generar. Es decir, en ningún momento se producirá una salida en el método definido en la subclase, que no cumpla las condiciones establecidas para las salidas, en el método definido en la clase padre. En la figura 5.8 se muestran los cuatro casos posibles que se pueden producir. Para que la nueva postcondición, Post′, sea correcta, debe cumplirse el caso 1, en el cual, todo el dominio de valores que satisfacen la nueva postcondición Post′ están incluidos en el dominio de valores que satisfacen la postcondición heredada Post.

Para comprobar de forma automática si se cumple o no esta regla, se reutilizará la metodología propuesta en el apartado 5.2. Los pasos serán:

- **Detección de errores**. Para comprobar la validez de la nueva postcondición se debe comprobar que el cumplimiento de la postcondición definida en la clase hija

debe implicar el cumplimiento de la postcondición definida en la clase padre, es decir: $\text{Post}' \Rightarrow \text{Post}$.

Esta regla debe cumplirse siempre. Para garantizar que no existe un caso en el que no se cumpla dicha implicación, la idea es buscar soluciones al inverso de la regla planteada. Si se encuentran soluciones, cada una de estas soluciones representará un caso en el que no se cumple que la nueva postcondición implica la postcondición heredada. Por tanto el problema CSP a resolver sería:

CSP: $\neg(\text{Post}' \Rightarrow \text{Post})$ que es equivalente al CSP: $\text{Post}' \wedge \neg\, \text{Post}$

Si este CSP no tiene solución, será debido a que la nueva postcondición es correcta. Si existe alguna solución, dicha solución será un contraejemplo en el que se demostrará que la nueva postcondición no está bien planteada. Plantear el CSP de esta forma tiene como ventaja que las soluciones pueden ayudar a detectar, y posteriormente eliminar, aquellos casos en los que la nueva postcondición no cumple las condiciones necesarias.

- **Localización de los defectos**. En caso de haber encontrado un contraejemplo, de forma análoga a como se explicó para el caso de la precondición, se plantearía un problema de diagnosis para poder conocer cuál es el defecto, o defectos, que impiden que la postcondición heredada sea consistente con la nueva postcondición.

  El contraejemplo demuestra, que hay un valor valido para la nueva postcondición, pero no para la postcondición heredada. Para no reducir el espectro de valores permitidos por la nueva postcondición, sería necesario modificar la postcondición heredada, de tal forma, que permita ampliar el espectro de valores permitidos. El problema de diagnosis permitirá conocer qué asertos de la postcondición heredada deben ser modificados para cumplir este objetivo.

  En el problema de diagnosis se aplicará como caso de prueba el contraejemplo encontrado al comprobar la postcondición. Las restricciones del problema serán obtenidas de transformar cada uno de los asertos que forman la postcondición heredada: Post.

  Además, para tener en cuenta la posibilidad de que los asertos tengan comportamiento anormal, será necesario añadir el predicado AB. El predicado AB será aplicado a cada aserto de los que forman la nueva postcondición heredada Post. Al resolver el problema de diagnosis se determinará cuáles de los asertos de la postcondición heredada deben ser modificados. Para ello se resolverá un problema Max-CSP, tal como se explicó cuando se mostró la metodología de diagnosis.

Ejemplo 5.7. Tal como aparece en el ejemplo utilizado en el apartado anterior (figura 5.7), las clases B y C añaden nuevas postcondiciones para el método *m*. Para garantizar que las nuevas postcondiciones son correctas se realizarían las siguientes comprobaciones.

- **Detección de errores**. Para garantizar si el cumplimiento de la nueva postcondición ($Post'$) implica siempre el cumplimiento de la postcondición heredada (Post), se debe comprobar que no hay solución para el CSP: Post′ ∧ ¬ Post.

  Esto se traduce en la clase B por el CSP: @result <= 2 ∧ @result > 3.

  No hay solución para dicho problema, ya que *@result* no puede ser a la vez menor o igual que 2 y mayor que 3, por tanto el contrato está bien establecido.

  Y en el caso de la clase C el CSP será: @result <= 5 ∧ @result > 3.

  Este problema sí tiene solución, si el resultado toma los valores 4 o 5. Cada uno de estos valores son soluciones al CSP y contraejemplos que demuestran que la postcondición heredada y la nueva no cumplen las condiciones necesarias.

- **Localización de los defectos**. Para localizar qué asertos de la postcondición heredada deben cambiar en el método m de la clase A, será necesario plantear un problema de diagnosis, transformando a restricciones cada uno de los asertos que forman la postcondición heredada, y utilizando como caso de prueba uno de los contraejemplos encontrados, por ejemplo p = 4. El objetivo es poder determinar qué debe cambiar en la postcondición heredada para que todo el espectro de valores permitidos en la nueva postcondición sean valores permitidos en la postcondición heredada. Las restricciones del problema de diagnosis serían:

Restricciones:	**Dominio**:
$\neg$AB(post$_1$) $\Rightarrow$ @result <= 3	@result = 4
$\forall$ post$_i$: post$_i$ $\in$ {post$_1$} : $\neg$AB(post$_i$) = cierto	

  La resolución del Max-CSP permitirá localizar qué asertos deben modificarse. En este ejemplo la resolución resulta trivial, ya que sólo existe un aserto en la postcondición heredada. En este problema, se obtienen como diagnosis mínima que se debe modificar el aserto @result <= 3 de la postcondición heredada del método m de la clase A.

5.4.4. Precondición, postcondición e invariantes de cada método

Cada vez que un método es ejecutado se debe cumplir la secuencia de asertos: {@old(invariantes) ∧ precondición ∧ postcondición ∧ invariantes}, que corresponde con los invariantes antes de la ejecución del método, la precondición y la postcondición del método, y los invariantes tras la ejecución del método. En esta secuencia, las condiciones impuestas por los invariantes deben ser consistentes con la precondición y la postcondición. Los invariantes establecen condiciones sobre los atributos, que las precondiciones y postcondiciones no deben violar. Si los invariantes entran en contradicción con las condiciones establecidas en la especificación de un método, no será posible desarrollar

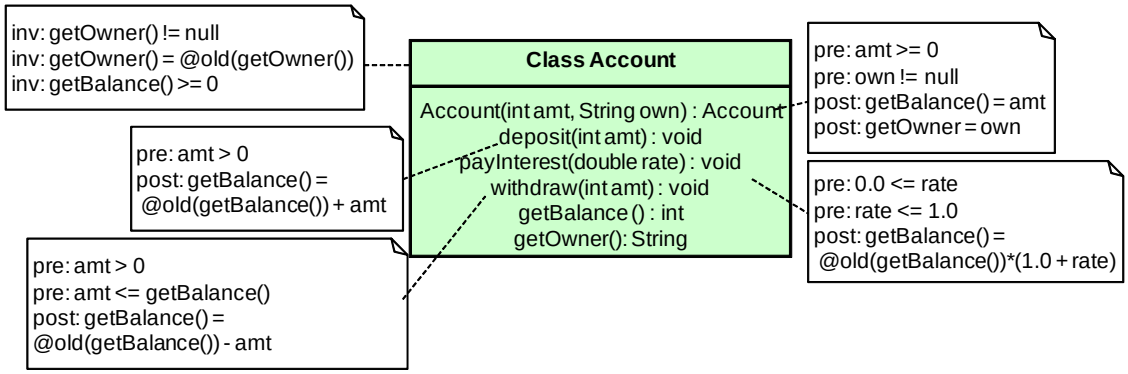


Figura 5.9: Clase account (cuenta bancaria)

la implementación de dicho método, por tanto, es necesario detectar y solucionar este tipo de defectos antes de pasar al desarrollo de la implementación.

El objetivo es comprobar si existe algún conflicto entre la especificación establecida en los invariantes, y la establecida en los métodos, y en caso de conflicto, determinar de forma automática qué parte de la especificación debería cambiar. La idea para conseguirlo será una vez más transformar la especificación a restricciones. Por cada precondición y postcondición más el conjunto de invariantes, la idea es obtener un problema de satisfacción de restricciones que permita detectar e identificar los elementos que deben cambiar en la especificación.

Tal como se ha explicado anteriormente, los invariantes deben cumplirse antes y después de la ejecución de cualquier método, y después de la ejecución de los métodos constructores. En el caso de los métodos constructores, no es necesario cumplir los invariantes antes de su ejecución, ya que es precisamente en el constructor donde deben establecerse los valores apropiados para los atributos del objeto. Supongamos por ejemplo una cuenta bancaria en la que el invariante obliga a tener siempre un titular, si el titular de la cuenta se establece en el constructor sería imposible satisfacer el invariante antes de la ejecución del constructor. Será posible cumplir el invariante una vez ejecutado el constructor, y asignado el titular.

Para comprobar de forma automática si se cumple o no esta regla, se reutilizará la metodología propuesta en el apartado 5.2. Los pasos serán:

- **Detección de errores**. Para comprobar si existen inconsistencias entre los asertos que participan en la ejecución de los métodos, se transformarán a restricciones los asertos que forman la secuencia {@old(invariantes) + precondición + postcondición + invariantes} en cada uno de los métodos. Si no se pueden satisfacer todas las restricciones a la vez, será debido a que son inconsistentes.

- **Localización de los defectos**. En caso de que existan inconsistencias entre los asertos, se planteará un problema de diagnosis que permitirá conocer cuáles

son los defectos que impiden que los asertos se puedan satisfacer a la vez. En el problema de diagnosis se aplicará el caso de prueba vacío. Las restricciones del problema serán obtenidas de transformar cada uno de los asertos que forman la secuencia {@old(invariantes) + precondición + postcondición + invariantes}.

Para tener en cuenta la posibilidad de que los asertos tengan comportamiento anormal, será necesario añadir el predicado AB a cada aserto de los que forman la secuencia. Al igual que en los casos anteriores, la diagnosis mínima se obtendrá al resolver el Max-CSP asociado al problema de diagnosis.

Ejemplo 5.8. En la figura 5.9 se muestra la clase *Account* que representa una cuenta bancaria. El ejemplo contiene dos atributos que representan al titular de la cuenta y el saldo de la cuenta. El saldo se ha representado con un entero para simplificar el problema, y el titular se representa con un objeto de tipo cadena de texto. Como invariantes de la clase se obliga a que el titular de la cuenta se conozca siempre y que no cambie a lo largo de la vida de dicha cuenta bancaria (desde que se crea la cuenta a través del constructor). Además la cuenta no puede tener un saldo negativo.

Los métodos para obtener el titular (*getOwner*) y el valor del saldo (*getBalance*) son métodos observadores y no tienen efectos sobre los atributos del objeto. Además se proponen los métodos depositar (*deposit*), pagar interés (*payInterest*) y retirar dinero (*withdraw*). Cada uno de estos métodos tiene precondición y postcondición. Por ejemplo, antes de depositar una cantidad se garantiza que dicha cantidad no sea negativa, y que el saldo una vez realizada la operación, sea igual al saldo original más la cantidad depositada. Para acceder al valor de un atributo antes de la llamada a un método se usa la función *@old*, que recibe como parámetro el atributo a consultar de la clase.

Supongamos que se quiere añadir a la clase mostrada en el ejemplo un nuevo método llamado *ownerChange*, que permite modificar el titular de la cuenta bancaria. En este ejemplo se supone que el titular de la cuenta es único, y se representa por un objeto de tipo String. La especificación de este método sería como sigue:

Precondición:	newAccountOwner ! = null
Precondición:	newAccountOwner ! = getOwner()
Método:	ownerChange(String newAccountOwner)
Postcondición:	getOwner() = newAccountOwner

Aplicando los pasos de la metodología propuesta se obtendrían los siguientes resultados.

- **Detección de errores**. Para comprobar si la especificación de este método es correcta habrá que transformar a restricciones los elementos que forman la especificación (precondición y postcondición) y los invariantes de la clase. Tal como se ha explicado anteriormente, los invariantes deben cumplirse antes de realizar la llamada al método. El segundo invariante definido en la clase, evita que durante la ejecución de un método se pueda cambiar el titular de la cuenta. Este invariante

debe comparar el titular antes y después de la ejecución del método, por tanto no tiene sentido utilizarlo antes de la ejecución del método. Como norma general, cuando un invariante está especificado usando la función *@old*, sólo se añadirán las restricciones correspondientes a este invariante tras las restricciones de la postcondición. Esta norma es la que se ha seguido en la transformación a restricciones del ejemplo. Por este motivo sólo se han añadido las restricciones que corresponden con el primer y tercer invariante, antes de las restricciones correspondientes a la precondición.

Desde el primer y tercer aserto se realizan llamadas a los métodos observadores *getOwner* y *getBalance*. Dichas llamadas no tienen efectos sobre los atributos de la clase. Tal como ya se ha explicado anteriormente, en la transformación a restricciones cada llamada a un método observador (también llamados de consulta) retornará una variable del tipo establecido en el método, pero con el dominio libre, ya que al no tener en cuenta la implementación de dicho método en las comprobaciones que se están realizando en este capítulo, se presupondrá que puede devolverse cualquier valor. Además, en este ejemplo concreto, dichos métodos observadores no tienen una postcondición que restrinja el dominio de los valores retornados. Si la hubiera tendría que transformarse a restricciones y añadirse al CSP.

Con las restricciones correspondientes a la secuencia {@old(invariantes) $\wedge$ precondición $\wedge$ postcondición $\wedge$ invariantes} se formaría el CSP que se muestra a continuación. Este CSP no tiene solución, por tanto el contrato establecido para el método que se pretende añadir no es consistente con la especificación establecida en los invariantes.

Restricciones:	**Dominio**:
$\neg$AB(oldInv1) $\Rightarrow$ @old(getOwner()) ! = null	@old(getOwner()) = libre
$\neg$AB(oldInv3) $\Rightarrow$ @old(getBalance()) >= 0	@old(getBalance()) = libre
$\neg$AB(Pre1) $\Rightarrow$ newAccountOwner ! = null	newAccountOwner = libre
$\neg$AB(Pre2) $\Rightarrow$ newAccountOwner ! = @old(getOwner())	
$\neg$AB(Post) $\Rightarrow$ getOwner() = newAccountOwner	getOwner() = libre
$\neg$AB(Inv1) $\Rightarrow$ getOwner() ! = null	
$\neg$AB(Inv2) $\Rightarrow$ getOwner() = @old(getOwner())	
$\neg$AB(Inv3) $\Rightarrow$ getBalance() >= 0	getBalance() = libre
$\forall$ a_i : $a_i \in$ {oldInv1, oldInv3, Pre1, Pre2, Post, Inv1, Inv2, Inv3} : $\neg$AB(a_i) = cierto	

- **Localización de los defectos**. Para determinar qué asertos de la especificación son defectuosos será necesario añadir la función objetivo y solucionar el problema Max-CSP que a continuación se muestra.

Restricciones:	**Dominio**:
¬AB(oldInv1) ⇒ @old(getOwner()) ! = null	@old(getOwner()) = libre
¬AB(oldInv3) ⇒ @old(getBalance()) >= 0	@old(getBalance()) = libre
¬AB(Pre1) ⇒ newAccountOwner ! = null	newAccountOwner = libre
¬AB(Pre2) ⇒ newAccountOwner ! = @old(getOwner())	
¬AB(Post) ⇒ getOwner() = newAccountOwner	getOwner() = libre
¬AB(Inv1) ⇒ getOwner() ! = null	
¬AB(Inv2) ⇒ getOwner() = @old(getOwner())	
¬AB(Inv3) ⇒ getBalance() >= 0	getBalance() = libre

F. Objetivo: Max(N r_i : $r_i \in$ {oldInv1, oldInv3, Pre1, Pre2, Post, Inv1, Inv2, Inv3} : ¬AB(r_i) = cierto)

La diagnosis mínima que se obtendría para este ejemplo estaría formada por {Pre2, Post, Inv2}. La solución a las inconsistencias en la especificación es modificar o eliminar cualquiera de estos tres asertos para que la secuencia de asertos sea consistente. El resto de la especificación no influye en el error detectado.

El nuevo método permite modificar el titular de una cuenta. Esta operación no fue contemplada cuando se establecieron los invariantes, ya que para realizar esta operación se exige que el nuevo titular sea diferente al antiguo. Tal como muestra la diagnosis mínima, para resolver esta inconsistencia es necesario modificar la especificación del nuevo método (Pre2 o Post) o modificar el invariante (Inv2). En concreto será necesario modificar alguno de los asertos incluidos en la diagnosis mínima obtenida a través de la metodología propuesta.

5.4.5. Diagnóstico utilizando casos de prueba concretos

Tal como se ha explicado anteriormente, para comprobar los asertos que forman una especificación, se pueden utilizar casos de prueba vacíos o concretos. Para los casos de prueba concretos es necesario realizar un trabajo previo que permita generar los casos de prueba más adecuados. Las comprobaciones con casos de prueba concretos pueden considerarse como complementarias a las ya realizadas usando los casos de prueba vacíos.

La idea es utilizar casos de prueba concretos para poder detectar defectos que las comprobaciones con casos de prueba vacíos no podrían detectar. El motivo es simple, cuando se realizan comprobaciones con casos de prueba vacíos no se establecen restricciones a las entradas y salidas, salvo las establecidas por la especificación. Sin embargo, al utilizar un caso de prueba concreto se establecen más condiciones que deben cumplirse, ya que las entradas y salidas están fijadas, y no pueden ser diferentes a las establecidas en el caso de prueba. La especificación debe ser consistente con las condiciones establecidas en los casos de prueba.

La comprobación realizada para un caso de prueba concreto sólo es válida para dicho caso de prueba. Es decir, si se comprueba que la especificación es correcta para un caso de prueba concreto, sólo se puede garantizar que es correcta para dicho caso. De manera análoga, los defectos detectados usando un caso de prueba concreto no tienen por qué manifestarse en otros casos de prueba concretos.

En esta tesis se supondrá que los casos de prueba concretos serán seleccionados siguiendo una metodología que permita una cobertura adecuada. No es el objetivo de esta tesis la generación de los casos de prueba concretos. Se supondrá que los casos de prueba vienen dados. Por ejemplo, se podrían utilizar para comprobar la validez de la especificación de los métodos los mismos casos de prueba que se hayan diseñado para realizar pruebas unitarias para el código fuente de dichos métodos.

En los ejemplos mostrados en las secciones anteriores para la detección y diagnóstico de defectos en los invariantes, precondiciones y postcondiciones, se han utilizado casos de prueba vacíos. El proceso de detección y diagnóstico no varía si se dispone de casos de prueba concretos, simplemente se añadirían las condiciones que establece el caso de prueba concreto en forma de restricciones sobre los dominios de las variables del CSP. E igual que en los ejemplos mostrados anteriormente, resolviendo el problema de diagnosis se identificarán los asertos con defectos en su diseño.

Ejemplo 5.9. En el ejemplo de la figura 5.9 (clase *Account*) visto anteriormente, aparece el método retirar dinero (*withdraw*). Para poder aplicar la metodología de diagnosis propuesta, vamos a introducir un defecto en dicha especificación, quedando la especificación como sigue:

Pre:	amt > 0
Método:	withdraw(int amt)
Post:	getBalance() = @old(getBalance())

El siguiente caso de prueba concreto se especifica cuáles son las salidas esperadas del método para las entradas propuestas:

Método a probar:	Withdraw
Entradas:	@old(getBalance()) = 100, amt = 100, @old(getOwner())= 'Bank'
Salidas:	getBalance() = 0, getOwner() = 'Bank'

- **Detección de errores**. El caso de prueba propuesto ejecutaría el método *withdraw*. Para comprobar si la especificación del método es correcta habrá que transformar a restricciones los elementos que forman la especificación (precondición y postcondición) y los invariantes de la clase. A continuación se muestra el CSP formado por las restricciones obtenidas al transformar la secuencia {@old(invariantes) $\wedge$ precondición $\wedge$ postcondición $\wedge$ invariantes}.

Restricciones:	**Dominio**:
¬AB(oldInv1) ⇒ @old(getOwner()) ! = null	@old(getOwner()) = libre
¬AB(oldInv3) ⇒ @old(getBalance()) >= 0	@old(getBalance()) = libre
¬AB(Pre) ⇒ amt > 0	amt = libre
¬AB(Post) ⇒ getBalance() = @old(getBalance())	getBalance() = libre
¬AB(Inv1) ⇒ getOwner() ! = null	getOwner() = libre
¬AB(Inv2) ⇒ getOwner() = @old(getOwner())	
¬AB(Inv3) ⇒ getBalance() >= 0	

$\forall$ a_i : $a_i \in$ {oldInv1, oldInv3, Pre, Post, Inv1, Inv2, Inv3} : ¬AB(a_i) = cierto

Restricciones debidas al caso de prueba:
@old(getBalance()) = 100 ∧ amt = 100 ∧ @old(getOwner())= 'Bank'
∧ getBalance() = 0 ∧ getOwner() = 'Bank'

Este problema no tiene solución debido a las condiciones impuestas por el caso de prueba.

- **Localización de los defectos**. Para determinar que asertos de la especificación son defectuosos habría que solucionar el problema que a continuación se muestra, donde se ha añadido la función objetivo.

Restricciones:	**Dominio**:
¬AB(oldInv1) ⇒ @old(getOwner()) ! = null	@old(getOwner()) = libre
¬AB(oldInv3) ⇒ @old(getBalance()) >= 0	@old(getBalance()) = libre
¬AB(Pre) ⇒ amt > 0	amt = libre
¬AB(Post) ⇒ getBalance() = @old(getBalance())	getBalance() = libre
¬AB(Inv1) ⇒ getOwner() ! = null	getOwner() = libre
¬AB(Inv2) ⇒ getOwner() = @old(getOwner())	
¬AB(Inv3) ⇒ getBalance() >= 0	

Restricciones debidas al caso de prueba:
@old(getBalance()) = 100 ∧ amt = 100 ∧ @old(getOwner())= 'Bank'
∧ getBalance() = 0 ∧ getOwner() = 'Bank'

F. Objetivo: Max(N a_i : $a_i \in$ {oldInv1, oldInv3, Pre, Post, Inv1, Inv2, Inv3} : ¬AB(a_i) = cierto)

De las restricciones mostradas, la correspondiente al aserto *Post* es la que debe cambiar, conforme a la solución del problema Max-CSP. El resto de los asertos no entran en conflicto con las condiciones impuestas en el caso de prueba. Como se puede observar, el problema está en que la postcondición no contiene la relación correcta entre el saldo de la cuenta tras la operación de reintegro, y el saldo de la cuenta antes de la operación.

5.5. Conclusiones

Tal como se explicó en la introducción de este documento, el primer objetivo de esta tesis es el diagnóstico sobre la especificación del software. Concretamente, el objetivo es comprobar la consistencia entre los asertos que forman la especificación, y determinar en qué partes de la especificación existen defectos.

La metodología propuesta se basa en la transformación a restricciones de los asertos que forman la especificación. Con estas restricciones se forman diferentes problemas de satisfacción de restricciones, cuya resolución permite detectar errores en la especificación. Cuando se detectan errores, las restricciones se incorporan a un problema de diagnosis que es resuelto mediante un Max-CSP. La resolución del problema de diagnosis permite determinar qué asertos de la especificación deben cambiar.

El código fuente debe cumplir la especificación establecida, por tanto, para evitar propagar defectos al desarrollo del código fuente, es necesario establecer mecanismos que permitan detectar y localizar los defectos contenidos en la especificación. Con la metodología propuesta es posible realizar comprobaciones de forma automática sobre la especificación antes de pasar a la implementación. Otra gran ventaja de la metodología propuesta está en la cantidad de información que se evita revisar. La metodología de diagnosis identifica de forma automática cuáles de los asertos deben cambiar, evitando revisar el resto de los asertos. Si no se tiene una metodología para el diagnóstico sería necesario revisar todos los asertos, hasta encontrar cuáles deben cambiar, con el consiguiente gasto en recursos humanos.

Capítulo 6

Diagnóstico del código fuente

6.1. Introducción

Los casos de prueba permiten comprobar el grado de conformidad que se ha alcanzado sobre los requisitos establecidos, y detectar si hay diferencias entre el comportamiento real y el esperado de un sistema software. En concreto, cuando se trata de comprobar un código fuente, los casos de prueba establecen para unas entradas determinadas, cuáles son los resultados correctos que se deberían obtener. De esta forma se pueden comparar los resultados esperados con los proporcionados en la ejecución del código fuente, y detectar si un código fuente genera, o no, los resultados establecidos como correctos.

Las discrepancias entre el comportamiento real y el esperado se considerarán errores. Si la ejecución del caso de prueba genera un error, para poder localizar el origen de este error, el desarrollador debe depurar de forma manual el código fuente. Los casos de prueba permiten detectar errores, pero no permiten localizar de forma automática qué defectos de diseño en el código fuente han dado lugar a los errores detectados.

En este capítulo se adaptará la metodología de diagnosis propuesta en el capítulo 4 con el objetivo de poder identificar defectos en el código fuente. En concreto los defectos a localizar son de tipo semántico, se trata de sentencias mal diseñadas, que deben ser modificadas o eliminadas para poder obtener los resultados especificados como correctos. El objetivo de esta tesis no es la generación de pruebas, se supondrá que los casos de prueba vienen dados, y que han sido seleccionados siguiendo una metodología que permita una cobertura adecuada.

En primer lugar, en este capítulo se mostrarán cuáles son las adaptaciones necesarias con respecto a la metodología general de diagnosis mostrada en el capítulo 4. En segundo lugar se mostrará de manera detallada cuáles son los pasos a seguir para la detección de errores y localización de los defectos.

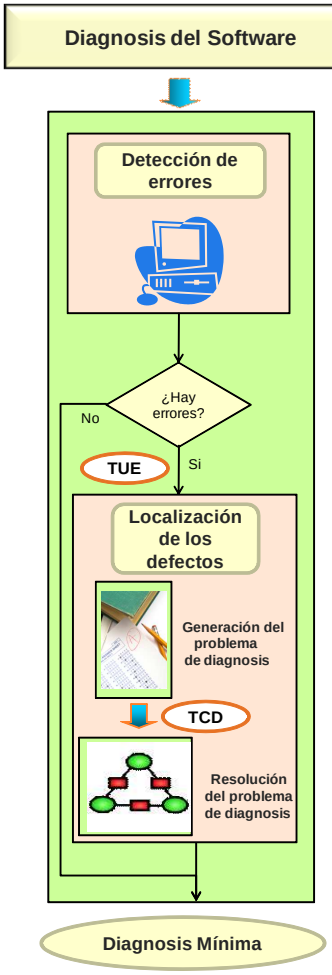


Figura 6.1: Fases de la metodología de diagnóstico aplicada al código fuente

6.2. Adaptación de la metodología de diagnosis

Un problema de diagnosis, tal como se expuso en el capítulo 4, está formado por la tupla (SD, STS, SPEC, TC), donde SD es la descripción del sistema, que incluye un conjunto de restricciones; STS es el conjunto de las sentencias que forma el código fuente; SPEC es el conjunto de asertos que forman la especificación; y TC representa un caso de prueba. En este capítulo, los elementos a diagnosticar serán las sentencias del código fuente (STS).

El proceso de diagnóstico consta de dos fases, la detección de errores y la localización de los defectos. De forma resumida, el objetivo de cada una de estas fases será:

1. **Detección de errores**. El primer paso será comprobar si aplicando los casos de prueba se producen errores. Los casos de prueba establecen las entradas que

se deben proporcionar al código fuente y los resultados correctos que se deben obtener. Si el sistema software proporciona los resultados que establece el caso de prueba como correctos, entonces no existen discrepancias, y en consecuencia, tampoco errores. Un error es una discrepancia entre los resultados especificados en el caso de prueba, y los que el sistema software genera cuando se aplica el caso de prueba.

2. **Localización de los defectos**. Para cada uno de los casos de prueba donde se hayan detectado errores se generará un problema de diagnosis. Las restricciones de la descripción del sistema se obtendrán como transformación de las sentencias del código fuente y los asertos de la especificación que pertenezcan a la traza obtenida como resultado de la ejecución de un caso de prueba. Estas restricciones permitirán simular el comportamiento real del código fuente y de los asertos que forman la especificación para los casos de prueba propuestos. Sólo las sentencias que han participado en la traza o trazas, pueden contener defectos, pues sólo ellas han influido en los resultados. La resolución del problema de diagnosis permitirá identificar cuáles son las sentencias del código fuente que deben cambiar para poder llegar a obtener el resultado esperado en las pruebas realizadas.

En la figura 6.1 se muestra qué se recibe como información, y qué se obtiene como resultado en cada uno de los pasos de la metodología de diagnosis. Aunque algunos de los conceptos que aparecen en dicha figura se definirán con detalle más adelante, esta figura permite tener una visión general del flujo de información entre los diferentes pasos de la metodología de diagnosis.

1. **Detección de errores**. Para la detección de errores se utilizarán como entradas el código fuente, los asertos de la especificación, y un caso de prueba. Si se detectan errores al ejecutar el caso de prueba, la información correspondiente a la traza ejecutada se almacenará en una TUE (Traza de Unidades Ejecutadas) que será tratada en la siguiente fase.

2. **Localización de los defectos**. Para obtener las restricciones de la descripción del sistema del problema de diagnosis, la TUE será transformada a una TCD, Traza de Componentes Diagnosticables. La TCD contiene todas las restricciones de la descripción del sistema, agrupadas en componentes, de tal forma que las restricciones relacionadas entre sí estarán agrupadas en componentes. Esta característica permitirá, tal como se verá en el capítulo siguiente, realizar la localización de los defectos de forma más eficiente. Con las restricciones recogidas en el TCD, y el caso de prueba se obtendrá la diagnosis mínima. La diagnosis mínima establece qué sentencias del código fuente deben cambiar.

```
01    SoftwareSystem ::= Class*
02    Class ::= ‘class’ Identifier (‘extends’ Identifier)? ‘{’ (Invariant
            | Field | Constructor | Method)* ‘}’
03    Invariant ::= ‘inv:’ AssertExp
04    Field ::= Type AsigStat
05    Constructor ::= Precondition* Identifier ‘(’ Param* ‘)’ ‘{’ Block ‘}’ Postcondition*
06    Precondition ::= ‘pre:’ AssertExp
07    Postcondition ::= ‘post:’ AssertExp
08    Method ::= Precondition* Type Identifier ‘(’ Param* ‘)’ ‘{’ Block ‘}’ Postcondition*
09    Param ::= Type ‘:’ Identifier
10    Block ::= Element*
11    Element ::= LocalVariable | Statement
12    LocalVariable ::= Type AsigStat
13    Statement ::= IfElse | While | Return | Assert | AsigStat | Call | Constructor
14    IfElse ::= ‘if’ ‘(’ Exp ‘)’ ‘{’ Block ‘}’ (‘else’ ‘{’ Block ‘}’)? ‘;’
15    While ::= ‘while’ ‘(’ Exp ‘)’ ‘{’ Block ‘}’ ‘;’
16    Return ::= ‘return’ Exp ‘;’
17    Assert ::= ‘assert:’ AssertExp ‘;’
18    AsigStat ::= Identifier  ‘=’ AsigExp
19    AsigExp ::= Exp ‘;’ | Call | Constructor
20    Call ::= Identifier ‘(’ Argument* ‘)’ ‘;’
21    Constructor ::=  ‘new’ Type ‘(’ Argument* ‘)’ ‘;’
22    Argument ::= Exp
23    Exp ::= AritmeticExp | LogicExp | UnaryExp | Literal | Identifier
```

Figura 6.2: Sintaxis del lenguaje soportado por la metodología propuesta

6.3. Detección de errores

En este apartado se describirán cuáles son los resultados que se pueden obtener al ejecutar un caso de prueba, y cómo debe plantearse el problema de diagnosis en función de dichos resultados. En concreto, si se detectan discrepancias al ejecutar un caso de prueba, la información correspondiente a la traza ejecutada se almacenará en una TUE (Traza de Unidades Ejecutadas). Esta información será posteriormente transformada en la etapa de localización de defectos con el objetivo de generar la descripción del sistema del problema de diagnosis.

Antes de comenzar a describir los puntos reseñados en el párrafo anterior, a continuación se describirán de forma detallada cuáles son los elementos que forman la gramática que debe cumplir el código fuente a tratar.

6.3.1. Características del código fuente a diagnosticar

El lenguaje utilizado en los ejemplos de código fuente que aparecerán en este capítulo es una simplificación del lenguaje Java, el lenguaje Orientado a Objetos más utilizado en la actualidad. En la figura 6.2 se muestra la gramática del lenguaje. La descripción se ha realizado utilizando la notación EBNF (Extended Backus Normal Form). Como se puede observar incluye las principales características de un lenguaje Orientado a Objetos de tipo imperativo, tales como la creación y gestión de objetos, la reutilización a través de la herencia entre clases, y la especificación de contratos.

El código fuente a diagnosticar debe cumplir la sintaxis establecida en dicho lenguaje

de programación. La metodología podría ser adaptable a cualquier sistema software desarrollado en un lenguaje de programación que pueda ser transformado a la sintaxis mostrada en la figura 6.2. A continuación se hará una descripción de forma breve de cada uno de los elementos que forman la gramática propuesta en la figura 6.2.

1. Sistema Software. Estará formado por un conjunto de clases, y la especificación asociada a dichas clases.

2. Clase. Cada clase puede incluir invariantes, el nombre identificador de otra clase de la que herede, atributos, métodos y constructores.

3. Invariante. Un invariante es una expresión que debe cumplirse siempre antes y después de la ejecución de cualquier método.

4. Atributo. Además del tipo correspondiente al atributo, cada atributo incluye una asignación inicial, que inicializa el valor correspondiente al atributo.

5. Declaración de un constructor. Cada constructor permite instanciar un objeto de una clase concreta. Incluye un nombre identificador (coincidirá con el nombre de la clase que lo contiene), la lista de parámetros de entrada, el bloque de sentencias asociado, la precondición y la postcondición del constructor.

6. Precondición. La precondición contiene una expresión que debe cumplirse antes de ejecutar el método o constructor al que este asociada.

7. Postcondición. La postcondición contiene una expresión que debe cumplirse después de ejecutar el método o constructor al que este asociada.

8. Declaración de un método. Un método es una operación que se puede realizar sobre los atributos de un objeto. Cada método incluye un nombre identificador, la lista de parámetros de entrada, el bloque de sentencias asociado, el tipo correspondiente a la expresión devuelta si el método genera un resultado, la precondición y la postcondición.

9. Parámetro. Cada parámetro de entrada de un método o constructor contiene el tipo del parámetro y el identificador del mismo.

10. Bloque de elementos. Almacena una lista de elementos que deben ejecutarse de forma secuencial.

11. Elemento. Puede ser una declaración de una variable local o una sentencia.

12. Declaración de variable local. Cada declaración de este tipo incluye el tipo correspondiente a la variable local y la asignación inicial sobre la variable local.

13. Sentencia. Puede ser una sentencia selectiva, una sentencia iterativa, el retorno de un método, un aserto, una asignación, o una llamada a un método o constructor.

14. Sentencia selectiva. Una sentencia selectiva contiene una expresión que al ser evaluada genera un valor lógico. Dependiendo del valor lógico se ejecutará alguno de los dos bloques de sentencias asociado. Debe existir al menos un bloque de sentencias asociado a la sentencia selectiva, que se ejecutará si la condición es evaluada y genera como resultado el valor verdadero.

15. Sentencia de tipo iterativo o bucle. Una sentencia tipo iterativo contendrá una expresión que se evaluará como condición para ejecutar un bloque de sentencias mientras la condición se cumpla.

16. Retorno de un método. Una sentencia de retorno permite almacenar una expresión que al evaluarse será devuelta como resultado del método que la contiene.

17. Aserto. Un aserto contiene una expresión que debe cumplirse al ejecutarse, es decir, al ser evaluada. Esta expresión forma parte de la especificación. Puede contener variables locales, parámetros de entrada y atributos de objetos. Aparecerán incluidos en un bloque de sentencias asociado a un método, y pueden utilizarse, por ejemplo, para establecer el invariante de una sentencia iterativa.

18. Asignación. Las asignaciones permiten almacenar una expresión asignable en una variable local, o parámetro de entrada, o atributo de un objeto. Almacena el nombre identificador de la variable local, parámetro de entrada o atributo del objeto, y la expresión asignable.

19. Expresión asignable. Puede ser una expresión, o una llamada a un método, o una llamada a un constructor. Si se trata de una llamada a un método, la expresión asignable será el resultado devuelto por dicho método. Si se trata de una llamada a un constructor, la expresión asignable será el objeto creado.

20. Llamada a un método. La llamada a un método permite ejecutar un método. Contiene el nombre identificador del método, y los argumentos de entrada.

21. Llamada a un constructor. La llamada a un constructor permite ejecutar un constructor. Contiene el nombre identificador del constructor, y los argumentos de entrada.

22. Argumento de entrada de un método o constructor. Almacena una expresión que será asignada a un parámetro de entrada, en la ejecución de un método o constructor.

23. Expresión. Se admitirán expresiones aritméticas, lógicas, literales, e identificadores de variables locales, parámetros de entrada o atributos de objetos.

Además de los asertos que forman los invariantes de las clases, precondiciones, y postcondiciones, se pueden insertar dentro del código fuente sentencias de tipo aserto

para obligar a cumplir ciertas condiciones. Este tipo de asertos permite por ejemplo añadir invariantes en los bucles. En el lenguaje presentado, la evaluación de los invariantes, asertos, precondiciones y postcondiciones no genera ningún resultado, ya que por definición su evaluación no debe tener efectos sobre el código fuente.

Los identificadores utilizados para referenciar las variables locales y los parámetros de entrada serán simples, es decir, sólo incluirán un nombre. Los identificadores utilizados para hacer referencia a los atributos y métodos de un objeto, podrán ser simples o compuestos. En el caso de ser compuestos, incluirán los nombres que permitirán navegar hasta llegar al objeto que almacena el atributo o método deseado. Por ejemplo, para acceder al atributo a del objeto almacenado en la variable local v de un método, el nombre compuesto sería *v.a*. Para acceder al atributo b que pertenece al propio objeto, desde cualquiera de sus métodos, basta con escribir el nombre, es decir, se utilizaría un identificador simple. El operador punto (.) será utilizado en la sintaxis para separar los distintos nombres que compongan un identificador compuesto. También serán válidos como identificadores las palabras reservadas *this* y *super* que hacen referencia respectivamente al propio objeto y al objeto correspondiente a la clase padre.

6.3.2. Resultados de las pruebas

Los casos de prueba establecen los resultados que se consideran correctos, es decir, cuando se aplica un caso de prueba, TC, a un sistema software formado por un conjunto de sentencias, STS, y un conjunto de asertos, SPEC, siempre se supondrá que el TC es correcto, y que los defectos pueden estar contenidos en los elementos que forman los conjuntos STS o SPEC.

Al ejecutar un caso de prueba sobre un sistema software se pueden producir las siguientes situaciones:

1. **Ejecución completa del código y de la especificación**. La ejecución de las sentencias del código fuente ha producido unos resultados en un tiempo finito, y las condiciones de la especificación se han cumplido. En esta situación, el siguiente paso es comprobar si se ha cumplido el caso de prueba o no:

   a) Si los resultados reales generados en la ejecución no contradicen a los resultados establecidos en el caso de prueba, al no haber discrepancia se supondrá que el sistema software no tiene defectos.
   Errores. No hay errores en los resultados obtenidos para el caso de prueba TC.
   Defectos. STS y SPEC no tienen defectos de diseño.

   b) Si los resultados reales generados en la ejecución contradicen a los resultados establecidos en el caso de prueba, será debido a que debe existir al menos un defecto que provoca esta discrepancia. Para poder determinar qué defectos en STS han provocado los errores detectados se planteará un problema

de diagnosis. Sólo las sentencias del código fuente influyen en los resultados generados, por tanto sólo se buscarán defectos en dichas sentencias, y se supondrá que la especificación es correcta.
Errores. Hay errores en los resultados obtenidos para el caso de prueba TC.
Defectos. Existen defectos de diseño en las sentencias que forman el conjunto STS. Se supondrá correcta la especificación, formada por los asertos del conjunto SPEC.

2. **Ejecución incompleta debido a que uno de los asertos de la especificación no se ha podido satisfacer**. La ejecución del código fuente ha dado lugar a que al menos uno de los asertos de la especificación, a_i, no se haya cumplido. Para tomar una decisión será necesario comprobar si el problema se debe al código fuente o a los asertos que forman la especificación. Para ello se seguirán los siguientes pasos:

   - Se creará un conjunto vacio de asertos SPEC′, donde se irán almacenando los asertos fallidos. Se añadirá a_i a dicho conjunto SPEC′, y se eliminará del conjunto SPEC.
   - Se ejecutará de nuevo el sistema software formado por las sentencias del conjunto STS y los asertos incluidos en el conjunto SPEC.
   - Si al ejecutar el caso de prueba no es posible satisfacer otro aserto, a_j, se añadirá dicho aserto en el conjunto SPEC′ y se eliminará del conjunto SPEC. Se volverá al paso anterior mientras no se consiga ejecutar todos los asertos incluidos en el conjunto SPEC.

   Cuando se cumpla que se pueden ejecutar todos los asertos que han quedado incluidos en SPEC para el caso de prueba TC, se pasará a comprobar si el resultado generado por las sentencias del conjunto STS es acorde al resultado establecido en el caso de prueba. En este punto se pueden dar las siguientes situaciones:

   a) Si no hay discrepancias entre el resultado establecido en el caso de prueba y el generado tras la ejecución, habría que revisar si los asertos no satisfechos (forman el conjunto SPEC′) son correctos, ya que no son compatibles con un código fuente que si está generando el resultado esperado.
   Errores. No hay errores en los resultados obtenidos para el caso de prueba TC, si se evita la evaluación de los asertos que forman el conjunto SPEC′.
   Defectos. Los asertos incluidos en SPEC′ son inconsistentes con el caso de prueba TC, deben existir defectos en su diseño.

   b) Si hay discrepancias entre el resultado establecido en el caso de prueba y el generado tras la ejecución, se supondrá que la especificación incluida en SPEC′ es correcta y se planteará un problema de diagnosis para encontrar los defectos en las sentencias que forman el conjunto STS.

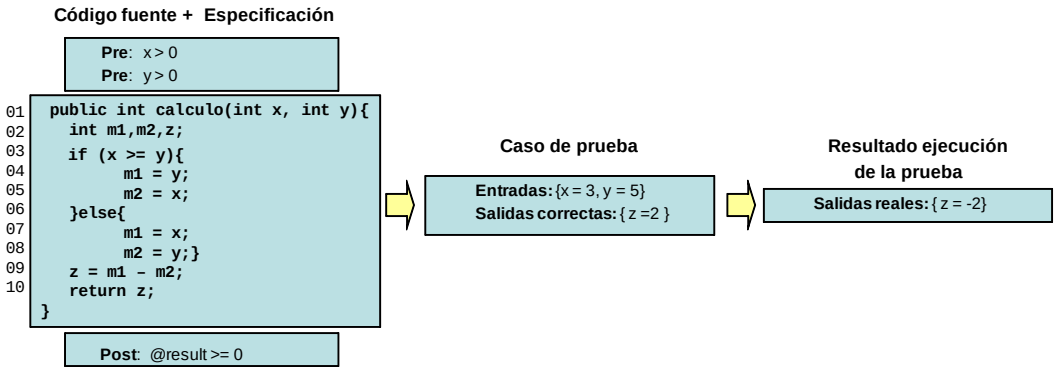


Figura 6.3: Detección de errores usando casos de prueba

Errores. Hay errores en los resultados obtenidos para el caso de prueba TC.
Defectos. Existen defectos de diseño en las sentencias que forman el conjunto STS. Se supondrá correcta la especificación, formada por los asertos del conjunto SPEC.

En este trabajo se supondrá que cuando se evalúa un aserto de la especificación, y no es posible cumplirlo, se producirá una parada en la ejecución, y que tras dicha parada será posible conocer qué aserto de la especificación es la que ha producido la parada. Este tipo de comportamiento puede ser modelado a través de excepciones. Es decir, cada vez que no se pueda satisfacer la condición impuesta en las expresiones que forman un aserto de la especificación, se interrumpirá la ejecución y se lanzará una excepción que contendrá información sobre el aserto de la especificación que no se puede satisfacer.

Es conveniente resaltar, tal como se ha expuesto en los casos 1b y 2b, que al plantear el problema de diagnosis se supondrán correctos los asertos que forman la especificación, SPEC. La idea que se persigue con esta suposición es aprovechar la información contenida en la especificación de forma que pueda ser útil en la localización de los defectos semánticos, pues los asertos establecen condiciones que el código fuente debe cumplir.

