Generación automática de objetivos de prueba a partir de casos de uso mediante partición de categorías y variables operacionales

Javier J. Gutiérrez, María J. Escalona, Manuel Mejías,
Jesús Torres, Arturo Torres-Zenteno

Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla
Avd. Reina Mercedes s/n. 41012. España
{javierj, escalona, risoto, jtorres}@lsi.us.es
arturo.torres.zenteno@everis.com

Resumen

Este trabajo complementa y amplia nuestros trabajos anteriores sobre generación de pruebas a partir de casos de uso presentando un proceso que, de manera sistemática y automática, permite generar objetivos de prueba a partir de casos de uso especificados en un lenguaje no formal. Este proceso aplica el método de categoría-partición y el patrón Use Case Test Pattern, el cual usa variables operacionales. Además se presenta los algoritmos necesarios para la automatización del proceso propuesto, acompañados de un caso práctico.

1. Introducción

La realización de pruebas es una tarea vital para el desarrollo de sistemas software de calidad. Siendo de mucha utilidad, aún estas tareas continúan siendo trabajosas y tediosas; circunstancias que muchas veces suelen conducirla y considerarla como impacto negativo en la producción de software. Por tanto, es necesario que existan procesos de pruebas que sean sistematizados y automatizados que subsanen estos inconvenientes.
Existen diferentes tipos de pruebas durante todo el ciclo de vida del desarrollo de software. En este trabajo nos centraremos en las pruebas del sistema. Las pruebas del sistema son un procedimiento de caja negra para verificar la satisfacción de los requisitos del sistema a prueba [5]. Las pruebas de sistema se dividen a su vez en las pruebas funcionales y las pruebas no funcionales (configuración, instalación, integridad, seguridad, desempeño, usabilidad, entre otros).

Nuestra propuesta especifica el comportamiento del sistema mediante casos de uso. Los casos de uso ofrecen una visión general del sistema y son fáciles de estudiar y validar por parte de usuarios no expertos en ingeniería del software. Es también muy común en la industria del software identificar casos de prueba a partir de los casos de uso [22]. En fases tempranas del desarrollo, cuando los requisitos están siendo descubiertos, definidos y negociados, es mucho más sencillo modificar los casos de uso definidos como texto libre o texto estructurado que requisitos formales.

En un trabajo anterior [15] presentamos un proceso sistemático y una herramienta de soporte para generación automática de objetivos de prueba a partir de la descripción textual de un caso de uso. Este proceso sistematizaba el análisis de escenarios de casos de uso o posibles caminos de ejecución, la cuál es una de las dos técnicas para la generación de objetivos identificada a partir de estudios comparativos de distintas propuestas [8] y [13]. Este trabajo expone un proceso diferente, aunque puede ser complementario con la presentada en [15], para la generación de objetivos de prueba y basado en el método de categoría-partición [1], [21] y variables operacionales [3].

A pesar de su antigüedad, la técnica de la partición en categorías, llamada en adelante CPM, se usa actualmente en muchos escenarios y ha sido incluida dentro del UML Testing Profile [19]. Esta técnica ha sido aplicada, por ejemplo, en las pruebas de software orientadas a objetos [20], de

XML [7] y, también, para la generación de pruebas a partir de los casos de uso. En este caso, Binder [3] propone un patrón de prueba llamado Extended Use Case Test Pattern consistente en la identificación de las variables operacionales de un caso de uso, las cuales tiene una definición similar al concepto de categoría, la división del dominio de cada variable, análogamente a las particiones, y la generación de combinaciones entre las distintas particiones. El resultado de este patrón es un conjunto de objetivos de prueba definidos como una colección de valores para las distintas variables operacionales.

Sin embargo, esta técnica de prueba presenta algunas deficiencias. En primer lugar los casos de uso se escriben en lenguaje sin formato, por lo que la identificación de variables y sus particiones es difícil de sistematizar y automatizar. En ocasiones hemos detectado que un mismo escenario puede representarse bien como una nueva variable, bien como una nueva partición de una variable ya identificada. Esta técnica no nos indica cuál de las dos opciones es la más adecuada. La generación de restricciones entre las distintas variables y la generación de combinaciones válidas también son pasos que deben realizarse a mano. Así mismo, existe una falta de sistematización y automatización de todo el proceso.