Ejemplo 6.1. En la figura 6.3 se muestra un sistema software de ejemplo. Aplicando el caso de prueba mostrado, se consigue ejecutar todo el código fuente, pero no se cumple la postcondición establecida. Para tomar una decisión, debido a que al menos un aserto de especificación no se ha podido satisfacer, será necesario ir eliminando del conjunto SPEC todos aquellos asertos que no se pueden satisfacer.

En el caso de prueba propuesto, basta eliminar la postcondición del conjunto SPEC para lograr que se ejecute todo el código fuente, y que se cumplan las condiciones establecidas en los asertos que quedan en el conjunto SPEC. Para decidir si el problema está en la postcondición o en el código fuente, el siguiente paso será comprobar si el

código fuente ha generado el resultado especificado en el caso de prueba. Se pueden dar dos situaciones:

1. Si el resultado de la salida z coincide con el valor establecido como correcto en el caso de prueba, habría que revisar si los asertos no satisfechos son correctos. Es decir, habrá que revisar la postcondición.

2. Si el resultado de la salida z no coincide con el valor establecido como correcto en el caso de prueba, se supondrá que la especificación es correcta, y se planteará un problema de diagnosis para localizar los defectos de diseño en el código fuente.

En el caso de prueba mostrado para el ejemplo, el resultado generado no coincide con el establecido en el caso de prueba, por tanto la opción a seguir será la segunda, es decir, plantear un problema de diagnosis y localizar los defectos en el código fuente.

6.3.3. Sentencias y asertos ejecutados

Cuando un código fuente se pone en funcionamiento, la traza de sentencias ejecutadas depende de las entradas recibidas, del estado inicial de los atributos de los objetos, y de las condiciones de las sentencias de control de flujo (sentencias selectivas e iterativas). Una traza de ejecución es la secuencia de sentencias que son ejecutadas como consecuencia de aplicar un caso de prueba a un código fuente. Para un mismo código fuente, diferentes casos de prueba pueden provocar trazas diferentes.

Sólo las sentencias que han participado en la traza ejecutada pueden contener defectos, pues sólo ellas han influido en los resultados. La metodología de diagnosis se centrará en estudiar trazas concretas, de esta forma, se conseguirá eliminar de la descripción del sistema aquellas sentencias que no han influido en los errores detectados, y se mantendrán sólo aquellas sentencias que pueden tener defectos semánticos. Por este motivo, las restricciones que formarán la descripción del sistema se obtendrán como resultado de la transformación de la traza ejecutada. Como consecuencia será necesario determinar cuál ha sido la traza seguida por el código fuente para cada caso de prueba, antes de generar el problema de diagnosis.

En la programación orientada a objetos la generación de un comportamiento se consigue mediante la ejecución conjunta de diferentes métodos. Cada método tendrá asociado un comportamiento que dependerá del estado del objeto, las entradas, y las instrucciones incluidas en éste. La traza ejecutada será una secuencia de sentencias obtenida a través de los enlaces de diferentes llamadas a métodos y constructores.

Para poder determinar qué parte del código fuente ha participado en la traza seguida, y qué asertos han sido evaluados, es necesario auditar el sistema software en ejecución, o simular su funcionamiento. De esta forma se puede recuperar la información concerniente a las trazas correspondientes a los casos de prueba. La idea es almacenar la información necesaria para conocer la traza seguida, como por ejemplo los métodos ejecutados o las iteraciones realizadas por los bucles. En las pruebas realizadas

en este trabajo para la metodología propuesta, la técnica seguida ha sido la de auditar el sistema software. Para ello se modificó el código fuente de tal forma que cada vez que se ejecutaba una sentencia, la ejecución quedaba recogida.

Para cada una de las trazas auditadas se generará una Traza de Unidades Ejecutadas (TUE). Cada TUE almacena la traza seguida por el sistema, es decir, la secuencia de sentencias ejecutadas y asertos evaluados para cada uno de los casos de prueba. Además contendrá información semántica extraída del código fuente y de los asertos, que será necesaria para generar las restricciones asociadas a la descripción del sistema.

6.3.4. Traza de Unidades Ejecutadas

Para manejar las trazas seguidas en los casos de prueba, se hará uso del concepto de **Unidad Ejecutada (UE)**. Cada uno de los asertos y sentencias pertenecientes a la traza ejecutada, será transformado a una UE. Una sentencia S_i del código fuente puede ejecutarse varias veces en una misma traza, y cada una de estas apariciones producirá una nueva UE asociada a dicha sentencia S_i. Lo mismo ocurrirá con los asertos. Por tanto, varias UE pueden corresponder con una misma sentencia del código fuente, ya que por cada ejecución de una sentencia en la traza, se obtendrá una UE diferente.

Cada una de las ejecuciones de una misma sentencia puede recibir diferentes entradas y producir diferentes salidas dentro de la traza. Es más, puede que un defecto en una sentencia sólo se manifieste en una determinada aparición de la sentencia en la traza, y no en el resto de las apariciones de la sentencia en la traza. Por tanto, todas las ejecuciones de las sentencias en la traza seguida deben tener un reflejo en la descripción del sistema utilizada en el proceso de diagnóstico.

Cada **Traza de Unidades Ejecutadas (TUE)** almacena la secuencia de sentencias ejecutadas y asertos evaluados para cada uno de los casos de prueba, en forma de Unidades Ejecutadas. Cada UE contiene información semántica extraída de la sentencia del código fuente o del aserto del que es representación. Esta información será utilizada más adelante para generar las restricciones asociadas a la descripción del sistema del problema de diagnosis. Por este motivo no sólo se almacena la traza seguida, sino que además se mantiene en cada UE toda la información necesaria para obtener un conjunto de restricciones que sea capaz de ofrecer el mismo comportamiento, ante el caso de prueba utilizado, que la sentencia o aserto original. El conjunto de restricciones que se obtendrá servirá de modelo de la traza seguida.

A continuación se muestra un pequeño ejemplo de transformación de una traza a una TUE.

Ejemplo 6.2. En la tabla 6.1 se muestra un caso de prueba para una porción de código fuente. En el código fuente aparecen varias sentencias de asignación y una sentencia selectiva.

La ejecución del código fuente utilizando el caso de prueba propuesto, produce la traza mostrada en la columna denominada 'Traza'. La traza está formada por el

Detección de errores		
Entradas	Traza	TUE

SC:
S_1 `s = 10;`
S_2 `if( b > a ){`
S_3 `a = b;`
`}`
S_4 `s = s + a;`
TC:
Entradas = {a = 1, b = 9}
Salidas = {s = 90}
	S_1 s = 10;
	S_2 if (b > a){
	S_3 a = b
	}
	S_4 s = s + a
		UE_Asig
		UE_If{
		UE_Asig
		}
		UE_Asig

Tabla 6.1: Ejemplo de generación de una TUE

conjunto de sentencias y asertos que son ejecutados y evaluados debido a las entradas del caso de prueba. Como se puede observar, la condición de la sentencia de tipo selectiva se ha cumplido en la traza ejecutada, y por tanto, la asignación incluida en el bloque asociado a la condición también pertenecerá a la traza seguida. Si no se cumpliese la condición, la traza no coincidiría con el código fuente.

La traza de sentencias y asertos será transformada a una Traza de Unidades Ejecutadas. En la columna TUE aparecen representadas las UE que se generarían. Dentro de cada UE se almacenará información semántica, que como se verá más adelante, permitirá generar las restricciones que forman la descripción del sistema del problema de diagnosis. Los tipos de UE que existen y la información que almacenan, se mostrarán en los siguientes apartados.

6.3.5. Identificadores para los elementos del código fuente y asertos

En la metodología propuesta en este capítulo, cada una de las sentencias del código fuente quedará identificada de manera unívoca a través de un identificador, y cada UE obtenida de la traza seguida, tendrá asociado el identificador correspondiente a la sentencia del código fuente de la que ha sido obtenida, de esta forma cada UE estará asociada a una sentencia del código fuente. Si el proceso de diagnóstico identifica una sentencia como un defecto, implica que dicha sentencia debe cambiar, pero aunque dicha sentencia aparezca repetidas veces en la traza, la modificación es única, ya que se realizará sobre el código fuente, en el que la sentencia sólo aparece una vez. Es decir, si existe un defecto en una sentencia, existe un único defecto, aunque dicha sentencia se repita en varias ocasiones en la traza.

Los principales identificadores que se utilizarán son:

- Por cada clase C_i existirá un identificador único dentro del sistema software que será del tipo IDClass. Para implementar este identificador se puede utilizar el nombre de la clase si se garantiza que no pueden existir dos clases con el mismo nombre.

- Cada uno de los métodos o constructores estará asociado a una clase C_i. Para identificar a un método o constructor, se tendrá un identificador único de tipo IDMethod o IDConstructor, que lo identificará unívocamente dentro del sistema software. Para implementar este identificador se puede utilizar el identificador de la clase que lo contiene, el nombre del método o constructor, el tipo y orden de los parámetros recibidos, y el tipo devuelto (si se trata de un método).

- Para identificar cada bloque de sentencias se tendrá un identificador único de tipo IDBlock, que lo identificará unívocamente dentro del sistema software. Cada bloque debe pertenecer a un método, a una sentencia selectiva, o a una sentencia iterativa, y en función de esta dependencia, la identificación del bloque se hará como sigue:

  - Los bloques asociados a un método estarán identificados unívocamente gracias a la identificación que tenga el método que lo referencia.
  - Los bloques asociados a sentencias selectivas estarán identificados con la misma identificación que tenga la sentencia selectiva que lo referencia, y un valor lógico que permitirá saber si es el bloque asociado a evaluación de la expresión como verdadero, o el bloque asociado a la evaluación de la expresión como falso.
  - El bloque asociado a una sentencia iterativa estará identificado unívocamente gracias a la identificación de la sentencia de tipo iterativa.

- Cada sentencia tendrá asignado un identificador de tipo IDStatement que lo identificará unívocamente dentro del sistema software.

En el código fuente, además de sentencias, existen datos que influyen directamente en los resultados que el sistema software genera. Estos datos son almacenados básicamente como atributos de clases u objetos, o como parámetros de entrada o variables locales en los métodos. Cada uno de estos elementos que aparecen en el código fuente y que permiten almacenar información, tendrán también asignado un identificador. De esta forma también cada uno de estos elementos quedará identificado de manera unívoca en el sistema software. Estos identificadores se aplicarán sobre:

- Atributos. Cada atributo tendrá asociado un identificador único dentro del sistema software que será del tipo IDField.

- Parámetros de entrada de un método. Cada parámetro de un método tendrá asociado un identificador único de tipo IDParam.

```
UE_Object ::= id:IDClass invs:UE_Assert* fields:UE_Field*
UE_Field ::= type:Type asigStat:UE_AsigStat
UE_Method ::= id:IDMethod params:UE_Param* retType:Type
              block:UE_Block pre:UE_Assert* post:UE_Assert*
UE_Constructor ::= id:IDConstructor params:UE_Param* ret:UE_Object
                   block:UE_Block pre:UE_Assert* post:UE_Assert*
UE_Param ::= id:IDParam type:Type
UE_Block ::= id:IDBlock stats:UE_Statement*
UE_Statement::= UE_LocalVariable | UE_If | UE_IfElse | UE_Return
                | UE_Assert | UE_AsigStat | UE_MCall | UE_CCall
UE_LocalVar ::= type:Type asigStat:UE_AsigStat
UE_IfElse ::= id:IDStatement exp:Exp if:UE_Block else:UE_Block
UE_Return ::= id:IDStatement exp:Exp
UE_AsigStat ::= id:IDStatement dest:ID* left:(IDLocal | IDField | IDParam)
                asigExp:AsigExp
UE_Assert ::= exp:Exp*
AsigExp ::= Exp | UE_MCall | UE_CCall
UE_CCall ::= id:IDStatement arguments:UE_Argument* met:UE_Constructor
UE_MCall ::= dest:ID* id:IDStatement arguments:UE_Argument* met:UE_Method
UE_Argument ::= exp:Exp
Exp ::= AritmeticExp | LogicExp | Literals | Identifier
Type ::= Boolean | Integer | Real | Class
```

Figura 6.4: Tipos de UE que pueden aparecer en una traza

- Variables locales de un bloque. Cada variable local dentro de un bloque tendrá asignado un identificador único de tipo IDLocal que la identificará en todo el sistema software.

6.3.6. Tipos de Unidades Ejecutadas

En la figura 6.4 aparecen los tipos de UE que se proponen en este trabajo para poder modelar cada uno de los tipos de sentencias que pueden aparecer en una traza. La descripción de las diferentes UE se ha realizado utilizando la notación EBNF (Extended Backus Normal Form) atribuida, de tal forma que cada símbolo que forme parte de otro tendrá asociada una etiqueta. Estas etiquetas serán utilizadas más adelante, y permitirán acceder a los diferentes campos que contiene cada UE.

La secuencia de UE se denomina Traza de Unidades Ejecutadas (TUE). Básicamente, en una TUE se almacena la secuencia de métodos que se han ejecutado en la traza seguida por el caso de prueba. Teniendo en cuenta que cada UE a su vez podrá contener unidades diagnosticables internas. Por ejemplo una unidad diagnosticable de tipo bloque puede contener otras unidades de tipo sentencia, e incluso llamadas a otros métodos.

En la figura 6.5 se muestra el orden de descomposición de las UE. Este orden se basa en el orden de ejecución normal en un programa. Por ejemplo, una UE de tipo sentencia selectiva (UE_IfElse) debe incluir al menos un bloque de sentencias internas, y a su vez cada bloque estará formado por otras sentencias.

Siguiendo el orden establecido en la figura 6.5, en los siguientes apartados se describirán los atributos que cada una de las UE posee.

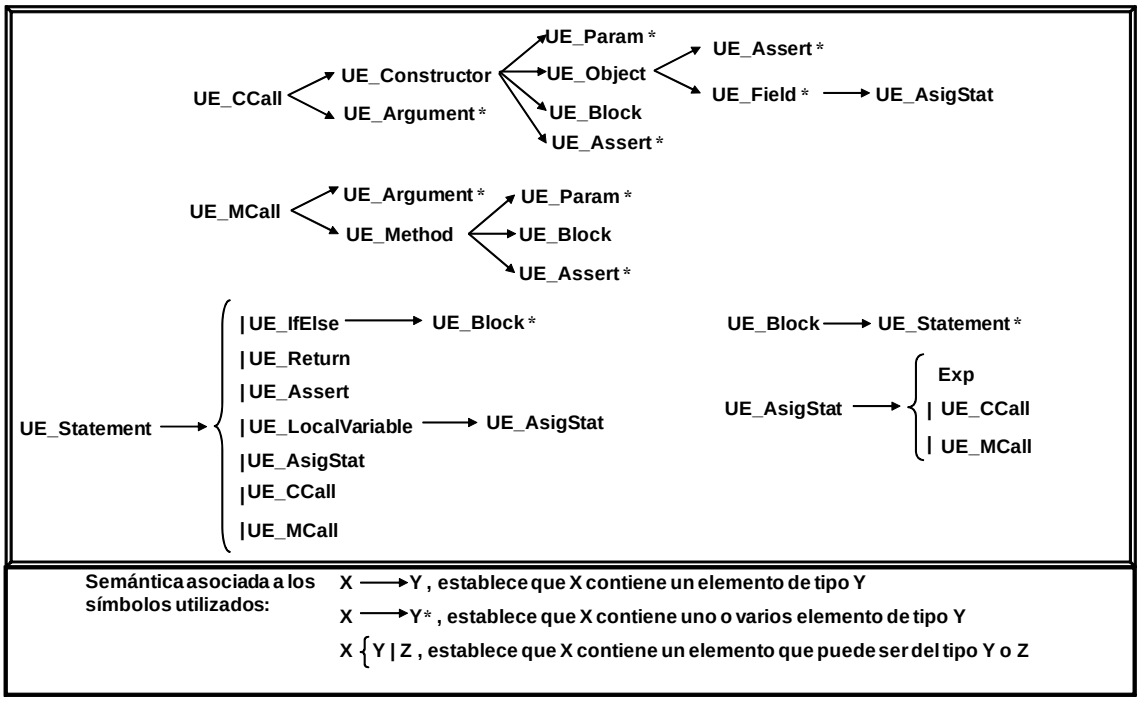


Figura 6.5: Orden de descomposición de las unidades diagnosticables de un programa

UE tipo llamada a método

```
UE_MCall ::= dest:ID* id:IDStatement arguments:UE_Arguments*
         met:UE_Method
```

Cada UE_MCall representa una llamada a un método dentro de la traza seguida. Cada UE_MCall estará identificada con el campo *id* de tipo IDStatement. Además, contiene la lista de expresiones utilizadas como argumentos en la llamada, y una UE de tipo UE_Method que contiene la UE correspondiente al método llamado.

En el caso de que el método llamado no pertenezca al propio objeto, el campo *dest* almacena los identificadores sobre los que se debe navegar para llegar al objeto al que pertenece el método llamado. Por ejemplo, para la sentencia siguiente, que representa una llamada al método *getNumber*:

this.myAccount.getNumber()

dicha llamada se hace sobre el objeto almacenado en el atributo *myAccount*. Por tanto para dicha llamada, en el campo *dest*, se almacenará la lista: {this, myAccount}.

UE tipo argumento de llamada

```
UE_Argument ::= exp:Exp
```

Cada UE_Argument contiene la expresión utilizada como argumento de una llamada a un método o a un constructor.

UE tipo llamada a constructor

```
UE_CCall ::= id:IDStatement arguments:UE_Argument*
             met:UE_Constructor
```

Cada UE_CCall representa una llamada a un constructor dentro de la traza seguida. El campo *id*, de tipo IDStatement, identifica a dicha llamada unívocamente dentro del código fuente. Dispone de otro campo para almacenar la lista expresiones que se han enviado como argumentos al constructor. La información sobre el constructor ejecutado se almacena en el campo *met*, que es de tipo UE_Constructor (se verá más adelante).

UE tipo declaración de método

```
UE_Method ::= id:IDMethod params:UE_Param* retType:Type block:UE_Block
              pre:UE_Assert* post:UE_Assert*
```

Cada UE_Method estará asociada a la ejecución de un método. El campo *id* es el encargado de almacenar el identificador (de tipo IDMethod) del método. Cada UE_Method incluye campos para almacenar los parámetros de entrada (*params*) y el tipo del resultado devuelto (*ret*), el bloque de sentencias (*block*), la precondición (*pre*) y la postcondición del método (*post*).

UE tipo parámetro

```
UE_Param ::= id:IDParam type:Type
```

Cada UE_Param representa un parámetro de entrada de un método o constructor de una clase. El campo *id* es el encargado de almacenar el identificador (de tipo IDParam) del parámetro. Además, cada UE_Param incluye un campo (*type*) que informa del tipo del parámetro (lógico, entero, objeto, etc.).

UE tipo bloque

```
UE_Block ::= id:IDBlock stats:UE_Statement*
```

Cada UE_Block representa la ejecución en la traza de un bloque de sentencias. El campo *id* es el encargado de almacenar el identificador (de tipo IDBlock) del bloque.

Este valor permite identificar al bloque de forma unívoca dentro del código fuente del sistema software. El campo *stats* contiene la lista de UE que corresponden con las sentencias ejecutadas dentro del bloque.

UE tipo declaración de contructor

```
UE_Constructor ::= id:IDConstructor params:UE_Param* ret:UE_Object
                   block:UE_Block pre:UE_Assert* post:UE_Assert*
```

Cada UE_Constructor representa la ejecución de una llamada a un constructor. El campo *id* contendrá el identificador (de tipo IDConstructor) de dicho constructor. Cada UE_Constructor incluye campos para almacenar los parámetros de entrada (*params*), la creación del nuevo objeto (*ret*), el bloque de sentencias asociado al constructor (*block*), la precondición (*pre*), y la postcondición (*post*).

UE tipo objeto

```
UE_Object ::= id:IDClass invs:UE_Assert* fields:UE_Field*
```

Cada UE_Object representa la creación de un nuevo objeto en la traza ejecutada. El objeto creado será de una clase concreta, cuyo identificador de tipo IDClass se almacenará en el campo *id*. El campo *invs* permite almacenar la lista de invariantes que el objeto debe cumplir; y el campo *fields* contendrá la lista de atributos y las asignaciones iniciales de dichos atributos.

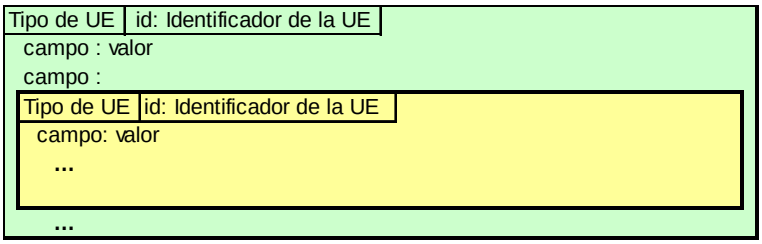


Figura 6.6: Representación visual de las UE

UE tipo declaración de atributo

```
UE_Field ::= type:Type asigStat:UE_AsigStat
```

Cada UE_Field representa la creación de un atributo de un objeto. Además del tipo correspondiente al atributo (campo *type*), cada UE_Field incluye un campo de tipo UE_AsigStat, denominado *asigStat*, que almacena la asignación inicial al atributo. Para

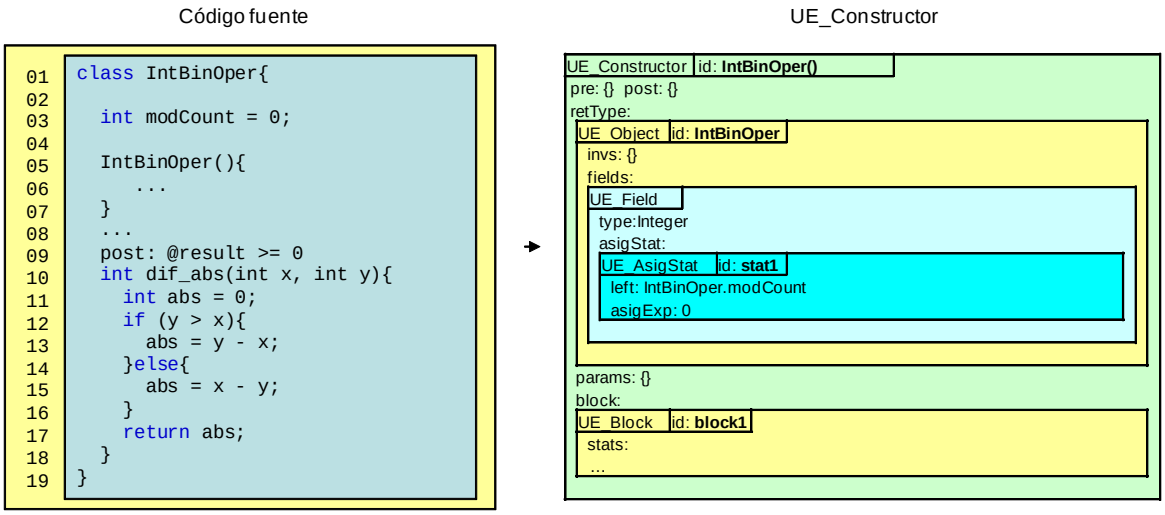


Figura 6.7: UE_Constructor obtenida de la ejecución del constructor de la clase IntBinOper

conocer el identificador del atributo (de tipo IDField), basta con acceder al campo *left* de la asignación inicial.

Ejemplo 6.3. El tratamiento de las TUE de forma automática se implementará a través de un árbol XML. De esta forma, toda la información asociada a una TUE podría ser tratada con las mismas herramientas que permiten manejar flujos de información en XML. El problema de usar XML para mostrar los ejemplos, y de cara al lector, es que se trata de texto plano y a veces es complicado seguir su contenido, por esta razón en los ejemplos se utilizará la representación visual como la que se muestra en la figura 6.6.

La representación visual, de forma general, contendrá en la esquina superior izquierda el nombre del tipo de UE, y a continuación el identificador de la sentencia o aserto al que está asociada dicha UE. Debajo se mostrarán cada uno de los campos y el valor asociado. Si el campo permite almacenar una UE o una lista de UE, se mostrarán directamente dichas UE utilizando la misma representación visual.

En la figura 6.7 se muestra cuál sería la UE resultante de la ejecución del método constructor de la clase *IntBinOper*. La llamada a dicho constructor dará lugar a una UE de tipo UE_Constructor. Para identificar al constructor dentro del sistema software se ha utilizado el nombre *IntBinOper*(), tal como aparece en la figura 6.7.

Internamente la UE_Constructor contiene los siguientes campos:

- Los campos *pre*, *post* y *params* encargados de almacenar la precondición, postcondición y los parámetros de entrada. En este caso los tres campos están vacíos, ya que no hay precondición, postcondición, ni parámetros de entrada para el constructor.

- El campo *ret*, almacena una UE de tipo UE_Object que contiene la información correspondiente a la creación del objeto de tipo *IntBinOper*. En el campo *id* se guarda el identificador del tipo del objeto, es decir, el identificador de la clase de la cuál el objeto es instancia. Los otros campos incluidos en esta UE son:

  - El campo *invs* encargado de almacenar los invariantes de la clase. En este caso la clase no posee ningún invariante.
  - El campo *fields* que almacena la lista de declaraciones de atributos. En este caso incluye una UE_Field para la declaración del atributo, *modCount*. A su vez, la UE_Field contiene como campos: el tipo (*type*), que en este caso es de tipo entero; y el campo correspondiente a la asignación inicial para el atributo, que contendrá una UE de tipo asignación (UE_AsigStat). Las UE de tipo asignación se explicarán con más detalle más adelante. Como se puede observar en la figura, para identificar el atributo se ha utilizado el identificador *IntBinOper.modCount*, y el valor inicial asignado es un cero.

- Y, por último, el campo que almacena el bloque de sentencias asociado al constructor (de tipo UE_Block). En este caso, el contenido de este campo se ha dejado en blanco, ya que en los siguientes apartados será donde se explique cuáles son los tipos de UE asociados a las sentencias que se pueden incluir en un bloque de sentencias.

UE tipo declaración de variable local

```
UE_LocalVar ::= type:Type asigStat:UE_AsigStat
```

Cada UE_LocalVar representa la declaración de una variable local dentro de un bloque de sentencias. Además del tipo correspondiente a la variable local, cada UE_LocalVar incluye un campo de tipo UE_AsigStat (este tipo de UE se verá más adelante) que realiza la asignación inicial sobre la variable local. Dicho campo, incluirá el identificador (de tipo IDLocal) de la variable local sobre la que se realiza la asignación inicial.

UE tipo sentencia selectiva

```
UE_IfElse ::= id:IDStatement exp:Exp if:UE_Block else:UE_Block
```

Cada UE_IfElse representa la ejecución de una sentencia selectiva. Cada UE_IfElse estará identificada con el campo *id* de tipo IDStatement. Además de la expresión establecida como condición, se tendrá un campo reservado para cada bloque posible (cada bloque será de tipo UE_Block). Sólo uno de los dos bloques asociado a la condición puede ejecutarse. El bloque ejecutado depende de la evaluación de la expresión como cierto o falso. Si en la sentencia selectiva sólo aparece un bloque asociado, éste se ejecutará si la condición se evalúa a cierto.

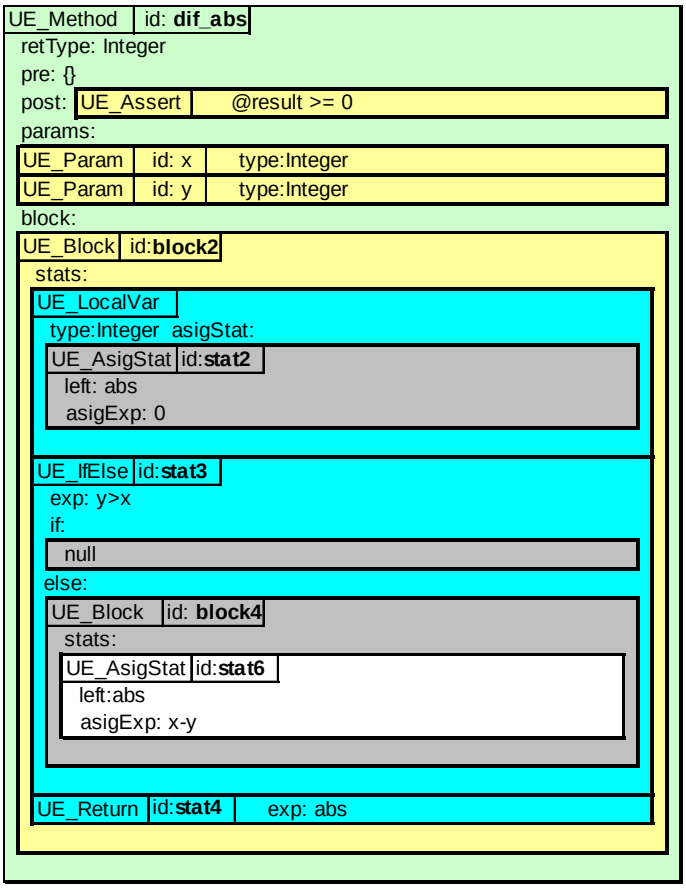


Figura 6.8: UE implicadas en la ejecución del método *dif_abs* de la clase *IntBinOper*

UE tipo retorno de método

```
UE_Return ::= id:IDStatement exp:Exp
```

Cada UE_Return representa la ejecución de una sentencia de retorno de un método. Almacena la expresión que debe evaluarse como salida del método. Cada UE_Return estará identificada con el campo *id* de tipo IDStatement.

UE tipo aserto

```
UE_Assert ::= exp:Exp
```

Cada UE_Assert representa la ejecución de un aserto. Un aserto es una expresión que debe cumplirse al evaluarse dentro de la traza seguida. El campo *exp* almacena la expresión asociada al aserto.

UE tipo asignación

```
UE_AsigStat ::= id:IDStatement dest:ID* left:(IDLocal | IDField
          | IDParam) asigExp:AsigExp
```

Cada UE_AsigStat representa una sentencia de asignación de la traza seguida. Contiene una expresión, o una llamada a un método o constructor, cuyo resultado será asignado a un atributo de un objeto, o a un parámetro de entrada, o a una variable local del método que contiene la asignación. Cada UE_AsigStat estará identificada con el campo *id* de tipo IDStatement. En el caso de que la asignación sea sobre un atributo que no pertenezca al propio objeto, el campo *dest* almacena los identificadores sobre los que se debe navegar, para llegar al objeto al que pertenece el atributo a asignar.

Ejemplo 6.4. En la figura 6.8 se muestra la UE_Method que se obtendría al realizar una llamada al método dif_abs de la clase IntBinOper que se mostró en la figura 6.7. Esta operación calcula la diferencia en valor absoluto de los dos enteros recibidos como parámetros.

El método dif_Abs sigue la secuencia de sentencias formada por la declaración de la variable local (que almacena el resultado que debe devolver el método), ejecución de la sentencia selectiva, y retorno del valor almacenado en la variable local que fue declarada en la primera sentencia.

Para identificar al método se ha usado el nombre *dif_abs*. El tipo devuelto por el método, almacenado en el campo *retType*, es el tipo entero. La postcondición es recogida en el campo denominado *post*. El bloque asociado al método (UE_Block) incluye tres UE de tipo sentencia. La primera corresponde con una declaración local (UE_Local) de una variable de tipo entero. La declaración local contiene una asignación inicial que establece el valor cero inicial para la variable identificada como *abs*.

La segunda sentencia es de tipo selectiva o condicional (UE_IfElse). Cuando el sistema software se ejecuta, dependiendo de la condición, una de las dos posibles ramas se almacenará en el campo *if* o en el campo *else* según corresponda. En el ejemplo mostrado en la figura 6.8, al no cumplirse la condición de la sentencia selectiva, se incorpora a la traza la sentencia de asignación que aparece en el bloque *else*. Por este motivo en la rama *else* aparece una UE_Block y la rama *if* queda vacía. En el campo *exp* de la UE_IfElse se almacena la expresión asociada a la sentencia selectiva. En el bloque correspondiente a la rama *else* hay una única sentencia de tipo asignación. La asignación es sobre la variable *abs*.

Aunque ya se ha explicado anteriormente, y se ha remarcado en los ejemplos mostrados, lo que se almacena en las expresiones que aparecen en los campos de las UE de la traza, no son los valores concretos obtenidos en la traza para dichas expresiones, sino las propias expresiones. El objetivo no es guardar en la traza de UE los valores concretos asignados, sino las expresiones que calculan el valor que debe ser asignado, ya que será a partir de estas expresiones de donde se obtengan las restricciones que formen la descripción del sistema del problema de diagnosis.

Detección de errores		
Entradas	Traza	TUE

TC:
{c = 2, a = 1, s = 9}
 	
SC:
S_1 `while(c > 0){`
S_2 `c = c - 1;`
S_3 `a = a + 3;`
`}`
S_4 `s = a;`
	S_1 if (c > 0){
	S_2 c = c − 1
	S_3 a = a + 3
	S_1 if (c > 0){
	S_2 c = c − 1
	S_3 a = a + 3
	assert: !(c > 0)
	}else{
	assert: !(c > 0)
	}
	}else{
	assert: !(c > 0)
	}
	S_4 s = a
		UE_IfElse{
		UE_Asig
		UE_Asig
		UE_IfElse{
		UE_Asig
		UE_Asig
		UE_Assert
		
		UE_Assert
		}
		
		UE_Assert
		}
		UE_Asig

Tabla 6.2: Ejemplo de generación de una TUE para una sentencia iterativa

La última UE que se incluye en el bloque asociado al método es de tipo UE_Return. En el campo *exp* se almacena la expresión que debe devolverse como resultado de la llamada al método.

Transformación de las sentencias iterativas a UE

Una sentencia iterativa contiene una condición y un bloque de sentencias que se ejecutará mientras se cumpla la condición. Dado un caso de prueba concreto, el número de iteraciones vendrá dado por el número de veces que la condición impuesta en la sentencia iterativa se haga cierta. Cada una de las sentencias del bloque incluido en la sentencia iterativa influye en cada iteración sobre el resultado obtenido al finalizar la sentencia iterativa.

Al desglosar una sentencia iterativa en sus diferentes iteraciones, se obtiene un comportamiento idéntico al de varias sentencias selectivas anidadas, que contengan la misma condición que el bucle original. Teniendo en cuenta esta propiedad, para simplificar el número de tipos de UE a tener en cuenta, la traza de la sentencia iterativa será representada a través de tantas sentencias selectivas anidadas como iteraciones haya realizado el bucle en su ejecución para el caso de prueba propuesto.

Cada iteración i será transformada a una sentencia selectiva, que tendrá como condición la misma condición evaluada en la iteración i-esima del bucle, y como bloque asociado al cumplimiento de la condición tendrá el mismo bloque ejecutado en la iteración i-esima del bucle, más, la sentencia selectiva correspondiente a la iteración i + 1 si el bucle se ha ejecutado al menos i + 1 veces. Además, asociado al no cumplimiento

de la condición, y que implicaría el final del bucle, se añadirá como aserto la condición del bucle negada. De esta forma se obliga a que siempre que no se itere se cumpla que la condición del bucle no se cumple.

Así una sentencia iterativa ejecutada n veces, al transformarla a sentencias selectivas, quedaría tal como aparece a continuación:

if ($\exp_1$){
  block_1
  if ($\exp_2$){
    block_2
    ...
    if ($\exp_n$){
      block_n
      assert: !($\exp_{n+1}$)
    }**else**{
      assert: !($\exp_n$)
    }
    ...
  }**else**{
    assert: !($\exp_2$)
  }
}**else**{
  assert: !($\exp_1$)
}

El termino exp_i hace referencia a la condición del bucle en la iteración i-esima, y el termino $block_i$ hace referencia al bloque de sentencias ejecutado en la iteración i-esima. Es importante reseñar, que en el bloque asociado al cumplimiento de la condición de la última sentencia selectiva anidada, se debe añadir como aserto, que una vez ejecutado dicho bloque, no debe cumplirse la condición establecida en el bucle. De esta forma se garantiza que tras la última iteración la condición impuesta en el bucle no se cumplirá.

A continuación se muestra un pequeño ejemplo de transformación de una traza a una TUE.

Ejemplo 6.5. En la tabla 6.2 se muestra un caso de prueba para una porción de código fuente. En el código fuente aparece una sentencia iterativa y una asignación detrás de dicha sentencia. Dentro de la sentencia iterativa a su vez hay dos asignaciones.

La ejecución del código fuente utilizando el caso de prueba propuesto, produce la traza mostrada en la columna denominada 'Traza'. La traza está formada por el conjunto de sentencias y asertos que son ejecutados y evaluados debido a las entradas del caso de prueba. Como se puede observar, la sentencia de tipo iterativa ha sido transformada a su equivalente usando sentencias selectivas. El bucle se ha ejecutado dos veces en la traza, lo que equivale a dos sentencias selectivas anidadas.

La traza de sentencias y asertos será transformada a una Traza de Unidades Ejecutadas. En la columna TUE aparecen representadas las UE que se generarían.

6.4. Localización de los defectos

Una vez detectado el error o errores en el código fuente, el siguiente paso es generar el problema de diagnosis. Tal como se explicó en el capítulo dedicado a la metodología (capítulo 4) un problema de diagnosis está formado por (SD, STS, SPEC, TC), donde SD es la descripción del sistema, formada por un conjunto de restricciones; STS es el conjunto de sentencias que forma el código fuente; SPEC es el conjunto de asertos que forman la especificación; y TC representa un caso de prueba. En los problemas que se resolverán en este capítulo, se supondrá que el conjunto SPEC es correcto, y que los defectos deben estar dentro del conjunto STS de sentencias del código fuente. Es decir, la unidad mínima donde se podrán detectar defectos será cada una de las sentencias que forman el conjunto STS.

Las restricciones que componen la descripción del sistema serán obtenidas como resultado de la transformación de la traza ejecutada para el caso de prueba TC. Tal como se explicó en el capítulo dedicado a la metodología de diagnosis (capítulo 4), el problema de diagnosis se resolverá como un problema de satisfacción de restricciones (CSP).

Para resolver el CSP de la manera más eficiente posible, resulta conveniente conocer qué restricciones tienen variables en común. De esta forma, por ejemplo, la propagación de las restricciones se podría realizar de forma más eficiente, y así encontrar soluciones en el tiempo deseado. Para que en la transformación a restricciones de la traza seguida no se pierda esta información, es decir, para conocer qué restricciones están relacionadas entre sí, cada UE de la traza seguida será transformada a un Componente Desplegado (CD). El objetivo es que cada grupo de restricciones que estén relacionadas entre sí permanezcan agrupadas en un CD. Por ejemplo, de esta forma se podrá mantener en un mismo CD todas las restricciones asociadas a la ejecución de un método.

El conjunto de componentes desplegados obtenidos de la transformación de la TUE se denominará Traza de Componentes Desplegados (TCD). Cada CD almacenará un conjunto de restricciones de la descripción del sistema, y para obtener el conjunto completo de restricciones bastará con recorrer por completo la TCD.

Para obtener los componentes que forman la TCD a partir de una TUE se han definido una serie de reglas de transformación. Estas reglas se desarrollarán con detalle en las siguientes secciones.

6.4.1. Traza de Componentes Desplegados

Los tipos de componentes desplegados que se utilizarán en este trabajo son: CD_Asig, CD_Statement, CD_IfElse, CD_Assert, CD_MCall, CD_CCall, y CD_Block. Cada uno

de ellos tendrá una serie de propiedades o atributos. A continuación se muestra un resumen donde aparecen todos los tipos de CD que se utilizarán, y entre llaves los atributos que cada uno posee. De cada atributo aparece primero el tipo y luego el nombre de dicho atributo. Cuando un CD hereda propiedades de otro se utiliza la palabra reservada 'extends'. Cuando el atributo es del tipo 'Set<...>'se trata de un conjunto de elementos del tipo reflejado entre los símbolos '<>'.

```
CD_Block { Set<CD_Statement>  stats; }
CD_Statement { SD sd; }
CD_Asig extends CD_Statement { }
CD_Assert extends CD_Statement { }
CD_IfElse extends CD_Statement { CD_Block if; CD_Block else; }
CD_CCall extends CD_Statement { Set<CD_Asig> arguments;
          Set<CD_Asig> fields; CD_Block met; }
CD_MCall extends CD_Statement { Set<CD_Asig> arguments;
          CD_Block met; }
```

En general, cada componente contendrá un conjunto de restricciones, y opcionalmente, y dependiendo del tipo de componente, otros componentes internos. En concreto, cada componente incluirá un campo denominado *sd*, que almacenará un objeto de tipo SD (SystemDescription), que será el encargado de almacenar las restricciones.

A continuación se describe el objetivo de cada uno de los tipos de componentes presentados:

- Cada componente de tipo CD_Block representa un conjunto de CD_Statement que deben ejecutarse de forma secuencial. Este tipo de componentes no tiene asociado ningún comportamiento por sí mismo, ya que es simplemente un recolector de otros componentes internos. Por este motivo es el único CD que no tiene restricciones asociadas. Todos los CD que un CD_Block puede incluir son de tipo CD_Statement. El tipo CD_Statement tiene diferentes subtipos que se explican a continuación.

- Los componentes de tipo CD_Asig permiten modelar el comportamiento de las asignaciones. Las sentencias de asignación pueden ser sobre variables locales, atributos de objetos, o parámetros de entrada. También el paso de argumentos de entrada en las llamadas a métodos y constructores será transformado a componentes de tipo CD_Asig, ya que se considera que el paso de parámetros de entrada es equivalente a una asignación entre los parámetros de entrada y las expresiones enviadas como argumento de entrada. En concreto, las asignaciones, las declaraciones de atributos y variables locales, los pasos de parámetros a los métodos y constructores, y el retorno de valores u objetos por parte de los métodos y constructores, serán modelados como asignaciones y a través de componentes de tipo CD_Asig. Las restricciones que se incluyan en el componente permitirán modelar

Componente Desplegado	Unidades Ejecutadas
CD_Block	UE_Block
CD_Statement	UE_Statement
CD_Asig	UE_Field, UE_Param, UE_Argument, UE_LocalVar, UE_Return, UE_AsigStat
CD_Assert	UE_Assert
CD_IfElse	UE_IfElse
CD_CCall	UE_CCall, UE_Constructor, UE_Object, UE_Invariant
CD_MCall	UE_MCall, UE_Method, UE_Invariant

Tabla 6.3: Relación entre las UE y los CD

la asignación, y la declaración de la variable, si se trata de la primera vez que aparece la variable en la traza.

- Cada componente de tipo CD_Assert guarda en forma de restricciones, el equivalente a un aserto introducido en la traza seguida. La condición establecida en el aserto debe satisfacerse en el problema de diagnosis.

- Cada componente de tipo CD_IfElse representa una sentencia selectiva. Para modelar el comportamiento asociado a la sentencia selectiva se usarán una serie de restricciones que se almacenarán en el campo *sd*. Además, se incluirán uno o dos posibles componentes de tipo CD_Block que corresponden a los bloques de sentencias asociados al cumplimiento o no de la condición establecida en la sentencia selectiva.

- Cada componente de tipo CD_MCall representa una llamada a un método. Tendrá asociado un conjunto de asignaciones recogidas a través de componentes de tipo CD_Asig, correspondientes a los emparejamientos de los argumentos enviados con los parámetros de entrada, y un componente de tipo CD_Block, que corresponde con la transformación del bloque de sentencias incluidas en el método llamado.

- Cada componente de tipo CD_CCall representa una llamada a un constructor. Al igual que en los componentes de tipo CD_MCall, tendrá asociado un conjunto de asignaciones de los argumentos enviados a los parámetros de entrada, recogidas a través de componentes de tipo CD_Asig; y un componente de tipo CD_Block, que almacena los componentes obtenidos de la transformación del bloque de sentencias incluidas en el método llamado. Además, tendrá un conjunto de componentes de tipo CD_Asig, que representan las asignaciones iniciales sobre los atributos del objeto creado. El campo *sd* almacenará las restricciones necesarias para la creación del nuevo objeto por parte del constructor llamado.

En la tabla 6.3 se muestra en qué tipo de componentes será transformada cada unidad ejecutada. Por ejemplo, para obtener un componente de tipo CD_MCall

se utilizará la información almacenada en las UE de tipo UE_MCall, UE_Method y UE_Invariant. Las UE de tipo UE_Field, UE_Param, UE_Argument, UE_LocalVar y UE_Return, serán transformadas a componentes del tipo CD_Asig.

En los siguientes apartados se mostrará el proceso para transformar cada UE a CD. Este proceso se basará en una serie de funciones de transformación. Cada una se encarga de la transformación de un tipo de UE.

6.4.2. Tipos de defectos

Los defectos que se podrán localizar a través de la resolución del problema de diagnosis, se pueden englobar en dos grandes tipos: defectos en asignaciones, y defectos en las sentencias selectivas e iterativas.

- Los defectos en las asignaciones pueden aparecer en las asignaciones de los atributos y variables locales. Además se considerarán asignaciones el paso de los argumentos de entrada a un constructor o método. Cada paso de un argumento de entrada será considerado como una asignación sobre el correspondiente parámetro de entrada. También, si el resultado de una llamada a un método es recogido en una asignación, la sentencia de retorno se considerará parte de dicha asignación, ya que la sentencia de retorno establece la expresión que debe ser asignada.

  Por tanto la metodología propuesta será capaz de localizar defectos en las asignaciones iniciales de las declaraciones de atributos, y variables locales; en el paso de argumentos a los métodos y constructores; en la asignación de variables locales, parámetros de entrada y atributos de objetos; y en el retorno de un resultado por parte de un método.

- Los defectos en las sentencias selectivas se denominarán defectos estructurales, y afectan a la traza ejecutada, eliminando o añadiendo sentencias que deberían o no deberían estar en la traza. La metodología propuesta buscará defectos en las expresiones utilizadas como condición en las sentencias selectivas. La condición tendrá un defecto cuando para unas entradas determinadas no se ejecute el bloque adecuado de sentencias. Tal como se ha explicado anteriormente, las sentencias iterativas serán tratadas como sentencias selectivas anidadas, por tanto, este tipo de defectos incluiría también defectos en las sentencias iterativas.

Es conveniente remarcar que la declaración de un atributo siempre se considerará correcta, los defectos se buscarán en la asignación inicial, es decir, se buscarán defectos en la expresión inicial asignada. De manera análoga, las declaraciones de variables locales se considerarán correctas, sólo se buscarán defectos en la expresión inicial asignada. La metodología propuesta tampoco buscará defectos en la definición de los parámetros de entrada de los métodos y constructores, los defectos podrán estar en las expresiones utilizadas como argumentos en las llamadas a los métodos o constructores. El conjunto

de defectos de un bloque de sentencias será la unión de los defectos del conjunto de sentencias que forman dicho bloque.

Para poder identificar defectos en asignaciones y en las sentencias selectivas se utilizarán restricciones que permitirán modelar, en la descripción del sistema, el siguiente comportamiento:

- Para las asignaciones, se contemplarán dos posibilidades. La primera que la asignación no tenga defectos, y que por tanto al solucionar el problema de diagnosis, la variable asignada sea igual a la expresión a asignar. Y la segunda, que la expresión a asignar no sea la correcta, en cuyo caso, sin realizar dicha asignación, el problema de diagnosis deberá poder tener solución.

- Para el caso de las condiciones en las sentencias selectivas, se contemplarán también dos posibilidades. La primera que la condición de la sentencia selectiva no tenga defectos, y que por tanto el bloque que debe ejecutarse será el que determine la sentencia selectiva. Y la segunda opción será, que la condición de la sentencia selectiva sea defectuosa, y que por tanto, cumpliéndose la negación de la condición de dicha sentencia selectiva, el problema de diagnosis deberá poder tener solución.

Para poder alcanzar el resultado esperado será necesario modificar las asignaciones y las condiciones de las sentencias selectivas donde se hayan localizado los defectos. En los siguientes apartados se concretarán las diferentes funciones de transformación que darán lugar a las restricciones del problema de diagnosis. Para poder aplicar las funciones de transformación es necesario introducir nuevos conceptos, como por ejemplo el predicado P; y añadir una serie de requisitos, como por ejemplo la gestión de objetos en las restricciones.

Predicado P

Cuando un código fuente se pone en funcionamiento, la secuencia final de sentencias ejecutadas depende de las entradas recibidas y de las condiciones evaluadas en las sentencias selectivas e iterativas. Estos elementos de control permiten seleccionar diferentes grupos de sentencias dependiendo de la evaluación de una condición lógica. La condición que debe ser evaluada para determinar el bloque de sentencias que debe ser ejecutado puede contener defectos. Este tipo de defectos influye en la traza seguida, y pueden provocar que no se alcance el resultado especificado como correcto, debido a que ciertas sentencias hayan sido eliminadas o añadidas a la traza ejecutada. Este tipo de defectos se denominan defectos estructurales.

Para poder localizar, a través del problema de diagnosis, este tipo de defectos, se hará uso del predicado P, y del concepto de traza correcta, que a continuación se define:

Definición 6.1. Traza Correcta. Para un código fuente, una especificación, y un caso de prueba concreto, será el conjunto de sentencias y asertos que deben ejecutarse para alcanzar el resultado especificado como correcto en el caso de prueba propuesto.

Definición 6.2. Predicado P. Se define el predicado P(s) de una sentencia o aserto *s*, como aquél que toma el valor verdadero si dicha sentencia o aserto pertenece a la traza correcta, y falso, en caso contrario.

Es decir, la metodología de diagnosis determinará para cada sentencia o aserto, si el predicado P debe tomar el valor cierto o falso, dependiendo si pertenece o no a la traza correcta. O dicho de otra forma, la traza correcta será la formada por las sentencias y asertos en los que el predicado P habrá tomado el valor verdadero. Este predicado es una propuesta de esta tesis, y es una novedad con respecto a la metodología DX, ya que el objetivo en dicha metodología no es buscar defectos estructurales, sin embargo en el caso del software sí es un objetivo, ya que las sentencias de decisión pueden tener defectos de este tipo.

En el apartado dedicado a las transformaciones de las sentencias selectivas se darán más detalles sobre el predicado P. En las funciones de transformación que se mostrarán más adelante, el predicado P se recibirá como parámetro de entrada. En la generación de las restricciones se utilizará el predicado P y el predicado AB. Tanto los predicados P como AB, tomarán un valor concreto (verdadero o falso) cuando el problema de diagnosis tenga solución. Cada uno persigue un objetivo diferente. Los predicados P permiten determinar cuál es la traza correcta; y para las sentencias y asertos que forman la traza correcta, a través de los predicados AB, se podrá determinar cuáles son las sentencias que pueden contener defectos. El predicado P servirá para identificar defectos en las condiciones de las sentencias selectivas, y el predicado AB se utilizará para identificar asignaciones que contengan defectos.

Gestión de objetos en las restricciones

Tal como se introdujo en el capítulo dedicado a la metodología (sección 4.4.1), por cada una de las clases que intervienen en el código fuente, se creará una clase equivalente, en la que los atributos de la clase, y los parámetros y variables locales de los métodos, serán transformados a los tipos disponibles en el entorno de programación con restricciones. Los tipos permitidos para las variables que se usarán en las restricciones son IntTypeVar, RealTypeVar, BooleanTypeVar y ObjectTypeVar, que permiten almacenar respectivamente, dominios de tipo entero, real, lógico, u objetos.

Cada una de las variables locales, parámetros de entrada y atributos de objetos que aparezcan en la traza, serán transformados a variables del tipo adecuado, de forma que puedan ser utilizados en las restricciones que se incluirán en los CD. En el caso de los atributos, parámetros o variables locales que sean de tipo referencia a un objeto, serán transformados a variables del tipo *ObjectTypeVar*, permitiendo así almacenar un conjunto referencias a objetos como dominio de una variable.

Aunque cada objeto es idéntico en comportamiento a otro de la misma clase, el estado de los diferentes objetos que son instancia de una misma clase puede ser diferente. El estado viene determinado por el conjunto de sus atributos. Para poder tener en cuenta el estado de cada objeto en la transformación a restricciones de la traza, por

cada objeto que se haya creado en la traza seguida, se creará otro objeto, para que pueda ser utilizado en las restricciones que se obtengan. Por ejemplo, en la función de transformación UE_CCall2CD que permite transformar una llamada a un método constructor a restricciones, se creará un nuevo objeto del tipo adecuado, que podrá ser utilizado en las restricciones que se generen.

De manera análoga a como ocurría con cada clase C_i que tiene asociado un identificador único de tipo IDClass, para identificar cada uno de los objetos que se creen, se usará un identificador de tipo IDObject, que identificará al objeto con respecto al resto de objetos del sistema.

Para crear objetos dentro del entorno de programación con restricciones, se utilizará la función *newObject* que recibe el identificador de la clase (de tipo IDClass) y devuelve un nuevo identificador de tipo IDObject que hará referencia a un objeto del tipo IDClass. Esta función se utilizará, por ejemplo, cuando más adelante se muestre la función de transformación UE_CCall2CD. Para acceder al atributo de un objeto dentro del entorno de programación con restricciones será necesario conocer el identificador del objeto (del tipo IDObject) y el identificador del atributo (del tipo IDField).

6.4.3. Funciones de transformación

Para transformar las UE a CD se han definido una serie de funciones de transformación. Las funciones de transformación son UE_AsigStat2CD, UE_IfElse2CD, UE_LocalVar2CD, UE_Assert2CD, UE_MCall2CD, UE_CCall2CD, UE_Block2CD, y UE_Return2CD. Cada una se encarga respectivamente de la transformación de las asignaciones, sentencias selectivas, declaraciones de variables locales, asertos, llamadas a métodos, llamadas a constructores, bloques de sentencias, y sentencias de retorno de un método.

Cada una de estas funciones recibirá al menos tres parámetros:

- La UE sobre la que se realizará la transformación. La función de transformación accederá a la información almacenada en las UE para generar como resultado un CD.

- El predicado P. Este predicado se verá con más detenimiento en la función de transformación de las sentencias selectivas. El valor de este predicado determina si la CD pertenece o no a la traza considerada como correcta. A medida que se vayan generando los CD, cuando exista la posibilidad de elegir entre varias alternativas, por ejemplo en una sentencia selectiva, se crearán nuevas instancias del predicado P que permitan controlar las posibles alternativas que se puedan tomar para llegar a la traza correcta.

- El identificador del objeto sobre el que se está ejecutando el CD. Cuando se ejecuta una traza de sentencias, cada método ejecutado actúa por defecto sobre un objeto concreto. Para poder controlar qué atributos son accesibles en cada momento, este parámetro irá tomando como valor el identificador del objeto activo en cada

Transformación: UE_LocalVar2CD			
Parámetros de entrada	UE_LocalVar: uE_LocalVar Path: P IDObject: idObj	Tipo devuelto	CD_Statement
Pasos 1) return **UE_AsigStat2CD**(uE_LocalVar.asigStat, P, idObj)			

Figura 6.9: Transformación de las declaraciones locales de variables

momento. De esta forma, las restricciones generadas para cada método tendrán acceso al objeto que corresponda.

6.4.4. Transformación de asignaciones, asertos y bloques de sentencias

Transformación de las declaraciones de variables locales

La función UE_LocalVar2CD (figura 6.9) es la encargada de realizar la transformación de las UE de tipo UE_LocalVar a componentes de tipo CD_Asig. Esta función recibe tres parámetros: la UE correspondiente a la declaración de la variable local, el predicado P cuyo valor indicará cuando se resuelva el problema de diagnosis si la sentencia pertenece o no a la traza correcta de ejecución, y el identificador del objeto que contiene el método que incluye la declaración de la variable local que se debe transformar.

Las UE que pueden estar implicadas en la transformación UE_LocalVar2CD son:

Unidades Ejecutadas implicadas
`UE_LocalVar ::= type:Type asigStat:UE_AsigStat`
`UE_AsigStat ::= id:IDStatement dest:ID* left:(IDLocal | IDField | IDParam) asigExp:AsigExp`

Cada UD de tipo declaración de variable local, y su correspondiente asignación inicial, será transformada a un componente de tipo CD_Asig. Para ello se reutilizará la función de transformación UE_AsigStat2CD que se mostrará en el siguiente apartado. Los argumentos enviados como entrada para esta función serán tres: la UE_AsigStat almacenada en el campo *uE_LocalVar.asigStat*; el mismo predicado P que se recibió como entrada, ya que si la declaración forma parte de la traza correcta, también la asignación inicial debe pertenecer a la traza correcta; y el mismo identificador del objeto que se recibió como entrada, ya que la asignación inicial pertenece al mismo objeto que la declaración de la variable.

Transformación: UE_AsigStat2CD			
Parámetros de entrada	UE_AsigStat: uE_AsigStat Path: P IDObject: idObj	Tipo devuelto	CD_Statement

Pasos

```
1) if (uE_AsigStat.asigExp instanceof Exp){
2)    return Asignment2CD(
         SSA.nextSSA(SSA.getIdObject(idObj, uE_AsigStat.dest), uE_AsigStat.left),
         uE_AsigStat.asigExp, SSA.newAB(uE_AsigStat.id), P, idObj)
   } else {
3)    if (uE_AsigStat.asigExp instanceof UE_CCall) {
4)       return UE_CCall2CD(uE_AsigStat.asigExp, P, idObj,
            SSA.nextSSA(idObj, uE_AsigStat.left))
5)    } elseIf (uE_AsigStat.asigExp instanceof UE_MCall) {
6)       return UE_MCall2CD(uE_AsigStat.asigExp, P, idObj,
            SSA.nextSSA(idObj, uE_AsigStat.left))
      }
   }
```

Figura 6.10: Transformación de sentencias de asignación

Transformación de las asignaciones

Las unidades ejecutadas que pueden estar implicadas en la transformación UE_AsigStat2CD son las siguientes:

Unidades Ejecutadas implicadas
`UE_AsigStat ::= id:IDStatement dest:ID* left:(IDLocal | IDField | IDParam) asigExp:AsigExp`
`AsigExp ::= Exp | UE_MCall | UE_CCall`
`UE_MCall ::= id:IDStatement dest:ID* met:UE_Method arguments:UE_Arguments*`
`UE_CCall ::= id:IDStatement met:UE_Constructor arguments:UE_Argument*`
`Exp ::= AritmeticExp | LogicExp | Literals | Identifier`

Para transformar cada UE_AsigStat a componentes se ha de aplicar la función UE_AsigStat2CD (ver figura 6.10). Esta función recibe tres parámetros: la UE correspondiente a la asignación, el predicado P cuyo valor indica si la sentencia de asignación pertenecerá o no a la traza correcta de ejecución, y el identificador del objeto que contiene al método que incluye la asignación que se desea transformar; y devuelve un componente que puede ser del tipo CD_Asig, CD_CCall, o CD_MCall, en función del tipo de asignación realizada.

La transformación consta de los siguientes pasos:

- Paso 1. Comprobar si la asignación es de una expresión, o de una llamada a un método o constructor. Para ello se utiliza el operador *instanceof* que permite reconocer si un elemento es de un tipo concreto.

- Paso 2. Si la expresión a asignar no es una llamada a un método o constructor, entonces se usará la función de transformación auxiliar Asignment2CD. Esta función transformará la asignación (UE_AsigStat) a un componente de tipo CD_Asig. La función de transformación auxiliar Asignment2CD recibe como parámetros de entrada, el objeto actual, un nuevo predicado AB, el predicado P recibido originalmente, la variable sobre la que se realizará la asignación, y la expresión a asignar (almacenada en el campo *uE_AsigStat.asigExp*).

  Para generar el nuevo predicado AB, se ha utilizado la función *SSA.nextAB*, que recibe el argumento *uE_AsigStat.id* que es el identificador de la asignación (de tipo IDStatement), y genera un nuevo predicado AB asociado a dicha asignación. Para calcular la variable sobre la que se realizará la asignación en la forma SSA, se ha utilizado la función *SSA.nextSSA*, que recibe como parámetros: el identificador del objeto actual, que ha sido calculado usando la función *SSA.getIdObject*; y el identificador de la variable, parámetro o atributo almacenado en *uE_AsigStat.left*, y que será sobre el que se realizará la asignación. La función *SSA.getIdObject* permite navegar hasta el objeto adecuado a partir de un objeto inicial y una lista de nombres de atributos. De esta forma es posible saber a qué objeto pertenece el atributo que se va a modificar en la asignación. En el caso de asignaciones sobre parámetros de entrada o sobre variables locales la función *SSA.getIdObject* simplemente devuelve el objeto que se recibió como parámetro de entrada.

- Pasos 3 y 4. Si la expresión a asignar es la llamada a un constructor, entonces el objeto creado en el constructor es el que debe ser asignado. La transformación de esta asignación a restricciones se realizará dentro de la función de transformación UE_CCall2CD. Para ello se envía a dicha función, como último parámetro de entrada, la variable sobre la que debe hacerse la asignación del objeto creado. Para transformar la variable enviada a la forma SSA se ha utilizado la función *SSA.nextSSA*. Será en la función de transformación UE_CCall2CD donde se añadirán las restricciones necesarias para modelar la asignación entre la variable a asignar, recibida como parámetro de entrada, y el objeto creado en la llamada al constructor.

- Pasos 5 y 6. Si la expresión a asignar es la llamada a un método, entonces el resultado devuelto por dicho método es el que debe ser asignado. La transformación de esta asignación a restricciones se realizará dentro de la función de transformación UE_MCall2CD. Para ello se envía a dicha función, como último parámetro de entrada, la variable sobre la que se debe realizar la asignación, y que previamente habrá sido transformada a la forma SSA utilizando la función *SSA.nextSSA*, de forma análoga a los pasos 3 y 4.

En el bloque de sentencias asociado al método llamado, debe existir al menos una sentencia de retorno, y será en la transformación de dicha sentencia donde se añadirán las restricciones necesarias para modelar la asignación entre la variable a asignar y la expresión a retornar. Para ello en la función de transformación UE_MCall2CD, la variable sobre la que se debe realizar la asignación, será reenviada a la función de transformación de la sentencia, o sentencias de retorno, que existan en el método llamado. Si existen varias sentencias de retorno, será el predicado P el que establezca cuál de las sentencias de retorno se debe ejecutar en la traza correcta, y por tanto, sólo las restricciones asociadas a dicha sentencia de retorno se tendrán que satisfacer en la solución al problema de diagnosis.

La función de transformación Asignment2CD es la encargada de crear los componentes de tipo CD_Asig. Tal como se verá a medida que se muestren todas las funciones de transformación, serán transformadas a componentes de tipo CD_Asig las asignaciones iniciales sobre los atributos y variables locales, la asignación de los argumentos a los parámetros de entrada de los métodos y constructores, las sentencias de asignación, y las sentencias de retorno en las llamadas a métodos. Todos estos componentes de tipo CD_Asig se generarán utilizando la misma función de transformación Asignment2CD.

En general, las asignaciones serán transformadas a restricciones de igualdad. La igualdad será entre la variable sobre la que se realiza la asignación y la expresión asignada. Esta transformación no puede realizarse directamente, ya que se pueden dar casos en los que aunque el código fuente sea correcto, sea imposible satisfacer la restricción generada a partir de una asignación. Por ejemplo, sea la asignación $x = x + 3$, de la que se obtendría la restricción de igualdad $x = x + 3$. Esta restricción nunca podría ser satisfecha, ya que no existe ningún valor para la variable x que cumpla dicha igualdad. En general, si una expresión es asignada a una variable, y dicha expresión a su vez contiene a dicha variable, aunque la asignación sea correcta, la restricción generada nunca podría ser satisfecha. Para resolver este problema se hará uso de la forma SSA (Static Single Assignment).

Antes de ver la función de transformación Asignment2CD, en el siguiente apartado se mostrarán las funciones encargadas de mantener la forma SSA en las restricciones que se generen.

Forma SSA

La ejecución de un programa sigue siempre una secuencia que garantiza que ciertas instrucciones se ejecutarán antes que otras. Esta característica no ocurre al resolver un problema de satisfacción con restricciones, ya que las restricciones pueden ser evaluadas repetidas veces, y en cualquier orden. Es importante mantener el orden en la ejecución de las sentencias para poder obtener los resultados esperados. Por esta razón el orden también debe mantenerse de forma implícita en las restricciones que formen la Descripción del Sistema. Para lograrlo, se mantendrá la forma SSA en las restricciones que se generen.

La forma SSA (Static Single Assignment) es equivalente al código fuente original pero sólo se permite una asignación para cada variable en la traza ejecutada. Por ejemplo para el código *x := x + 3* la forma SSA equivalente sería *x1 = x0 + 3*. De esta forma, en x0 se mantiene el valor de la variable x antes de la ejecución de esta sentencia, y en x1 se mantiene el valor de la variable x después de la ejecución de esta sentencia. Para una mayor detalle de la forma SSA se pueden consultar los trabajos [2] y [35].

En la forma SSA, las n asignaciones realizadas sobre una variable v por todas las sentencias que forman la traza ejecutada quedarían recogidas en la secuencia $v_1...v_n$ de variables. La transformación a la forma SSA permite conservar la secuencia seguida en la traza para las asignaciones sobre las variables, aunque obliga a manejar un mayor número de variables en el modelo basado en restricciones, ya que puede haber varias asignaciones para una misma variable del código fuente en una traza.

De manera análoga a como sucede con las asignaciones sobre cada variable local del código fuente, para un programa Orientado a Objetos, también la forma SSA debe mantenerse sobre cada asignación que se realice sobre cada atributo de un objeto, y sobre los parámetros de entrada de los métodos y constructores.

Con el fin de aislar la metodología para la generación de las restricciones con respecto a la metodología para el mantenimiento de la forma SSA, se hará uso de diferentes funciones auxiliares que serán las encargadas de proporcionar los nombres de las variables locales, parámetros de entrada y atributos de los objetos en la forma SSA.

La generación de la forma SSA se realizará durante la transformación de las UE a CD, concretamente en la creación de las restricciones asociadas a los componentes. Para ello se utilizarán los métodos que a continuación se exponen, y que son aplicables sobre parámetros de entrada, atributos de objetos y variables locales:

- La función *nextSSA* se utilizará para transformar a restricciones cada nueva asignación sobre un atributo de un objeto, o una variable local o parámetro de entrada de un método. Esta función auxiliar genera y devuelve una nueva variable que permite mantener la forma SSA.

- La función *lastSSA* debe ser utilizada al transformar cada lectura o acceso sobre un atributo de un objeto, o una variable local o parámetro de entrada de un método. Este método no genera una nueva variable, simplemente devuelve el nombre de la variable que se utilizó en la última asignación.

Las variables que devuelven ambas funciones auxiliares sustituirán a los atributos de objetos, parámetros de entrada y variables locales del código fuente original en las restricciones generadas. Para acceder a dichas funciones auxiliares, se supondrá la existencia de un objeto denominado SSA, que dará acceso a dichas funciones auxiliares. Ambas funciones reciben como entradas el identificador del atributo junto con el identificador del objeto (de tipo IDObject) al que pertenece el atributo; o el identificador del parámetro de entrada o variable local, junto con el identificador del objeto al que pertenece el método que incluye el parámetro de entrada o variable local.

Para simplificar la transformación de las expresiones a la forma SSA, se supondrá implementada además la función *lastSSAExp*. Esta función auxiliar recibe como parámetros la expresión que debe transformarse a la forma SSA, y el identificador del objeto sobre el que se evalúa dicha expresión. El resultado de esta función es una expresión donde cada una de las variables, atributos y parámetros han sido sustituidos por sus correspondientes en la forma SSA. Además de las funciones expuestas anteriormente, a medida que se vayan comentando las diferentes funciones de transformación, se irán introduciendo otras funciones necesarias para mantener la forma SSA, y que sólo se aplican en algunas funciones de transformación.

La transformación a la forma SSA obliga a manejar un mayor número de variables en el modelo basado en restricciones, ya que para una misma variable pueden realizarse muchas asignaciones, y para cada una de ellas será necesario definir una nueva variable. En la ejecución de un código fuente, el espacio de memoria reservado para una variable se reutiliza en cada asignación, sin embargo, en las restricciones asociadas al modelo, se utilizará una nueva variable para cada una de estas asignaciones. Esta característica debe ser tenida en cuenta al plantear el problema de diagnosis, ya que los entornos de programación con restricciones pueden poner límites en el número de variables disponibles, lo que implicaría una limitación sobre la metodología propuesta.

Ejemplo 6.6. A continuación se muestra un ejemplo de transformación de un código fuente a la forma SSA aplicando las funciones descritas anteriormente:

Código fuente	Transformación a la forma SSA	Forma SSA
int x = 1;	SSA.nextSSA(x, idObj) = SSA.lastSSAExp(1, idObj)	x0 = 1
...	...	...
int y = 8;	SSA.nextSSA(y, idObj) = SSA.lastSSAExp(8, idObj)	y0 = 8
...	...	...
y = x ∗ y;	SSA.nextSSA(y, idObj) = SSA.lastSSAExp(x ∗ y, idObj)	y1 = x0 ∗ y0
assert : y > 0;	SSA.lastSSAExp(y, idObj) > 0	y1 > 0
x = x + y	SSA.nextSSA(x, idObj) = SSA.lastSSAExp(x + y, idObj)	x1 = x0 + y1

Tal como se puede observar, en la última columna aparece el resultado de transformar el código fuente de la primera columna. Para realizar la transformación, en la columna central se han aplicado las funciones correspondientes, dependiendo de si se trataba de una asignación o de una lectura. El identificador *idObj* identifica al objeto al que pertenece el método que incluye las líneas de código mostradas en la primera columna. Este identificador se utiliza, por ejemplo, en la función *lastSSAExp*, para poder acceder a los atributos del objeto que forman parte de la expresión a transformar a la forma SSA.

Función de transformación Assignment2CD

En la figura 6.11 se muestran los pasos que componen la función de transformación Assignment2CD, y que son los siguientes:

Transformación: Assignment2CD		
Parámetros de entrada	Var: leftVar Exp: rightExp Abnormal: AB Path: P IDObject: idObj	Tipo devuelto CD_Asig

Pasos
1) CD_Asig cd = new CD_Asig()
2) if (¬(leftVar instanceof ObjectTypeVar)){
3) cd.sd.addCons(P ∧ ¬AB ⇒ (leftVar = SSA.lastSSAExp(idObj, rightExp)))
} else {
4) cd.sd.addCons(P ⇒ (leftVar = SSA.lastSSAExp(idObj, rightExp)))
}
5) return cd		

Figura 6.11: Transformación de asignaciones

- Paso 1. Crear un componente de tipo CD_Asig.
- Paso 2 y 3. Comprobar si la asignación no es de un objeto. Si la expresión a asignar no es un objeto, se añadirá la restricción de implicación mostrada en el paso 3 de la función de la transformación. El método *addCons* se utiliza para añadir nuevas restricciones al componente. La restricción añadida se podrá satisfacer sólo si se cumple alguno de los tres casos posibles que a continuación se muestran, y que cubren las 3 posibilidades que se pueden dar:

  - No se cumple el predicado P, y por tanto la asignación no pertenece a la traza correcta.
  - Se cumple el predicado P y se cumple el predicado AB. En este caso hay un comportamiento anormal. Es decir, este caso contemplado en la restricción, es el que permite modelar la posibilidad de que exista un defecto en la asignación.
  - Se cumple el predicado P y no se cumple el predicado AB. En este caso se debe cumplir el comportamiento esperado para una asignación, que corresponde a que la variable asignada debe ser igual al resultado de la expresión a asignar establecida en la sentencia. Este comportamiento es el que está expresado en la parte derecha de la implicación, a través de una igualdad entre la variable (*leftVar*) y la expresión a asignar (*rightExp*) transformada a la forma SSA.

  Al resolver el problema de diagnosis, esta restricción debe cumplirse, y para ello debe ocurrir alguna de las 3 posibilidades expuestas anteriormente. Dependiendo

de la opción que se cumpla, la interpretación será diferente, tal como se verá en el apartado dedicado a la localización de los defectos 6.4.

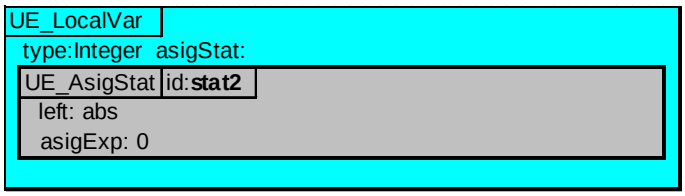
- Paso 4. Si la expresión a asignar es un objeto, se añadirá la restricción de implicación mostrada en la figura figura 6.11. Dicha restricción se podrá satisfacer sólo si:

  - Si no se cumple el predicado P, y por tanto la asignación no pertenece a la traza correcta.
  - Si se cumple el predicado P, y se cumple el comportamiento esperado para una asignación, que corresponde a que la variable asignada debe ser igual al resultado de la expresión a asignar establecida en la sentencia. Este comportamiento es el que está expresado en la parte derecha de la implicación, a través de una igualdad entre la variable (*leftVar*) y la expresión a asignar (*rightExp*) transformada a la forma SSA.

  La metodología propuesta no permite localizar defectos en asignaciones de objetos sobre variables. Este tipo de defectos se han dejado como trabajo futuro, ya que su estudio requiere un análisis más profundo. Por tanto en la restricción generada no se contempla la posibilidad de que exista un defecto en la asignación del objeto.

- Paso 5. Devolver el componente creado en el paso 1.

Ejemplo 6.7. Dada la UE de tipo declaración de variable local que se muestra a continuación:



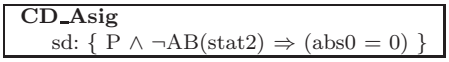
El primer paso en su transformación será utilizar la función de transformación UE_LocalVar2CD que a su vez realizará una llamada a la función de transformación UE_AsigStat2CD. En esta función se creará un nuevo predicado AB asociado a la sentencia de asignación inicial de la variable local, identificada como *stat2*. Dentro de la función UE_AsigStat2CD se llamará a su vez a la función de transformación Assignment2CD, enviando como parámetros entre otros el nuevo predicado AB. La restricción correspondiente a la asignación será almacenada en el campo *sd* del componente de tipo CD_Asig creado.

Una vez completadas todas las llamadas a las funciones de transformación citadas, el resultado será el componente de tipo asignación que a continuación se presenta. Para facilitar la visión de los CD, en este apartado, y en los ejemplos que se mostrarán más

Transformación: UE_Assert2CD			
Parámetros de entrada	UE_Assert: uE_Assert Path: P IDObject: idObj	Tipo devuelto	CD_Assert
Pasos 1) CD_Assert cd = new CD_Assert() 2) cd.sd.addCons(P ⇒ SSA.lastSSAExp(idObj, uE_Assert.exp)) 3) return cd			

Figura 6.12: Transformación de los asertos

adelante, se utilizará una representación visual, que contendrá en la esquina superior izquierda el tipo de CD y a continuación la lista de campos y los valores asociados a dichos campos. Si el campo permite almacenar un CD o una lista de CD, se mostrarán directamente dichos CD utilizando la misma representación visual.



Transformación de los asertos

Para transformar cada UE_Assert a restricciones se aplicará la función UE_Assert-2CD (figura 6.12). Esta función genera un componente de tipo CD_Assert, y de forma análoga a las funciones de transformación vistas anteriormente, recibe tres parámetros: la UE de tipo aserto, el predicado P cuyo valor indicará si el aserto pertenecerá o no a la traza correcta cuando el problema de diagnosis se resuelva, y el identificador del objeto al que pertenece el método que incluye el aserto a transformar.

Cada componente de tipo aserto contendrá las restricciones necesarias para que la condición establecida en el aserto se tenga en cuenta en el problema de diagnosis. En esta transformación la única UE implicada es UE_Assert:

Unidades Ejecutadas implicadas
`UE_Assert ::= exp:Exp`

Las expresiones asociadas a los asertos se supondrán correctas, y por tanto libres de defectos. El primer paso a seguir en la transformación UE_Assert2CD será crear un componente de tipo CD_Assert. En el segundo paso se añade como restricción, que si el aserto forma parte de la traza correcta, es decir, si se cumple P, entonces debe cumplirse la expresión asociada al aserto. La expresión asociada al aserto debe transformarse a la forma SSA, para ello se hará uso de la función *SSA.lastSSAExp*, que como se explicó anteriormente, sustituye cada parámetro, variable o atributo, por la variable que corresponda en la forma SSA. El último paso será devolver el componente creado en el paso 1, de tipo CD_Assert.

Transformación: UE_Block2CD			
Parámetros de entrada	UE_Block: uE_Block Path: P IDObject: idObj Var: retVar	Tipo devuelto	CD_Block

Pasos
1) CD_Block cd = new CD_Block()
2) foreach (uE_Statement : uE_Block.stats) {
3) if (uE_Statement instanceof UE_LocalVariable) {
cd.stats.add(**UE_LocalVar2CD**(uE_Statement, P, idObj)) }
4) if (uE_Statement instanceof UE_IfElse) {
cd.stats.add(**UE_IfElse2CD**(uE_Statement, P, idObj, retVar)) }
5) if (uE_Statement instanceof UE_Return $\wedge$ retVar $\neq$ null) {
cd.stats.add(**UE_Return2CD**(uE_Statement, P, idObj, retVar)) }
6) if (uE_Statement instanceof UE_Assert) {
cd.stats.add(**UE_Assert2CD**(uE_Statement, P, idObj)) }
7) if (uE_Statement instanceof UE_AsigStat) {
cd.stats.add(**UE_AsigStat2CD**(uE_Statement, P, idObj)) }
8) if (uE_Statement instanceof UE_MCall) {
cd.stats.add(**UE_MCall2CD**(uE_Statement, P, idObj, null)) }
9) if (uE_Statement instanceof UE_CCall) {
cd.stats.add(**UE_CCall2CD**(uE_Statement, P, idObj, null)) }
}
10) return cd

Figura 6.13: Transformación de bloques de sentencias

Transformación de los bloques de sentencias

Para transformar cada UE_Block a componentes se aplicará la función de transformación UE_Block2CD (figura 6.13). Las unidades ejecutadas que pueden estar implicadas en la transformación UE_Block2CD son:

Unidades Ejecutadas implicadas
`UE_Block ::= id: IDBlock stats:UE_Statement*`
`UE_Statement::= UE_LocalVariable | UE_IfElse | UE_Return | UE_Assert | UE_AsigStat | UE_MCall | UE_CCall`

La transformación de una UE_Block implica la transformación de cada una de las UE incluidas en el bloque. Para saber de qué tipo es cada UE almacenada se utiliza el operador *instanceof* que permite reconocer si un elemento es un tipo concreto. Para cada UE se realiza una llamada a la función de transformación correspondiente, reenviando los parámetros recibidos que correspondan.

```
if (a < b) {
    max = a
    min = b
} else {
    max = b
    min = a
}
dif = max - min
```

Figura 6.14: Sentencia selectiva con defecto en la condición

Los parámetros que recibe esta función son análogos a los que aparecen en las funciones de transformación vistas anteriormente. El último parámetro recibido no es utilizado directamente en la transformación, el objetivo es que sea utilizado en algunas de las transformaciones de las UE incluidas en la UE_Block. Por ejemplo, en las sentencias de tipo retorno que incluya el bloque de sentencias. Si se recibe este parámetro se debe a que existe una asignación que está esperando el resultado devuelto por el método que incluye a este bloque de sentencias. Este parámetro corresponde con la variable sobre la que se debe realizar la asignación (tal como se explicó en la función de transformación UE_AsigStat2CD). Será en la transformación de la sentencia de retorno, UE_Return2CD, donde se utilizará esta variable. En concreto, se añadirán las restricciones necesarias para modelar la asignación entre la variable recibida como parámetro, y la expresión a retornar.

Además de la función UE_Return2CD, también la función UE_IfElse2CD recibe este cuarto argumento, la razón es simple, se recibe para poder ser reenviado en el caso de que la UE_IfElse incluya en alguno de los dos bloques de sentencias, una UE de tipo UE_Return.

6.4.5. Transformación de sentencias selectivas

Para la transformación de las UD de tipo sentencia selectiva se utilizará la función de transformación UE_IfElse2CD. Las sentencias selectivas permiten, en función de una condición, que otras sentencias participen o no en la traza ejecutada. Para tener en cuenta esta característica en el modelo basado en restricciones se utilizará el predicado P (que se presento anteriormente). La función UE_IfElse2CD se explicará con detalle al final de esta sección, pero antes se mostrarán algunos conceptos necesarios para entender dicha función.

Defectos estructurales

En la figura 6.14 se muestra un código fuente que debería devolver como resultado en la variable *dif* la diferencia entre el valor máximo y mínimo de las variables *a* y *b*.

Sin embargo, debido a la condición establecida, no se ejecuta el bloque de sentencias correcto, y el resultado almacenado en la variable *dif* no es el esperado. Cuando se aplica un caso de prueba, el código fuente sigue una traza concreta, que tal como se ha visto en el ejemplo anterior, puede no ser la correcta. El defecto en la condición impuesta en la sentencia selectiva provoca que la traza ejecutada no sea la correcta.

Si en el código fuente existe una sentencia selectiva, la condición de dicha sentencia puede contener un defecto. En el caso de las sentencias selectivas, siempre hay dos opciones, una asociada al valor verdadero de la condición y otra al valor falso. Con el uso del predicado P será posible comprobar si la condición es incorrecta, y si una modificación en la traza seguida es suficiente para alcanzar el resultado esperado. Es decir, será posible detectar defectos estructurales debidos a las guardas de las sentencias condicionales y en base a ello, definir cuál debe ser la traza correcta. Para ello, en el problema de diagnosis se añadirán una serie de restricciones encaminadas a determinar cuál es la traza correcta a seguir para alcanzar el resultado esperado en el caso de prueba propuesto; y dentro de la traza correcta, determinar qué sentencias o asertos pueden contener defectos, y por tanto deben ser modificados.

Sea uE_IfElse una unidad ejecutada que representa una sentencia selectiva del código fuente, que contiene una condición uE_IfElse.exp, y dos bloques de sentencias uE_IfElse.if y uE_IfElse.else, asociados respectivamente a que la condición se cumpla y al caso contrario. Sean AB(uE_IfElse) y P(uE_IfElse) los predicados que establecen respectivamente el comportamiento anormal o no de uE_IfElse, y si la sentencia selectiva pertenece o no a la traza correcta; y sean P(uE_IfElse.if) y P(uE_IfElse.else) los predicados que establecen si los bloques de sentencias uE_IfElse.if y uE_IfElse.else pertenecen o no a la traza correcta. Para modelar el comportamiento asociado a uE_IfElse, se deben añadir las siguientes restricciones a la descripción del sistema.

1. $\neg$ P(uE_IfElse) $\Rightarrow$ P(uE_IfElse.if) = false
2. $\neg$ P(uE_IfElse) $\Rightarrow$ P(uE_IfElse.else) = false
3. P(uE_IfElse) $\Rightarrow$ P(uE_IfElse.if) $\neq$ P(uE_IfElse.else)
4. P(uE_IfElse) $\wedge$ $\neg$AB(uE_IfElse) $\Rightarrow$ P(uE_IfElse.if) = uE_IfElse.exp

La restricciones 1. y 2. establecen que si la sentencia selectiva (uE_IfElse) no pertenece a la traza correcta, tampoco pueden pertenecer a la traza correcta ninguno de los bloques de sentencias asociados al cumplimiento o no, de la condición impuesta en la sentencia selectiva.

La restricción 3. garantiza que si la sentencia selectiva pertenece a la traza correcta, sólo uno de los dos bloques de sentencias pertenecerá a la traza correcta. Para ello se obliga a que los predicados P(uE_IfElse.if) y P(uE_IfElse.else) siempre tengan un valor diferente.

Por último, la restricción 4. obliga a que si la sentencia selectiva pertenece a la traza correcta y su comportamiento es no anormal (es decir, la condición establecida

en la sentencia selectiva es correcta), entonces el bloque asociado al cumplimiento de la condición de la sentencia selectiva pertenecerá a la traza correcta si la condición establecida en la sentencia selectiva se cumple. Para ello se obliga a que el predicado P(uE_IfElse.if) tome el mismo valor que resulte de la evaluación de la expresión que condiciona la sentencia selectiva (uE_IfElse.exp). Conviene recordar, que por la restricción 3, es imposible que los predicados P(uE_IfElse.if) y P(uE_IfElse.else) tomen un mismo valor, y por tanto si se instancia P(uE_IfElse.if) a un valor concreto, de forma automática también se instanciará P(uE_IfElse.else) al valor contrario.

Estás restricciones permiten modelar el comportamiento asociado a una sentencia selectiva. En concreto, incluyendo estas restricciones en la descripción del sistema del problema de diagnosis, será posible comprobar si suponiendo defectuosa la condición de la sentencia selectiva, es posible alcanzar el resultado especificado como correcto.

Las sentencias selectivas pueden tener hasta dos bloques de sentencias asociados. El primer bloque se ejecuta si la condición se cumple. El segundo bloque puede no existir, pero si existe, se ejecuta si la condición impuesta en la sentencia selectiva no se cumple. En el caso de que sólo exista un bloque asociado al cumplimiento de la condición de la sentencia selectiva, las restricciones 2. y 3. no serían añadidas a la descripción del sistema.

Para que estas restricciones puedan ser utilizadas en la descripción del sistema será necesario añadir más elementos a la TUE (Traza de Unidades Ejecutadas), y conseguir lo que se definirá como la TUE ampliada. Para cada una de las sentencias selectivas, en la TUE se han guardado sólo los bloques de sentencias que se han ejecutado. Para poder añadir las restricciones propuestas anteriormente a la descripción del sistema, las dos alternativas posibles de cada sentencia selectiva deberían estar disponibles en la TUE. De esta forma se podría contemplar en las restricciones cuál sería el comportamiento del código fuente en caso de cambiar las condiciones impuestas en las guardas de las sentencias selectivas.

En el siguiente apartado se explicará la forma de obtener la TUE ampliada, y a continuación, y antes de mostrar la función de transformación UE_IfElse2CD, se presentará la función ϕ necesaria para mantener la forma SSA dentro de las restricciones asociadas a la transformación de las sentencias selectivas.

TUE Ampliada

Tal como se ha expuesto anteriormente, sólo las sentencias que han participado en la traza o trazas, pueden contener defectos, pues sólo ellas han influido en los resultados. La metodología de diagnosis propuesta en esta tesis se centra en estudiar trazas concretas, en las que las diferentes sentencias del código fuente se han ido ejecutando. De esta forma, se consigue eliminar de la descripción del sistema aquellas sentencias que no han influido en los errores detectados, y se mantienen sólo aquellas sentencias que pueden ser causa de los errores detectados. Las restricciones de la descripción del sistema se obtienen de la transformación de las sentencias que forman la traza.

En el caso de las sentencias selectivas, siempre hay dos opciones, una asociada al valor verdadero de la condición y otra al valor falso; y uno, o dos bloques de sentencias asociados. Para poder determinar si existe un defecto en la condición de las sentencias selectivas, sería necesario mantener en la descripción del sistema el resultado que generaría cualquiera de las dos opciones. De esta forma, la metodología de diagnosis podría disponer de las dos alternativas, y con esta información podría comprobar cuál de las dos alternativas permite alcanzar los resultados esperados, y de forma indirecta, cuál debe ser la traza correcta a seguir para poder alcanzar el resultado esperado.

Para poder ampliar la traza seguida se utilizará el código fuente original. Las UE introducidas para completar la traza, se obtendrán como transformación de las sentencias y asertos de la alternativa no ejecutada. Las sentencias selectivas, llamadas a métodos y constructores también serán traducidas a UE.

La ampliación de la TUE no siempre será posible e implica ampliar la complejidad de la descripción del sistema, pero permitirá la localización de los defectos estructurales. Este paso se realizará sólo en las sentencias selectivas, pero no en las sentencias iterativas, aunque éstas hayan sido transformadas a sentencias selectivas. Ampliar la TUE en el caso de las sentencias iterativas, implicaría una complejidad mayor que en el caso de las sentencias selectivas, pues en el caso de las sentencias iterativas tendría que acotarse el número de posibles iteraciones. En el caso de las sentencias selectivas, sólo es necesario añadir una opción más. Por tanto la ampliación de la descripción del sistema, sólo se realizará en aquellos casos en los que la alternativa a introducir en la sentencia selectiva, no incluya ninguna sentencia iterativa.