Todo esto nos lleva a proponer en este trabajo un nuevo proceso sistemático y automático para identificar variables operacionales, identificar sus particiones, descubrir restricciones entre ellas y generar objetivos de prueba definidos como combinaciones de valores.

Además, los distintos resultados de este proceso (variables, particiones, restricciones y combinaciones) pueden refinarse manualmente por parte de los ingenieros de prueba. Al igual que el patrón Extended Use Case Test Pattern, nuestro punto de partida son plantillas de casos de uso escritas en lenguaje no formal.

La organización de este trabajo se describe a continuación. En la sección 2 se resumen brevemente nuestros trabajos anteriores, los cuáles son el punto de partida para la propuesta que presentamos en la sección 3. En esa sección, se muestra cómo aplicar la técnica CPM y el patrón propuesto por Binder en [3] a los casos de uso de una manera sistemática y automática. En la sección 4 se comentan los resultados obtenidos de la aplicación del proceso propuesto. Seguidamente, en la sección 5 se comentan brevemente otros trabajos relacionados. Finalmente, en la sección 6, se exponen las conclusiones.

2. Un resumen de la generación de objetivos de prueba a partir de casos de uso

En las referencias [13], [14] y [15] se explica en detalle el proceso, modelos y algoritmos para generar un diagrama de actividades a partir de un caso de uso y obtener objetivos de prueba definidos como recorridos sobre el diagrama de actividades. En los siguientes párrafos se resumen las ideas fundamentales.

A lo largo del proceso, se mostrará un ejemplo tomado a partir de un caso de uso de un sistema de gestión de eventos on-line desarrollado por la empresa CodeCharge (www.codecharge.com) y utilizado con su permiso.

Una técnica ampliamente utilizada en la industria para definir casos de uso son las plantillas. Una plantilla combina la prosa con una estructura concreta. La generación sistemática de objetivos de prueba implica una serie de restricciones. Una de ellas es la necesidad de definir un modelo concreto de plantillas de casos de uso que pueda ser manipulado de manera sistemática y automática sin perder la ventaja de usar texto en prosa. Como modelo hemos utilizado el modelo de requisitos propuesto en la metodología Navigational Development Technique (NDT) [9], la cual ofrece un modelo de requisitos completo, formal y flexible. NDT propone una plantilla tabular para definir requisitos funcionales mediante casos de uso. Hemos elegido el modelo de requisitos y las plantillas de NDT por dos motivos principales. En primer lugar, este modelo está basado en un metamodelo formal definido mediante UML [9], [19]. En segundo lugar, el modelo de requisitos de NDT ha sido aplicado en proyectos reales y complejos, con resultados muy exitosos [10] y [11].

A continuación, se construye de manera automática un diagrama de actividades para representar el comportamiento del caso de uso. En la tabla 1 se muestra un caso de uso para realizar una búsqueda sobre el catálogo de enlaces definido como un documento XML que sigue el formato de la herramienta de soporte (el DTD

correspondiente puede descargarse de la página de la herramienta).

Este caso de uso es el mismo que se incluyó como ejemplo en el trabajo [14] con el fin de comparar los resultados. El diagrama de actividades generado automáticamente se muestra en la figura 1.

Actualmente, nuestras herramientas de soporte sólo pueden trabajar con casos de uso definidos en lengua inglesa, por lo que todos los ejemplos mostrados a continuación están en dicho idioma.
Después, los objetivos de prueba se obtienen de manera sistemática. Los objetivos de pruebas se definen como caminos a través del modelo de comportamiento o como diagramas de actividades dónde sólo existe un único camino. Los objetivos de prueba obtenidos se muestran en la tabla 5.

3. Aplicación del método categoría-partición a los casos de uso

El proceso presentado en este artículo consta de cuatro actividades: construcción de un modelo de actividades y un conjunto de objetivos a partir de un caso de uso, identificación de variables operaciones y particiones, identificación de restricciones y generación de combinaciones de valores. La primera actividad ya ha sido resumida en la sección 2. Las tres restantes se definen en las siguientes secciones.

Tabla 1. Caso de uso de ejemplo para la búsqueda de enlaces.

```
<useCase id="Search link by description">
  <description> A use case for searching a set of links by their description
   and to show the results. </description>
  <mainSequence>
     <step id="1"> The visitor asks the system for searching links by
      description.
     </step>
     <step id="2"> The system asks for the description. </step>
     <step id="3"> The visitor introduces a piece of the description of
       the searched links. </step>
     <step id="4"> The system searches for the links which matches up with
       the introduced description. </step>
     <step id="5"> The system shows the results found.</step>
    </mainSequence>

     <alternativeSteps>
       <astep id="3.1"> At any time, the visitor may cancel the search,
         then this use case ends. </astep>
       <astep id="4.1"> If the visitor introduces an empty piece of text,
         then the system searches for all stored links and step 4.2 is executed.
       </astep>
       <astep id="5.1">  If no links were found, then the system shows a
         message and this use case ends. </astep>
     </alternativeSteps>

     <errorSteps>
       <estep id="4.2"> If the system finds any error performing the
         search, then an error message is shown and this use case ends.
       </estep>
     </errorSteps>
   </useCase>
```

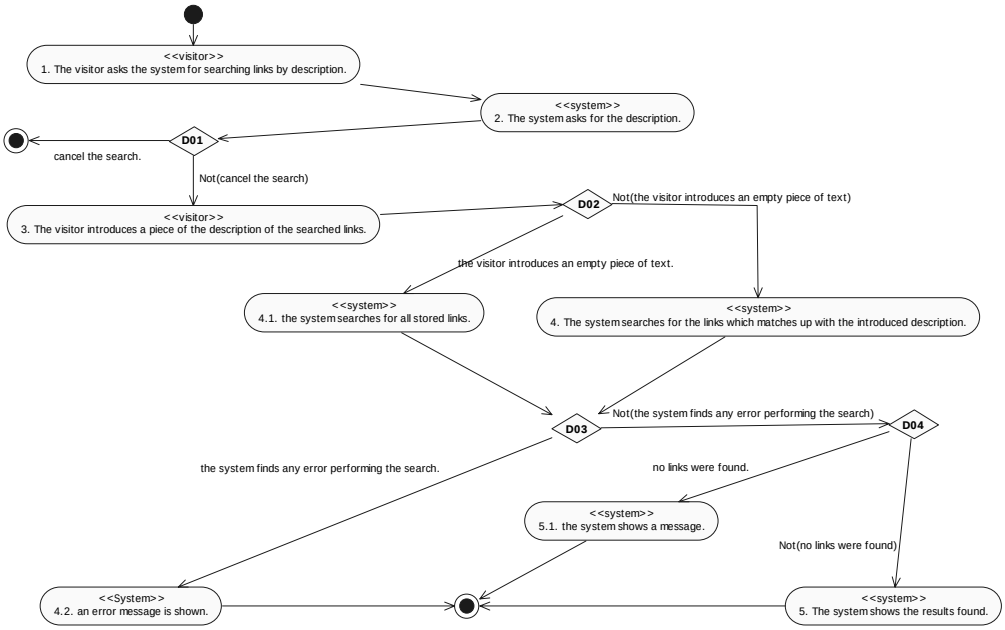

Figura 1. Diagrama de actividades obtenido a partir del caso de uso.

3.1. Identificación de variables operacionales y particiones

El punto de partida son los diagramas de actividades generados por el proceso descrito en el punto anterior para cada caso de uso.

Las variables operacionales se identifican, o bien en aquellas acciones que representen una entrada o salida de información del sistema, o bien en los nodos de decisión del diagrama de actividades.

Los dominios de dichas variables operacionales se identifican a partir de los nodos de decisión del diagrama de actividades. Cada una de las transiciones salientes del nodo decisión será una partición distinta del dominio de cada variable operacional evaluada en dicha transición. A continuación, se muestra el Algoritmo 1, que recorre un diagrama de actividades e identifica las variables operacionales y sus particiones

```
vl: [0..x] of VARIABLE
pl: [0..y] of PARTITION

function traverse (in ad:AD)
  for each n:NODE in AD
    traverseNode (n)
  end-for
end-function
function traverseNode(in n:NODE)
  if ( isActivity(n) )
    if ( containsVariable(n)
      addVariable(vl, n)
    end-if
    traverseNode(getNextNode(n))
  else-if
    v : VARIABLE
    extractVariale(v, n)
    if( notContains(vl, v) )
      addVariable(vl, v)
    end-if
    getAlternatives(n, as)
    for each a:ALTERNATIVE in as

setVariablePartition(pl,v,a)

traverseNode(getNextNode(a))
    end-for
  end-else-if
end-function
```

Algoritmo 1. Algoritmo para la generación de variables operacionales y particiones.