Tampoco podrá ampliarse la TUE en el caso de llamadas recursivas, ya que de forma análoga a como ocurre con los bucles, no existe una función universal que permita acotar el número de llamadas recursivas que se podrían dar hasta alcanzar el caso base, para cualquier programa recursivo. Por tanto, los defectos estructurales serán contemplados, sólo en aquellas sentencias selectivas que permitan la ampliación de la TUE, o que sólo tengan un bloque asociado a la condición de la sentencia selectiva y dicho bloque haya sido ejecutado y por tanto aparezca en al TUE. En este punto conviene recordar que es inviable transformar directamente las sentencias iterativas y recursivas a restricciones, ya que en la programación con restricciones no se permiten bucles ni llamadas recursivas. Este es otro de los motivos por los que en la metodología propuesta, para tratamiento de las sentencias iterativas se hace primero una conversión a sentencias selectivas, en las cuales sí es posible hacer una transformación a restricciones.

Ejemplo 6.8. En la figura 6.15 se muestra la TUE original (parte central de la figura) obtenida aplicando el caso de prueba a = 4 y b = 3, sobre el código fuente que aparece a la izquierda. Para este caso de prueba el código fuente no genera el resultado correcto, por tanto el código fuente es defectuoso de acuerdo con el caso de prueba.

En la parte derecha se muestra la TUE ampliada en la que se han añadido las UE correspondientes al bloque de sentencias de la sentencia selectiva que no se ha ejecutado. En concreto se ha añadido las UE correspondientes al bloque asociado a que se cumpla la condición de la sentencia selectiva.

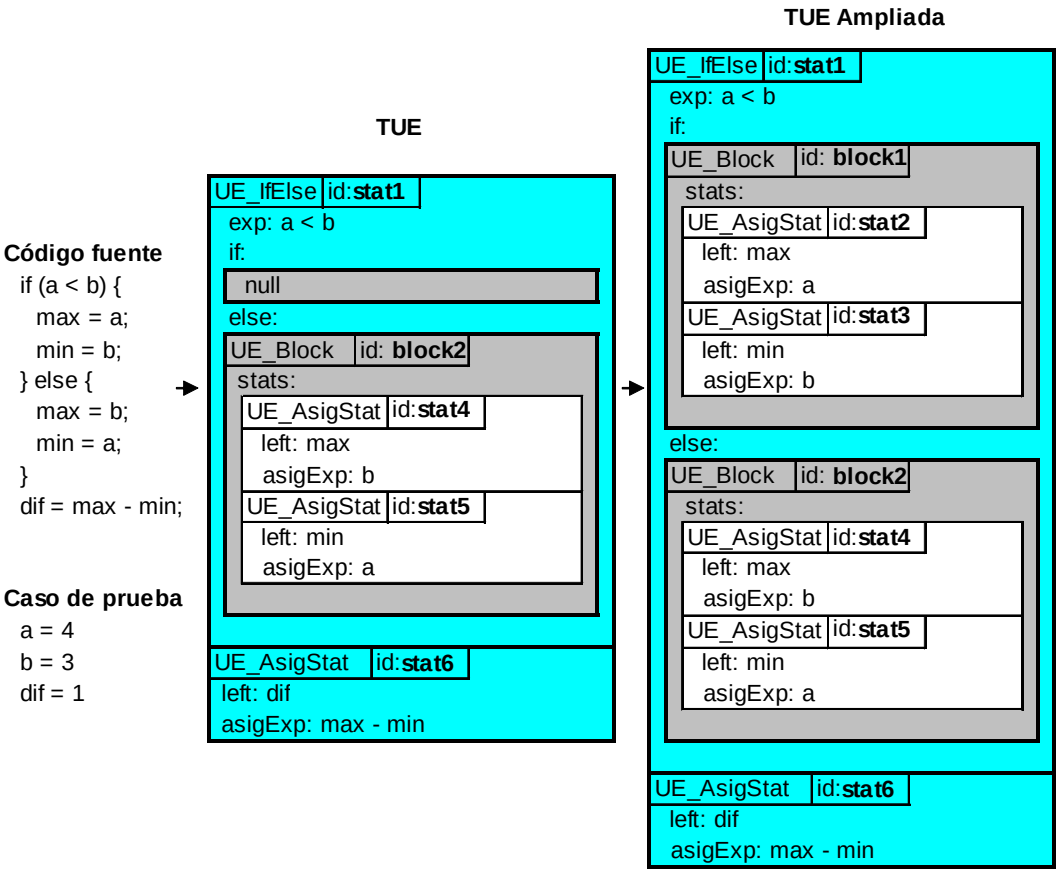


Figura 6.15: TUE original y TUE ampliada

Mantenimiento de la forma SSA en las sentencias selectivas

Las sentencias selectivas permiten añadir nuevas sentencias a la traza seguida en función de la condición establecida en dichas sentencias. Sea por ejemplo el siguiente código fuente, y su transformación a la forma SSA:

	Código Fuente		Forma SSA
01	if (a < b) {		if (a0 < b0) {
02	max = a		max1 = a0
03	min = b		min1 = b0
	} else {	$\longrightarrow$	} else {
04	max = b		max2 = b0
05	min = a		min2 = a0
	}		}
06	dif = max - min		dif0 = max? - min?

Como se puede observar, en cada uno de los bloques asociados se realiza la asignación sobre las dos variables *max* y *min*. La sentencia número 6 realiza una asignación en la que es necesario leer el valor de las variables *max* y *min*. Para realizar dicha asignación es necesario conocer cuál de las alternativas de la función selectiva ha sido seleccionada, de ahí que en la figura los nombres de las variables aparezcan con una interrogación. Para lograr este objetivo será necesario adaptar la función ϕ, que se utiliza en la bibliografía asociada a la forma SSA [35].

La función ϕ recibe la expresión usada como condición en la sentencia selectiva, y asigna para cada variable asignada dentro de la sentencia selectiva, el valor adecuado para mantener en la forma SSA el mismo comportamiento que en el código fuente original. La función ϕ también sería aplicable a atributos de objetos y a parámetros de entrada de métodos.

Dada una sentencia selectiva S con dos bloques de sentencias asociados, se define la función $\phi(\mathrm{S}_{Cond}, \mathrm{v}_{Post}, \mathrm{v}_{If}, \mathrm{v}_{Else})$ como:

$\phi(\mathrm{S}_{Cond}, \mathrm{v}_{Post}, \mathrm{v}_{Pre}, \mathrm{v}_{If}, \mathrm{v}_{Else})\{$
  if (v_{If} = null)
    $\mathrm{v}_{If} = \mathrm{v}_{Pre}$
  if (v_{Else} = null)
    $\mathrm{v}_{Else} = \mathrm{v}_{Pre}$
  if S_{cond} then $\mathrm{v}_{Post} = \mathrm{v}_{If}$ else $\mathrm{v}_{Post} = \mathrm{v}_{Else}$
}

Donde:

- S_{cond} es la condición de la sentencia selectiva.
- V es el conjunto de variables que son asignadas dentro de alguno de los dos bloques de sentencias.
- $v \in$ V representa cada una de las variables que pueden ser asignadas en cada uno de los dos bloques de la sentencia selectiva.
- v_{Pre} representa la variable que corresponde con la ultima transformación a la forma SSA de la variable v, antes de realizar la transformación a la forma SSA de la sentencia selectiva S.
- v_{If} representa la variable que corresponde con la última transformación a la forma SSA de la variable v, en el bloque de sentencias asociado al cumplimiento de la condición de la sentencia selectiva.
- v_{Else} representa la variable que corresponde con la última transformación a la forma SSA de la variable v, en el bloque de sentencias asociado al no cumplimiento de la condición de la sentencia selectiva.

- v_{Post} representa la variable que corresponde con una nueva transformación a la forma SSA de la variable v una vez transformados a la forma SSA los dos bloques de sentencias asociados a la sentencia selectiva.

La función ϕ iguala la variable v_{Post} al valor generado por el bloque de sentencias que se ejecutaría, teniendo en cuenta la evaluación de la condición S_{cond}. De forma general, con el uso de la función ϕ, se consigue proporcionar en la forma SSA el mismo comportamiento de la sentencia selectiva original.

En el ejemplo mostrado anteriormente, aplicando la función ϕ, la transformación a la forma SSA quedaría como sigue:

	Código Fuente		Forma SSA
(S01)	if (a < b) {		if (a0 < b0) {
(S02)	max = a		max1 = a0
(S03)	min = b		min1 = b0
	} else {	⟶	} else {
(S04)	max = b		max2 = b0
(S05)	min = a		min2 = a0
	}		}
			ϕ(a0 < b0, max3, max0, max1, max2)
			ϕ(a0 < b0, min3, max0, min1, min2)
(S06)	dif = max - min		dif0 = max3 - min3

La función ϕ establece cuál es el valor que deben tomar las variables *max3* y *min3*. Las sentencias que aparezcan a continuación en el código fuente, podrán leer el contenido de *max3* y *min3*, que guardarán respectivamente el resultado equivalente al que se generaría la sentencia selectiva.

Para mantener la forma SSA en las restricciones generadas a partir de las UE_IfElse, la función ϕ debe adaptarse. En concreto, la función ϕ que se utilizará en la transformación a restricciones de las UE de tipo sentencia selectiva será la siguiente:

ϕ(S_{cond}, v_{Post}, v_{Pre}, v_{If}, v_{Else}, SD sd) {
  if (v_{If} = null)
    v_{If} = v_{Pre}
  if (v_{Else} = null)
    v_{Else} = v_{Pre}
  sd.addCons(if S_{Cond} then v_{Post} = v_{If}
                else v_{Post} = v_{Else})
}

Donde sd representa un contenedor de restricciones del tipo SD. La función ϕ añade una restricción para establecer que el valor de la variable v_{Post} será igual a v_{If} si la

Transformación: UE_IfElse2CD			
Parámetros de entrada	UE_IfElse: uE_IfElse Path: P IDObject: idObj Var: retVar	Tipo devuelto	CD_IfElse

```
01 CD_IfElse cd = new CD_IfElse()
02 Abnormal AB = SSA.newAB(uE_IfElse.id)
03 Path P(uE_IfElse.if) = SSA.newPath(uE_IfElse.if)
04 cd.sd.addCons(¬P ⇒ P_If = false)
05 cd.sd.addCons(P ∧ ¬AB ⇒ P(uE_IfElse.if) = (SSA.lastSSAExp(idObj, uE_IfElse.exp)))
06 if (uE_IfElse.if ≠ null) {
07   cd.if = UE_Block2CD(uE_IfElse.if, P(uE_IfElse.if), idObj, retVar)
   }
08 if (uE_IfElse.else ≠ null) {
09   Path P(uE_IfElse.else) = SSA.newPath(uE_IfElse.else)
10   cd.sd.addCons(¬P ⇒ P(uE_IfElse.else) = false)
11   cd.sd.addCons(P ⇒ P(uE_IfElse.if) ≠ P(uE_IfElse.else))
12   cd.else = UE_Block2CD(uE_IfElse.else, P(uE_IfElse.else), idObj, retVar)
   }
13 SSA.IfElseAsign(uE_IfElse, idObj, cd.sd)
14 return cd
```

Figura 6.16: Transformación de sentencias selectivas

condición se cumple, y en caso contrario, la variable v_{Post} será igual a v_{Else}.

Transformación de las sentencias selectivas

Una vez vistos los conceptos de TUE ampliada, y las extensiones necesarias para mantener la forma SSA, en este apartado se mostrará la función de transformación para las sentencias selectivas. La unidad ejecutada implicada en la transformación de una sentencia condicional es la UE_IfElse:

Unidad Ejecutada implicada
`UE_IfElse ::= id:IDStatement exp:Exp if:UE_Block else:UE_Block`

Para transformar cada UE_IfElse a restricciones, se aplicará la función UE_IfElse2CD mostrada en la figura 6.16. Esta función recibe cuatro parámetros. De forma análoga a las funciones de transformación vistas anteriormente necesita recibir: la UE correspondiente a la sentencia selectiva; el predicado P cuyo valor indicará si la sentencia selectiva pertenece o no a la traza correcta cuando se solucione el problema de diagnosis; y el identificador del objeto actual.

El cuarto parámetro recibido no es utilizado directamente en la transformación, el objetivo es que llegue a las UE_Return que pueden estar incluidas en los bloques asociados a la UE_IfElse. Si se recibe este parámetro se debe a que existe una asignación que está esperando el resultado devuelto por el método que incluye este bloque de sentencias; este parámetro corresponde con la variable sobre la que se debe realizar la asignación (tal como se explicó en la función de transformación UE_AsigStat2CD). Será en la transformación de la sentencia de retorno donde se utilizará este parámetro.

La función de transformación UE_IfElse2CD consta tres partes. Una primera parte (pasos 1 al 7) encargada de transformar la condición y el primer bloque de sentencias, una segunda parte (pasos 8 al 12) encargada de transformar el segundo bloque de sentencias si existiese, y una tercera parte (pasos 13 al 14) encargada de aplicar la función ϕ y devolver el componente creado. Los pasos son:

- Paso 1. Creación del componente de tipo CD_IfElse. Este componente es el que se devolverá una vez completada la transformación.
- Paso 2. Creación del predicado AB asociado a la sentencia selectiva. Para crear el nuevo predicado se hará uso de la función *SSA.newAB*. Este predicado almacena si la sentencia selectiva tiene o no un comportamiento anormal, es decir, si tiene defectos y debe modificarse.
- Paso 3. Creación del predicado P asociado al primer bloque. Para crear un nuevo predicado se hará uso de la función *SSA.newPath*. Este predicado recoge si el primer bloque de sentencias pertenecerá o no a la traza correcta.
- Paso 4. Añadir la restricción 1 (presentada en el apartado 6.4.5). Tal como se ha explicado anteriormente, esta restricción establece que si la sentencia selectiva (uE_IfElse) no pertenece a la traza correcta, tampoco pueden pertenecer a la traza correcta el bloque de sentencias asociado al cumplimiento de la condición impuesta en la sentencia selectiva.
- Paso 5. Añadir la restricción 4 (presentada en el apartado 6.4.5). Esta restricción establece que si la sentencia selectiva pertenece a la traza correcta y su comportamiento es no anormal, entonces el bloque asociado al cumplimiento de la condición de la sentencia selectiva pertenecerá a la traza correcta si la condición de la sentencia selectiva se cumple.
- Pasos 6 y 7. Transformación del bloque de sentencias asociado al cumplimiento de la condición. En el caso de que existan sentencias dentro de dicho bloque, para su transformación se hará uso de la función de transformación UE_Block2CD, que devolverá un componente de tipo CD_Block.
- Paso 8. Transformación del bloque de sentencias asociado al no cumplimiento de la condición. En el caso de que existan sentencias dentro de dicho, se procederá a su transformación a través de los pasos 9, 10, 11 y 12.

- Paso 9. Creación del predicado P asociado al segundo bloque. Para crear un nuevo predicado se hará uso de la función *SSA.newPath*. Este predicado recoge si el bloque de sentencias asociado al no cumplimiento de la condición de la sentencia selectiva pertenecerá o no a la traza correcta.

- Paso 10. Añadir la restricción 2 (presentada en el apartado 6.4.5). Tal como se ha explicado anteriormente, esta restricción establece que si la sentencia selectiva (uE_IfElse) no pertenece a la traza correcta, tampoco pueden pertenecer a la traza correcta el bloque de sentencias asociado al no cumplimiento de la condición impuesta en la sentencia selectiva.

- Paso 11. Añadir la restricción 3 (presentada en el apartado 6.4.5). Esta restricción establece que si la sentencia selectiva pertenece a la traza correcta, entonces sólo uno de los dos bloques, el asociado al cumplimiento de la condición, o el asociado al no cumplimiento de la condición, debe pertenecer a la traza correcta.

- Paso 12. Transformación del bloque de sentencias asociado al no cumplimiento de la condición. De forma análoga al paso 7, para su transformación se hará uso de la función de transformación UE_Block2CD.

- Paso 13. Aplicación de la función *SSA.IfElseAsign* encargada de añadir las restricciones necesarias para mantener la forma SSA una vez acabada la transformación de la sentencia selectiva. Esta función realizará una llamada a la función ϕ descrita anteriormente, por cada atributo, parámetro o variable local que pueda ser asignada dentro de la sentencia selectiva. Para ello recoge tres parámetros: la UD_IfElse a transformar, el identificador del objeto actual, y el contenedor donde se deben almacenar las restricciones generadas por la función ϕ.

- Paso 14. Retorno del componente de tipo CD_IfElse.

Ejemplo 6.9. A continuación se muestra como sería la transformación para la TUE mostrada en la figura 6.17. En dicha TUE se incluyen dos UE, una primera de tipo UE_IfElse y una segunda de tipo UE_AsigStat.

Para transformar la primera UE será necesario aplicar la función de transformación UE_IfElse2CD. En esta transformación se obtendrían las siguientes restricciones, como consecuencia de aplicar los pasos 4, 5, 10 y 11.

$\neg P \Rightarrow P(block1) = false$
$P \wedge \neg AB(stat1) \Rightarrow P(block1) = (a < b)$
$\neg P \Rightarrow P(block2) = false$
$P \Rightarrow P(block1) \neq P(block2)$

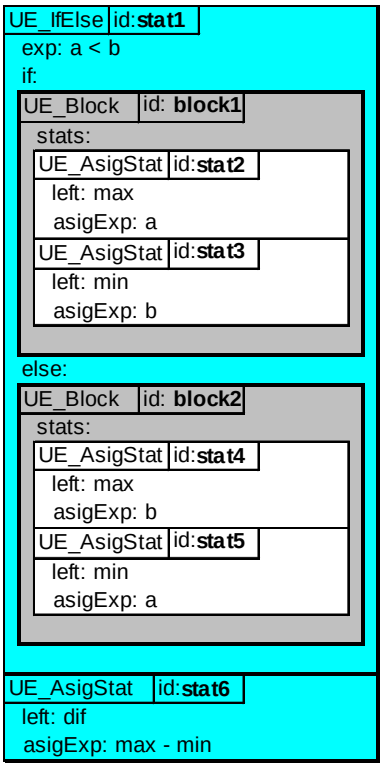


Figura 6.17: TUE Ampliada

Cada predicado P estará asociado al identificador de la UE a que corresponda, por ejemplo para el bloque asociado al cumplimiento de la condición en la sentencia selectiva se ha utilizado el identificador *block1*. En el paso 13, se añadirían las restricciones que genera la función ϕ, para las dos variables que son asignables en los dos bloques, *max* y *min*. En concreto, las llamadas a la función ϕ generan las restricciones:

if P(a < b) then max3 = max1 else max3 = max2
if P(a < b) then min3 = min1 else min3 = min2

Previamente, en los pasos 7 y 12, se habrán realizado las transformaciones de los bloques de sentencias asociados, usando la función UE_Block2CD. A su vez, dentro de la transformación de los bloques, habrá sido necesario utilizar la función de transformación UE_AsigStat2CD para transformar las asignaciones. Con las restricciones mostradas anteriormente, y con los resultados obtenidos de la transformación de los bloques, el CD_IfElse quedaría como sigue:

CD_IfElse
sd:
{ ¬P ⇒ P(block1) = false,
P ∧ ¬AB(stat1) ⇒ P(block1) = (a < b),
¬P ⇒ P(block2) = false,
P ⇒ P(block1) ≠ P(block2),
if P(block1) then max3 = max1 else max3 = max2,
if P(block1) then min3 = min1 else min3 = min2 }
if:
CD_Asig
sd: { P(block1) ∧ ¬AB(stat2) ⇒ max1 = a }
CD_Asig
sd: { P(block1) ∧ ¬AB(stat3) ⇒ min1 = b }
else:
CD_Asig
sd: { P(block2) ∧ ¬AB(stat4) ⇒ max2 = b }
CD_Asig
sd: { P(block2) ∧ ¬AB(stat5) ⇒ min2 = a }

Para transformar el segundo componente de la TUE, será necesario aplicar la función UE_AsigStat2CD, que daría como resultado el siguiente CD_Asig:

CD_Asig
sd: { P ∧ ¬AB(stat6) ⇒ dif = max3 - min3 }

6.4.6. Transformación de llamadas a métodos y constructores

Transformación de las llamadas a constructores

Para transformar cada UE_CCall a componentes se aplicará la función de transformación UE_CCall2CD (figura 6.18). La función de transformación UE_CCall2CD recibe cuatro parámetros. Los tres primeros son análogos a los vistos en las funciones de transformación ya explicadas. El último de los parámetros, *retVar*, si es recibido es debido a que existe una asignación que está esperando el objeto que se creará en el constructor llamado. Este parámetro se utilizará en el paso 3 de la transformación UE_CCall2CD.

Cada transformación de una llamada a un constructor implica transformar a restricciones la propia llamada y las sentencias incluidas en el bloque de sentencias asociado al constructor. Por tanto, las unidades ejecutadas que pueden estar implicadas en la transformación UE_CCall2CD son:

Unidades Ejecutadas implicadas
`UE_CCall ::= id:IDStatement met:UE_Constructor arguments:UE_Argument*`
`UE_Argument ::= exp:Exp`
`UE_Constructor ::= id:IDConstructor par:UE_Param* block:UE_Block ret:UE_Object pre:UE_Assert* post:UE_Assert*`
`UE_Param ::= id:IDParam type:Type`
`UE_Object ::= id:IDClass invs:UE_Invariant* fields:UE_Field*`
`UE_Invariant ::= exp:Exp`
`UE_Field ::= type:Type asigStat:UE_AsigStat`

Transformación: UE_CCall2CD			
Parámetros de entrada	UE_CCall: uE_CCall Path: P IDObject: idObj Var: retVar	Tipo devuelto	CD_CCall

Pasos
1) CD_CCall cd = new CD_CCall()
2) IDObject newObjId = SSA.newObject(uE_CCall.met.ret.id)
3) cd.sd.addCons([P ⇒ (retVar = newObjId)])
4) foreach (uE_Field : uE_CCall.met.ret.fields) {
cd.fields.add(**UE_AsigStat2CD**(uE_Field.asigStat, P, newObjId)) }
5) foreach ((uE_Param, uE_Argument) : (uE_CCall.met.par, uE_CCall.arguments)) {
cd.arguments.add(**Assignment2CD**(SSA.nextSSA(uE_Param.id), uE_Argument.exp,
SSA.newAB(uE_Param.id), P, idObj)) }
6) foreach (UE_Assert uE_Assert: uE_CCall.met.pre) {
cd.met.add(**UE_Assert2CD**(uE_Assert, P, newObjId)) }
7) cd.met.add(**UE_Block2CD**(uE_CCall.uE_Method.block, P, newObjId, null))
8) foreach (UE_Assert uE_Assert: uE_CCall.met.post) {
cd.met.add(**UE_Assert2CD**(uE_Assert, P, newObjId)) }
9) foreach (UE_Assert uE_Assert: InvStore.getInvs(uE_CCall.met.id)) {
cd.met.add(**UE_Assert2CD**(uE_Assert, P, newObjId)) }

Figura 6.18: Transformación de las llamadas a contructores

En la transformación UE_CCall2CD se han de seguir una serie de pasos que transformarán a restricciones los argumentos enviados como parámetros, el nuevo objeto creado, los atributos del objeto creado, la precondición, el bloque de sentencias del constructor, la postcondición, y los invariantes después de ejecutar el constructor. La transformación UE_CCall2CD devuelve un componente de tipo CD_CCall. Los pasos a realizar en la transformación son:

- Paso 1. Creación del componente de tipo CD_CCall. Este componente es el que se devolverá una vez completada la transformación.
- Paso 2. Creación del nuevo objeto. Para crear un nuevo objeto que pueda ser utilizado en las restricciones, se hará uso de la función *SSA.newObject* que recibe como entrada el tipo o clase del cual el objeto será instancia. El identificador del nuevo objeto se almacena en la variable *newObjId*. Tal como se explicó anteriormente, en la descripción del sistema basada en restricciones, cada objeto debe tener un identificador único de tipo IDObject. En adelante, este será el objeto sobre el que actuarán las restricciones obtenidas al transformar el bloque de sentencias asociado al constructor.
- Paso 3. Asignación del nuevo objeto. Una vez creado el nuevo objeto, se añadirá la

restricción que garantice que en el caso de que el predicado P sea cierto, es decir, la llamada al constructor pertenece a la traza correcta, la variable *retVar*, recibida como parámetro, debe ser igual al objeto creado por el constructor.

- Paso 4. Creación de los atributos del nuevo objeto. La creación de un objeto, implica la creación e inicialización del conjunto de atributos incluidos en cada objeto. Cada asignación inicial de cada atributo será transformada a restricciones reutilizando la función de transformación UE_AsigStat2CD. Para cada atributo se obtendrá un componente de tipo CD_Asig, y el conjunto de estos componentes se almacenará en el campo *fields*.

- Paso 5. Transformación de los parámetros de entrada. Cuando se realiza una llamada a un método o constructor se asigna a cada parámetro de entrada la expresión recibida como argumento de entrada. Se debe cumplir que para cada uno de los parámetros de entrada debe existir un argumento de entrada. El envío de un argumento de entrada puede entenderse como una asignación del argumento de entrada al parámetro de entrada, por tanto, puede ser transformado como si se tratase de una asignación. Para realizar esta transformación se hará uso de la función auxiliar de transformación Asignment2CD vista en el apartado 6.4.4. Para cada pareja formada por el parámetro y el argumento de entrada se obtendrá un componente de tipo CD_Asig, y el conjunto de estos componentes se almacenará en el campo *arguments*. La variable sobre la que se realizará la asignación será la correspondiente al parámetro de entrada en la forma SSA, y la expresión a asignar será el argumento de entrada.

  Para realizar la llamada a la función auxiliar de transformación Asignment2CD, es necesario crear un nuevo predicado AB por cada parámetro de entrada. La función *SSA.newAB* es la encargada de crear el nuevo predicado AB. Si el predicado AB toma el valor verdadero implicará que la expresión enviada como argumento de entrada al constructor no es correcta, y se considerará un defecto.

- Paso 6. Transformación de la precondición. Antes de ejecutar el constructor debe cumplirse la precondición. Para añadir esta información al conjunto de restricciones del componente, cada aserto de la precondición será transformado a restricciones reutilizando la función de transformación UE_Assert2CD vista anteriormente.

- Paso 7. Transformación del bloque de sentencias. En este paso se hará uso de la función de transformación UE_Block2CD. Esta función devuelve un componente de tipo CD_Block. El último de los argumentos enviados a dicha función de transformación está reservado para reenviar la variable que debe recoger la expresión devuelta por una UE_Return. Un método constructor no tiene sentencia de retorno, por ese motivo se envía *null* como cuarto argumento. Este cuarto argumento, tendrá sentido más adelante, cuando se muestre la función de transformación UE_MCall2CD encargada de transformar llamadas a métodos, y en cuyo bloque de sentencias si puede aparecer una UE_Return.

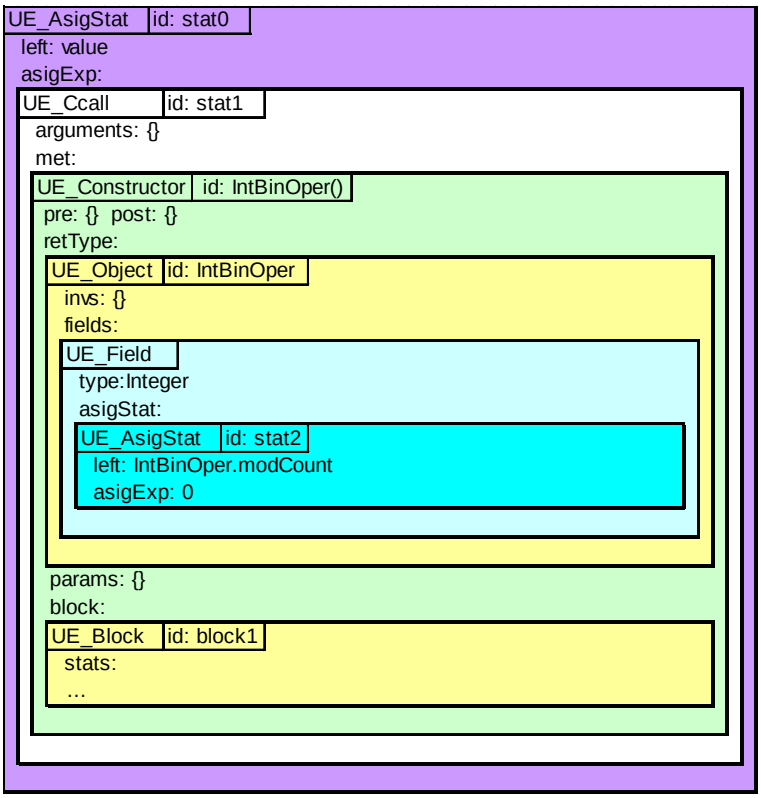


Figura 6.19: UE obtenida de la asignación del resultado de un constructor a una variable

- Paso 8. Transformación de la postcondición. Tras la ejecución del constructor debe cumplirse la postcondición. Para añadir esta información en forma de restricciones, de forma análoga a como se hizo en la precondición, cada aserto de la postcondición será transformado a restricciones reutilizando la función de transformación UE_Assert2CD.

- Paso 9. Transformación de los invariantes al acabar el constructor. Una vez ejecutado el constructor, deben cumplirse los invariantes de la clase. Para recuperar los invariantes asociados a cada clase, se hará uso de la función *InvStore.getInvs*, que recibiendo el identificador de un método o constructor, devuelve el conjunto de invariantes asociados a la clase que contiene dicho método o constructor. Cada invariante será transformado a restricciones reutilizando la función de transformación UE_Assert2CD.

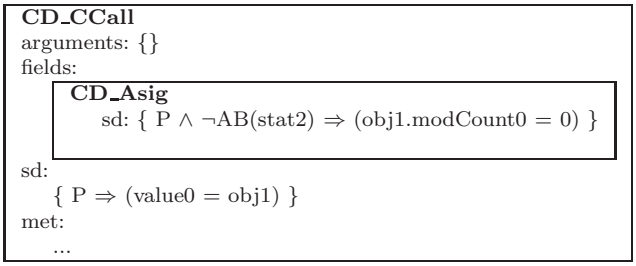
- Paso 10. Retorno del componente de tipo CD_CCall, que se creó en el paso 1.

Ejemplo 6.10. En la figura 6.19 se muestra una UE de tipo asignación, en la que se asigna el resultado de la llamada a un constructor sobre una variable. Para transformar este ejemplo a CD, el primer paso será utilizar la función UE_AsigStat2CD, que se

presentó en el apartado 6.4.4. En dicha función, en el paso 4, se realizará una llamada a la función de transformación UE_CCall2CD, encargada de realizar la transformación de la llamada al constructor.

El método constructor no tiene parámetros, por lo que no habrá que añadir restricciones para los parámetros. Una vez creado el nuevo objeto, al que se referencia como *obj1*, el siguiente paso será inicializar los atributos. El nuevo objeto tiene un atributo, identificado como *obj1.modCount*, que se inicializa a través de una llamada a la función de transformación de asignaciones UE_AsigStat2CD. Para procesar el bloque de sentencias se hará uso de la función de transformación UE_Block2CD, aunque para simplificar este ejemplo, no se han incluido sentencias en dicho bloque (en su lugar aparecen tres puntos suspensivos). Una vez transformado el bloque de sentencias, al no existir postcondición ni invariantes, se habría completado la transformación.

A continuación se muestra el resultado completo de la transformación, del que se obtendría un componente de tipo CD_CCall.



La restricción incluida en el componente de tipo CD_CCall (P ⇒ (var0 = obj1)), establece a que si dicho componente pertenece a la traza correcta, entonces la variable *var0* debe tener como dominio asociado para dicha variable el objeto *obj1*.

Transformación de las llamadas a métodos

Para transformar cada UE_MCall a componentes, se aplicará la función de transformación UE_MCall2CD (figura 6.20). Cada transformación de una llamada a un método implica transformar a restricciones la llamada y las sentencias que forman el bloque de sentencias asociado al método llamado. Las unidades ejecutadas que pueden estar implicadas en la transformación UE_MCall2CD son:

Unidades Ejecutadas implicadas
`UE_MCall ::= id:IDStatement dest:ID* met:UE_Method arguments:UE_Arguments*`
`UE_Argument ::= exp:Exp`
`UE_Method ::= id:IDMethod params:UE_Param* block:UE_Block retType:Type pre:UE_Assert* post:UE_Assert*`
`UE_Param ::= id:IDParam type:Type`

Transformación: UE_MCall2CD			
Parámetros de entrada	UE_MCall: uE_MCall Path: P IDObject: idObj Var: retVar	Tipo devuelto	CD_MCall

Pasos
1) CD_MCall cd = new CD_MCall()
2) foreach ((uE_Param, uE_Argument) : (uE_MCall.met.par, uE_MCall.arguments)) {
cd.arguments.add(**Asignment2CD**(SSA.nextSSA(uE_Param.id), uE_Argument.exp, SSA.newAB(uE_Param.id), P, idObj)) }
3) foreach (UD_Assert uD_Assert : InvStore.getInvs(uE_MCall.met.id)) {
cd.met.add(**UE_Assert2CD**(uD_Assert, P, idObj)) }
4) foreach (UD_Assert uD_Assert : uE_MCall.met.pre) {
cd.met.add(**UE_Assert2CD**(uD_Assert, P, idObj)) }
5) cd.met.add(**UE_Block2CD**(uE_MCall.uE_Method.block, P, idObj, retVar))
6) foreach (UD_Assert uD_Assert : uE_MCall.met.post) {
cd.met.add(**UE_Assert2CD**(uD_Assert, P, idObj)) }
7) foreach (UD_Assert uD_Assert : InvStore.getInvs(uE_MCall.met.id)) {
cd.met.add(**UE_Assert2CD**(uD_Assert, P, idObj)) }
8) return cd

Figura 6.20: Transformación de las llamadas a métodos

La función UE_MCall2CD recibe cuatro parámetros. Los tres primeros son análogos a los vistos en las funciones de transformación ya explicadas. El cuarto parámetro, *retVar*, será recibido cuando exista una asignación que está esperando el resultado a devolver por el método. Este parámetro se recibe y es reenviado a la función de transformación UE_Block2CD en el paso 5. Tal como se explicó en la función de transformación UE_Block2CD, dicho parámetro será a su vez reenviado para ser luego utilizado en las transformaciones de las UE de tipo retorno (UD_Return).

En la transformación UE_MCall2CD se transformarán a restricciones, los argumentos enviados como parámetros, los invariantes antes de la ejecución del método, la precondición, el bloque de sentencias asociado al método, la postcondición, y los invariantes después de ejecutar el método llamado. Los pasos a dar en la función de transformación UE_MCall2CD son:

- Paso 1. Creación del componente de tipo CD_MCall. Este componente es el que se retornará una vez completada la transformación.

- Paso 2. Transformación de los parámetros de entrada. Para realizar esta transformación se ha seguido el mismo procedimiento expuesto en la función de transformación UE_CCall2CD (transformación de llamadas a constructores, paso 5), generando un componente de tipo CD_Asig por cada pareja formada por un parámetro

Transformación: UE_Return2CD			
Parámetros de entrada	UE_Return: uE_Return Path: P IDObject: idObj Var: retVar	Tipo devuelto	CD_Asig
Pasos 1) return **Asignment2CD**(retVar, uE_Return.exp, SSA.newAB(uE_Return.id), P, idObj)			

Figura 6.21: Transformación de la sentencia de retorno

y un argumento de entrada.

- Paso 3. Transformación de los invariantes antes del bloque de sentencias asociado al método. Para recuperar los invariantes asociados a cada clase, se hará uso de la función *InvStore.getInvs*, que recibiendo el identificador de un método o constructor, devuelve el conjunto de invariantes asociados a la clase que contiene dicho método o constructor. Cada uno de los invariantes será transformado a restricciones reutilizando la función de transformación UD_Assert2CD.

- Paso 4. Transformación de la precondición. Para realizar esta transformación se ha seguido el mismo procedimiento expuesto en la función de transformación UE_CCall2CD (transformación de llamadas a constructores, paso 6).

- Paso 5. Transformación del bloque de sentencias del método. En este paso se hará uso de la función de transformación UE_Block2CD. Esta función devuelve un componente de tipo CD_Block. En la llamada a dicha función de transformación, se reenvía el parámetro recibido *retVar*. Esta variable procede de una asignación, tal como se explicó en la transformación UE_AsigStat2CD, y será reenviada hasta llegar a ser utilizada en la transformación UE_Return2CD (aplicable a las UE de tipo UE_Return).

- Pasos 6 y 7. Transformación de la postcondición e invariantes después del bloque de sentencias asociado al método. Para realizar estas transformaciones se utilizarán las mismas operaciones que se propusieron para tal fin, en la función de transformación UE_CCall2CD (transformación de llamadas a constructores, pasos 8 y 9).

- Paso 8. Retorno del componente de tipo CD_MCall que se creó en el paso 1.

Transformación de las sentencias de retorno

Las UE_Return se tratarán de manera análoga a las asignaciones. En concreto, la función de transformación UE_Return2CD, permite obtener las restricciones necesarias

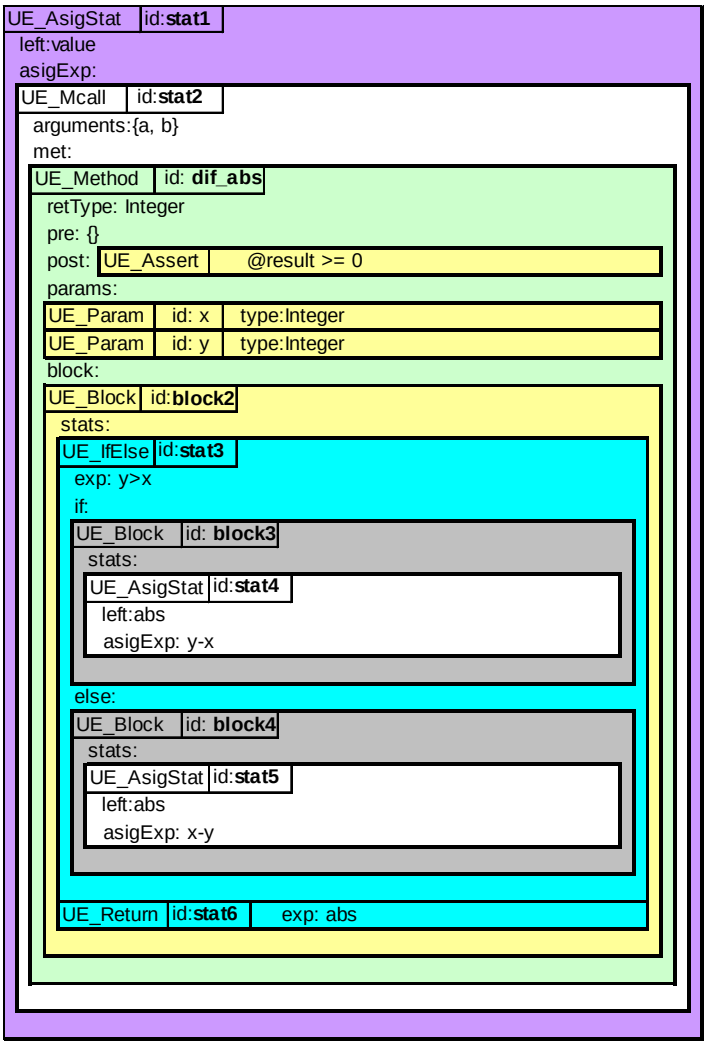


Figura 6.22: UE correspondiente a la asignación de la llamada al método *dif_abs*

para modelar el comportamiento asociado a la asignación del resultado de una llamada a un método, a un atributo de un objeto o a una variable. En esta transformación la única unidad ejecutada implicada es UE_Return:

Unidades Ejecutadas implicadas
`UE_Return ::= id:IDStatement exp:Exp`

La función UE_Return2CD (figura 6.21) recibe cuatro parámetros. El cuarto parámetro es la variable en la forma SSA sobre la que se asignará la expresión de retorno de la UE_Return. Esta variable, debe haber sido enviada por una asignación anterior,

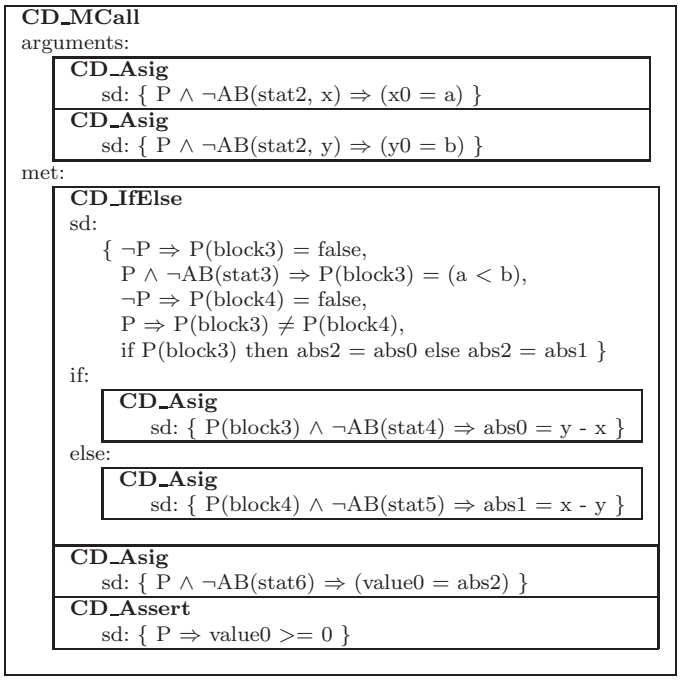


Figura 6.23: Resultado de la transformación a CD de la UE mostrada en la figura 6.22

en el caso de que se quiera recoger el resultado de la ejecución del método (tal como aparecía en la función de transformación de las asignaciones, UE_AsigStat2CD).

La sentencia de retorno será transformada reutilizando la función de transformación Asignment2CD vista anteriormente. Para crear el nuevo predicado AB asociado a la asignación, se utilizará el identificador *uE_Return.id* (de tipo IDStatement) que identifica a la sentencia de retorno de forma unívoca dentro del código fuente. Este predicado será el encargado de establecer si existe o no un defecto (comportamiento anormal o no) en la expresión retornada.

Ejemplo 6.11. A continuación se muestra el resultado de transformar la UE_AsigStat mostrada en la figura 6.22. Se trata de la asignación del resultado de la llamada al método *dif_abs* sobre una variable local. En concreto se trata de transformar la UE resultado de la ejecución de la sentencia: `value := dif_abs(a, b);`

El primer paso en la transformación es llamar a la función UE_AsigStat2CD encargada de transformar la asignación. En dicha función, a su vez se realizará una llamada a la función de transformación UE_MCall2CD, encargada de realizar la transformación de la llamada al método *dif_abs*. El resultado de esta transformación será el componente de tipo CD_MCall que se muestra en la figura 6.23. Tal como se puede ver en dicha figura, en el campo *arguments* se han incluido dos componentes de tipo CD_Asig. Estos dos componentes de tipo asignación permiten modelar, en forma de restricción, la asignación de los argumentos de entrada recibidos sobre los parámetros de entrada del método. En el campo *sd* del componente CD_MCall se ha añadido la restricción

correspondiente a la transformación de la postcondición del método.

Y, por último, en el campo *met*, se han incluido dos componentes como resultado de la transformación del bloque de sentencias asociado al cuerpo del método llamado. El primer componente del bloque es el resultado de la transformación de la sentencia selectiva, y es de tipo CD_IfElse; y el segundo es el resultado de la transformación de la sentencia de retorno, y es de tipo CD_Asig. Concretamente, este componente modela la asignación de la expresión a devolver por el método, sobre la variable *value*, ya que en dicha variable debe almacenarse el resultado de la ejecución del método, tal como aparece en la UE original.

6.4.7. Problema de diagnosis para defectos en el código fuente

Tal como se explicó en el capítulo 4, el problema de diagnosis está formado por la descripción del sistema (SD), el conjunto de sentencias del código fuente (STS), el conjunto de asertos que forman la especificación (SPEC), y un caso de prueba (TC). Las restricciones que forman la descripción del sistema están distribuidas en los componentes que forman la TCD, de tal forma que realizando un recorrido por la TCD es posible obtener el conjunto completo de restricciones que la forman. Cada TCD proviene de la transformación de la Traza de Unidades Ejecutadas (TUE) correspondiente a un caso de prueba donde se hayan detectado errores.