Tabla 2. Variables obtenidas automáticamente a partir del diagrama de actividades.

```
<variables>
 <variable id="V_4">
 <description> The links which
matches up with the introduced
description.</description>
  <partition id="P_1">
   <condition> No links were found.
</condition>
  </partition>
  <partition id="P_2">
   <condition> Not(no links were
         found)  </condition>
  </partition>
 </variable>
 <variable id="V_3">
 <description>a piece of the
description of the searched
links.</description>
  <partition id="P_1">
   <condition> the visitor
     introduces an empty piece of
     text. </condition>
  </partition>
  <partition id="P_2">
   <condition> Not(the visitor
     introduces an empty piece of
      text)  </condition>
  </partition>
 </variable>
 <variable id="V_D02">
  <partition id="P_1">
   <condition> the system finds any
     error performing the search.
   </condition>
  </partition>
  <partition id="P_2">
   <condition> Not(the system finds
     any error performing the
search)
   </condition>
  </partition>
 </variable>
 <variable id="V_D01">
  <partition id="P_1">
   <condition> cancel the search.
</condition>
  </partition>
  <partition id="P_2">
   <condition> Not(cancel the
search)  </condition>
  </partition>
 </variable>
</variables>
```

A continuación, se describe brevemente la misión de las funciones auxiliares del Algoritmo 1. La función *getNextNode*, devuelve el siguiente nodo que se visita. La función *getAlternatives* toma como parámetro de entrada una decisión y devuelve la lista de alternativas de dicho nodo. La función *containsVariables* utiliza patrones lingüísticos [9] para detectar variables operacionales. Los patrones implementados actualmente en la herramienta se muestran en la tabla 3. Al estar estos patrones almacenados en archivos externos, pueden modificarse o añadirse más patrones fácilmente. En este ejemplo, el patrón 3 detecta la variable V_4 en el paso 4 del caso de uso de la tabla 1.

Tabla 3. Patrones para la detección de variables.

	Patrón
1	`loads(.*)`
2	`recovers (.*)`
3	`searches for (.*)`

El resultado del algoritmo 1 es un listado de variables con sus correspondientes particiones. En la tabla 2 se muestra el resultado mediante un documento XML para el diagrama de actividades de la figura 1 generado con nuestra herramienta de soporte (el DTD correspondiente puede descargarse de la página de la herramienta).

Dos de las variables de la tabla 2: V_3 y V_4, se han obtenido de las actividades 3 y 4 del diagrama de actividades. Las decisiones D03 y D04 han permitido identificar dos particiones relevantes para cada una. Las otras dos variables, V_D01 y V_D02, se han obtenido a partir de las decisiones D01 y D02. Es posible refinar el resultado obtenido a mano, bien añadiendo variables adicionales o refinando las particiones.

3.2. Identificación de restricciones

Muy a menudo, los valores que toma una variable operacional establecen restricciones sobre los valores que pueden tomar otras variables operacionales. Por ejemplo, en el conjunto de variables y particiones de la tabla 2, si la variable V_D01 toma el valor de cancelar (partición P_1), el caso de uso finaliza y ninguna de las variables restantes necesita tomar ningún valor. Es necesario identificar y tener en cuenta

este tipo de restricciones para generar combinaciones válidas y para reducir el número de combinaciones posibles.

Nuestra propuesta es capaz de identificar automáticamente alguna de estas restricciones. En concreto, se van a identificar restricciones que indiquen cuando no es necesario asignar un valor a una variable operacional A ya que, debido al valor que toma otra variable operacional B, A nunca será utilizada, como se ha visto en el párrafo anterior. Este tipo de restricciones evitará combinar valores de variables que nunca vayan a ser utilizados.

Para la identificación de este tipo de restricciones, en primer lugar será necesario construir un mapa de dependencias para luego realizar la construcción de restricciones.