La diagnosis es una hipótesis que permite hacer consistente el Problema de Diagnosis. Más concretamente, la diagnosis $\mathcal{D}$ debe cumplir que SD $\cup$ TC $\cup$ $\{AB(a) \mid a \in \mathcal{D}\} \cup \{\neg$ AB(a) $\mid$ a $\in$ SPEC - $\mathcal{D}\}$, siendo $\mathcal{D} \subseteq$ SPEC. En este capítulo el interés se centra en los defectos semánticos de las sentencias que forman el código fuente, por tanto, debe cumplirse que $\mathcal{D} \subseteq$ STS.

Con respecto a los casos de prueba, se supondrá que el valor de las entradas y salidas especificadas será establecido a través de expresiones que cumplan la gramática propuesta para las expresiones del lenguaje. Además, será necesario transformar a la forma SSA las entradas y salidas especificadas en el caso de prueba. Para ello se reutilizarán las mismas funciones auxiliares que se utilizaron para mantener la forma SSA en las restricciones incluidas en la TCD.

En la transformación de las sentencias y asertos a restricciones se han utilizado dos tipos de predicados: el predicado P y el predicado AB. El predicado P(c) establece para cada sentencia o aserto c, si dicha sentencia o aserto pertenece a la traza correcta. El predicado AB(s) establece si la sentencia s tiene un comportamiento anormal o no, es decir, si contiene, o no, defectos.

Cada solución al problema de diagnosis proporciona los valores de los predicados P y AB. Del conjunto de sentencias, sólo se tendrán en cuenta aquellas que forman la traza correcta, es decir, aquellas cuyo predicado P ha tomado el valor verdadero. De todas las sentencias incluidas en la traza correcta, formarán parte de una diagnosis mínima sólo aquellas cuyo predicado AB tome el valor verdadero. Las diagnosis mínimas se proporcionarán al usuario de forma ordenada, de menor a mayor, por el número de

STS:	**Codigo fuente**:	**TC:**
	public dif_abs(int a, int b) {	Entradas: {a = 10, b = 2}
S_1	if (a < b) {	Salidas: {ret0 = 8}
S_2	max = a ;	
S_3	min = b ;	
	} else {	
S_4	max = b ;	
S_5	min = a ;	
	}	
S_6	return max - min ;	

SD:

Restricciones obtenidas del código fuente

$$\{ \neg P \Rightarrow P(block1) = false,$$
$$P \wedge \neg AB(S_1) \Rightarrow P(block1) = (a < b),$$
$$\neg P \Rightarrow P(block2) = false,$$
$$P \Rightarrow P(block1) \neq P(block2),$$
$$\text{if } P(block1) \text{ then } max2 = max0 \text{ else } max2 = max1,$$
$$\text{if } P(block1) \text{ then } min2 = min0 \text{ else } min2 = min1 \}$$
$$\{ P(block1) \wedge \neg AB(S_2) \Rightarrow max0 = a \}$$
$$\{ P(block1) \wedge \neg AB(S_3) \Rightarrow min0 = b \}$$
$$\{ P(block2) \wedge \neg AB(S_4) \Rightarrow max1 = b \}$$
$$\{ P(block2) \wedge \neg AB(S_5) \Rightarrow min1 = a \}$$
$$\{ P \wedge \neg AB(S_6) \Rightarrow (ret0 = max2 - min2) \}$$

Restricciones debidas al caso de prueba

$$a = 10 \wedge b = 2 \wedge @result = 8 \wedge P = true$$

F. Objetivo: $\text{Max}(N\ s_i : s_i \in \{S_1, S_2, S_3, S_4, S_5, S_6\} : \neg AB(s_i) = \text{cierto})$

Figura 6.24: Problema de diagnosis para el método *dif_abs*

sentencias que incluyan.

Los defectos que se podrán identificar se pueden englobar en dos grandes tipos: defectos en asignaciones, y defectos en las sentencias selectivas e iterativas.

- Si el predicado AB(s) está asociado a una sentencia selectiva o iterativa, entonces se trata de un defecto estructural. El defecto de diseño está en la expresión utilizada como condición en las sentencias selectiva o iterativa. En concreto, para el caso de prueba utilizado no se ha ejecutado el bloque adecuado de sentencias.

- Si el predicado AB(s) está asociado a una asignación inicial de un atributo o una variables local; o a un paso de argumentos a un método o constructor; o a una asignación de una variable local, parámetro de entrada o atributo de un objeto; o al retorno de un resultado por parte de un método; entonces se trata de un defecto en la asignación. El defecto de diseño se encuentra en la expresión asignada.

STS	AB	Traza correcta sobre el código fuente
		public dif_abs(int a, int b) {
S_1	$AB(S_1)$ = True	if (a < b) {
S_2	$AB(S_2)$ = False	max = a ;
S_3	$AB(S_3)$ = False	min = b ;
		} else {
		max = b ;
		min = a ;
		}
S_6	$AB(S_6)$ = False	return max - min ;
		}

Dominio final de las variables del problema de diagnosis:
a = 10, b = 2, max0 = 10, min0 = 2, max1 = libre, min1 = libre,
max2 = 10, min2 = 2, @result = 8

Figura 6.25: Traza y domino de las variables para la diagnosis mínima S_1

Para solucionar los errores encontrados, será necesario modificar las asignaciones y las condiciones de las sentencias selectivas donde se hayan identificado defectos. Es importante recordar que cada sentencia del código fuente es única, y aunque se repita varias veces en la traza ejecutada, sólo aparecerá una vez en la diagnosis mínima, ya que sólo debe modificarse una vez en el código fuente. Como se puede observar, no hemos hecho referencia a las sentencias de tipo iterativo (bucles), ya que como se ha explicado anteriormente (sección 6.3.6), este tipo de sentencias serán tratadas como sentencias selectivas anidadas.

Ejemplo 6.12. A continuación se muestra como ejemplo el código fuente correspondiente al método *dif_abs*. Como se puede comprobar, el resultado no es el esperado cuando se aplica el caso de prueba propuesto (se espera que el método devuelva el valor 8, y devuelve el valor -8), por lo que se procede a plantear el problema de diagnosis que aparece en la figura 6.24. Los elementos a diagnosticar son las sentencias que forman el código fuente. Las restricciones de la descripción del sistema se han obtenido como transformación de la traza seguida para el caso de prueba propuesto. Los predicados P(block1) y P(block2) hacen referencia a los predicados P asociados a los bloques de sentencias incluidos en la sentencia selectiva S_1.

El problema de diagnosis se solucionará tal como se explicó en el capítulo dedicado a la metodología (capítulo 4). Para este problema se obtendrían en total dos diagnosis mínimas de tipo simple (implican modificar una única sentencia), que son: S_1 o S_6.

En el primer caso, S_1 es una sentencia selectiva. Si existe un defecto, dicho defecto debe estar en la expresión utilizada para seleccionar el bloque de sentencias que debe

STS	AB	Traza correcta sobre el código fuente
		public dif_abs(int a, int b) {
S_1	$AB(S_1)$ = False	if (a < b) {
		max = a ;
		min = b ;
		} else {
S_4	$AB(S_2)$ = False	max = b ;
S_5	$AB(S_3)$ = False	min = a ;
		}
S_6	$AB(S_6)$ = True	return max - min ;
		}

Dominio final de las variables del problema de diagnosis:
a = 10, b = 2, max0 = libre, min0 = libre, max1 = 2, min1 = 10,
max2 = 2, min2 = 10, @result = 8

Figura 6.26: Traza y domino de las variables para la diagnosis mínima S_6

ejecutarse. En la figura 6.25 se muestran los valores de los predicados AB para las sentencias que pertenecen a la traza correcta (resaltado en amarillo), y los dominios finales para las variables del problema de diagnosis que se han obtenido como transformación de las variables utilizadas en el código fuente. En esta solución, la traza correcta viene dada por el valor de los predicados P, como el predicado P(block1) toma el valor verdadero, se deduce que de los dos bloques asociados a la sentencia selectiva, es el primero el que debe ejecutarse en la traza correcta. Es decir, la condición de la sentencia S_1 debe evaluarse a cierto, en lugar de a falso como ocurría en la traza real. De esta forma, se ejecutaría el primer bloque asociado a la sentencia selectiva.

Para solucionar el problema de diagnosis habría que modificar la sentencia S_1. En concreto se debe cambiar la condición de dicha sentencia para que devuelva justo el valor contrario al evaluarse para este caso de prueba. Si se observa dicha sentencia, aparece como condición que (a < b), cuando la expresión correcta debería ser (a > b). Realizando este cambio el problema estaría solucionado. Si además del caso de prueba propuesto se tienen otros casos de prueba, el cambio en el código fuente debería ser validado con el resto de casos de prueba. De esta forma se comprobaría si no entra en contradicción con otros casos de prueba establecidos, ya que los cambios introducidos en el código fuente, no deben entrar en contradicción con los casos de prueba disponibles.

En el segundo caso, S_6 es una sentencia de retorno. Si existe un defecto en ella se debe a que la expresión a devolver no es la correcta. De manera análoga a como se hizo en el caso anterior, en la figura 6.26 se muestran los valores de los predicados AB para las sentencias que pertenecen a la traza correcta (resaltado en amarillo), y

los dominios finales para las variables del problema de diagnosis que se han obtenido como transformación de las variables que aparecen en el código fuente. La solución obtenida establece que la traza correcta es igual a la traza real obtenida para el caso de prueba propuesto, es decir, se debe ejecutar el segundo bloque de sentencias asociado a la sentencia selectiva S_1.

En este caso, para solucionar el problema de diagnosis habría que modificar la sentencia S_6. En concreto se debe cambiar la expresión a devolver, de forma que genere el valor esperado, que es el valor 8, tal como establece el caso de prueba. Realizando este cambio el problema estaría solucionado. Pero antes de realizar el cambio, es necesario comprobar si es apropiado para el resto de casos de prueba que estén disponibles, es decir, comprobar si el cambio en el código fuente permite validar el resto de casos de prueba. Este cambio en el código no debe hacerse si provoca que otros casos de prueba no se cumplan.

Aunque en este ejemplo sólo aparece un caso de prueba, seguramente el cambio propuesto para la sentencia S_6 entraría en contradicción con otros casos de prueba. Tal como se verá en el siguiente capítulo, la selección de la diagnosis mínima más apropiada para solucionar el problema, puede mejorarse si se dispone de varios casos de prueba. La idea es proporcionar diagnosis mínimas que permitan validar todos los casos de prueba donde no se han obtenido los resultados correctos.

6.5. Mejora de los resultados utilizando asertos en el código fuente

Tal como se ha expuesto en este capítulo, la descripción del sistema se ha obtenido como transformación del código fuente y la especificación del sistema. La especificación consta de asertos, precondiciones, postcondiciones e invariantes. El objetivo al incluir la especificación es mejorar la precisión en la metodología de diagnosis, tal como se mostrará a continuación.

La detección de los defectos en un sistema software es posible gracias a la propagación de los fallos hasta llegar a un punto donde sea posible comparar el resultado real con el establecido como correcto por un caso de prueba, dando lugar a los errores. Los defectos que provocan fallos en el software, son detectables gracias a los errores. Si no existiese la propagación de los fallos hasta convertirse en errores, sería imposible comprobar si existen defectos, tal como se expuso en la introducción de esta tesis (apartado 1.1.1).

Al transformar la especificación contenida en los asertos, invariantes, precondiciones y postcondiciones, a restricciones de la descripción del sistema, es posible garantizar que las soluciones al problema de diagnosis satisfarán las restricciones obtenidas de la especificación. La información contenida en la especificación no sólo puede servir de cortafuegos en la propagación de los defectos cuando un sistema se ejecuta, sino que además, puede ser de utilidad en la metodología de diagnosis para mejorar la precisión

Código fuente	STS	SPEC	SD
`int x = a * c`	S_1		$P \wedge \neg AB(S_1) \Rightarrow (x = a * c)$
`int y = b * d`	S_2		$P \wedge \neg AB(S_2) \Rightarrow (y = b * d)$
`int z = c * e`	S_3		$P \wedge \neg AB(S_3) \Rightarrow (z = c * e)$
`assert: x * e = z * a`		A_1	$P \Rightarrow x * e = z * a$
`int f = x + y`	S_4		$P \wedge \neg AB(S_4) \Rightarrow (f = x + y)$
`int g = y * z`	S_5		$P \wedge \neg AB(S_5) \Rightarrow (g = y$ ***** $z)$

TC = {Entradas: {a = 3, b = 2, c = 2, d = 3, e = 3, P = true},
Salidas: {f = 12, g = 12}}

Figura 6.27: Problema de diagnosis

al determinar la diagnosis mínima.

En concreto, si la especificación es lo suficientemente restrictiva, será posible determinar en determinados puntos de la traza, cuáles son los valores correctos para ciertas variables, o que relación deberían guardar los valores almacenados en diferentes variables. De esta forma, será posible comprobar si las sentencias generan los resultados establecidos como correctos, sin tener que propagar dichos resultados hasta las variables cuyo valor correcto esté establecido en los casos de prueba.

Para comprobar las ventajas que el uso de la información almacenada en la especificación proporciona, a continuación se muestra un pequeño ejemplo.

Ejemplo 6.13. La figura 6.27 se muestra un código fuente de ejemplo y su correspondiente caso de prueba. Además de sentencias de asignación, el código fuente incluye un aserto: $x * e = z * a$. La traza seguida por el código fuente no proporciona los resultados correctos. Al usar el caso de prueba el resultado real obtenido para la variable f es 12 (igual al establecido en el caso de prueba), mientras que para la variable g el resultado es 36 (diferente al establecido en el caso de prueba).

Usando las funciones de transformación explicadas anteriormente, se obtendría la descripción del sistema mostrada a la derecha de la figura. Resolviendo el problema de diagnosis se obtendría que las diagnosis mínimas son $\{\{S_5\}, \{S_2, S_4\}\}$. Sin el aserto incorporado al código fuente, en total se obtendrían cuatro diagnosis mínimas, que corresponden con: $\{\{S_3\}, \{S_5\}, \{S_1, S_2\}, \{S_2, S_4\}\}$.

El aserto A_1 evita que la sentencia S_3 sea considerada como diagnosis mínima. Si en el ejemplo se elimina dicho aserto, la sentencia S_3 aparecía como diagnosis mínima, ya que si dicha sentencia establece el valor 2 en la variable z, entonces el resultado generado por el programa sería el correcto para la variable g. Si se incluye el aserto, el valor 2 para la variable z no estaría permitido mientras la variable x tuviera el valor 6. De manera análoga, el aserto introducido evita que la pareja $\{S_1, S_2\}$ sea considerada como diagnosis mínima. En general, el aserto A_1 permite saber si existen errores en los valores de las variables x y z, que son asignadas respectivamente por las sentencias S_1

Modelo del sistema

SD:
$P \wedge \neg AB(S_1) \Rightarrow (x = a * c)$
$P \wedge \neg AB(S_2) \Rightarrow (y = b * d)$
$P \wedge \neg AB(S_3) \Rightarrow (z = c * e)$
$P \wedge (x * e = z * a)$
$P \wedge \neg AB(S_4) \Rightarrow (f = x + y)$
$P \wedge \neg AB(S_5) \Rightarrow (g = y * z)$
STS:
$\{S_1, S_2, S_3, S_4, S_5\}$
SPEC:
$\{A_1\}$
TC:
Entradas: {a = 3, b = 2, c = 2,
d = 3, e = 3, P = true }
Salidas: {f = 12, g = 12}

Soluciones del problema Max-CSP

Diag. mínima: $\{S_5\}$
cierto $\wedge$ cierto $\Rightarrow (6 = 3 * 2)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge (6 * 3 = 6 * 3)$
cierto $\wedge$ cierto $\Rightarrow (12 = 6 + 6)$
cierto $\wedge$ falso $\Rightarrow (12 = 6 * 6)$

Diag. mínima: $\{S_2, S_4\}$
cierto $\wedge$ cierto $\Rightarrow (6 = 3 * 2)$
cierto $\wedge$ falso $\Rightarrow (2 = 2 * 3)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge (6 * 3 = 6 * 3)$
cierto $\wedge$ falso $\Rightarrow (12 = 6 + 2)$
cierto $\wedge$ cierto $\Rightarrow (12 = 2 * 6)$

Figura 6.28: Resolución del problema Max-CSP

y S_3.

6.5.1. Uso del invariante de los bucles

En algunos lenguajes de programación, es posible añadir asertos como invariantes de bucles. El invariante es a la vez precondición y postcondición del cuerpo del bucle en cada iteración. Este elemento de la especificación se debe satisfacer antes de la primera iteración, después de cada una de ellas, y en particular, después de la última, es decir, a la terminación del bucle.

En el invariante se describen los estados por los que atraviesa el cómputo realizado por el bucle, observados justo antes de evaluar la condición de repetición. Es decir, puede entenderse como la unión de todos los estados por los que un bucle debe pasar. Las instrucciones del bucle modifican el estado de las variables pero no las relaciones entre las variables, que es justo la información que el invariante almacena.

El invariante establece condiciones que deben cumplir las variables después de cada iteración. La idea es sacar partido de la información contenida en los invariantes para que la metodología de diagnosis sea más precisa. Si las condiciones impuestas por los invariantes de los bucles son transformadas a restricciones y añadidas a la descripción del sistema, tras cada iteración del bucle se garantiza que el problema de diagnosis sólo propondrá como diagnosis mínimas aquellas que no incumplan los invariantes de los bucles.

Para que el diagnóstico sea lo más preciso posible, es necesario que los invariantes

```
Pre: N >= 0
int r = 0;
int s = 1;
int t = 1;
while (s <= N){
  r = r + 2;
  s = s + t + 2;
  t = t + 2;
  assert: s = (r + 1)^2;
  assert: t = 2 * r + 1;
}
Post: r^2 <= N < (r + 1)^2
```

Caso de prueba {N = 41, r = 6}

Iteración	r	s	t	inv
Previo bucle	0	1	1	cierto
Iteración 1	2	4	3	cierto
Iteración 2	4	9	5	cierto
Iteración 3	6	16	7	cierto
Iteración 4	8	25	9	cierto
Iteración 5	10	36	11	cierto

Figura 6.29: Algoritmo cálculo raíz cuadrada de un número entero

sean lo suficientemente fuertes como para poder restringir al máximo las condiciones que deben cumplir las iteraciones de los bucles. Por ejemplo, el invariante cierto siempre se cumple, pero no ofrecería ninguna información al proceso de diagnóstico.

Exigir que se cumpla el invariante no garantiza que un bucle termine. En un caso extremo, se podría proponer como cuerpo del bucle la instrucción nula o nada, que mantendría la invariabilidad de cualquier invariante, y sin embargo, haría que el bucle iterase indefinidamente. En la metodología de diagnosis propuesta, no se contempla este tipo de defectos en el código (bucle infinito).

Para utilizar los invariantes de los bucles en el problema de diagnosis, simplemente por cada iteración que se realice del bucle dentro de la traza ejecutada, se obtendría una restricción asociada al invariante del bucle, utilizando para ello las funciones de transformación explicadas anteriormente para los asertos. De esta forma, en la descripción del sistema aparecerían las restricciones correspondientes a los invariantes de los bucles.

Ejemplo 6.14. En la figura 6.29 se muestra el algoritmo encargado de calcular la raíz cuadrada de un número entero N. En cada una de las iteraciones debe cumplirse el invariante del bucle. Si se recibe como entrada el número N = 41, el número de iteraciones sería 5. En el cuadro que aparece a la derecha del código fuente, se muestran los valores asociados a las variables en cada iteración. En todas las iteraciones se cumplen los invariantes. En el capítulo dedicado de pruebas, apartado 7.1.3, se mostrará como el uso de los invariantes mejora la precisión en la determinación de la diagnosis mínima para este ejemplo.

6.6. Conclusiones

Tal como se explicó en la introducción de este documento, el segundo objetivo de esta tesis es el diagnóstico del código fuente. Concretamente, el objetivo es detectar y localizar defectos de tipo semántico en las sentencias que forman el código fuente.

La propuesta presentada en este capítulo se basa en la transformación a restricciones de las sentencias y asertos que forman el sistema software. Para cada caso de prueba donde los resultados obtenidos han sido diferentes a los correctos, se planteará un problema de diagnosis, modelado como un problema Max-CSP. La resolución del problema de diagnosis proporcionará las diagnosis mínimas. Cada diagnosis mínima establece qué sentencias deben cambiar para poder alcanzar los resultados establecidos en el caso de prueba, sin violar los asertos que forman la especificación.

La metodología es capaz de identificar defectos estructurales (en sentencias selectivas e iterativas), haciendo uso del predicado P, que a su vez proporciona, de forma implícita, la traza correcta (conjunto de sentencias y asertos que deben ejecutarse para alcanzar el resultado correcto). Además, la metodología permite identificar defectos en las asignaciones iniciales de las declaraciones de atributos, y variables locales; en el paso de argumentos a los métodos y constructores; en la asignación de variables locales, parámetros de entrada y atributos de objetos; y en el retorno de un resultado por parte de un método.

Al localizar los defectos de forma automática, se evita tener que revisar de forma manual todas las sentencias que forman la traza seguida, ya que propone de forma ordenada (por número de sentencias a modificar), cuáles son las sentencias, que una vez modificadas, permiten alcanzar los resultados establecidos como correctos en los casos de prueba. Sólo las sentencias propuestas como diagnosis mínima deben ser revisadas, con el consiguiente ahorro en tiempo y recursos.

Capítulo 7

Experimentos realizados y mejora de la precisión en la identificación de los defectos

En este último capítulo se persiguen dos objetivos principales. Por una parte presentar las pruebas que se han realizado para comprobar la eficiencia y eficacia de la metodología propuesta en el diagnóstico de programas. Y por otra parte, ampliar la metodología propuesta para mejorar la eficacia a la hora de reparar los defectos. Para ello se propondrán dos mejoras. La primera mejora se basará en analizar las diagnosis mínimas obtenidas en el caso de disponer de varios casos de prueba donde se hayan detectado errores. Y la segunda se basará en el análisis de la propagación de los defectos, y sus implicaciones en la reparación eficiente de los defectos.

7.1. Pruebas sobre la metodología aplicada al código fuente

Tal como se expuso en la introducción de la metodología, los programas a diagnosticar deben cumplir, entre otras, las siguientes características:

- Deben cumplir la gramática presentada en el capítulo 6 (figura 6.2).
- El código fuente no debe tener errores sintácticos.
- La ejecución del código fuente, para el caso de prueba propuesto, debe acabar en un tiempo finito, y deber haber existido un único hilo de ejecución.

La diagnosis del software se plantea en esta tesis como un complemento a los casos de prueba, en concreto a las pruebas unitarias (sección 2.4.1). El valor añadido que ofrece la diagnosis del software es la identificación automática de los defectos. Para la

aplicación de la metodología se supondrá que el código fuente es casi correcto. Para comprobar la eficiencia y eficacia de la metodología como complemento a las pruebas unitarias, se propondrán diferentes ejemplos con sus correspondientes casos de prueba. En dichos programas se introducirán defectos, que impedirán alcanzar los resultados correctos para los casos de prueba propuestos. Estos defectos serán modificaciones en el código fuente. La idea es comprobar si la diagnosis del software permite:

- Determinar en un tiempo finito en qué sentencias pueden estar localizados los defectos que han provocado los errores detectados. El tiempo necesario para la determinación de las diagnosis mínimas, será la medida que determinará la eficiencia de la metodología propuesta.

- Reducir el número de sentencias que deben ser revisadas en un porcentaje considerable con respecto al número de sentencias del código fuente; y sobre todo, que los defectos reales estén incluidos en el grupo de sentencias que la metodología proponga revisar. Es decir, evitar en lo posible, los falsos positivos (sentencias correctas, pero que la metodología propone revisar), y los falsos negativos (sentencias con defectos, pero que la metodología no propone revisar). El porcentaje de sentencias que deben ser revisadas, será la medida del la eficacia de la metodología propuesta.

Las técnicas de mutación, tal como se explicó en el apartado 2.4.4, permiten medir la calidad de un conjunto de pruebas. En nuestro caso, el objetivo al introducir un defecto es comprobar si dicho defecto está incluido en alguna de las diagnosis mínimas obtenidas por la metodología propuesta, por tanto el objetivo es comprobar la calidad de la metodología propuesta. En los siguientes apartados se mostrarán los resultados obtenidos para los ejemplos propuestos. Tras analizar los resultados obtenidos se propondrán posibles ampliaciones para la metodología. Estas ampliaciones serán expuestas en las dos últimas secciones de este capítulo.

7.1.1. Implementación

Para poder realizar las pruebas se ha procedido a la implementación de la metodología. Se ha optado por usar un entorno de programación con restricciones que implemente los tipos utilizados en la gramática soportada por la metodología (sección 4.4.1). En concreto, el entorno utilizado es ILOG JSolver^{TM}, que para los clases definidas por el usuario incluye el tipo *IlcAny* capaz de almacenar un conjunto de objetos como dominio para una variable. Las pruebas de la metodología se han realizado en un ordenador con procesador Pentium Dual CPU 1.73 GHz y memoria RAM de 2GB. Los tiempos que se mostrarán corresponden a las pruebas realizadas en dicha máquina.

Dado que no se ha encontrado un conjunto de ejemplos y pruebas que sean utilizados de forma general en los trabajos relacionados con la tesis, se ha recopilado un conjunto de programas y casos de prueba para tratar de cubrir el más amplio abanico

Nombre	CT	Defecto	STS	LCT	Tiempo(ms)	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
ToyProgram	1	S_5	5	5	0,01	2	Sí	40 %

Tabla 7.1: Resultados obtenidos para el ejemplo ToyProgram

de posibilidades. En el conjunto de experimentos propuesto, existe al menos un ejemplo por cada uno de los elementos que forman la gramática presentada en el capítulo 6, y que pueden contener defectos.

En algunos de los ejemplos recopilados se utilizan arrays para almacenar información. Para que la metodología propuesta sea capaz de utilizar arrays, son necesarios dos requisitos:

- Que el entorno de programación con restricciones permita trabajar con arrays. En este caso, el entorno seleccionado, JSolver, sí permite utilizar arrays de todos los tipos de variables necesarios.
- Que se pueda aplicar la forma SSA a las variables de tipo array. Como no es el objetivo de esta tesis obtener la forma SSA para las variables de tipo array, y su uso no aportaría ninguna mejora en la eficiencia o eficacia de la metodología de diagnosis, pero implicaría extender más este trabajo, se ha optado por dejar la ampliación de la forma SSA a los arrays como trabajo futuro. En su lugar, en los ejemplos que se propondrán en las pruebas propuestas, se permitirán todas las lecturas necesarias sobre los elementos que componen un array, pero sólo se permitirá una escritura en cada uno de los elementos que forman dicho array. De esta forma será posible utilizar arrays, pero no será necesario realizar su transformación a la forma SSA.

7.1.2. Medidas y consideraciones previas

Los programas y los casos de prueba utilizados aparecen en el apéndice B. A modo de ejemplo, en la tabla 7.1 se muestran los resultados que se han obtenido para el programa ToyProgram. Para cada experimento realizado sobre un programa se recogerán los siguientes datos y medidas:

- Nombre del programa sobre el que se aplica la metodología de diagnosis.
- CT. Número del caso de prueba utilizado.
- Defecto. Sentencia que contiene el defecto introducido.
- STS. Número de sentencias que forman el código fuente.
- LCT. Número de sentencias ejecutadas en la traza seguida para el caso de prueba utilizado.

- Tiempo (en milisegundos) necesario para el cálculo de las diagnosis mínimas.
- $\mathcal{D}$min. Número de sentencias implicadas en las diagnosis mínimas obtenidas. Es decir, es el número de sentencias que pueden contener defectos y que por tanto deben ser revisadas.
- DC. Defecto cubierto, es decir, indica si el defecto introducido está incluido en alguna de las diagnosis mínimas obtenidas.
- $\mathcal{D}$min/STS. Porcentaje de sentencias del código fuente que aparecen en alguna diagnosis mínima. Este porcentaje indica qué número máximo de sentencias tendrán que revisarse, pues sólo ellas pueden contener defectos.

Los defectos introducidos serán modificaciones sobre sentencias del código fuente original, y todo el código fuente será correcto salvo una sentencia modificada. De esta forma se cumplirá el supuesto de que el programa a diagnosticar es casi correcto.

La metodología propuesta permite obtener todas las diagnosis mínimas, tal como se mostró en los capítulos anteriores. Para el experimento propuesto, en la tabla 7.1, se han obtenido las siguientes diagnosis mínimas: S_3, S_5, $\{S_2, S_4\}$, $\{S_1, S_2\}$. En total son 4 diagnosis mínimas. Como se puede observar, prácticamente todas las sentencias están implicadas en al menos una diagnosis mínima. Por tanto, en el peor de los casos, si se comprobaran todas las diagnosis mínimas, sería necesario revisar todas las sentencias del código fuente.

En la comunidad de Diagnosis Basada en Modelos, las diagnosis mínimas son ordenadas siguiendo el principio de parsimonia, ofreciendo normalmente como resultado sólo las diagnosis simples, o las dobles si no existen diagnosis simples. Este mismo criterio será el que se siga al mostrar los resultados obtenidos en los ejemplos propuestos en este capítulo.

El principio de la parsimonia supone que la explicación más simple y suficiente es la más probable, pero no tiene que ser necesariamente la verdadera. En ciertas ocasiones, la opción más compleja puede ser la correcta. Por ejemplo, para el caso del ToyProgram, las diagnosis mínimas que se propondrían serían S_3 y S_5. Si el defecto se encuentra en una única sentencia, obligatoriamente dicha sentencia debe ser S_3 o S_5. De esta forma sólo sería necesario revisar 2 de las 5 sentencias que forman el código fuente. Si el defecto no estuviera en dichas sentencias, se pasaría a revisar las diagnosis mínimas que incluyan dos sentencias. Las diagnosis mínimas de tipo doble eran: $\{S_1, S_2\}$ y $\{S_2, S_4\}$. Al repasar estas últimas sentencias se habría repasado todo el código fuente.

Para todos los ejemplos de este capítulo se seguirá el principio de la parsimonia a la hora de mostrar el número de diagnosis mínimas, ofreciendo como resultado sólo las diagnosis simples, o las dobles si no existen diagnosis simples, o las triples si no existen dobles, y así sucesivamente. De esta forma, cuando se calcule el porcentaje de sentencias que estén implicadas en dichas diagnosis mínimas, dicho porcentaje será más cercano al caso real, ya que si se tuvieran en cuenta todas las diagnosis mínimas, el porcentaje

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
IndexOf	1	S_4	8	44	0,01	2	Sí	25 %

Tabla 7.2: Resultados obtenidos para el ejemplo IndexOf

que se obtendría sería en muchos casos cercano al 100 %, ya que para un mismo caso de prueba pueden existir muchas diagnosis mínimas que impliquen modificar a la vez varias sentencias, pero que tendrían una escasa probabilidad de ser reales teniendo en cuenta que el código fuente es casi correcto.

7.1.3. Defectos en bucles

Para realizar los primeros experimentos sobre sentencias iterativas o bucles, se ha utilizado el ejemplo denominado IndexOf, que incluye sólo un único bucle. El programa (apéndice B.2) permite obtener la posición en la que se encuentra un elemento dentro del intervalo [*start*, *end*] de un array *a*.

La tabla 7.2 muestra los resultados obtenidos para dicho ejemplo. En este caso, se ha introducido un defecto en la sentencia S_4. La metodología ha obtenido dos diagnosis mínimas simples: S_4 y S_6. El defecto se introdujo en la sentencia S_4, por tanto ha conseguido aislar el defecto, aunque también ofrece como alternativa modificar la sentencia S_6.

En este ejemplo el número de sentencias del código fuente difiere del número de sentencias de la traza ejecutada. Esta diferencia en el número de sentencias se debe al bucle. Para el caso de prueba propuesto se han realizado 10 iteraciones. Aunque la sentencia S_4 se ejecute 10 veces, sólo debe modificarse en el código original una vez, por tanto la metodología considera dicho defecto como simple, aunque en la traza dicha sentencia aparezca repetidas veces.

Como se ha podido comprobar, la metodología es capaz de detectar defectos en la condición que debe ser evaluada en cada iteración. En concreto, puede detectar que sobran iteraciones. Cuando se detecta un error en una de las condiciones de una iteración, implica que dicha iteración y las siguientes, no deberían realizarse para obtener el resultado correcto. Pero no es posible diagnosticar en qué caso un bucle debería realizar un número mayor de iteraciones. Es decir, la metodología puede identificar defectos en la condición del bucle que provoquen más iteraciones de las debidas, pero no puede identificar defectos en la condición del bucle cuando lo que se provoca es que se realicen menos iteraciones de las debidas.

La razón está en que la metodología, tal como se explicó en el capítulo anterior, transforma los bucles a sentencias selectivas anidadas, generando tantas sentencias anidadas como iteraciones se ejecuten en el caso de prueba. Por tanto, el conjunto de restricciones del modelo no permiten simular un número mayor de iteraciones de las que se produjeron al ejecutar el caso de prueba, y en consecuencia, la metodología no puede comprobar si realizando más iteraciones se podría alcanzar el resultado correcto

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
RaizCuadrada	1	S_2	7	16	0,01	7	Sí	100 %
RaizCuadrada+Inv	1	S_2	7	16	0,01	2	Sí	28,6 %

Tabla 7.3: Resultados obtenidos para el ejemplo RaizCuadrada

establecido en el caso de prueba.

De forma análoga, cuando en el programa aparecen llamadas recursivas, la metodología permite identificar defectos en la condición que permita la llamada recursiva cuando el defecto provoca más llamadas de las debidas, pero no puede localizar defectos en la condición de la llamada recursiva cuando dicho defecto provoca que se realicen menos llamadas de las debidas.

Invariantes de bucles

Para el caso de las sentencias iterativas o bucles, la metodología puede ofrecer mejores resultados si se definen invariantes que deban cumplirse en cada iteración. En la tabla 7.3 se muestran los resultados obtenidos para el programa denominado RaizCuadrada, sin utilizar invariantes (resultados mostrados en la primera fila), y utilizando invariantes (resultados mostrados en la segunda fila).

El código fuente del programa puede verse en el apéndice B.3. Consta de varias asignaciones iniciales y un bucle donde se han añadido dos invariantes. El objetivo del programa es calcular la raíz cuadrada entera del número entero recibido como parámetro de entrada.

Tal como se puede observar en la tabla 7.3, los resultados de la metodología son diferentes, aunque se utilice el mismo caso de prueba y el mismo defecto en ambos experimentos. Esta diferencia de resultados se debe a la ventaja proporcionada por los invariantes:

- Si no se incluyen invariantes. Como se puede observar el número de diagnosis mínimas es 7. Es decir, todas las sentencias del código fuente serían diagnosis mínimas simples. Por tanto no se reduciría el número de sentencias a revisar, pues habría que revisar el 100 % de las sentencias.

- Si se incluyen invariantes. Los resultados son mejores, ya que la metodología permite incorporar dichos invariantes en forma de restricciones. En el programa de ejemplo los invariantes permiten establecer en cada iteración cuál es la relación que deben guardar las variables r, s y t. Utilizando esta información, la metodología puede descartar que las sentencias de asignación que forman el bucle deban modificarse, pues puede comprobar que modificaciones en las asignaciones de valores a las variables r, s y t que permitirían cumplir el caso de prueba, no permitirían validar los invariantes, y por tanto no podrían ser tenidas en cuenta como posibles diagnosis. Como se puede observar en los resultados mostrados,

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
Polinomio	1	S_6	7	16	0,01	4	Sí	57 %

Tabla 7.4: Resultados obtenidos para el ejemplo Polinomio

el número de diagnosis mínimas ha pasado de 7 a 2 utilizando los invariantes. Las sentencias que forman las dos diagnosis mínimas son S_2 y S_3, siendo S_2 la sentencia donde se introdujo el defecto.

7.1.4. Variables reales

La metodología propuesta permite también el uso de variables de tipo real, siempre y cuando el entorno de programación con restricciones permita definir variables de dicho tipo, como ha sido el caso en los experimentos realizados. El problema puede surgir cuando los tipos reales utilizados en el código fuente tengan una precisión mayor (número de decimales) al tipo equivalente en el entorno de programación con restricciones. Si se da este caso, en el modelo basado en restricciones se realizarían redondeos al propagar las restricciones, que podrían influir en el resultado generado por la metodología de diagnosis. Para las variables de tipo real es necesario establecer un margen a partir del cual dos números se considerarán iguales, es decir, una precisión a partir de la cuál debe aplicarse el redondeo. El redondeo es importante en las restricciones que hagan comparaciones entre números reales, ya que por ejemplo, los números 1.996 y 2.004 se considerarán iguales si la precisión son dos decimales, pues ambos tomarán el valor redondeado a 2.00.

Para que el modelo basado en restricciones produzca el mismo comportamiento que el código fuente original, es necesario estudiar los valores que las variables reales puedan tomar, y fijar la precisión de las variables en el modelo basado en restricciones de tal forma que todos los valores que las variables del código fuente puedan tomar, estén dentro del dominio de las variables del modelo basado en restricciones. Así por ejemplo, si en el caso de prueba aparece una variable que toma un valor real con 3 decimales, la variable definida en el modelo basado en restricciones debe tener una precisión de al menos 3 decimales. De otra forma se podrían producir redondeos que pueden afectar a la determinación de las diagnosis mínimas. Si no es posible tener una precisión de al menos 3 decimales, los valores deberían expresarse en forma de intervalos. Por ejemplo, si se espera que una variable tome el valor 1.934, y la precisión que se puede aplicar es 2 decimales, el valor se adaptaría al intervalo [1.93, 1.94].

Ante este problema se puede actuar de dos formas:

- O bien establecer la precisión en decimales al máximo permitido por el entorno de programación con restricciones, y adaptar todos los valores del caso de prueba a intervalos redondeados a la precisión máxima.
- O bien realizar un estudio más profundo del caso de prueba y del código fuente,

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
CuentaBancaria	1	S_{15}	18	18	0,01	2	Sí	11,1 %
CuentaBancaria	2	S_{13}	25	25	0,01	1	Sí	4 %
CuentaBancaria	3	S_{20}	25	25	0,01	1	Sí	4 %

Tabla 7.5: Resultados obtenidos para el ejemplo CuentaBancaria

para establecer qué precisión debe ser la adecuada para cada variable que influya en los resultados establecidos en el caso de prueba.

La primera opción en principio implicaría un mayor gasto en recursos, y seguramente más tiempo en el cálculo de las diagnosis mínimas. La segunda opción requiere un estudio previo más profundo, pero permite un mayor aprovechamiento de los recursos.

En cualquiera de los dos casos, si la precisión necesaria es demasiado elevada para el entorno de programación con restricciones, la metodología podría determinar menos posibles defectos de los que realmente existen, debido a los redondeos realizados por el entorno de programación con restricciones. Para intentar evitar los efectos de dichos redondeos sería necesario utilizar intervalos en lugar de valores absolutos, en aquellas variables que almacenen valores donde se haya realizado un redondeo. De esta forma, la metodología podría determinar que existen más posibles defectos de los que realmente existen, pero no menos, tal como podría ocurrir si no se utilizan los intervalos.

En las pruebas realizadas en este trabajo se ha aplicado la primera opción, ya que la segunda opción requiere un estudio más profundo y se escapa de los objetivos iniciales de este trabajo. En la tabla 7.4 aparecen los resultados que se han obtenido para el ejemplo Polinomio (apéndice B.4). Para el caso de prueba propuesto, en las variables se ha establecido una precisión de 6 decimales, lo que permite garantizar que el comportamiento del modelo basado en restricciones no tendrá redondeos con respecto al establecido en el programa original. En total han sido 4 diagnosis mínimas, que corresponden con las sentencias S_3, S_4, S_6, S_7. El defecto se introdujo en la sentencia S_6.

7.1.5. Defectos en las expresiones enviadas a métodos

La metodología propuesta es capaz de identificar defectos en las expresiones que son enviadas como parámetros de entrada a métodos. Para comprobarlo, se ha utilizado como ejemplo el programa CuentaBancaria, que incluye varias llamadas. La clase CuentaBancaria dispone de varios métodos (apéndice B.5), y permite modelar las operaciones más típicas de una cuenta corriente, como por ejemplo realizar ingresos o retirar efectivo.

La tabla 7.5 muestra los resultados obtenidos para dicho ejemplo. Se han utilizado tres casos de prueba, pero en este apartado sólo es interesante el primero de ellos, los otros dos serán útiles en el siguiente sub-apartado. Para el primer caso de prueba se ha introducido un defecto en la sentencia S_{15}, en concreto se ha sustituido la expresión

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
SumaSubSecMaxima	1	S_5,S_{14}	20	44	0,04	10	Si	50 %

Tabla 7.6: Resultados obtenidos para otros ejemplos

enviada como parámetro de entrada. La metodología propuesta ha obtenido dos diagnosis mínimas simples: S_{15} y S_{25}. La metodología ofrece la sentencia defectuosa como una de las diagnosis mínimas posibles. Por tanto ha conseguido aislar el defecto, aunque también ofrece como alternativa modificar la sentencia S_{25}, ya que en dicha sentencia se utiliza el parámetro recibido como entrada en una asignación, y modificando dicha asignación también se podrían conseguir los resultados esperados en el caso de prueba.

7.1.6. Defectos en las sentencias selectivas

Continuando con el mismo ejemplo denominado CuentaBancaria, a continuación se va a comprobar la eficacia de la metodología en la localización de defectos en las sentencias selectivas. En concreto, la metodología propuesta es capaz de identificar defectos en las expresiones utilizadas como condiciones en las sentencias selectivas. Para comprobarlo se han realizado dos experimentos, introduciendo en cada uno de ellos un defecto en cada una de las dos sentencias selectivas que existen en el código fuente del ejemplo. La tabla 7.5 muestra, en las dos últimas filas, los resultados obtenidos para estos dos experimentos:

- En el primer experimento se ha introducido un defecto en la sentencia S_{13}. En concreto se ha sustituido la expresión utilizada como condición. La metodología propuesta ha obtenido una única diagnosis mínima simple: S_{13}, que coincide con la sentencia defectuosa. Por tanto se ha conseguido aislar el defecto.

- En el segundo experimento se ha introducido un defecto en la sentencia S_{20}. En este caso también se ha sustituido la expresión utilizada como condición. La metodología propuesta ha obtenido como única diagnosis mínima simple la sentencia S_{13}, que coincide con la sentencia defectuosa. Por tanto, en este experimento también se ha conseguido aislar el defecto.

7.1.7. Defectos multiples

Para comprobar la metodología sobre defectos múltiples, en el programa SumaSubSecMaxima, que aparece en el apéndice B.11, se han introducido dos defectos a la vez. En la tabla 7.6 se muestran los resultados obtenidos para dicho ejemplo.

En los resultados obtenidos, la metodología no ha propuesto ninguna diagnosis mínima simple, por tanto, garantiza que no es posible solucionar los errores detectados modificando únicamente una sentencia. Esto representa una ventaja frente a otras metodologías (como por ejemplo las técnicas de Slicing) que tratarían de explicar los errores detectados con defectos simples.

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
BusqBinaria	1	S_{15}	11	30	0,01	2	Sí	18,18 %
MochilaVZ	1	S_8	8	21	0,01	1	Sí	12,5 %
MonedasPD	1	S_{21}	25	324	0,01	4	Sí	16 %
NTree	1	S_{38}	49	131	0,01	4	Sí	8,16 %

Tabla 7.7: Resultados obtenidos para otros ejemplos

Se han obtenido diferentes diagnosis dobles, entre ellas la pareja de sentencias donde se han introducido los dos defectos, S_5 y S_{14}. En estás diagnosis dobles están implicadas un total de 10 sentencias, lo que representa un 50 % del código del ejemplo propuesto. A medida que el número de defectos introducidos aumenta, el número de posibles diagnosis también aumenta, y en general también el número de sentencias que pueden contener defectos.

7.1.8. Otros ejemplos

Para completar los experimentos, en la tabla 7.7 se muestran los resultados obtenidos en otros ejemplos:

- Problema de la búsqueda binaria (apéndice B.8). El programa propuesto realiza la búsqueda binaria de un valor en un array ordenado. En este caso se ha introducido un defecto en la sentencia S_6, encargada de almacenar la posición del elemento buscado. La metodología propone dos diagnosis mínimas simples, las sentencias S_2 y S_6.

- Problema de la mochila (apéndice B.7). El programa propuesto resuelve el problema de la mochila utilizando la técnica voraz. El defecto introducido ha sido en la sentencia S_8, encargada de guardar el peso máximo que queda disponible en la mochila. La metodología propone como única diagnosis mínima simple dicha sentencia S_8.

- Problema de las monedas (apéndice B.10). El programa propuesto resuelve el problema de las monedas utilizando la técnica de programación dinámica. En este caso se ha introducido un defecto en la sentencia S_{21}, encargada de almacenar el número de monedas cuando la opción elegida es no utilizar un tipo concreto de moneda. La metodología propone cuatro diagnosis mínimas simples, las sentencias S_4, S_{11}, S_{21} y S_{25}.

- Problema con el tipo árbol n-ario (apéndice B.9). La clase NTree permite guardar una serie de números enteros en una estructura de datos de tipo de árbol. El caso de prueba propuesto permite ejecutar el método *incMax* de la clase Prueba, que recibe un objeto de tipo NTree. En este caso se ha introducido un defecto en la sentencia S_{38}, encargada de aumentar el valor del elemento máximo. La

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
IndexOf	1	S_4	8	44	0,01	2	Sí	25 %
RaizCuadrada+Inv	1	S_2	7	16	0,01	2	Sí	28,6 %
Polinomio	1	S_6	7	16	0,01	4	Sí	57 %
SumaSubSecMaxima	1	S_5,S_{14}	20	44	0,04	10	Si	50 %
CuentaBancaria	1	S_{15}	18	18	0,01	2	Sí	11,1 %
CuentaBancaria	2	S_{13}	25	25	0,01	1	Sí	4 %
CuentaBancaria	3	S_{20}	25	25	0,01	1	Sí	4 %
BusqBinaria	1	S_{15}	11	30	0,01	2	Sí	18,18 %
MochilaVZ	1	S_8	8	21	0,01	1	Sí	12,5 %
MonedasPD	1	S_{21}	25	324	0,01	4	Sí	16 %
NTree	1	S_{38}	49	131	0,01	4	Sí	8,16 %

Tabla 7.8: Resultados obtenidos para los casos de prueba propuestos

metodología propone como diagnosis mínima modificar la sentencia S_{38} que es precisamente la que se ha modificado.

7.1.9. Conclusiones sobre los experimentos

La tabla 7.8 muestra los resultados obtenidos en los experimentos que se han mostrado anteriormente. A continuación vamos a analizar los resultados desde dos puntos de vista, la eficiencia y la eficacia.

- Eficiencia. El tiempo necesario para la determinación de las diagnosis mínimas ha sido muy parecido en todos los experimentos, y razonable desde el punto de vista práctico. Resolver un sistema de satisfacción con restricciones es un problema exponencial en el peor de los casos. A groso modo, es necesario comprobar las 2^n diagnosis posibles, donde n es el número de sentencias que pueden contener defectos.

  En los experimentos realizados sólo se han obtenido diagnosis que fuesen mínimas. Primero se ha parado la búsqueda una vez obtenidas todas las diagnosis mínimas simples. En caso de no existir diagnosis simples automáticamente se pasa a comprobar si los errores se pueden explicar con diagnosis mínimas dobles, y así sucesivamente hasta parar la búsqueda cuando las diagnosis mínimas permiten explicar los errores. De esta forma, aunque en teoría el número de diagnosis posibles es 2^n, en la práctica el espacio de búsqueda es mucho menor, permitiendo que la metodología genere los resultados en un tiempo razonable.

- Eficacia. En general se produce un claro descenso en el número de sentencias que deben ser revisadas, ya que la metodología determina que sentencias son las únicas que al ser modificadas permiten alcanzar los resultados establecidos en los casos

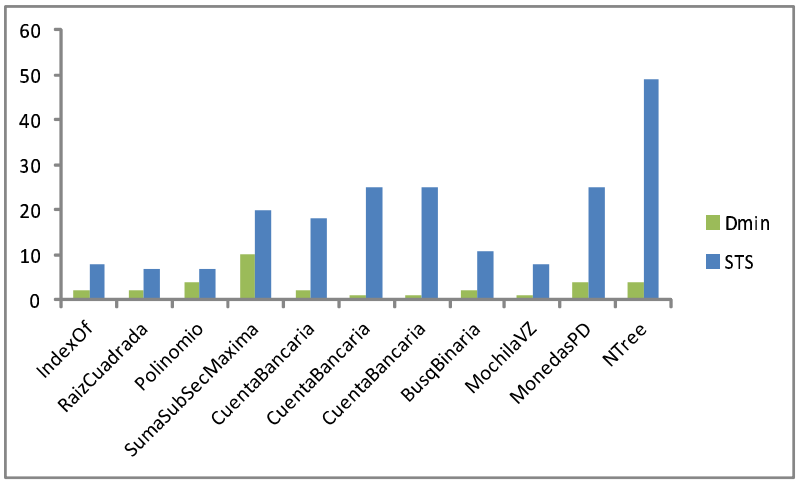


Figura 7.1: Relación entre el número de sentencias implicadas en las diagnosis mínimas (Dmin) y el número de sentencias del programa (STS)

de prueba. En algunos casos, la metodología ha sido capaz incluso de determinar exactamente la sentencia donde se produjo el defecto, como por ejemplo cuando se han modificado las condiciones de las sentencias selectivas (experimentos 2 y 3 de la cuenta bancaria).

Para observar de forma visual las mejoras que ofrece la metodología en cuanto a la eficacia, en la figura 7.1 se muestra la relación entre las diagnosis mínimas propuestas por la metodología y el número de sentencias implicadas en dichas diagnosis.

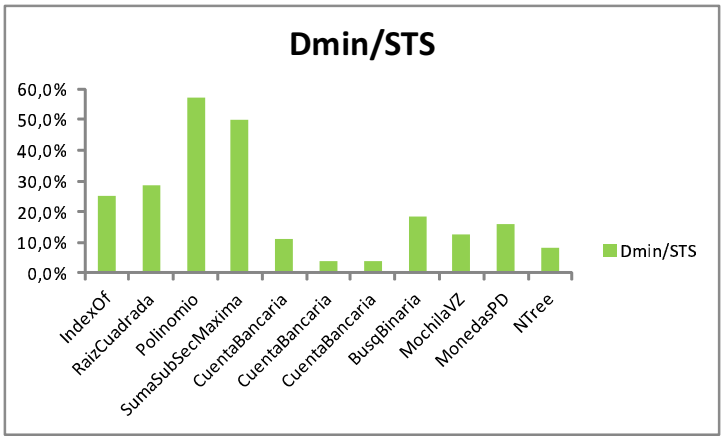


Figura 7.2: Porcentaje de sentencias del código fuente incluidas en las sentencias implicadas en las diagnosis mínimas obtenidas

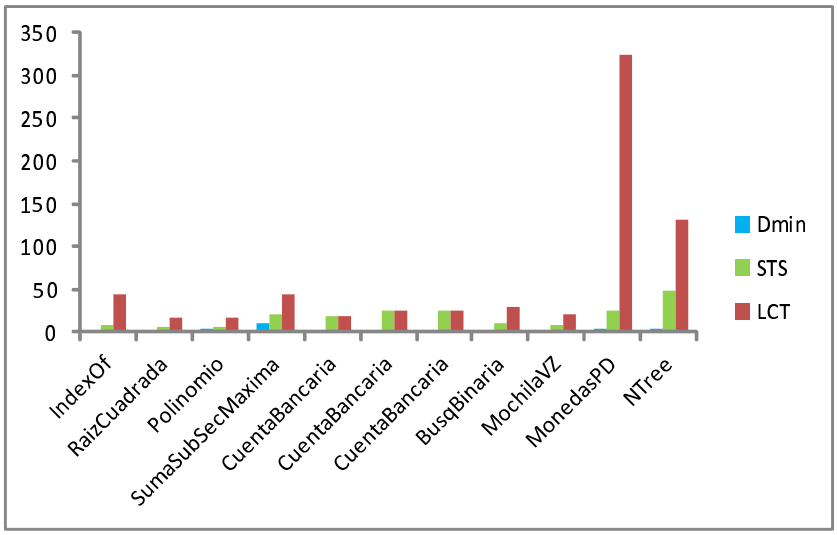


Figura 7.3: Nº de sentencias implicadas en las diagnosis mínimas (Dmin) en comparación con el nº de sentencias del programa (STS) y el nº de sentencias que forman la traza (LCT)

En la figura 7.2 se muestra la relación (en forma de porcentaje) entre el número de sentencias implicadas en las diagnosis mínimas, y el número de sentencias que forman el código fuente. Por último, en la figura 7.3, con respecto a la figura 7.1, se ha añadido como dato el número de sentencias que forman parte de la traza ejecutada para el caso de prueba utilizado en el experimento.

7.2. Ampliaciones para mejorar el proceso de identificación de defectos

Una vez obtenidas las diagnosis mínimas, el siguiente paso es la reparación de los defectos. Para ello deben repasarse una a una las diagnosis mínimas, hasta encontrar cuál corresponde con el defecto, o defectos, reales. La reparación es un proceso manual, no automático, pero puede mejorarse la eficiencia a través de dos ampliaciones que se proponen sobre la metodología presentada en esta tesis.

La primera ampliación se basará en analizar las diagnosis mínimas obtenidas en el caso de disponer de varios casos de prueba donde se hayan detectado errores. La segunda se basará en el análisis de la propagación de los defectos, y sus implicaciones en la reparación eficiente de los defectos.

En los siguientes apartados se mostrarán las ampliaciones propuestas y los beneficios que se consiguen al aplicarlas sobre los ejemplos utilizados en la primera parte de este capítulo.

7.2.1. Utilización de varios casos de prueba

Para comprobar si un sistema software está bien diseñado, se pueden realizar diferentes pruebas encaminadas a ejecutar las diferentes trazas posibles. Una buena selección de los casos de prueba permite un mejor cubrimiento de las posibles trazas.

Para que la metodología de diagnosis también pueda sacar ventaja de la disponibilidad de varios casos de prueba, en este trabajo se propone procesar las diagnosis mínimas obtenidas en cada caso de prueba, e intentar ofrecer al usuario una diagnosis mínima que sea capaz de explicar todos los casos de prueba donde se han producido errores, y que reduzca el número de sentencias a revisar en el proceso de depuración.

La idea es revisar primero aquellas sentencias que expliquen el comportamiento anómalo de un mayor número de casos de prueba. Como consecuencia también aumentaría la eficacia del proceso de reparación, ya que para un mismo defecto se tendrían agrupados los casos de prueba afectados.

Tal como se introdujo en la sección 4.4.1, un problema de diagnosis (DP, Diagnosis Problem) está formado por la tupla (SD, STS, SPEC, TC), donde SD es la descripción del sistema, que incluye un conjunto de restricciones; STS es el conjunto de elementos diagnosticables, que en general será un subconjunto de las sentencias que forma el código fuente; SPEC es el conjunto de asertos que forman la especificación; y TC representa un caso de prueba.

Diferentes casos de prueba pueden generar diferentes trazas de ejecución, por tanto el uso de diferentes casos de prueba en principio generaría diferentes problemas de diagnosis. La aplicación del proceso de diagnóstico a cada uno de los casos de prueba por separado, permite disponer del conjunto de sentencias que forma la diagnosis mínima para cada caso de prueba.

Para encontrar la diagnosis mínima común a todos los casos de prueba en conjunto, la idea es reutilizar las diagnosis mínimas obtenidas para cada caso de prueba, y deducir el menor grupo de sentencias que cubra las diagnosis mínimas de todos los casos de prueba. Es decir, buscar defectos que expliquen el mal comportamiento del sistema para todos los casos de prueba que no han producido los resultados esperados. La diagnosis del sistema para n casos de prueba, de manera análoga a como sucedía con un caso de prueba, será una hipótesis sobre qué sentencias o expresiones deben cambiar para poder alcanzar el resultado establecido como correcto en los n casos de prueba.

La diagnosis para n problemas (SD_i, STS, SPEC, TC_i) será un conjunto $\mathcal{D} \subseteq$ STS, tal que para todo $\text{SD}_i \cup \text{TC}_i \cup \text{SPEC} \cup \{AB(s) \mid \text{s} \in \mathcal{D}\} \cup \{\neg AB(s) \mid s \in STS \text{ - } \mathcal{D}\}$.

Si sólo se está interesado en los defectos simples, este conjunto mínimo será la intersección de los defectos simples que son diagnosis mínimas en los diferentes casos de prueba. Estos defectos sólo podrían encontrarse en aquellas sentencias que cubran todas las trazas fallidas. Pero la intersección entre las diagnosis mínimas asociadas a los casos de prueba puede incluso ser vacía, si hay varios defectos en el programa, por lo que la intersección no siempre es la solución.

Sean n casos de prueba en los que se han detectado errores en un sistema software.

Una vez resueltos los problemas de diagnosis asociados a cada caso de prueba TC_i, existirá un conjunto $\mathcal{D}_i$ de diagnosis mínimas, $\mathcal{D}_i = \{\mathcal{D}_i^1, ..., \mathcal{D}_i^j, ..., \mathcal{D}_i^m\}$, donde cada diagnosis mínima $\mathcal{D}_i^j$ será un subconjunto del conjunto STS de sentencias del sistema software.

Para encontrar la diagnosis mínima común a todos los casos de prueba en conjunto, la idea es reutilizar las diagnosis mínimas obtenidas para cada caso de prueba, y deducir el menor grupo de sentencias que cubra las diagnosis mínimas de todos los casos de prueba. Es decir, buscar defectos que expliquen el mal comportamiento del sistema para todos los casos de prueba que no han producido los resultados esperados. La diagnosis del sistema para n casos de prueba, de manera análoga a como sucedía con un caso de prueba, será una hipótesis sobre qué sentencias o expresiones deben cambiar para poder alcanzar el resultado establecido como correcto en los n casos de prueba.

Definición 7.1. Diagnosis para n casos de prueba. Dados n casos de prueba que han producido errores para un sistema software, existirá un conjunto $\mathcal{D}_i$ de diagnosis mínimas, $\mathcal{D}_i = \{\mathcal{D}_i^1, ..., \mathcal{D}_i^j, ..., \mathcal{D}_i^m\}$, donde cada diagnosis mínima $\mathcal{D}_i^j$ será un subconjunto del conjunto STS que explicará el error detectado para el caso de prueba TC_i. La diagnosis para n problemas (SD_i, STS, SPEC, TC_i) será un conjunto $\mathcal{D} \subseteq STS$, tal que cumpla para cada conjunto $\mathcal{D}_i$ de diagnosis mínimas asociadas al caso de prueba TC_i, existe al menos una diagnosis mínima $\mathcal{D}_i^j \in \mathcal{D}_i$, tal que $\mathcal{D}_i^j \subseteq \mathcal{D}$.

Una diagnosis $\mathcal{D}$ será mínima para n casos de prueba, si para ningún subconjunto $\mathcal{D}' \subset \mathcal{D}$ se cumple que $\mathcal{D}'$ es una diagnosis para los n casos de prueba. Las diagnosis mínimas que explican los errores detectados en los n casos de prueba permiten enfocar el proceso de corrección de defectos en aquellas sentencias que explican el comportamiento anómalo de un mayor número de casos de prueba.

Tal como se explicó en el capítulo 6 cada sentencia tiene asociado un identificador, que la identifica unívocamente dentro del sistema software. Estos identificadores se han utilizado a la hora de generar las restricciones asociadas a la descripción del sistema, en particular se han utilizado en los predicados AB para establecer qué elementos tienen un comportamiento anormal o no, es decir, que elementos contienen defectos. Por tanto, si todas las trazas obtenidas para los casos de prueba provienen de un mismo sistema software, los identificadores serán comunes, y será posible determinar el conjunto M que contendrá todos las sentencias y asertos que forman el sistema software. Cada diagnosis mínima asociada a cada caso de prueba será un subconjunto de M.

Para obtener la diagnosis común a todos los casos de prueba se planteará un problema de optimización basado en restricciones, donde el objetivo será minimizar el número de sentencias que formen parte de la diagnosis común. Para toda sentencia del conjunto STS se definirá una nueva variable b_i de tipo lógico. Un sentencia de STS forma parte de la diagnosis común a todos los casos de prueba si la variable b_i toma el valor verdadero, es decir $\forall s_i \in STS : s_i \in \mathcal{D} \Leftrightarrow b_i$.

Para cada diagnosis mínima $\mathcal{D}_i^j$ del conjunto de diagnosis mínimas obtenidas para el caso de prueba i, se definirá una nueva variable de tipo lógico d_i^j. El valor de dicha

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	DC	$\mathcal{D}$min/STS
SumDiferencia	1	S_{10}	12	10	0,01	1	Sí	8.3 %
SumDiferencia	2	S_{10}	12	22	0,01	2	Sí	16.7 %
SumDiferencia	3	S_{10}	12	19	0,01	2	Sí	16.7 %
CuentaBancaria	2	S_{13}	25	25	0,01	1	Sí	4 %
CuentaBancaria	3	S_{20}	25	25	0,01	1	Sí	4 %

Tabla 7.9: Resultados obtenidos para los casos de prueba propuestos

variable será $d_i^j = \bigwedge b_k : s_k \in \mathcal{D}_i^j$.

Para poder cubrir al menos una diagnosis mínima de cada $\mathcal{D}_i$ que explique los errores detectados en cada caso de prueba, es necesario añadir que se cumpla que $\bigvee_{j=1}^{j=m_i} d_i^j$ es cierto para cada caso de prueba i, siendo m_i el número de diagnosis mínimas obtenidas para el caso de prueba i.

Por último, para que el conjunto $\mathcal{D}$ sea mínimo, el objetivo debe ser minimizar el número de sentencias incluidas en el conjunto $\mathcal{D}$, es decir se debe cumplir el objetivo: $\text{Min}(N\ b_i) : b_i \mid s_i \in \text{STS}$.

Para reducir la complejidad del cálculo, el proceso puede pararse cuando los conjuntos $\mathcal{D}$ alcancen un número de sentencias. Por ejemplo se puede añadir una restricción al problema para que sólo se generen conjuntos $\mathcal{D}$ de tamaño 1 o 2, dando lugar a defectos simples o dobles que explicarían los errores de los casos de prueba donde se han detectado errores.

Aplicación a los experimentos propuestos

En la tabla 7.9 se muestran los resultados obtenidos para diferentes casos de prueba sobre dos ejemplos: el programa SumDiferencia, que aparece en el apéndice B.6, y el programa CuentaBancaria, que aparece en el apéndice B.5.

Para el primero de los ejemplos (programa SumDiferencia), para cada uno de los tres casos de prueba se han obtenido las siguientes diagnosis mínimas: $\mathcal{D}_1 = \{S_{10}, S_{12}\}$, $\mathcal{D}_2 = \{S_{10}, S_{12}\}$, y $\mathcal{D}_3 = \{S_{10}\}$. Siguiendo los pasos expuestos en este apartado, en la figura 7.4 se muestran las restricciones que permiten determinar el conjunto de diagnosis mínimas que explican todos los casos de prueba. Al resolver el CSP, y siguiendo la función objetivo, sólo una diagnosis simple podría explicar todos los casos de prueba, $\mathcal{D} = \{S_{10}\}$. Modificando esta sentencia se podrían resolver los tres casos de prueba donde se han detectado errores. El código correcto sería `s = m1*(z+1)` en lugar de `s = m1*z` que fue el defecto introducido.

Es importante resaltar que aunque los tres casos de prueba han sido aplicados a un mismo programa, las trazas seguidas no son iguales. Hay sentencias que aparecen sólo en algunas trazas. Al aplicar la mejora propuesta, la idea es dar preferencia a aquellas sentencias o grupos de sentencias que permitan explicar los errores detectados en todos los casos de prueba, por tanto, las sentencias que aparezcan en todas los casos de prueba serán revisadas antes.

$\text{STS} = \{\text{S}_1, \text{S}_2, \text{S}_3, \text{S}_4, \text{S}_5, \text{S}_6, \text{S}_7, \text{S}_8, \text{S}_9, \text{S}_{10}, \text{S}_{11}, \text{S}_{12}, \text{S}_{13}, \text{S}_{14}\}$

Diagnosis mínimas:	**Restricciones**:
$\mathcal{D}_1 = \{\mathcal{D}_1^1\}$	$d_1^1 = \text{true}$
$\mathcal{D}_1^1 = \{S_{10}\}$	$d_1^1 = (b_{10})$
$\mathcal{D}_2 = \{\mathcal{D}_2^1, \mathcal{D}_2^2, \mathcal{D}_2^3\}$	$(d_2^1 \vee d_2^2 \vee d_2^3) = \text{true}$
$\mathcal{D}_2^1 = \{S_{10}\}$	$d_2^1 = (b_{10})$
$\mathcal{D}_2^2 = \{S_{12}\}$	$d_2^2 = (b_{12})$
$\mathcal{D}_2^3 = \{S_{13}\}$	$d_2^3 = (b_{13})$
$\mathcal{D}_3 = \{\mathcal{D}_3^1, \mathcal{D}_3^2, \mathcal{D}_3^3\}$	$(d_3^1 \vee d_3^2 \vee d_3^3) = \text{true}$
$\mathcal{D}_3^1 = \{\text{S}_{10}\}$	$d_3^1 = (b_{10})$
$\mathcal{D}_3^2 = \{\text{S}_{12}\}$	$d_3^2 = (b_{12})$
$\mathcal{D}_3^3 = \{\text{S}_{13}\}$	$d_3^3 = (b_{13})$
Función objetivo::	$\text{Min}(N\ b_i) : b_i \mid s_i \in \text{STS}$

Figura 7.4: Obtención de la diagnosis mínima común a varios casos de prueba

Pero esto no quiere decir que exista una probabilidad mayor de que contengan defectos, la idea es simplemente revisar el menor número de sentencias que expliquen el problema detectado en todos los casos de prueba. Recordemos que se parte siempre del supuesto de que el código fuente a revisar es casi correcto, ya que la metodología debe aplicarse cuando el sistema software está en un punto de desarrollo estable, o cuando el desarrollo ya ha finalizado.

Para el segundo de los ejemplos (programa CuentaBancaria), en cada uno de los dos casos de prueba se han obtenido las siguientes diagnosis mínimas: $\mathcal{D}_1 = \{\text{S}_{13}\}$, y $\mathcal{D}_2 = \{\text{S}_{20}\}$. Siguiendo los pasos expuestos en este apartado, la diagnosis mínima que podría explicar todos los casos de prueba sería doble $\mathcal{D} = \{\text{S}_{13}, \text{S}_{20}\}$.

En general, a la hora de revisar el código fuente y resolver los defectos, resulta interesante la información sobre las diagnosis mínimas que resuelven todos los casos de prueba, cuando se consiguen diagnosis simples, o con pocos elementos en comparación a las diagnosis mínimas obtenidas para los casos de prueba por separado. Ya que revisando pocas sentencias (en algunos casos sólo una), es posible resolver a la vez todos los casos de prueba que han producido errores. Es el caso por ejemplo del primer ejemplo (SumDiferencia).

Sin embargo, cuando la diagnosis mínima que explica todos los casos de prueba es doble, o en general de mayor grado que las diagnosis mínimas obtenidas para los casos de prueba por separado, es más conveniente revisar cada caso de prueba por separado, ya que seguramente el número de defectos será mayor que uno. Es el caso del segundo ejemplo (CuentaBancaria).

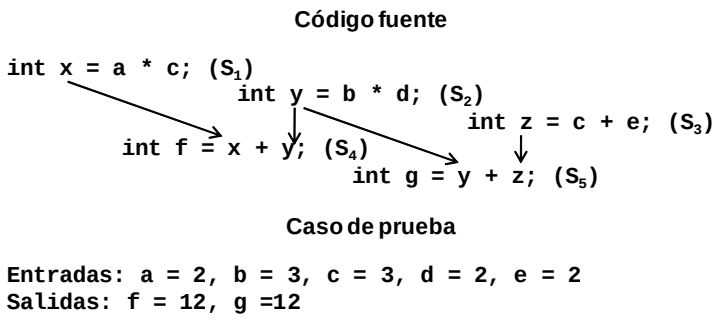


Figura 7.5: Propagación de los fallos en el código fuente

7.2.2. Propagación de los fallos

Tal como se expuso en la introducción de esta tesis (apartado 1.1.1), es importante diferenciar entre los conceptos de fallo, error y defecto. Por ejemplo, cuando se escribe el código fuente, es el desarrollador el que puede introducir un defecto en una sentencia. Como consecuencia de este defecto se producirá un fallo, es decir, el resultado generado por dicha sentencia será diferente al correcto. Este resultado se propagará a otras sentencias, hasta que se llegue a generar un resultado del sistema software que sea diferente al establecido como correcto en el caso de prueba. Esta diferencia entre el resultado real y el establecido en el caso de prueba es lo que denominamos error.

La detección de los errores en un sistema software es posible gracias a que los fallos producidos en un determinado elemento del sistema son propagados por el resto de los elementos que están conectados a éste, hasta llegar a un punto donde sea posible comparar el resultado real con el establecido como correcto por un caso de prueba. Si no existiese esta propagación sería imposible detectar los errores, y por consiguiente diagnosticar los defectos. Por tanto, la detección de los errores, y el posterior diagnóstico de un sistema software, se basa en la propagación de los fallos por los elementos que forman el sistema software, en otras palabras, los defectos que provocan fallos en el software, son detectables gracias a los errores.

Pero lamentablemente, los mismos elementos que propagan los fallos pueden aparecer como los causantes de los errores detectados, es decir, la metodología de diagnosis puede determinar que contienen defectos, cuando en realidad han generado un resultado erróneo debido a que las entradas recibidas no son las correctas.

Por ejemplo, sea el programa de la figura 7.5. Las flechas indican cómo se propagarían los fallos a través de las sentencias. Como se puede observar un fallo en la sentencia que genera la variable x, influiría en la sentencia que genera la variable f. Un fallo en la sentencia que genera la variable y influiría en las sentencias que generan los valores para las variables f y g.

Supongamos que hay un defecto en la sentencia que asigna el valor a la variable z (sentencia S_3). De esta variable no se conoce cuál debe ser el valor correcto, ya que no

está establecido en el caso de prueba, y por tanto, no es posible determinar si ese valor es un error. La propagación de un fallo en dicha sentencia influiría en la sentencia que genera el valor de la variable g (sentencia S_5), en donde sí es posible comprobar si existe un error, ya que el valor correcto de la variable g si aparece en el caso de prueba.

Si la metodología de diagnosis propone dos diagnosis mínimas simples, correspondientes a las sentencias S_3 y S_5, hay que tener en cuenta que el defecto en S_5 puede ser debido a una propagación de un defecto en S_3. Teniendo en cuenta esta información, sería conveniente revisar S_3 antes que S_5, pues si existe un defecto en S_3, lo más probable es que afecte a S_5. Lo prioritario debe ser revisar primero el origen más probable del problema. Por tanto, S_3 y S_5 deben revisarse de forma secuencial.

El objetivo de este apartado es ofrecer un método que permita averiguar estas relaciones de propagación, para agrupar las diagnosis mínimas que deberían ser revisadas de forma secuencial, y de esta forma ahorrar tiempo a la persona encargada de solventar los defectos.

Nodos de componentes

En este apartado se explicará el concepto de variable no prescindible. Este tipo de variables permitirán deducir cuándo determinadas diagnosis mínimas deben revisarse de forma secuencial, debido a una posible propagación de los fallos.

Cada una de las sentencias del código fuente necesita para su ejecución los valores de determinadas variables (y puede que también algunos valores constantes). A su vez, cada sentencia puede asignar valores a determinadas variables. En función de esta relación entre las variables y las sentencias, se pueden definir dos tipos de variables.

- **Variables de entrada** de una sentencia. Serán aquellas variables cuyo valor es leído por dicha sentencia.
- **Variable de salida** de una sentencia. Será aquella variable cuyo valor es asignado por dicha sentencia.

Una misma variable puede ser a la vez de entrada y salida para una misma sentencia, ya que se puede leer, y posteriormente asignar un valor a una variable en una misma sentencia.

Cada caso de prueba, TC, contendrá los valores correctos de las variables de entrada de algunas de las sentencias del sistema software, y los valores correctos de las variables de salida de algunas de las sentencias del sistema software. Estas variables se denominarán entradas y salidas del sistema software para el caso de prueba TC.

En la metodología propuesta, la traza ejecutada se ha transformado a una TCD (Traza de Componentes Desplegables, ver apartado 6.4), y las variables del código fuente han sido transformadas a variables incluidas en las restricciones que componen cada CD (Componente Desplegable, ver apartado 6.4). Por tanto, para conocer las relaciones entre las variables que forman la traza de sentencias ejecutada, será necesario estudiar la TCD.

Antes de seguir, es necesario introducir nuevas definiciones:

Definición 7.2. Variable validada. Una variable estará validada si su dominio es restringido por el caso de prueba que originó la TCD o por alguna restricción obtenida de la especificación.

Definición 7.3. Variable no prescindible. Una variable es no prescindible si se cumplen las siguientes condiciones: no es una variable que esté validada, y sólo dos CD la incluyen en sus restricciones, siendo la única variable de salida de un CD y la variable de entrada de otro único CD.

En una TCD no todas las variables están validadas. Para validar el comportamiento de los componentes, la única posibilidad es propagar los valores de las variables intermedias hasta llegar a una variable validada, y comprobar si los componentes proporcionan la misma información que está establecida como correcta para dicha variable. Los componentes implicados en esa propagación de valores serán exonerados si el resultado es correcto, es decir, se supondrán correctos, y en caso contrario serán candidatos a formar parte de la diagnosis mínima.

En el caso de las variables no prescindibles, se da la circunstancia de que los dos componentes que utilizan cada variable no prescindible necesitan la información asociada a esa variable, para que pueda ser propagada y llegar hasta conexiones ya validadas. En un caso la información será de entrada, y en otro caso la información será de salida. Para resolver el problema de satisfacción con restricciones será necesario establecer un valor para dicha variable. Este valor puede venir generado por un CD y comprobado en el otro, o viceversa, pero no será posible determinar exactamente cuál de los dos componentes tiene un defecto, pues ambos son necesarios para la comprobación del error, no es posible prescindir de ninguno de los dos.

Para tener en cuenta esta característica, los componentes que están relacionados por variables no prescindibles se agruparán en conjuntos denominados nodos. Un nodo se define como:

Definición 7.4. Nodo de componentes. Es un grupo de componentes en donde se cumple que cada pareja de componentes del nodo tienen al menos una variable no prescindible en común, y además, es posible navegar desde cualquier componente c_i a otro componente c_j a través de las variables no prescindibles.

De la propia definición, se puede deducir que en un nodo existirán al menos n - 1 variables no prescindibles que unirán los n componentes del nodo, y que llamaremos **variables no prescindibles internas**.

Proposición. Dado un nodo con n componentes, será posible comprobar si existe un error en el nodo completo, pero no se podrá discernir qué CD de los incluidos en el nodo es el causante del error. Es decir, si un CD de un nodo tiene un defecto, aplicando un caso de prueba no será posible determinar qué CD del nodo es el origen concreto

```
   CD2Nodos(Set<CD> comp) return (Graph<Nodo, Var>){
01   Graph<CD, Var> cg = compToGraph(comp)
02   for (Var v: cg.edges()) {
03     if (isValidada(v))
04       cg.removeEdge(v)
05   }
06   Graph<Nodo, Var> ret = CDGraphToNodoGraph(cg)
07   boolean exit = false
08   while (¬ exit) {
09     exit = true
10     for (Var v: ret.edges()) {
11       if (isNonPresVar(v)) {
12         exit = false
13         Nodo ni = ret.getVertexs(v).get(0)
14         Nodo nj = ret.getVertexs(v).get(1)
15         nj.CDSet.addAll(ni.CDSet)
16         nj.NoPresIntVarSet.addAll(ni.NoPresIntVarSet)
17         nj.NoPresIntVarSet.add(v)
18         for (Nodo nk: ret.adjacentTo(ni))
19           ret.addEdge(nk, nj, ret.removeEdge(nk, ni))
20         ret.removeVertex(ni)
21         break
22       }
23     }
24   }
   }
```

Figura 7.6: Algoritmo para convertir un grafo de componentes en un grafo de nodos de componentes

del error detectado.

El objetivo en este apartado es obtener los nodos de componentes de una TCD, para poder establecer que diagnosis mínimas están incluidas en un nodo, y por tanto deberían ser revisadas a la vez, ya que posiblemente se haya producido una propagación de un fallo a todos los miembros del nodo.

Algoritmo para la determinación de los nodos

En la figura 7.6 aparece el algoritmo que permite pasar de un conjunto de componentes a un grafo que contendrá tantos vértices como nodos se hayan obtenido al agrupar los componentes recibidos como entrada. Cada uno de los nodos se representa

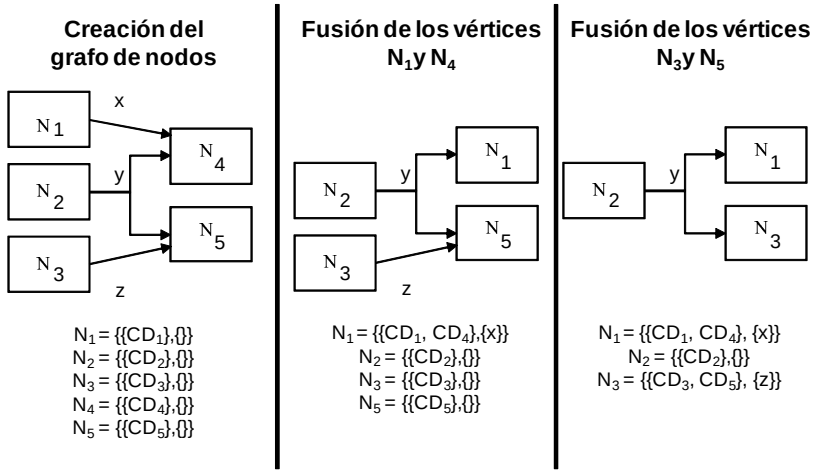


Figura 7.7: Nodos de CD obtenidos para el ejemplo propuesto

como un objeto de tipo Nodo, que tiene dos propiedades: el conjunto de componentes que incluye (propiedad CDSet), y el conjunto de variables no prescindibles internas (propiedad NoPresIntVarSet). Estas dos propiedades son modificables.

La primera parte del algoritmo empieza obteniendo un grafo de dependencias a partir de los componentes, que se guardará en la variable *cg*. Para crear este grafo se utiliza el algoritmo que aparece en el apéndice A.1. De la línea 2 a la 5 del algoritmo se recorren todas las aristas del grafo para eliminar aquellas aristas asociadas a variables que cumplen la definición de variable validada. De esta forma sólo quedarán en el grafo variables no validadas.

La segunda parte del algoritmo comienza obteniendo un grafo a partir del grafo guardado en la variable *cg*, en el que cada vértice v_i representa un nodo que contendrá un único CD y ninguna variable no prescindible interna. Cada arista representa una variable común entre las restricciones asociadas a dos nodos. A partir de ahí, se inicia un proceso incremental que irá fusionando los nodos hasta usar todas las variables no prescindibles que aparezcan en el grafo.

De la línea 8 a la 24 del algoritmo los vértices del grafo se irán fusionando a medida que se encuentren nuevas variables no prescindibles. Este proceso acabará cuando no existan más variables no prescindibles. En la línea 10 se itera sobre las aristas del grafo que contienen las variables comunes entre los nodos, y para cada variable no prescindible, de las líneas 13 a la 20 se realiza su tratamiento, que básicamente consiste en fusionar los dos nodos que conectan dicha variable. El tratamiento de cada variable no prescindible v incluye los siguientes pasos:

- En las líneas 13 y 14 se determinan cuáles son los nodos (ni, nj) que utilizan la variable v. Para obtener los nodos se utiliza el método *getVertexs* que devuelve la lista de vértices asociada a la arista que contiene la variable v. Cada arista

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	Nodos	DC
ToyProgram	1	S_5	5	5	0,01	2	1	Sí
IndexOf	1	S_4	8	44	0,01	2	2	Si
RaizCuadrada+Inv	1	S_2	7	16	0,01	2	2	Sí
Polinomio	1	S_6	7	16	0,01	4	1	Sí
CuentaBancaria	1	S_{15}	18	18	0,01	2	1	Sí
CuentaBancaria	2	S_{13}	25	25	0,01	1	1	Sí
CuentaBancaria	3	S_{20}	25	25	0,01	1	1	Sí
BusqBinaria	1	S_{15}	11	30	0,01	2	2	Sí
MochilaVZ	1	S_8	8	21	0,01	1	1	Sí
MonedasPD	1	S_{21}	25	324	0,01	4	1	Sí
NTree	1	S_{38}	49	131	0,01	4	1	Sí

Tabla 7.10: Resultados obtenidos para los casos de prueba propuestos

puede estar asociada a más de dos vértices, ya que una misma variable puede ser utilizada por más de dos componentes.

- En la línea 15 se añaden los componentes incluidos en el nodo *ni* al nodo *nj*, y en las líneas 16 y 17 se añaden al conjunto de variables no prescindibles del nodo *nj*, las variables no prescindibles de *ni*, y la variable *v*.
- En las líneas 18 y 19 se actualizan todos los nodos que conectaban con *ni*, para que ahora conecten con el nodo *nj* resultado de la fusión entre *ni* y *nj*.
- Por último en la línea 20 se elimina del grafo el nodo *ni*.

A medida que se van generando nuevos nodos, es posible que variables que a priori no cumplían la definición de variable no prescindible, pasen a cumplir dicha definición, por este motivo es necesario revisar de nuevo todas las aristas (y sus variables asociadas) cuando dos nodos son fusionados. Una vez acabado el proceso, se tendrán dos tipos de variables: variables no prescindibles internas, incluidas en el nodo que corresponda; y variables que unen a nodos, que al finalizar el algoritmo quedarán como aristas del grafo.

Ejemplo 7.1. En la figura 7.7 se muestra cuál sería el proceso y el resultado de aplicar el algoritmo CD2Nodos mostrado en la figura 7.6 al problema que aparece en la figura 7.5. Se parte de un grafo de dependencias donde cada vértice v_i representa un nodo N_i que incluirá al componente c_i, y cada arista representa una variable común entre dos componentes. La figura 7.7 está dividida en tres partes. La primera, que está en la parte izquierda de la figura, muestra cuál sería el grafo inicial, y cuál es el conjunto de componentes incluido en cada nodo, y el conjunto de variables no prescindibles internas.

En la siguiente escena de la figura se muestra como quedaría el grafo y los nodos, una vez fusionado el nodo N_1 y N_4 que tienen en común la variable no prescindible x. En la última escena de la figura se muestra como quedaría el grafo y los nodos, una vez

fusionado el nodo N_3 y N_5 que tienen en común la variable no prescindible z. En total quedan 3 nodos, unidos por la variable y.

Aplicación a los experimentos propuestos

En la tabla 7.10 se ha añadido una columna denominada Nodos. Para cada experimento se han obtenido las diagnosis mínimas, y se han calculado los nodos. En dicha columna se muestra el número de nodos implicados en las diagnosis mínimas propuestas. Para los ejemplos de la MochilaVZ, y los casos de prueba 2 y 3 del ejemplo de la CuentaBancaria, la metodología propuesta había obtenido una única diagnosis mínima, por lo tanto el número de nodos resulta irrelevante.

En los ejemplos ToyProblem, Polinomio, primer caso de prueba de la CuentaBancaria, MonedasPD y NTree, todas las diagnosis mínimas propuestas pertenecen a un mismo nodo. Por tanto, aunque existen varias diagnosis mínimas propuestas en estos ejemplos, todas pueden deberse a un mismo fallo que se ha propagado a los elementos del nodo que se ofrecen como posibles diagnosis mínimas.

En los ejemplos IndexOf, RaizCuadrada+Inv y BusqBinaria, se han obtenido dos diagnosis mínimas, y cada una de ellas pertenece a nodos diferentes. Por tanto, en ambos ejemplos, las diagnosis mínimas son independientes, y no se puede deducir que exista la propagación de un fallo entre ellas.

Reducción de la complejidad de cálculo para diagnosis mínimas simples

Las variables de salida del sistema software para un caso de prueba TC son aquellas variables de salida cuyo valor viene dado por el caso de prueba TC. Si al detectar errores en las variables de salida del sistema software, se supone que el defecto que puede explicar los errores detectados es de tipo simple, sólo las sentencias que influyen en los errores detectados, es decir, sólo las sentencias que influyen en las variables de salida del sistema software en donde no coincida el resultado real (obtenido de la ejecución del sistema software) con el correcto (establecido en el caso de prueba), serán las que pueden contener dicho defecto.

Es decir, si el defecto es simple, debe afectar a las variables de salida del sistema software en las que la sentencia que contiene dicho defecto influye. Teniendo en cuenta esta información, sólo sería necesario buscar defectos en aquellas sentencias que influyan en todos los errores detectados. De esta forma se reduciría la complejidad de cálculo de la diagnosis mínima, ya que previamente a la resolución del Max-CSP, se podrían descartar la posibilidad de que ciertas sentencias contengan defectos.

Para ello será necesario clasificar las sentencias que forman la traza ejecutada para un caso de prueba concreto, según el conjunto de variables de salida del sistema software a las que afecten. Los CD que influyen en una misma variable de salida del sistema software serán agrupados en lo que denominaremos cluster.

Definición 7.5. Cluster de componentes. Dado un caso de prueba TC, se define el cluster asociado a una variable de salida del sistema software v_{out}, como el conjunto de

Modelo del sistema

SD:
$P \wedge \neg AB(S_1) \Rightarrow (x = a * c)$
$P \wedge \neg AB(S_2) \Rightarrow (y = b * d)$
$P \wedge \neg AB(S_3) \Rightarrow (z = c * e)$
$P \wedge \neg AB(S_4) \Rightarrow (f = x + y)$
$P \wedge \neg AB(S_5) \Rightarrow (g = y * z)$
$P = true$
$AB(S_1) = false$
$AB(S_2) = false$
$AB(S_4) = false$
STS:
$\{S_1, S_2, S_3, S_4, S_5\}$
TC:
Entradas: $\{a = 3, b = 2, c = 2, d = 3, e = 3\}$
Salidas: $\{f = 12, g = 12\}$

Soluciones del problema Max-CSP

Diag. mínima: S_3
cierto $\wedge$ cierto $\Rightarrow (6 = 3 * 2)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge$ falso $\Rightarrow (2 = 2 * 3)$
cierto $\wedge$ cierto $\Rightarrow (12 = 6 + 6)$
cierto $\wedge$ cierto $\Rightarrow (12 = 6 * 2)$
Diag. mínima: S_5
cierto $\wedge$ cierto $\Rightarrow (6 = 3 * 2)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge$ cierto $\Rightarrow (6 = 2 * 3)$
cierto $\wedge$ cierto $\Rightarrow (12 = 6 + 6)$
cierto $\wedge$ falso $\Rightarrow (12 = 6 * 6)$

Figura 7.8: Ejemplo de resolución de un problema Max-CSP

todos los CD que influyen en el valor final de dicha variable v_{out}, en la traza seguida por el sistema software para el caso de prueba TC.

Si se supone que un defecto en una sentencia debe afectar a todas las variables de salidas en las que dicha sentencia influye, sólo podrán contener defectos aquellas CD que pertenecen al conjunto C, tal que $C = \bigcap C_i$: error(S_i) - $\bigcup C_i$: ¬error(S_i), donde C_i representa el cluster de CD que influye en la variable de salida del sistema software S_i. El predicado error(S_i) devuelve verdadero si en la variable de salida S_i se ha producido un error, y falso en el caso contrario.

Sólo los predicados AB asociados a los CD que forman dicho conjunto C, podrán tomar el valor verdadero, pues sólo estos componentes podrían explicar con su comportamiento anormal los defectos detectados. De esta forma, realizando un análisis de los cluster que influyen en cada una de las variables de salida del sistema, previamente a la resolución del Max-CSP, se podría reducir el dominio inicial de los predicados AB, ya que todos aquellos que no pertenezcan al conjunto C tendrían el dominio del predicado AB acotado al valor falso.

El algoritmo encargado de obtener los clusters de componentes aparece en el apéndice A.2.

Ejemplo 7.2. En la figura 7.8 se muestra cómo quedaría planteado el problema Max-CSP para el ejemplo ToyProgram (apéndice B.1). En el ejemplo mostrado, la variable g toma un valor final de 36, que es diferente al establecido en el caso de prueba.

Nombre	CT	Defecto	STS	LCT	Tiempo	$\mathcal{D}$min	Clusters	DC
ToyProgram	1	S_5	5	5	0,01	2	2	Sí
IndexOf	1	S_4	8	44	0,01	2	1	Sí
RaizCuadrada+Inv	1	S_2	7	16	0,01	2	1	Sí
Polinomio	1	S_6	7	16	0,01	4	1	Sí
CuentaBancaria	1	S_{15}	18	18	0,01	2	2	Sí
CuentaBancaria	2	S_{13}	25	25	0,01	1	2	Sí
CuentaBancaria	3	S_{20}	25	25	0,01	1	2	Sí
BusqBinaria	1	S_{15}	11	30	0,01	2	1	Sí
MochilaVZ	1	S_8	8	21	0,01	1	4	Sí
MonedasPD	1	S_{21}	25	324	0,01	4	1	Sí
NTree	1	S_{38}	49	131	0,01	4	2	Sí

Tabla 7.11: Resultados obtenidos para los casos de prueba propuestos

En este problema existen dos salidas, las variables f y g, por tanto existen dos clusters que llamaremos C_1 y C_2, que incluyen a las sentencias C_1=$\{S_1, S_2, S_4\}$, y C_2=$\{S_2, S_3, S_5\}$. El cluster C_1 influye en la variable de salida del sistema software f, y el cluster C_2 influye en la variable de salida del sistema software g. El conjunto C = $\bigcap C_i$: error(S_i) - $\bigcup C_i$: $\neg$error(S_i) se obtendría como resultado de la operación $\{S_2, S_3, S_5\}$ - $\{S_1, S_2, S_4\}$ = $\{S_3, S_5\}$. Por tanto, y tal como se ha explicado anteriormente, si se supone que un defecto en una sentencia debe afectar a todas las variables de salida en las que dicha sentencia influye, sólo podrán ser diagnosis simples las sentencias $\{S_3, S_5\}$.

Aplicando la metodología propuesta para la diagnosis del código fuente, se obtendría una TCD que estaría formada por cinco CD de tipo asignación, correspondientes a las cinco sentencias que forman el código fuente tal como se muestra en la misma figura A.1. En la figura 7.8 se han añadido como restricciones que las sentencias $\{S_1, S_2, S_4\}$ no podrán tener un comportamiento anormal. De un total de 5 sentencias, sólo 2 tendrán el predicado AB libre, y por tanto son candidatas a contener algún defecto, la reducción en este ejemplo es de un 60 %.

Del total de diagnosis posibles simples permitidas, con esta modificación, en la resolución del problema Max-CSP sólo se tendrían en cuenta las sentencias $\{S_3, S_5\}$, que son las únicas que influyen exclusivamente en la variable de salida g donde se ha detectado el error. En la parte derecha de la figura se muestran las dos diagnosis mínimas que se obtendrían, y cuáles son los valores a los que se reduciría el dominio de las variables en la resolución del Max-CSP.

En la tabla 7.11 se ha añadido una columna denominada Clusters. Para cada experimento se han obtenido los clusters y las diagnosis mínimas. En dicha columna se muestra el número de clusters (existe un cluster por cada variable de salida cuyo valor correcto esté establecido en el caso de prueba). Para los ejemplos del ToyProgram, la CuentaBancaria, la MochilaVZ, y NTree, la metodología propuesta ha obtenido 2 o más clusters. Sin embargo el tiempo de cálculo de las diagnosis mínimas no ha mejorado

significativamente. Aunque teóricamente debe existir una mejora, ya que ciertas sentencias dejarían de ser analizadas como posibles diagnosis mínimas, este tiempo ahorrado no es significativo en el tiempo total del calculo de las diagnosis mínimas. En el resto de los experimentos, el número de clusters es uno, por lo que no es posible aplicar la mejora.

7.3. Conclusiones

Los resultados obtenidos de la metodología, analizados desde el punto de vista de la eficiencia y la eficacia han sido satisfactorios. El tiempo necesario para la determinación de las diagnosis mínimas ha sido muy parecido en todos los experimentos, y aceptable desde el punto de vista práctico. En general se produce un descenso en el número de sentencias que deben ser revisadas, ya que la metodología determina qué sentencias son las únicas que al ser modificadas permiten alcanzar los resultados establecidos en los casos de prueba. En algunos de los experimentos, la metodología ha sido capaz incluso de determinar exactamente la sentencia donde se produjo el defecto.

Tal como se expuso en la introducción la detección de los errores en un sistema software es posible gracias a que los fallos producidos en un determinado elemento del sistema son propagados por el resto de los elementos que están conectados a éste. Pero los mismos elementos que propagan los fallos pueden aparecer como los causantes de los errores detectados. Por este motivo se ha propuesto una ampliación de la metodología que tenga en cuenta en los resultados ofrecidos, la propagación de los defectos. Las diagnosis mínimas relacionadas en la propagación de un posible defecto se ofrecen agrupadas. Esta información resulta interesante a la hora de reparar los defectos, pues permite al programador revisar de forma secuencial aquellas sentencias relacionadas con un mismo defecto, y de esta forma ahorrar tiempo a la hora de corregir el código fuente.

Por último, para que la metodología de diagnosis también pueda sacar ventaja de la disponibilidad de varios casos de prueba, se ha añadido la posibilidad de una vez procesadas las diagnosis mínimas de cada caso de prueba, ofrecer al usuario una diagnosis mínima que sea capaz de explicar todos los casos de prueba donde se han producido errores, y que reduzca el número de sentencias a revisar en el proceso de depuración. La idea es revisar primero aquellas sentencias que expliquen el comportamiento anómalo de un mayor número de casos de prueba, optimizando el tiempo de reparación de los defectos.

La Diagnosis del Software permite a una organización dedicada a la producción del software, realizar la identificación de los defectos de forma automática, evitando el gasto en tiempo y recursos que conllevaría la realización de la misma tarea de manera manual. Aunque en un principio la organización tendría que realizar una inversión inicial para implantar la metodología, el retorno de la inversión se lograría a corto plazo teniendo en cuenta los experimentos realizados. Ya que como se ha mostrado en los experimentos realizados, existe una reducción significativa en el número de elementos a revisar en la

depuración, y por tanto, un ahorro de recursos dedicados a la localización de defectos, que podrían ser dedicados a trabajo productivo.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Contribuciones de esta tesis

La metodología de diagnosis propuesta en esta tesis permite identificar defectos en el software a través de un modelo basado en restricciones. En concreto, los defectos pueden estar localizados en los asertos de la especificación y en las sentencias del código fuente.

- El primer objetivo de esta tesis ha sido el diagnóstico de la especificación del software, expresada en forma de asertos de un Diseño por Contrato. Primero se han propuesto diferentes comprobaciones para chequear la consistencia entre los asertos que forman la especificación, y posteriormente, una vez comprobada las inconsistencias, se ha aplicado la metodología de diagnosis propuesta con la idea de identificar de forma automática cuáles de los asertos deben cambiar.

  La metodología permite aislar de forma automática los defectos de la especificación antes de pasar a la implementación. El código fuente debe cumplir la especificación establecida, por tanto, la metodología propuesta permite evitar la propagación de los defectos al desarrollo del código fuente.

- El segundo objetivo de esta tesis ha sido el diagnóstico del código fuente. Para cada caso de prueba donde los resultados obtenidos han sido diferentes a los correctos, se ha planteado un problema de diagnosis.

  La metodología evita tener que revisar de forma manual todas las sentencias que forman la traza seguida, ya que propone de forma automática cuáles son las sentencias, que una vez modificadas, permiten alcanzar los resultados establecidos como correctos en los casos de prueba.

Los resultados obtenidos de la metodología, analizados desde el punto de vista de la eficiencia y la eficacia han sido satisfactorios. El tiempo necesario para la determinación de las diagnosis mínimas ha sido adecuado desde el punto de vista práctico.

Además, en el caso del diagnóstico del código fuente, se ha ampliado la metodología, para que ofrezca información relevante para reparar los defectos. Por ejemplo, se

ha ampliado la metodología para tener en cuenta en los resultados ofrecidos, la propagación de los defectos. Esta ampliación es interesante, pues permite al programador revisar de forma secuencial aquellas sentencias que pueden estar relacionadas debido a la propagación de un mismo defecto. También, para que la metodología de diagnosis pueda sacar ventaja de la disponibilidad de varios casos de prueba, se ha añadido la posibilidad de ofrecer al usuario una diagnosis mínima que sea capaz de explicar todos los casos de prueba donde se han detectado errores, y que permita revisar primero aquellas sentencias que expliquen el comportamiento anómalo de un mayor número de casos de prueba.

En comparación con las técnicas que se vieron en el estado del arte, la Diagnosis del Software ofrece las siguientes ventajas:

- Diseño por Contrato. La especificación establece qué es lo que el software debe cumplir y permite comprobar si los resultados generados por la implementación son correctos. Sin embargo, no es posible garantizar en todos los casos, que el producto software desarrollado bajo el Diseño por Contrato esté libre de defectos, ya que la especificación no es generada de forma automática, y los defectos introducidos en los contratos se pueden propagar al código fuente. Es más, aunque la especificación no tuviera defectos, si no es completa o es poco restrictiva, el código fuente podría contener defectos aun cumpliendo lo establecido en la especificación.

  La metodología propuesta en esta tesis sirve de complemento al Diseño por Contrato ya que permite detectar y aislar defectos en los asertos, realizando previamente para ello, comprobaciones en la consistencia de la especificación. Además, en el código fuente existen defectos que la especificación no puede detectar por las limitaciones citados en el parrafo anterior, y que la metodología propuesta si puede aislar utilizando casos de prueba.

- Analisis estático. El análisis de los grafos de control de flujo, de las dependencias del código fuente, o el Slicing estático, permite detectar defectos antes de ejecutar el código fuente. Estas técnicas, o variantes de ellas, ya han sido incorporadas en muchos compiladores, para detectar defectos como variables no inicializadas, sentencias no alcanzables, etc.

  Pero lamentablemente existen defectos que sólo se manifiestan ante determinados casos de prueba, y es necesario ejecutar el código fuente para comprobar sus efectos. Ante este tipo de defectos el análisis estático se encuentra limitado.

- Analisis dinámico. Se basa en utilizar casos de prueba para comprobar si un código fuente es correcto o no. El uso de las pruebas permite detectar errores, pero no determinan cuál es el defecto que los origina. La diagnosis del software se plantea como un complemento a los casos de prueba, aportando como valor añadido la localización automática de los defectos.

- Slicing dinámico. En general las técnicas de Slicing Dinámico son más precisas que las de Slicing Estático, ya que en una traza concreta, se conocen todos los valores de las variables, y por tanto, se puede reducir más el tamaño de las slices. Las técnicas de slicing dinámico permiten aislar un defecto dado un caso de prueba, pero lamentablemente no pueden utilizar la información contenida en la especificación.

  Se basan en ver las dependencias estructurales, pero no tienen en cuenta restricciones que plantea la especificación a los datos. La metodología propuesta en esta tesis sí puede utilizar la información contenida en la especificación. Dicha información puede utilizase para eliminar falsos positivos, y así aislar los defectos de forma más precisa y eficaz.

- Dicing. Cuando se tienen dos casos de prueba que ejercitan un mismo código, pero uno provoca un error y el otro no, la técnica de Dicing puede determinar la parte del código fuente que ha influido en el error, y que a su vez no ha influido en el caso de prueba donde el comportamiento ha sido correcto.

  Se basa en suponer que siempre que exista un defecto, éste debe manifestarse en todos los casos de prueba donde participe. Sin embargo esta suposición no siempre es cierta. Por ejemplo, supóngase la operación de suma de dos enteros de una sentencia ha sido sustituida por una operación de resta de dos enteros. Ambas operaciones generarían el mismo resultado si debido a un caso de prueba el elemento sumado o restado fuera el cero, por lo que en principio no se produciría un error, y este caso de prueba exoneraría a la sentencia que incluye la operación defectuosa. Sin embargo existe un defecto en dicha operación, es decir, aunque el caso de prueba no detecta el error, estamos ante un falso negativo. Por este motivo, en la metodología propuesta en esta tesis, sólo se han tenido en cuenta aquellos casos de prueba en los que se han detectado errores, lo que evita la aparición de estos falsos negativos.

- Depuración manual. La depuración del software es una actividad costosa en tiempo. Frente a la depuración manual, la metodología propuesta permite el aislamiento de los defectos de forma automática.

- Delta Debbuging. Delta Debugging podría considerarse como una instancia del Testing Adaptativo, en el que la generación de cada prueba depende de los resultados obtenidos en las pruebas anteriores. La idea es ir simplificando los casos de prueba eliminando los elementos irrelevantes, a través de pruebas sistemáticas sobre el código. Un elemento es irrelevante si el error detectado ocurre estando o no presente dicho elemento. Para poder aplicar la metodología de forma automática, es necesario definir una herramienta que permita verificar de forma automática las pruebas propuestas, y una estrategia para determinar qué elementos son relevantes o no. La estrategia más utilizada es la basada en la búsqueda binaria, en la que los elementos que forman la traza son divididos en dos subconjuntos (sin elementos en común).

	Slicing Dinámico	Dicing	Delta Debugging	MB Debbuging	Métodología propuesta
Contratos	No	No	No	Si/No	**Si**
Diag. Contratos	No	No	No	No	**Si**
Uso de varios casos de prueba	No	Si (falsos negativos)	Si/No	No	**Si**

Figura 8.1: Comparación de la metodología propuesta frente a otras técnicas

El problema de Delta Debbuging, y de otras técnicas de testing adaptativo, es que se necesitan muchos casos de prueba, o una forma de generarlos de forma automática. En la mayoría de los programas no se dispone de esta herramienta, pues los casos de pruebas suelen ser generados de forma manual y por personas. Este requisito no es necesario en la diagnosis del software, pues simplemente con un caso de prueba sería suficiente para poder aplicar la metodología propuesta.

- Model Based Debugging. Otras aproximaciones han intentado aprovechar las técnicas de diagnosis (DX) para realizar la depuración del software de forma automática. Es el caso del proyecto JADE (Java Diagnosis Experiments), de depuración de código basado en modelos. El modelo representa las sentencias y expresiones como si fuesen componentes de un sistema físico.

  Los trabajos de Model Based Debugging son aplicados a la programación estructurada, mientras que la metodología propuesta en esta tesis permite tener en cuenta programas orientados a objetos. A diferencia de los trabajos de MBD, la metodología propuesta en esta tesis se ha basado en la programación con restricciones, lo que permite un mejor tratamiento de la especificación. Una vez comprobada la especificación, ésta forma parte de las restricciones utilizadas para el diagnóstico del código fuente, mejorando la precisión en la obtención de las diagnosis mínimas. Por último, en la diagnosis del software se han añadido mejoras basadas en el uso de varios casos de prueba, y en el análisis estructural, permitiendo mejorar los resultados que ofrece la metodología tradicional de diagnosis basada en modelos (DX), y el proceso de reparación de los defectos.

En la tabla 8.1 se han resumido las conclusiones que se han detallado anteriormente al comparar la metodología propuesta con las técnicas más utilizadas.

- En la primera fila se hace referencia a si la metodología es capaz o no utilizar la información contenida en asertos que forman el Diseño por Contrato.

- En la segunda fila se hace referencia a si la metodología es capaz de identificar defectos en los asertos que forman el Diseño por Contrato.
- En la tercera fila se hace referencia a si la metodología es capaz de identificar defectos en el código fuente utilizando a la vez varios casos de prueba. En este apartado es conveniente remarcar, que aunque la técnica de Dicing permite el uso de varios casos de prueba a la vez, tiene el problema de los falsos negativos que se explico anteriormente.

Por último, a modo de resumen, a continuación se engloban las principales ventajas que la aplicación de la metodología aporta al desarrollo del software:

- **Identificación automática de los defectos**. La metodología de diagnóstico presentada en esta tesis permite encontrar las causas que han dado lugar a los errores detectados. La ventaja reside en poder determinar estas causas de forma automática. Además, para los casos en los cuales existen varias causas posibles, la metodología de diagnosis ofrece como resultado una relación con las posibles causas ordenadas siguiendo el principio de la parsimonia, y de esta forma conseguir que la revisión sea lo más eficiente posible.
- **Reducción del tiempo de desarrollo**. Al seleccionar la información a revisar de forma automática se reduce el tiempo de desarrollo, pues la solución a los errores encontrados se puede acometer antes.
- **Reducción de los costes de mantenimiento**. Hoy en día, el tamaño de las aplicaciones en desarrollo crece hasta niveles que los hacen difíciles de asimilar por una persona, por lo que cada vez es más importante disponer de herramientas que permitan identificar de forma automática el origen de los errores encontrados durante el mantenimiento.
- **Disminución de la propagación de defectos**. La detección y posterior diagnóstico de los defectos, evita su propagación a las etapas posteriores del desarrollo. Como es sabido, el coste asociado a los defectos del sistema, aumenta a medida que el proyecto avanza.

8.2. Líneas de trabajo futuro

El uso de diferentes casos de prueba permite refinar los resultados propuestos por la metodología de diagnosis. Por esta razón es muy importante seleccionar buenos casos de prueba que cubran un amplio espectro de posibles defectos. Cuanto mejor sea la selección de los casos de prueba, más precisa será la diagnosis mínima obtenida. Esta debe ser una de las líneas de investigación a seguir, intentar refinar el proceso de obtención del conjunto de casos de prueba a utilizar con el objetivo final de mejorar el proceso de diagnosis.

Otro objetivo a cubrir será la extensión de la gramática utilizada para expresar los asertos que forman la especificación. De forma que se permitan por ejemplo operadores como el sumatorio, el cuantificador existencial ($\exists$), el cuantificador universal ($\forall$), u operaciones sobre conjuntos como por ejemplo la pertenencia a conjuntos ($\in$).

Por último, aunque no por ello menos importante, otro objetivo es extender la metodología para que se pueda aplicar a ejemplos más complejos, donde el mapeo a restricciones requiera un mayor número de adaptaciones. Por ejemplo, se podría extender la metodología a más características propias de un lenguaje orientado a objetos, como son la herencia, gestión de excepciones, concurrencia, etc.

Apéndice A

Algoritmos auxiliares

A.1. Transformación de una TCD a un grafo de dependencias

Una Traza de Componentes Desplegados (TCD) está compuesta por todos los Componentes Desplegados (CD) que se han ejecutado en un sistema software para un caso de prueba concreto (tal como se explicó en el apartado 6.4). Cada CD almacena una serie de restricciones que utilizarán determinadas variables.

El objetivo al transformar la TCD a un grafo de dependencias es tener una vista sobre que variables son comunes a los diferentes CD, y más concretamente, qué conjunto de componentes influyen en el dominio de cada variable. La representación en forma de grafo de dependencias se ha utilizado en el estudio de la propagación de los fallos y la determinación de los clusters de componentes vista en el capítulo 7.

Ejemplo A.1. En la figura A.1 se muestra un ejemplo de obtención del grafo de dependencias. Se trata del ejemplo ToyProgram que aparece en el apéndice B.1. Para el caso de prueba propuesto, el resultado real obtenido para la variable f es 12 (igual al establecido en el caso de prueba), mientras que para la variable g el resultado es 36 (diferente al establecido en el caso de prueba).

Aplicando la metodología propuesta para la diagnosis del código fuente, se obtendría una TCD que estaría formada por cinco CD de tipo asignación, correspondientes a las cinco sentencias que forman el código fuente, tal como se muestra en la misma figura A.1. En la parte baja de la figura se muestra el grafo de dependencia que se obtendría al transformar la TCD obtenida para el caso de prueba propuesto. Como se puede observar cada vértice contiene un CD, y los vértices están unidos por aristas que almacenan variables que son comunes a las restricciones asociadas a los CD. En los siguientes apartados se mostrarán los pasos a seguir para obtener el grafo de dependencias.

Código fuente
int x = a ∗ c;
int y = b ∗ d;
int z = c ∗ e;
int f = x + y;
int g = y ∗ z;

Caso de prueba
Entradas: a = 2, b = 3,
c = 3, d = 2, e = 2
Salidas: f = 12, g =12

TCD

CD$_1$	entradas:{a, c} salidas:{x}
	sd: { P ∧ ¬AB(S_1) ⇒ (x = a ∗ c) }
CD$_2$	entradas:{b, d} salidas:{y}
	sd: { P ∧ ¬AB(S_2) ⇒ (y = b ∗ d) }
CD$_3$	entradas:{c, e} salidas:{z}
	sd: { P ∧ ¬AB(S_3) ⇒ (z = c ∗ e) }
CD$_4$	entradas:{x, y} salidas:{f}
	sd: { P ∧ ¬AB(S_4) ⇒ (f = x + y) }
CD$_5$	entradas:{y, z} salidas:{g}
	sd: { P ∧ ¬AB(S_5) ⇒ (g = y ∗ z) }

Grafo de dependencias resultante

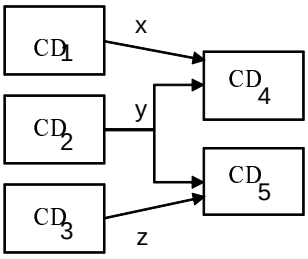


Figura A.1: Grafo de dependencias obtenido para el ejemplo ToyProgram

A.1.1. Cálculo de las entradas y salidas de cada CD

Cada una de las sentencias del código fuente necesita para su ejecución los valores de determinadas variables, y/o de determinadas constantes. A su vez, cada sentencia puede asignar valores a determinadas variables. En función de esta relación entre las variables y las sentencias, se pueden definir dos tipos de variables.

- **Variables de entrada** de una sentencia. Serán aquellas variables cuyo valor es leído por dicha sentencia.
- **Variables de salida** de una sentencia. Serán aquellas variables cuyo valor es asignado por dicha sentencia.

Cada CD se ha obtenido como transformación de las Unidades Ejecutadas (UE) que forman la traza seguida para un caso de prueba. Cada CD almacena un conjunto de restricciones que incluyen una serie de variables. Estas variables pueden ser consideradas de entrada o salida en función de si eran de entrada o salida en las sentencias que dieron lugar a las UE que fueron transformadas a CD. Por tanto, para poder saber si

UE	Entradas	Salidas
UE_Invariant	inputs(exp)	
UE_Field	asigStat.entradas	asigStat.salidas
UE_Method	inputs(params) ∪ block.entradas	block.salidas
UE_Constructor	inputs(params) ∪ block.entradas	@result ∪ block.salidas
UE_Block	noLocal(inputs(stats))	noLocal(outputs(stats))
UE_LocalVariable	asigStat.entradas	asigStat.salidas
UE_If	inputs(exp) ∪ if.entradas	if.salidas
UE_IfElse	inputs(exp) ∪ if.entradas ∪ else.entradas	if.salidas ∪ else.salidas
UE_Return	inputs(exp)	@result
UE_Assert	inputs(exp)	
UE_AsigStat	AsigExp.entradas	AsigExp.salidas ∪ left
UE_MCall	inputs(dest) ∪ met.entradas ∪ inputs(arguments)	met.salidas
UE_CCall	met.entradas ∪ inputs(arguments) ∪ met.ret.invs.entradas ∪ met.ret.fields.entradas	met.salidas ∪ met.ret.fields.entradas
UE_Argument	inputs(exp)	

Tabla A.1: Entradas y salidas para cada una de las UE de un sistema software

las variables son de entrada o salida es necesario añadir a cada una de las UE cuáles son las variables de entrada y cuáles son de salida.

En la tabla A.1 aparece de forma resumida, la forma de determinar para cada UE, cuál será el conjunto de entradas y salidas. Las entradas y salidas de cualquiera de las UE dependerán también del alcance de las definiciones realizadas. Concretamente las variables locales sólo son visibles y modificables en el método donde han sido declaradas, pero no fuera de él. Para completar la tabla A.1 se han utilizado una serie de funciones auxiliares. Las más importantes son *noLocal* e *inputs*. La función *inputs* devuelve el conjunto de identificadores de los atributos, parámetros y variables, presentes en el parámetro recibido como entrada. La función *noLocal* aplicada a la lista de entradas o salidas de un bloque devuelve sólo las entradas o salidas que no correspondan con variables declaradas localmente dentro de ese bloque.

En la transformación de las UE a CD, la información sobre las variables de entrada y de salida debe incorporarse a los componentes. En la tabla A.2 aparece de forma resumida cuál será el conjunto de entradas y salidas para cada CD en función de la función de transformación utilizada (las funciones de transformación se mostraron en el apartado 6.4.3).

Transformación	CD resultante	Entradas del CD	Salidas del CD
UE_LocalVar2CD	CD_Asig	UE_LocalVar.entradas	UE_LocalVar.salidas
UE_AsigStat2CD	CD_Asig	UE_AsigStat.entradas	UE_AsigStat.salidas
UE_Assert2CD	CD_Assert	UE_Assert.entradas	
UE_Block2CD	CD_Block	UE_Block.entradas	UE_Block.salidas
UE_IfElse2CD	CD_IfElse	UE_IfElse.entradas	UE_IfElse.salidas
UE_CCall2CD	CD_CCall	UE_CCall.entradas	UE_CCall.salidas
UE_MCall2CD	CD_MCall	UE_MCall.entradas	UE_MCall.salidas
UE_Return2CD	CD_Asig	UE_Return.entradas	UE_Return.salidas

Tabla A.2: Relación de las entradas y salidas entre las UE y los CD

Una vez incorporado en cada CD cuáles son las variables de entrada y de salida, ya será posible utilizar el algoritmo que permite pasar de un conjunto de CD a un grafo. En el siguiente apartado se mostrará dicho algoritmo.

```
compToGraph(Set<CD> compSet) return (Graph<CD, Var>){
01    Graph<CD, Var> g = new GraphImp<CD, Var>()
02    for (CD comp: compSet){
03      addVertexs(comp, g)
04      Map<Var, Set<CD>> inputs = addInputs(comp, inputs)
05      Map<Var, CD> salidas = addOutputs(comp, salidas)
06      addEdges(g, inputs, salidas)
      }
07    return g
 }
```

Figura A.2: Algoritmo para la conversion de un conjunto de CD a un grafo de dependencias

A.1.2. Obtención del grafo de dependencias a partir de los CD

En la figura A.2 se muestra el algoritmo encargado de transformar un conjunto de CD (Componentes Desplegados) en un grafo; donde cada vértice representa un CD, y cada arista representa una variable común entre las restricciones asociadas a dos componentes.

El algoritmo comienza creando el grafo g (línea 1) que se devolverá como resultado. Se trata de un grafo dirigido en el cuál se almacenan componentes en los vértices y variables en las aristas. Desde las líneas 2 a la 6, se irán añadiendo al grafo en forma de vértices cada uno de los componentes incluidos en el conjunto recibido como parámetro de entrada del algoritmo.

En la línea 3 se realiza una llamada al método *addVertex* encargado de añadir los vértices al grafo que correspondan con el CD guardado en la variable *comp*. Conviene recordar que cada CD puede incluir a su vez otros CD internos. En la línea 4 se crea una función auxiliar (*inputs*) encargada de almacenar para cada variable, cuál es el conjunto de componentes que la usan como entrada. Para ello se utiliza el método *addInputs*. De forma análoga, en la línea 5 se crea una función auxiliar (*outputs*) encargada de almacenar para cada variable, cuál es el CD que genera dicha variable como salida. Para ello se utiliza el método *addOutputs*. En la línea 6 se realiza una llamada al método *addEdges*, encargado de añadir las aristas al grafo, que añade una arista por cada variable que es común a dos componentes incluidos en dos vértices del grafo.

En la línea 7 se da el último paso del algoritmo que consiste en devolver el grafo que se ha construido. En los siguientes apartados se mostrarán los detalles de cada uno de los métodos a los que este algoritmo realiza llamadas.

```
addVertex(CD comp, Graph<CD, Var> g){
01    if (comp instanceOf CD_Block)
02       addVertexs(((CD_Block)comp).stats, g)
03    if (comp instanceOf CD_Asig || comp instanceOf CD_Assert)
04       g.addVertex(comp)
05    if (comp instanceOf CD_IfElse) {
06       if (((CD_IfElse)comp).if != null)
07          addVertex(((CD_IfElse)comp).if, g)
08       else
09          addVertex(((CD_IfElse)comp).else, g)
10    }
11    if (comp instanceOf CD_CCall) {
12       CD_CCall cc = (CD_CCall) comp
13       addVertexs(cc.fields, g)
14       addVertexs(cc.arguments, g)
15       addVertex(cc.met, g)
16    }
17    if (comp instanceOf CD_MCall) {
18       CD_MCall mc = (CD_MCall) comp
19       addVertexs(mc.arguments, g)
20       addVertex(mc.met, g)
21    }
}
addVertexs(Set<CD> scomp, Graph<CD, Var> g){
22    for (CD c: scomp)
23       addVertex(c, g)
}
```

Figura A.3: Método addVertex y addVertexs

Método addVertex

En este apartado se mostrarán los detalles del método *addVertex* encargado de transformar los CD en vértices de un grafo. Tal como se expuso en el capítulo 6, existen diferentes tipos de CD. Algunos de estos, por ejemplo, poseen otros componentes internos. Antes de añadir cada uno de los componentes como vértices del grafo, será necesario tener en cuenta de qué tipo es dicho CD, ya que cada tipo de CD tendrá asociado una transformación diferente:

- CD_Asig y CD_Assert. De cada componente de estos dos tipos se obtendrá un vértice del grafo. En general, sólo este tipo de componentes serán los que aparezcan en los vértices, pues el resto de tipos de componentes serán descompuestos.

```
addInputs(CD comp) return (Map<Var, Set<CD>> inputs){
01    Map<Var, Set<CD>> inputs = new MapImp<Var, Set<CD>>()
02    addInputsAux(comp, inputs)
03    return inputs
 }

addOutputs(CD comp) return (Map<Var, CD> outputs){
04    Map<Var, CD> outputs = new MapImp<Var, CD>()
05    addOutputsAux(comp, outputs)
06    return outputs
 }
```

Figura A.4: Métodos addInputs y addOutputs

Al descomponer los componentes, sólo las asignaciones y asertos de la traza serán transformados a vértices, y se podrá determinar de forma más precisa, qué componentes concretos de la secuencia influyen en cada variable del sistema software.

- CD_Block. Cada bloque de componentes será descompuesto, y cada CD interno será transformado teniendo en cuenta de qué tipo es.
- CD_IfElse: Aunque estos componentes pueden incluir internamente hasta dos componentes de tipo CD_Block, se supondrá que los CD que se incorporan al grafo forman parte de la traza correcta, y que cada componente de tipo CD_IfElse sólo almacena el bloque asociado a la traza correcta. Dicho bloque será transformado tal como se comento en el punto anterior.
- CD_CCall. Una llamada a un constructor incluirá un componente de tipo CD_Block que contendrá la secuencia de componentes correspondientes al constructor. Además, los argumentos enviados al constructor (*arguments*), y los atributos inicializados por defecto (*fields*), se almacenarán como conjuntos de componentes de tipo CD_Asig; y por tanto también serán transformados a vértices del grafo.
- CD_MCall. Una llamada a un método incluirá un componente de tipo CD_Block que contendrá la secuencia de componentes correspondientes al método llamado, y un conjunto de asignaciones que corresponden con los argumentos enviados al método (*arguments*). Ambos serán descompuestos y transformados a vértices del grafo.

En la figura A.3 se muestra el método *addVertex* encargado de convertir cada CD a vértices de un grafo. Para realizar la transformación de cada componente recibido a vértices del grafo se han seguido las reglas expuestas anteriormente, ya que dicha transformación es diferente para cada tipo de componente.

```
addInputsAux(CD comp, Map<Var, Set<CD>> inputs){
01    if (comp instanceOf CD_Asig || comp instanceOf CD_Assert)
02       for (Var var: comp.inputs){
03          if (¬inputs.containsKey(var))
04             inputs.put(var, new Set<CD>())
05          inputs.get(var).add(comp)
06       }
07    if (comp instanceOf CD_Block)
08       addInputsSet(((CD_Block)comp).stats, inputs)
09    if (comp instanceOf CD_IfElse)
10       if (((CD_IfElse)comp).if != null)
11          addInputsAux(((CD_IfElse)comp).if, inputs)
12       else
13          addInputsAux(((CD_IfElse)comp).else, inputs)
14    if (comp instanceOf CD_CCall) {
15       CD_CCall cc = (CD_CCall) comp
16       addInputsSet(cc.fields, inputs)
17       addInputsSet(cc.arguments, inputs)
18       addInputsAux(cc.met, inputs) }
19    if (comp instanceOf CD_MCall) {
20       CD_MCall mc = (CD_MCall) comp
21       addInputsSet(mc.arguments, inputs)
22       addInputsAux(mc.met, inputs) }
 }
addInputsSet(Set<CD> scomp, Map<Var, Set<CD>> inputs){
23    for (CD c: scomp) addInputsAux(c, inputs)
 }
```

Figura A.5: Método addInputsAux y addInputsSet

El método *addVertex* recibe un CD (*comp*), que puede ser a su vez un bloque, y contener internamente otros CD; y un grafo (g), en el que una vez acabado el algoritmo se almacenarán los vértices obtenidos de transformar los CD. También existe el método *addVertexs* que se utiliza cuando en lugar de recibir un componente, se recibe un conjunto de componentes. En este caso, el método *addVertexs* se encarga de iterar sobre el conjunto de componentes, y realizar la transformación de cada uno de ellos llamando al método *addVertex*.

Tal como se puede observar en el algoritmo, sólo los CD de tipo CD_Asig y CD_Assert aparecerán como vértices del grafo de dependencias, ya que el objetivo al generar el grafo es tener una estructura que permita analizar de forma sencilla la influencia de las asignaciones y los asertos sobre el dominio de las variables del problema de diagnosis.

```
addOutputsAux(CD comp, Map<Var, Set<CD>> outputs){
01    if (comp instanceOf CD_Asig)
02       for (Var var: comp.outputs)
03          outputs.put(var, comp)
04    if (comp instanceOf CD_Block)
05       addOutputsSet(((CD_Block)comp).stats, outputs)
06    if (comp instanceOf CD_IfElse)
07       if (((CD_IfElse)comp).if != null)
08          addOutputsAux(((CD_IfElse)comp).if, outputs)
09       else
10          addOutputsAux(((CD_IfElse)comp).else, outputs)
11    if (comp instanceOf CD_CCall) {
12       CD_CCall cc = (CD_CCall) comp
13       addOutputsSet(cc.fields, outputs)
14       addOutputsSet(cc.arguments, outputs)
15       addOutputsAux(cc.met, outputs)
16    }
17    if (comp instanceOf CD_MCall) {
18       CD_MCall mc = (CD_MCall) comp
19       addOutputsSet(mc.arguments, outputs)
20       addOutputsAux(mc.met, outputs)
21    }
 }
addOutputsSet(Set<CD> scomp, Map<Var, CD> outputs){
22    for (CD c: scomp)
23       addOutputsAux(c, outputs)
 }
```

Figura A.6: Método addOutputsAux y addOutputsSet

Método addInputs y addOutputs

En el apartado A.1.1 se mostró la forma de obtener cuáles son las entradas y salidas de cada CD. Se supondrá que esta información está disponible, simplemente accediendo a las propiedades *inputs* y *outputs*, que contendrán respectivamente el conjunto de variables de entrada y salida de un CD.

En la figura A.5 se muestra el método encargado de asociar por cada variable, cuáles son los componentes que utilizan dichas variables como entradas (método *addInputs*), y el método encargado de asociar por cada variable, cuál es el componente encargado de establecer su valor, es decir cuál es el componente utiliza dicha variable como salida (método *addOutputs*). En el primer caso se recorren todos los componentes con sus correspondientes entradas, y en el segundo caso se hace lo mismo con las salidas. En

```
addEdges(Graph<CD, Var> g, Map<Var, Set<CD>> inputs, Map<Var, CD> outputs)
01    Iterator<Var> it = outputs.keySet().iterator()
02    for (it.hashNext(): Var v = it.next()){
03       if (inputs.containsKey(v)){
04          CD source = outputs.get(v)
05          g.addEdge(source, inputs.get(v), v)
07       }
08    }
 }
```

Figura A.7: Método addEdges

las figuras A.1.2 y A.6 se muestran los métodos auxiliares necesarios.

Es importante remarcar que cada variable que es considerada salida de un CD, es salida establecida sólo por dicho CD, pero puede ser recibida como entrada por varios CD. Por ese motivo cada salida es asociada a un sólo CD, y cada entrada puede tener asociados varios componentes.

Método addEdges

En la figura A.7 se muestra el método encargado de añadir al grafo las aristas. Cada arista une vértices del grafo que corresponden a CD que tendrán una variable en común, en el sentido de que para alguno de los componentes la variable es de salida y para los otros componentes la variable es de entrada. El método recibe el grafo (g) donde ya se han añadido los vértices correspondientes a los componentes, y dos funciones (*inputs* y *outputs*) donde están almacenadas respectivamente, de qué grupo de componentes se considera como entrada cada variable, y de qué componente se considera como salida cada variable.

Tal como se explicó anteriormente cada salida es generada por un único componente, y puede ser recibida como entrada por varios componentes. En el algoritmo, por cada salida v almacenada en la función, se recuperan todos los componentes que usan dicha salida v como entrada.

Ejemplo

En la figura A.8 se muestra un programa de ejemplo y su correspondiente caso de prueba. La traza seguida por el código fuente no proporciona los resultados correctos. Al usar el caso de prueba el resultado real obtenido es -1, cuando debería ser 1.

Aplicando la metodología propuesta para la diagnosis del código fuente, se obtendrían los CD que se muestran en la figura. Aplicando el algoritmo propuesto al ejemplo mostrado, se obtendría el grafo de dependencias que se muestra al final de la figura.

Código fuente

```
int dif_abs(int x, int y){
   int abs = y − x;
   if (y > x){
      abs = x − y;
   }
   return abs;
}
```

Caso de prueba

Entradas: x = 2, y = 3
Salidas: @result = 1

TCD

CD_MCall
arguments:
CD1 entradas:{} salidas:{x0}
sd: { P ∧ ¬AB(stat1, x) ⇒ (x0 = 2) }
CD2 entradas:{} salidas:{y0}
sd: { P ∧ ¬AB(stat1, y) ⇒ (y0 = 3) }
met:
CD3 entradas:{x0, y0} salidas:{abs0}
sd: { P ∧ ¬AB(stat2) ⇒ (abs0 = y0 - x0) }
CD4 entradas:{x0, y0} salidas:{abs1}
sd:
{ ¬P ⇒ P(block1) = false,
P ∧ ¬AB(stat3) ⇒ P(block1) = (y > x),
if ¬P(block1) then abs1 = abs0 }
if:
CD5 entradas:{x0, y0} salidas:{abs1}
sd: { P(block1) ∧ ¬AB(stat4) ⇒ abs1 = x0 - y0 }
CD6 entradas:{abs1} salidas:{@result}
sd: { P ∧ ¬AB(stat5) ⇒ (@result = abs1) }

Grafo de dependencias resultante

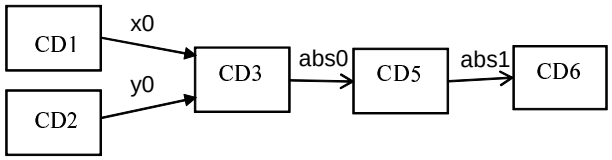


Figura A.8: Grafo de dependencias obtenido para la llamada al método dif_abs

A.2. Algoritmo para la determinación de los cluster

El algoritmo CD2Cluster mostrado en la figura A.9 es el encargado de determinar qué conjunto de CD influye en cada una de las variables de salida de un sistema software. El algoritmo recibe un conjunto de CD (*comp*) y devuelve una función (Map<Var, Set<CD>>), donde cada variable que es salida del sistema software tiene asociado el conjunto de CD que influyen sobre ella.

El primer paso es la transformación de los CD a un grafo dirigido g, donde cada vértice v_i representa un CD, c_i, y cada arista representa una variable común entre dos componentes. La forma de obtener este grafo se explicó con detalle en el apéndice A.1. Este grafo auxiliar permitirá determinar la influencia de cada CD en las variables del sistema software. Las variables que se han modificado en un CD se consideran salidas, y las variables que son leídas en los componentes se consideran entradas. Las aristas representan la transmisión del valor de una variable que es salida de un CD y entrada de otro CD diferente.

La idea del algoritmo es ir tratando los vértices del grafo que no sean alcanzables

```
CD2Cluster(Set<CD> comp) return (Map<Var, Set<CD>>) {
01   Graph<CD, Var> g = compToGraph(comp)
02   Map<Var, Set<CD>> retMap = new MapImp<Var, Set<CD>>()
03   Map<CD, Set<CD>> auxMap = new MapImp<CD, Set<CD>>()
04   for (CD c1: g.vertices())
05     auxMap.put(c1, new Set<CD>().add(c1))
06   while (¬ g.isEmpty()) {
07     Iterator it = g.vertices().iterator()
08     for (it.hasNext() : c1 = it.next()) {
09       if (g.adjacentsTo(c1).size() = 0) {
10         foreach (CD c2 : g.adjacents(c1))
11           auxMap.get(c2).addAll(auxMap.get(c1))
12         if (g.adjacents(c1).isEmpty()) {
13           foreach (Var v : c1.outputs)
14             foreach (CD c2 : auxMap.get(c1)){
15               if (¬retMap.containsKey(v))
16                 retMap.put(v, new Set<CD>())
17               retMap.get(v).add(c2)
18             }
19         }
20         it.remove()
21       }
22     }
23   }
24   return retMap
 }
```

Figura A.9: Algoritmo CD2Cluster encargado de enlazar cada componente con el conjunto de salidas del sistema software sobre las que influye

por ningún otro vértice, e ir eliminándolos a medida que sean tratados. Una vez eliminados, aparecerían nuevos vértices que no serán alcanzables por ningún otro vértice, y el tratamiento se repetirá para estos vértices. El proceso acabaría una vez eliminados todos los vértices del grafo.

El tratamiento de los vértices que se van eliminando consistirá en almacenar en una función auxiliar, denominada *auxMap* (creada en la línea 3), qué componentes son los que influyen en las salidas que genera. En las salidas de un componente c_i influyen el propio c_i, y recursivamente, todos los componentes c_j que generan entradas para c_i. Para obtener estas dependencias se irá completando, a medida que se eliminan los vértices del grafo, una función auxiliar *auxMap* que a cada componente c_i le hace corresponder el conjunto de componentes que pueden influir en cada salida de c_i.

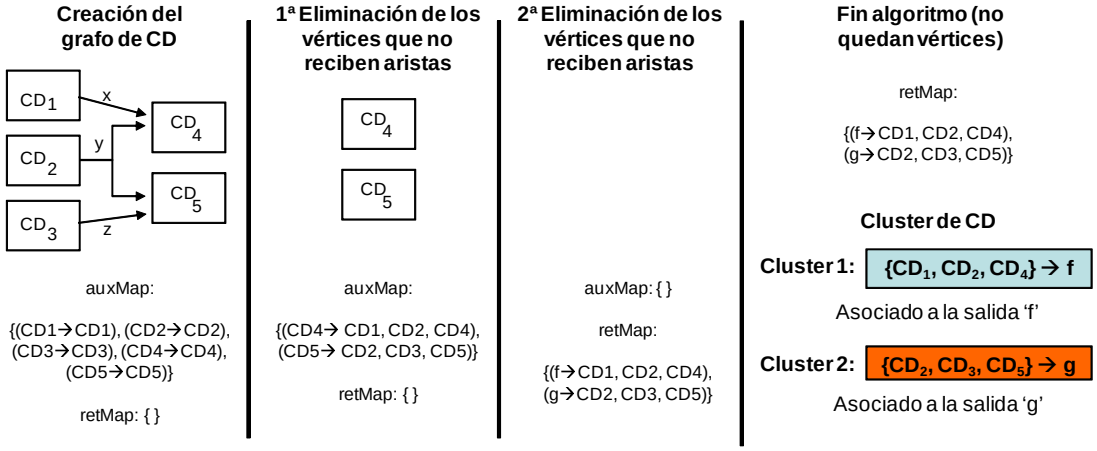


Figura A.10: Clusters de CD obtenidos para el ejemplo propuesto

Este tratamiento de los vértices se realiza de las líneas 6 a la 23. Los métodos del grafo *adjacents* y *adjacentsTo* devuelven respectivamente el conjunto de vértices que son alcanzables desde un vértice dado, y el conjunto de vértices que pueden alcanzar a un vértice dado.

En la línea 3 se genera la función *auxMap* encargada de guardar la correspondencia entre cada CD y el conjunto de componentes que influye en sus salidas. Al principio (líneas 4 y 5) se inicializará dicha función haciendo corresponder cada componente c_i con un conjunto que incluya a sólo a c_i, lo que supone que cada CD puede influir en sus propias salidas.

Previamente, en la línea 2 se inicializa la función *retMap* (Map<Var, Set<CD>>) que será devuelta en la línea 24. Por cada variable que se considere salida del sistema software, se almacenará el conjunto de componentes que influyen en ella. El contenido de esta estructura de datos se irá completando a medida que se vayan eliminando los vértices del grafo. Concretamente, de las líneas 13 a la 18, para cada vértice que sea eliminado, se recorrerán las salidas en las que el componente almacenado en dicho vértice influye. Con cada una de las salidas v, se actualizará el conjunto de componentes que influyen en ella, y que está almacenado en la función *retMap*. Para saber qué componentes influyen en las variables, se utiliza la información almacenada en la función *auxMap*.

El algoritmo hace un recorrido de todos los vértices del grafo una única vez, por tanto es de orden lineal con respecto al número de vértices, o lo que es lo mismo, con respecto al número de componentes. A continuación se muestra cuál sería el resultado del algoritmo para el ejemplo mostrado anteriormente (figura A.1).

Ejemplo A.2. El primer paso del algoritmo CD2Cluster mostrado en la figura A.9 es la transformación de los componentes a un grafo dirigido g, donde cada vértice v_i representa un componente c_i, y cada arista representa una variable común entre dos CD.

Tal como se ha dicho anteriormente, el algoritmo para obtener este grafo aparece en el apéndice A.1. La figura A.10 está dividida en cuatro trozos. El primero, que está en la parte izquierda de la figura, muestra cuál sería el grafo resultante, y cuál sería el estado de las variables *auxMap* y *retMap* que se utilizarán en el algoritmo antes de empezar a eliminar los vértices de dicho grafo.

En la siguiente escena de la figura se muestra como quedaría el grafo y las variables reseñadas, una vez eliminados los primeros tres vértices a los que no llega ninguna arista dirigida. Los vértices borrados son los correspondientes a los componentes CD_1, CD_2 y CD_3. En la penúltima escena de la figura se muestra como quedaría el grafo y las variables reseñadas, una vez eliminados los dos últimos vértices a los que no llega ninguna arista dirigida. Los vértices borrados son los correspondientes a los componentes CD_4 y CD_5. Como las variables de salida de estos CD no son utilizadas por otros CD, se consideran salidas del sistema software. En concreto las variables de salida del sistema software son f del CD_4, y g del CD_5. Estas variables son las que se incluyen en la función *retMap*, junto con el conjunto de componentes que influyen en ellas (ver algoritmo CD2Cluster, figura A.9).

En la última escena de la figura se muestran los dos cluster resultantes, que influyen respectivamente en las variables f y g.

Apéndice B

Ejemplos utilizados en las pruebas

A continuación se muestran los ejemplos utilizados en las pruebas presentadas en el capítulo 7.

B.1. ToyProgram

Nombre:	**ToyProgram**
Código fuente correcto:	S_1 `int x = a * c;` S_2 `int y = b * d;` S_3 `int z = c * e;` S_4 `int f = x + y;` S_5 `int g = y + z;`
Propiedades	If = 0, While = 0, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {a = 2, b = 3, c = 3, d = 2, e = 2} Código: `toyprogram()` Salidas: {f = 12, g = 12}
Defecto 1:	S_5: `int g = y * z;`

El programa ToyProgram es una adaptación del sistema polybox [39], utilizado muchos trabajos de diagnosis basada en modelos (DX). Es iterativo y sólo tiene sentencias de asignación. Sobre este ejemplo se propone un caso de prueba, y un defecto sobre una de las asignaciones, concretamente la sentencia (S_5).

B.2. IndexOf

El método que se muestra a continuación (IndexOf) permite obtener la posición en la que se encuentra un elemento dentro del intervalo [*start*, *end*] de un array *a*. El programa es iterativo y se basa en un único bucle. En este caso se propone un defecto sobre la condición del bucle, y un caso de prueba que permite detectar dicho defecto.

Nombre:	**IndexOf**
Código fuente:	S_1 `public int indexOf(Integer e, int start, int end){` S_2 `int i = start;` S_3 `int index = -1;` S_4 `while (i <= end && index == -1){` S_5 `if (this.a[i].equals(e)){` S_6 `index = i;` `}` S_7 `i = i + 1;` `}` S_8 `return index;` `}`
Propiedades	If = 1, While = 1, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: { a = {4, 2, 5, 6, 1, 5, 1, 4, 5, 0}, e = 5, start = 0, end = 9} Código: `indexOf(5, 0, 9)` Salidas: {index = 2}
Defecto 1:	S_4: `while (i <= end)`

B.3. Raiz cuadrada

Nombre:	**RaizCuadrada**
Código fuente correcto:	S_1 `int raizCuadrada(int N) {` S_2 `int r = 0;` S_3 `int s = 1;` S_4 `int t = 1;` S_5 `while (s <= N) {` S_6 `r = r + 1;` S_7 `s = s + t + 2;` S_8 `t = t + 2;` A_1 `assert: s = (r + 1)`^{2}`;` A_2 `assert: t = 2 * r + 1;` `}` S_9 `return r;` `}`
C. de prueba 1:	Entradas: {N = 11} Código: `raizCuadrada(N)` Salidas: {r = 3}
Defecto 1:	S_2: `r = 1;`

El programa propuesto permite obtener la raíz cuadrada entera del número entero recibido como entrada. El programa es iterativo y se basa en un único bucle. Sobre

dicho bucle existen dos invariantes que deben cumplirse en cada iteración.

El objetivo de este ejemplo es ver las diferencias en los resultados que la metodología de diagnosis proporciona en función de si utiliza o no la información contenida en los invariantes del bucle. Sobre este ejemplo se propone un caso de prueba, y se introducirá un defecto sobre una de las sentencias de asignación que se encuentran antes del bucle, concretamente la sentencia (S_2).

B.4. Polinomio

El programa Polinomio calcula el valor de un polinomio para un determinado valor de la variable x. Para ello recibe como parámetros de entrada los coeficientes del polinomio, el valor de la variable x, y el número de coeficientes que deben utilizarse. El programa es iterativo y se basa en un único bucle. En este caso se propone un defecto y un caso de prueba que permite detectar dicho defecto.

Nombre:	**Polinomio**

Código fuente correcto:	S_1 `double valor(double[] coef, double x,`
	`int length) {`
	S_2 `int i = 0;`
	S_3 `double valor = 0.0;`
	S_4 `double acum = 1.0;`
	S_5 `while (i < length){`
	S_6 `valor = valor + coef[i]*acum;`
	S_7 `acum = acum * x;`
	S_8 `i = i + 1;`
	`}`
	S_9 `return valor;`
	`}`

Propiedades	If = 0, While = 1, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {coef = {1.94, 3.13, 10.24}, x = 4.12, length = 3} Código: `valor(coef, x, length)` Salidas: {valor = 188.653456}
Defecto 1:	S_6: `valor = valor + acum;`

B.5. CuentaBancaria

La clase CuentaBancaria permite modelar las operaciones más típicas de una cuenta corriente, como por ejemplo realizar ingresos o retirar efectivo. Dispone de varios métodos, algunos con precondiciones y postcondiciones. Se han propuesto tres defectos y tres casos de prueba que permiten detectar dichos defectos.

Nombre:	**CuentaBancaria (I)**
Código fuente correcto:	

```
     class CuentaBancaria {
S1     double balance;
S2     double minBalance;
S3     double ret = 0.01;

A1     post: minBalance > 0 && balance == 0.0
S4     CuentaBancaria(double bal,
         double minB) {
S5       minBalance = minB;
S6       balance = bal;
       }

A2     pre: income > 0
A3     post: balance == @old(balance) + income
S7     void deposit(double income) {
S8       balance = balance + income;
       }

A4     pre: withdrawal > 0
A5     post: balance == @old(balance) - withdrawal
S9     void withdraw(double withdrawal) {
S10      balance = balance - withdrawal;
       }

S11    double balanceUpdate(double interest
         double accountCost) {
S12      double points = 0;
S13      if (balance < accountCost) {
S14        points = 0;
         } else {
S15        withdrawAccountCost(accountCost);
S16        points = getPoints();
S17        payInterest(interest);
         }
S18      return points;
       }

S19    void payInterest(double interest) {
S20      if (balance > minBalance) {
S21        balance = balance + balance * interest;
         }
       }
```

Nombre:	**CuentaBancaria (II)**
	S_{22} `double getPoints(){` S_{23} `return balance * ret;` `}` S_{24} `void withdrawAccountCost(double accountCost){` S_{25} `balance = balance - accountCost;` `}` `}`
Propiedades	If = 2, While = 0, Metodos = 7, Clases = 1
C. de prueba 1:	Entradas: {minBalance = 100.0, initialBalance = 310.0, interest = 0.02, accountCost = 10.0} Código: `CuentaBancaria acc = new CuentaBancaria(initialBalance, minBalance);` `double points = acc.balanceUpdate(interest, accountCost);` Salidas: {acc.balance = 312, points = 3}

C. de prueba 2:	Entradas: {minBalance = 100.0, initialBalance = 310.0, withdrawal = 300.0, interest = 0.02 accountCost = 15.0, income = 100.0}
	Código: `CuentaBancaria acc = new CuentaBancaria(initialBalance, minBalance);`
	`acc.withdraw(withdrawal);`
	`double points = acc.balanceUpdate(interest, accountCost);`
	`acc.deposit(income);`
	Salidas: {acc.balance = 110, points = 0}

C. de prueba 3:	Entradas: {minBalance = 100.0, initialBalance = 320.0, withdrawal = 200.0, interest = 0.05, accountCost = 20.0, income = 40.0}
	Código: `CuentaBancaria acc = new CuentaBancaria(initialBalance, minBalance);`
	`acc.withdraw(withdrawal);`
	`double points = acc.balanceUpdate(interest, accountCost);`
	`acc.deposit(income);`
	Salidas: {acc.balance = 140, points = 1}

Defecto 1: Defecto 2: Defecto 3:	S_{15}: `withdrawAccountCost(interest);` S_{13}: `if (balance <0)` S_{20}: `if (balance >= minBalance)`

Además de las operaciones de ingreso (*deposit*) y retirada (*withdraw*), al final de cada periodo establecido por el banco, se ejecuta la operación de actualización del saldo (*balanceUpdate*). Esta operación a su vez utiliza los métodos encargados de realizar el descuento en el saldo de los gastos de la cuenta (*withdrawAccountCost*), calcular los puntos promocionales en función del saldo (*getPoints*), y abonar el interés por el saldo (*payInterest*). Aunque el interés se suelen calcular sobre el saldo medio de las cuentas bancarias, para simplificar el ejemplo, se ha utilizado directamente el saldo en el momento correspondiente a la actualización. Los puntos son acumulados por los clientes, y posteriormente éstos pueden cambiarlos por regalos que ofrece el banco.

B.6. SumDiferencia

Nombre:	**SumDiferencia**

Código fuente correcto:	S_1 `int sumDiferencia(int x, int y){`
	S_2 `int m1;`
	S_3 `int m2;`
	S_4 `if (x >= y){`
	S_5 `m1 = y;`
	S_6 `m2 = x;`
	`}else{`
	S_7 `m1 = x;`
	S_8 `m2 = y;`
	`}`
	A_1 `assert: m2 >= m1`
	S_9 `int z = m2 - m1;`
	S_{10} `int s = m1 * (z + 1);`
	S_{11} `while (z > 0){`
	S_{12} `s = s + z;`
	S_{13} `z = z - 1;`
	`}`
	S_{14} `return s;`

C. de prueba 1:	Entradas: {x = 4, y = 4} Código: `sumdiferencia(x, y)` Salidas: {s = 4}
C. de prueba 2:	Entradas: {x = 3, y = 7} Código: `sumdiferencia(x, y)` Salidas: {s = 25}
C. de prueba 3:	Entradas: {x = 5, y = 2} Código: `sumdiferencia(x, y)` Salidas: {s = 14}
Defecto 1:	S_{10}: `int s = m1 * z;`

El programa propuesto permite obtener el sumatorio de los números enteros incluidos en el rango [*x*, *y*], siendo los límites de dicho rango los valores recibidos como entradas. El programa es iterativo y se basa en una sentencia selectiva, varias asignaciones y un bucle. Sobre los valores las variables intermedias *m1* y *m2*, se ha establecido un aserto que debe cumplirse en la ejecución del programa. Sobre este ejemplo se proponen 3 casos de prueba que permiten detectar un mismo defecto.

B.7. Problema de la mochila

Nombre:	**MochilaVZ**

Código fuente correcto:	S_1 `void voraz(int n, int pesoMax,`
	`int[] pesos, int[] valores, int[] sol){`
	S_2 `int i = 0;`
	S_3 `int valor = 0;`
	S_4 `while (i < n){`
	S_5 `int aux = pesoMax / pesos[i];`
	S_6 `sol[i] = aux;`
	S_7 `valor = valor + aux * valores[i];`
	S_8 `pesoMax = pesoMax - aux * pesos[i];`
	S_9 `i = i + 1;`
	`}`
	S_{10} `return valor;`
	`}`

Propiedades	If = 0, While = 1, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {n = 3, pesoMax = 10, pesos = {7, 5, 2}, valores = 14, 10, 2} Código: `voraz(n, pesoMax, pesos, valores, sol)` Salidas: {sol = {1, 0, 1}, valor = 16}
Defecto 1:	S_8: `pesoMax = pesoMax + aux * pesos[i];`

El programa propuesto resuelve el problema de la mochila utilizando la técnica voraz. En concreto se dispone de una serie de objetos que tienen asociado un peso y un valor. La idea es almacenar los objetos en una mochila sin sobrepasar el peso máximo que dicha mochila puede soportar. Se reciben como entrada el número de objetos disponibles (n), el peso máximo que puede soportar la mochila (*pesoMax*), los pesos de cada uno de los objetos (*pesos*), los valores de cada uno de los objetos (*valores*). Se supondrá que sólo existe un objeto de cada tipo. La solución al problema se almacenará en el array *sol* (recibido como último parámetro de entrada).

El algoritmo voraz intentará insertar objetos en la solución mientras quede peso disponible en la mochila. Para que la solución sea lo más optima posible, el orden de inserción de los objetos en la mochila debe ser en función de la mejor relación

valor/peso. Se supondrá que los objetos vienen ordenados según dicho criterio, por lo que no es necesario realizar dicha ordenación. El programa es iterativo y se basa en un bucle. Para el ejemplo se ha propuesto un defecto y un caso de prueba.

B.8. BusqBinaria

Nombre:	**BusqBinaria**

Código fuente correcto:	S_1 `int binSearch(int value, int[] values,`
	`  int start, int length) {`
	S_2 `  int result = -1;`
	S_3 `  while ((length > 0) && (result < 0)){`
	S_4 `  if (length == 1){`
	S_5 `  if (values[start] == value){`
	S_6 `  result = start;`
	`  }else {`
	S_7 `  length = 0;`
	`  }`
	`  }else{`
	S_8 `  int m = start + (length / 2);`
	S_9 `  if (value < values[m]){`
	S_{10} `  length = length / 2;`
	`  }else{`
	S_{11} `  length = length - (length / 2);`
	S_{12} `  start = m;`
	`  }`
	`  }`
	`  }`
	S_{13} `  return result;`
	`  }`

Propiedades	If = 3, While = 1, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {value = 7, values = {1, 7, 13, 20, 21}, start = 0, length = 5} Código: `binSearch(value, values, start, length)` Salidas: {result = 1}
Defecto 1:	S_6: `result = value;`

El programa propuesto realiza la búsqueda binaria (o búsqueda dicotómica) de un valor en un array ordenado. Recibe como parámetros de entrada el valor búscado (*value*), los elementos del array (*values*), y el intervalo ([*start*, *length*]) del array donde se puede encontrar el valor buscado. Devuelve la posición del array donde se encuentra el valor buscado, o -1 si no se encuentra en el array.

B.9. NTree

Nombre:	**NTree**
Código fuente correcto:	class NTree{
	S_1 int info;
	S_2 NTree tizq;
	S_3 NTree tder;
	S_4 NTree(int info) {
	S_5 this.info = info;
	S_6 this.tizq = null;
	S_7 this.tder = null;
	}
	S_8 NTree(int info,
	S_9 NTree izq, NTree der) {
	S_{10} this.info = info;
	S_{11} this.tizq = izq;
	S_{12} this.tder = der;
	}
	S_{13} int count(int elem){
	S_{14} int ret = 0;
	S_{15} if (info == elem){
	S_{16} ret = ret + 1;
	}
	S_{17} if (tizq != null){
	S_{18} int countIzq = tizq.count(elem);
	S_{19} ret = ret + countIzq;
	}
	S_{20} if (tder != null){
	S_{21} int countDer = tder.count(elem);
	S_{22} ret = ret + countDer;
	}
	S_{23} return ret;
	}
	S_{24} NTree max(){
	S_{25} NTree tMax = this;
	S_{26} if (tizq != null){
	S_{27} NTree izq = tizq.max();
	S_{28} if (izq.info > tMax.info){
	S_{29} tMax = izq;

Nombre:	**NTree (II)**

```
            }
          }
S30       if (tder != null){
S31         NTree der = tder.max();
S32         if (der.info > tMax.info){
S33           tMax = der;
            }
          }
S34       return tMax;
        }
        ...
      }

      class Prueba {
        ...
A1      pre: @pre(count(@pre(raiz.max()).info)) == 1
A2           && @pre(raiz.max()).info > 0
A3      post: count(raiz.max().info) == 1
S35     int incMax(Tree raiz, int max){
S36       int ret = 0;
S37       if (raiz.info == max){
S38         int aux = raiz.info + 1;
S39         raiz.info = aux;
S40         ret = aux;
          }
S41       if (raiz.tizq != null){
S42         int auxIzq = incMax(raiz.tizq, max);
S43         if (auxIzq > ret){
S44           ret = auxIzq;
            }
          }
S45       if (raiz.tder != null){
S46         int auxDer = incMax(raiz.tder, max);
S47         if (auxDer > ret){
S48           ret = auxDer;
            }
          }
S49       return ret;
        }
        ...
      }
```

Nombre:	**NTree (III)**
Propiedades	If = 12, While = 0, Metodos = 5, Clases = 2
C. de prueba 1:	Entradas: {raiz = new NTree(2, new NTree(7, new NTree(4), new NTree(5)), new NTree(9, null, new NTree(1)))} Código: `NTree maxTree = raiz.max();` `int newMax = incMax(raiz, maxTree.info);` Salidas: {ret = 10, raiz.tder.info = 10}
Defecto 1:	S_{38}: `int aux = raiz.info;`

La clase NTree permite guardar una serie de números enteros en forma de árbol. Dispone de varios métodos, entre los que se encuentra:

- El método *count* que permite saber el número de veces que un número se encuentra almacenado en el árbol.
- El método *max* que devuelve el subárbol que tenga como raíz el mayor de todos los números almacenados en el árbol.

Se proporciona un caso de prueba para ejercitar el método *incMax* de la clase Prueba. El objetivo de dicho método es incrementar el valor del mayor de los elementos almacenados en el árbol, suponiendo que dicho elemento es único (no se repite en el resto del árbol). El método *incMax* es recursivo. Para el ejemplo se ha propuesto un defecto y un caso de prueba.

B.10. Problema de las monedas

El programa propuesto resuelve el problema de las monedas utilizando la técnica de programación dinámica. En concreto se dispone de una serie de monedas que tienen asociado un valor. Se debe encontrar el número de monedas mínimo necesario para devolver una cantidad concreta (sin sobrepasarla). Se reciben como entrada el número de monedas disponibles (n), la cantidad a devolver (*cantidad*) y los valores de cada una de las monedas (*valores*). La solución al problema será la última posición de la matriz *sol* , que se recibe como último parámetro de entrada, y que será rellenada por el algoritmo. Tendrá tantas filas como tipos de monedas existen, y tantas columnas como indique el parámetro *cantidad* más uno.

Para que la solución sea la optima, los valores de las monedas están ordenados de menor a mayor. El programa es iterativo y se basa en dos bucles anidados. Se supondrá que no hay límite en el número disponible de monedas de cada tipo. Para el ejemplo se ha propuesto un defecto y un caso de prueba que permite su detección.

Nombre:	**MonedasPD**

Código fuente correcto:	S_1 `int pd(int n, int cantidad, int[] valores, int[][] sol){`
	S_2 `int i = 0;`
	S_3 `while (i < n){`
	S_4 `sol[i][0] = 0;`
	S_5 `i = i + 1;`
	`}`
	S_6 `int j = 1;`
	S_7 `while (j <= cantidad){`
	S_8 `if (valores[0] > j){`
	S_9 `sol[0][j] = cantidad + 1;`
	S_{10} `}else{`
	S_{11} `sol[0][j] = sol[0][j - valores[0]] + 1;`
	`}`
	S_{12} `j = j + 1;`
	`}`
	S_{13} `i = 1;`
	S_{14} `while (i < n){`
	S_{15} `j = 1;`
	S_{16} `while (j <= cantidad){`
	S_{17} `if (valores[i] > j){`
	S_{18} `sol[i][j] = sol[i - 1][j];`
	`}else{`
	S_{19} `int aux = sol[i][j - valores[i]] + 1;`
	S_{20} `if (sol[i - 1][j] < aux){`
	S_{21} `sol[i][j] = sol[i - 1][j];`
	`}else{`
	S_{22} `sol[i][j] = aux;`
	`}`
	`}`
	S_{23} `j = j + 1;`
	`}`
	S_{24} `i = i + 1;`
	`}`
	S_{25} `return sol[(n - 1)][cantidad];`
	`}`

Propiedades	If = 3, While = 4, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {n = 3, cantidad = 8, valores={1, 4, 6}, Código: `pd(n, cantidad, valores, sol)` Salidas: {@ret = 2}
Defecto 1:	S_{21}: `sol[i][j] = aux;`

B.11. SumaSubSecMaxima

El método que se muestra a continuación recibe un objeto de tipo *Problema*. Un problema contiene un array con valores de tipo entero (positivos, negativos o cero) y un intervalo $[i, j]$ que delimita las posiciones del array que son accesibles. El objetivo es encontrar el valor de la subsecuencia cuyo sumatorio sea el máximo de todas las subsecuencias posibles dentro de los limites marcados por el intervalo. En este caso se han introducido dos defectos a la vez, y un único caso de prueba que permite detectar ambos defectos.

Nombre:	**SumaSubSecMaxima**
Código fuente:	S_1 `public void sumaSubSecMax(Problema p){` S_2 `int k = (p.i +p.j)/2;` S_3 `int s1 = p.a[k];` S_4 `int s11 = p.a[k];` S_5 `int r1 = k - 1;` S_6 `while (r1 >= p.i){` S_7 `s11 = s11 + p.a[r1];` S_8 `if (s11 > s1){` S_9 `s1 = s11;` `}` S_{10} `r1 = r1 - 1;` `}` S_{11} `p.maxL = s1;` S_{12} `int s2 = p.a[k];` S_{13} `int s22 = p.a[k];` S_{14} `int r2 = k + 1;` S_{15} `while (r2 <= p.j){` S_{16} `s22 = s22 + p.a[r2];` S_{17} `if (s22 > s2){` S_{18} `s2 = s22;` `}` S_{19} `r2 = r2 + 1;` `}` S_{20} `p.maxR = s2;` `}`
Propiedades	If = 2, While = 2, Metodos = 0, Clases = 0
C. de prueba 1:	Entradas: {p.a = {0, -2, 3, -4, 5, 0, -5}, p.i = 0, p.j = 6} Código: `sumaSubSecMax(p)` Salidas: {p.maxL = -1, p.maxR = 1}
Defecto 1: Defecto 2:	S_5: `int r1 = k;` S_{14}: `int r2 = k;`

Bibliografía

[1] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, San Diego, California, 1988.

[3] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical Fault Localization for Dynamic Web Applications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 265–274, Cape Town, South Africa, 2010.

[4] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

[5] Melinda-Carol Ballou. Improving Software Quality to Drive Business Agility, 2008.

[6] Federico Barber and Miguel A Salido. Introduction to constraint programming. *Inteligencia Artificial Iberoamerican Journal of AI*, 7(20):13–30, 2003.

[7] Mike Barnett, Manuel Fähndrich, Peter Müller, K. Rustan, M. Leino, Wolfram Schulte, and Herman Venter. Specification and Verification : The Spec # Experience. *Communications of the ACM*, 2010.

[8] Kent Beck. *Test Driven Development: By Example*. Addison Wesley, 2003.

[9] Bernhard Beckert and Vladimir Klebanov. Proof Reuse for Deductive Program Verification. In *2nd International Conference on Software Engineering and Formal Methods (SEFM04)*, pages 77–86, Beijing,China, 2004.

[10] Peter Van Beek. On the minimality and decomposability of constraint networks. In *AAAI*, pages 447–452, 1992.

[11] B. Beizer. *Software testing techniques*. Van Nostrand, 1990.

[12] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[13] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI*, pages 108–113, 1993.

[14] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, pages 309–315, 2001.

[15] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[16] R.V. Binder. *Testing Object-Oriented Systems : Models, Patterms, and Tools*. Object Technology Series. Addison Wesley, 2000.

[17] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[18] G. Biswas, M.-O. Cordier, J. Lunze, L. Travé-Massuyés, and M. Staroswiecki. Diagnosis of complex systems: Bridging the methodologies of the FDI and DX communities. *IEEE Transactions on Systems, Man and Cybernetics*, 34(5):2159–2162, 2004.

[19] Blast. http://mtc.epfl.ch/software-tools/blast/.

[20] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, 1:75–88, January 1984.

[21] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat : Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA02)*, number July, pages 123–133, Roma, Italy, 2002.

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

[23] R Caballero, C. Hermanns, and H. Kuchen. Algorithmic Debugging of Java Programs. *Electronic Notes in Theoretical Computer Science*, 177:75–89, 2007.

[24] J. Cassar and M. Staroswiecki. A structural approach for the design of failure detection and identification systems. In *IFAC-IFIP-IMACS Conf. on Control of Industrial Processes, Belfort, France*, 1997.

[25] CBMC. http://www.cprover.org/cbmc/.

[26] Zhengqiang Chen and Xu Baowen. Slicing object-oriented java programs. *SIGPLAN Notices*, 36(4):33–40, 2001.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(244), 1986.

[28] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. In *27 th International Conference on Software Engineering*, pages 342–351, St. Louis, MO, USA, 2005.

[29] Hélène Collavizza and Michel Rueher. Exploring Different Constraint-Based Modelings. In *CP*, pages 49–63, Providence, 2007.

[30] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV : A Constraint-Programming Framework. In *CP*, pages 327–341, Sydney, 2008.

[31] L. Console and O. Dressler. Model-based diagnosis in the real world: Lessons learned and challenges remaining. In *Proceedings IJCAI99*, pages 1393–1400, 1999.

[32] Marie-Odile Cordier, Philippe Dague, François Lévy, Jacky Montmain, Marcel Staroswiecki, and Louise Travé-massuyès. A comparative analysis of AI and control theory approaches to model-based diagnosis. In *14th European Conference on Artificial Intelligence*, pages 136–140, Berlin, Germany, 2000.

[33] Marie-odile Cordier, Philippe Dague, François Lévy, Jacky Montmain, Marcel Staroswiecki, and Louise Travé-massuyès. Conflicts Versus Analytical Redundancy Relations : A Comparative Analysis of the Model Based Diagnosis Approach From the Artificial Intelligence and Automatic Control Perspectives. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 34(5):2163–2177, 2004.

[34] J. Cuadrado-Gallego, D. Rodríguez, M. Garre, and R. Rejas. Epistemological and Ontological Representation in Software Engineering. *LNCS*, 4488:1162–1169, 2007.

[35] Ron Cytron, Ferrante Kenneth, Barry K. Rosen, and Mark N. Wegman. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[36] Valentin Dallmeier and Andreas Zeller. Lightweight Bug Localization with AMPLE. In *6th International Symposium on Automated Analysis-Driven Debugging*, pages 99–104, Monterey, California, USA, 2005.

[37] Adnan Darwiche. Model-based diagnosis using structured system descriptions. *Artificial Intelligence Research*, 8(1):165–222, 1998.

[38] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

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

[40] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI*, pages 412–417, 1997.

[41] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[42] Giorgio Delzanno and Andreas Podelski. Model Checking in CLP. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, pages 223–239, The Netherlands, 1999.

[43] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[44] Michael J. Dent and Robert E. Mercer. Minimal forward checking. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 432–438, 1994.

[45] O. Dressler and P. Struss. *The Consistency-based Approach to Automated Diagnosis of Technical Devices*. Principles of Knowledge Representation. CSLI Publications, Stanford, 1996.

[46] E. H. Dürr and J. van Katwijk. Vdm++. a formal specification language for object-oriented design. *Proceeding of CompEuro92. In Computer Systems and Software Engineering*, pages 214–219, 1992.

[47] Desmond DŚouza. *Catalysis - Objects, Components, and Frameworks with UML*. Addison-Wesley, 1998.

[48] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Second IEEE International Conference on Fuzzy Systems*, volume 2, pages 1131 – 1136, 1993.

[49] R. Duke, P. King, G. A. Rose, and G. Smith. The object-z specification language. *In Technology of Object-Oriented Languages and Systems (TOOLS 5)*, pages 465–483, 1991.

[50] V. Encontre. Empowering the developer to be a tester tool. In *Int. Symposium on Software Testing and Analysis, Industry panel*, Rome, 2002. ACM SIGSOFT.

[51] M D Ernst, J Cockrell, W G Griswold, and D Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *21st International Conference on Software Engineering (ICSE99)*, pages 213–224, Los Angeles, CA, USA, 1999.

[52] Michael D Ernst, Jake Cockrell, William G Griswold, David Notkin, and Ieee Computer Society. Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[53] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[54] ESC/Java. http://kind.ucd.ie/products/opensource/ESCJava2/.

[55] EUREKA. http://www.ai-lab.it/eureka/.

[56] Hélène Fargier and Jérôme Lang. Uncertainty in constraint satisfaction problems: a probalistic approach. In *ECSQARU*, pages 97–104, 1993.

[57] Y.el Fattah and C. Holzbaur. A CLP Approach to Detection and Identification of Control System Component Failures. In *Proceedings of the Workshop on Qualitative and Quantitative Approaches to Model-Based Diagnosis, Second International Conference on Intelligent Systems Engineering*, pages 97–107, 1994.

[58] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[59] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[60] Paul M. Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy : A survey and some new results. *Automatica*, 26(3):459–474, 1990.

[61] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[62] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[63] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[64] Francisco José Galán Morillo and José Miguel Cañete Valdeón. Métodos Formales Orientados a Objetos. *Novática: Revista de la Asociación de Técnicos de Informática ISSN 0211-2124*, 165, 2003.

[65] Philippe Galinier and Jin-Kao Hao. Tabu search for maximal constraint satisfaction problems. In *Third International Conference on Principles and Practice of Constraint Programming (CP)*, pages 196–208, 1997.

[66] José García-Fanjul, Claudio de la Riva, and Javier Tuya. Generation of conformance test suites for compositions of web services using model checking. In *TAIC PART, Windsor, United Kingdom*, 2006.

[67] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.

[68] M. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[69] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[70] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, USA, 1996.

[71] D.E Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[72] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002.

[73] Alex Groce and Willem Visser. Model Checking Java Programs using Structural Heuristics. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA02)*, pages 12–21, Roma, Italy, 2002.

[74] Alex Groce and Willem Visser. Heuristics for Model Checking Java Programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[75] Grégoire Hamon, Leonardo De Moura, and John Rushby. Generating Efficient Test Sets with a Model Checker. In *Second International Conference on Software Engineering and Formal Methods (SEFM04)*, pages 261–270, Beijing, China, 2004.

[76] Sudheendra Hangal and Monica S Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, 2002.

[77] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[78] Mark Harman, Zheng Li, Phil Mcminn, Jeff Offutt, and John Clark. The Journal of Systems and Software TAIC PART 2007 and Mutation 2007 special issue editorial. *The Journal of Systems & Software*, 82(11):1753–1754, 2009.

[79] A Hartman. Is ISSTA Research Relevant to Industry ? In *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 205–206, Rome, Italy, 2002.

[80] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[81] U. Heller and P. Struss. G^+de - the generalized diagnosis engine. *DX-2001, 12th International Workshop on Principles of Diagnosis*, pages 79–86, 2001.

[82] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[83] Jinbo Huang and Adnan Darwiche. On Compiling System Models for Faster and More Scalable Diagnosis. In *Aaai Conference On Artificial Intelligence*, number Darwiche, pages 300–306, Pittsburgh, Pennsylvania, 2005.

[84] IEEE. *IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology.* 1990.

[85] R. Isermann. Supervision, fault-detection and fault-diagnosis methods, an introduction. *Control Engineering Practice*, 5(5):639–652, 1997.

[86] Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of Multithreaded Object-Oriented Programs with Invariants. In *Third Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, Newport Beach, CA, USA, 2004.

[87] Tao Jiang, Nicolas Gold, Mark Harman, and Zheng Li. Locating dependence structures using search based slicing. *Journal of Information and Software Technology (IST)*, Volume 50(Issue 12):1189–1209, 2008.

[88] Cliff B. Jones. *Systematic software development using VDM.* Prentice-Hall, Inc., 1990.

[89] James A Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, USA, 2005.

[90] Deepak Kapur. Automatically Generating Loop Invariants. In *IMACS Intl. Conf. on Applications of Computer Algebra (ACA2004)*, Beaumont, Texas, 2004.

[91] K. Kask. New search heuristics for max-CSP. In *Sixth International Conference on Principles and Practice of Constraint Programming, Singapore*, pages 262–277, September 2000.

[92] M. Khalil. An experimental comparason of software diagnosis methods. In *25th Euromicro Conference*, 1999.

[93] Sunwoo Kim, John A. Clark, and John A. McDermid. *Investigating the effectiveness of object-oriented strategies with the mutation method.* Kluwer Academic Publishers, 2001.

[94] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[95] Johan De Kleer and Brian C Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[96] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[97] Laura Kovács. Reasoning Algebraically About P-Solvable Loops. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 249–264, 2008.

[98] Mattias Krysander and Mattias Nyberg. Structural analysis utilizing mss sets with application to a paper plant. In *13th Int. Workshop Principles Diagnosis*, pages 51–57, Semmering, Austria, 2002.

[99] J. E. Labra Gayo, J. M. Cueva Lovelle, M. C. Luengo Díaz, A. Cernuda Del Rio, and L. Joyanes Aguilar. Comparación de Técnicas de Especificación Semántica de Lenguajes de Programación. In *SISOFT 2001. Simposio Iberoamericano de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento*, Santa Fé de Bogotá. Colombia. ISBN: 95897030-3-8, 2001.

[100] Javier Larrosa and Pedro Meseguer. Algoritmos para satisfacción de restricciones. *Revista Iberoamericana de Inteligencia Artificial*, 20:31–42, 2003.

[101] Javier Larrosa and Pedro Meseguer. Restricciones Blandas : Modelos y Algoritmos. *Revista Iberoamericana de Inteligencia Artificial*, 20:69–82, 2003.

[102] J. Larrossa and P. Meseguer. Partition-based lower bound for MAX-CSP. In *Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, pages 303–315, October 1999.

[103] Janusz Laski. Programming faults and errors: Towards a theory of software incorrectness. *Annals of Software Engineering*, 4:79–114, 1997.

[104] Janusz Laski and William Stanley. *Software Verification and Analysis*. Springer, 2009.

[105] Yves Le Traon, Farid Ouabdesselam, Chantal Robach, and Benoit Baudry. From diagnosis to diagnosability : axiomatization , measurement and application. *Journal of Systems and Software*, 65:31–50, 2003.

[106] Gary T Leavens, Albert L Baker, and Ruby Clyde. Preliminary Design of JML : A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes. ACM*, 31(3):1–38, 2006.

[107] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[108] Bixin Li. Analyzing information-flow in java program based on slicing technique. *SIGSOFT Software Engineering Notes*, 27(5):98–103, 2002.

[109] Donglin Liang and Kai Xu. Debugging Object-Oriented Programs with Behavior Views. In *Sixth International Workshop on Automated Debugging (AADEBUG)*, pages 19–21, Monterey, California, USA, 2005.

[110] Li Lin and Yunfei Jiang. The computation of hitting sets : Review and new algorithms. *Information Processing Letters*, 86:177–184, 2003.

[111] Gary Lindstrom, Peter C Mehlitz, and Willem Visser. Using Java PathFinder. In *Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 444–456. Springer Verlag Lecture Notes in Computer Science vol. 3707, 2005.

[112] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[113] J. R. Lyle and M. Weiser. Automatic bug location by program slicing. In *Second International Conference on Computers and Applications*, pages 877–883, Beijing, China, June 1987.

[114] Alan K. Mackworth. Consistency in networks of relations. Technical report, Univ. of B.C. Vancouver, Canada, 1975.

[115] Felip Manyà and Carla P. Gomes. Técnicas de resolución de problemas de satisfacción de restricciones. *Revista Iberoamericana de Inteligencia Artificial*, 19:169–180, 2003.

[116] Chengying Mao, Xiaohua Hu, and Yansheng Lu. Towards a Software Diagnosis Method Based on Rough Set Reasoning. In *8th IEEE International Conference on Computer and Information Technology (CIT2008)*, number 2, pages 718–723, Sydney, 2008.

[117] Ricardo Peña Marí. *Diseño de programas. Formalismo y abstracción.* Prentice Hall, 1998.

[118] Vincent Massol and Ted Husted. *JUnit in Action.* Manning Publications Co., 2003.

[119] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Model-based debugging of java programs. In *AADEBUG*, August 2000.

[120] W Mayer and M Stumptner. Evaluating models for model-based debugging. In *ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, volume pages, pages 128–137, 2008.

[121] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verifation and Reliability*, 11(2):81–96, 2001.

[122] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

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

[124] B. Meyer. *Eiffel: the Language.* Prentice Hall, 1992.

[125] B. Meyer. The significance of components. *In Beyond Objects, Software Development*, 7(11), November 1999.

[126] R. Mitchell and J. McKim. *Design by Contract, by Example.* Addison Wesley Professional, 2001.

[127] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2):225–233, 1986.

[128] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *ECAI*, pages 651–656, 1988.

[129] G. J. Myers. *The art of software testing.* John Wiley, 1979.

[130] Syeda Nessa, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. Software Fault Localization Using N -gram Analysis. In *Wireless Algorithms, Systems, and Applications 2008*, pages 548–559, Dallas, USA, 2008.

[131] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, 29(2):167–193, 1999.

[132] Object Management Group (OMG). *Object Constraint Language Specification.* Chapter 7 of OMG Unified Modeling Language Specification. Version 1.3, March 2000 (first edition). 2000.

[133] Object Management Group (OMG). *Object Constraint Language OMG Available Specification Version 2.0.* 2006.

[134] Roy Osherove. *The Art of Unit Testing: With Examples.* Manning Publications Co., 2009.

[135] R. J. Patton and J. Chen. A review of parity space approaches to fault diagnosis. *IFAC-SAFEPROCESS Symposium*, 1991.

[136] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-fai Wong, Yoav Zibin, D Ernst, and Martin Rinard. Automatically Patching Errors in Deployed Software. In *SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, New York, USA, 2009.

[137] Mark G Pleszkoch and Richard C Linger. Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior. In *International Conference on System Sciences*, volume 00, pages 1–10. Society Press, 2004.

[138] G. D. Plotkin. A structural approach to operational semantics. In *Technical Report DAIMI FN-19. Computer Science Department, Aarhus University, Aarhus, Denmark*, 1981.

[139] Andreas Podelski and Andrey Rybalchenko. ARMC : The Logical Choice for Software Model Checking with Abstraction Refinement. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007*, pages 245–259, Nice, France, 2007.

[140] Roger S. Pressman. *Software Engineering.* McGraw-Hill, 1998.

[141] Roger S. Pressman. *Ingeniería de software: Un enfoque práctico (Séptima edición).* McGraw-Hill, 2010.

[142] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[143] Belarmino Pulido and Carlos J Alonso. Revisión del concepto de posible conflicto como técnica de pre-compilación. *Revista Iberoamericana de Inteligencia Artificial*, 5(14):41–53, 2001.

[144] Belarmino Pulido and Carlos Alonso González. Possible Conflicts : A Compilation Technique for Consistency-Based Diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 34(5):2192–2206, 2004.

[145] J.B. Pulido. Posibles conflictos como alternativa al registro de dependencias en línea para el diagnóstico de sistemas continuos. *PhD. degree, Universidad de Valladolid*, 2000.

[146] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P Reiss. Automated Fault Localization Using Potential Invariants 1 Debugging in Principle : An Example. In *Fifth International Workshop on Automated Debugging (AADEBUG2003)*, Ghent, 2003.

[147] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[148] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S Mukherjee. Perturbation-based Fault Screening. In *IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, USA, 2007.

[149] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence 32*, 1:57–96, 1987.

[150] Enric Rodríguez-Carbonell and Deepak Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. *Static Analysis.Lecture Notes in Computer Science.*, 3148:280–295, 2004.

[151] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic Generation of Polynomial Loop Invariants : Algebraic Foundations. In *Static Analysis.Lecture Notes in Computer Science.*, pages 266–273, Santander, Spain, 2004.

[152] S. Ntafos. On random and partition testing. In *ACM SIGSOFT international symposium on Software testing and analysis(ISSTA98)*, pages 42–48. ACM Press, 1998.

[153] Martin Sachenbacher and Brian Williams. Diagnosis as Semiring-based Constraint Optimization. In *16th Eureopean Conference on Artificial Intelligence (ECAI2004)*, pages 873–877, Valencia, Spain, 2004.

[154] René Santaolaya-salgado, Liliana Badillo-sánchez, and Olivia Graciela Fragosodiaz. Process for Contract Extraction. In *The Third International Conference on Software Engineering Advances*, pages 37–42, Sliema, Malta, 2008.

[155] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints?". In *UAI*, pages 268–275, 1992.

[156] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *International Joint Conference on Artificial Intelligence (IJCAI95)*, 1995.

[157] Tobias Schuele and Klaus Schneider. Global vs . Local Model Checking : A Comparison of Verification Techniques for Infinite State Systems. In *2nd International Conference on Software Engineering and Formal Methods (SEFM04)*, pages 67–76, Beijing, China, 2004.

[158] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., 1989.

[159] M. Staroswiecki and P. Declerk. Analytical redundancy in non linear interconnected systems by means of structural analysis. In *IFAC Advanced Information Processing in Automatic Control (AIPAC-89)*, pages 51–55, Nancy, France, June 1989.

[160] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes, International Summer School in Computer Programming. Reprinted in* Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000*, 1967.

[161] Markus Stumptner and Franz Wotawa. Coupling csp decomposition methods and diagnosis algorithms for tree-structured systems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 388–393, Acapulco, Mexico, 2003.

[162] J. Tremblay and G.A. Cheston. *Data Structures and Software Development in an Object Oriented Domain: Eiffel Edition*. Prentice Hall, 2001.

[163] Eclipse Website. http://www.eclipse.org/.

[164] SPIN Website. http://www.spinroot.com/.

[165] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Transition-Based Directed Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009*, pages 186–200, York, UK, 2009.

[166] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. In *ISSTA '10 Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, Trento, Italy, 2010.

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

[168] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[169] W Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79:891–903, 2006.

[170] Franz Wotawa. A variant of Reiter's hitting-set algorithm. *6th International Symposium on Automated Analysis-Driven Debugging*, 79(1):45–51, 2001.

[171] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

[172] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-Based Debugging or How to Diagnose Programs Automatically. In *IEA/AIE*, pages 746–757, Cairns, Australia, 2002.

[173] K. Lano y H. Houghton. *The Z++ manual*. 1994.

[174] Andreas Zeller. Yesterday , my program worked . Today , it does not . Why ? In *Software Engineering - ESEC/FSE'99*, number September, pages 253–267, Toulouse, France, 1999.

[175] Jian Zhang, Chen Xu, and Xiaoliang Wang. Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, number 60125207, pages 242–250, Beijing, China, 2004.