3.2.1. Generación del mapa de dependencias

Un mapa de dependencias es una matriz cuadrada donde cada fila y columna corresponde a una variable operacional y donde cada celda sólo puede contener un 0 o un 1. La intersección de la variable operacional en la de la fila X con la variable operacional en la fila Y, indica si existe algún posible valor para X que permita a un camino alcanzar la variable Y (un 1), o si no existe ninguna manera de llegar a la variable Y (un 0). Por definición, no existe dependencia de una variable consigo misma. El mapa de dependencias del diagrama de actividades de la figura 1 generado mediante la herramienta de soporte se muestra en la tabla 4.

Tabla 4. Mapa de dependencias del ejemplo.

	V_D01	**V_D02**	**V_3**	**V_4**
V_D01	0	1	1	1
V_D02	0	0	1	1
V_3	0	0	0	1
V_4	0	0	0	0

El mapa de dependencias se construye examinando todos los posibles caminos del diagrama de actividades. El algoritmo para extraer todos los caminos de un diagrama de actividades que representa el comportamiento de un caso se uso se definió en [14]. El algoritmo implementado para construir mapas de dependencias se muestra en el Algoritmo 2.

```
ps : [1..n] of PATH
m : [1..n][1..n] of {0,1}
c1, c2: INTEGER

fill(m, 0)
for each p:PATH in ps
  for c1 = 1 to getNumOfNodes(p)
    n1:NODE
    n1 = getNode(p, c1)
    if ( isDesicion(n1) )
      for c2 = c1 to
        getNumberOfNodes(p)
       n2:NODE
        if (isDesicion(n2))
          m[c1][c2] = 1
        end-if
      end-for
    end-if
  end-for
end-for
```

Algoritmo 2. Algoritmo para la construcción del mapa de dependencias.

Las funciones auxiliares se describen a continuación. La función *fill* es una función auxiliar que llena una matriz con el valor indicado. La función *getNumOfNodes* devuelve el número de nodos (actividades y decisiones) por las que pasa dicho camino. La función *getNode*, devuelve un nodo concreto del camino, ya sea una acción o una desición.

3.2.2. Construcción de restricciones

A partir del conjunto de variables operacionales y particiones, del mapa de dependencias de las variables operacionales y del conjunto de caminos se puede obtener de manera automática un conjunto de restricciones. Para ello, se aplica la siguiente regla: si una variable operacional A permite alcanzar un conjunto de variables operacionales CV y encontramos un camino P dónde A toma un valor perteneciente a la partición PD de la variable A y dicho camino no contiene al subconjunto SCV de CV, entonces, para cualquier valor de la partición PD de la variable A, las variables del subconjunto SCV no toman ningún valor. A continuación se muestra un ejemplo de esta regla. Aplicando esta regla al diagrama de actividades de la figura 1 y al conjunto de valores y particiones obtenidos en las secciones anteriores, se ve que la variable V_D01 permite alcanzar el conjunto de variables {V_D02, V_3, V_4}. Sin embargo, existe un

camino (fila 7, tabla 6) donde V_D01 toma un valor de la partición P_1 y dicho camino no contiene el subconjunto {V_D02, V_3, V_4} (que, en este ejemplo, coincide con todo el conjunto de variables dependientes). Por este motivo, se identifica la siguiente restricción: Si V_D01 vale P_01 ("The visitor cancles the search") entonces V_D02, V3 y V4 no toman ningún valor. A continuación se presenta el Algoritmo 3, para calcular este tipo de restricciones.

```
tos : [1..n] of TESTOBJECTIVE
vs : [1..m] of VARIABLE
m : [1..n][1..n] of {0,1}

for each to:TESTOBJECTIVE in tos
 svs : [1..x] of VARIABLE
 getVariablesInTO (to, vs, svs)
 if ( notEmpty(svs) )
  for each vaux:VARIABLE in svs
   dvs : [1..x] of VARIABLE
   getDepVarsFor(vaux,dvs,m)
   if ( notEmpty(dvs) )
    for each daux:VARIABLE in dvs
     if (notContains(svs, daux))
      addWithoutVal(vaux, daux);
     end-if
    end-for
   end-if
  end-for
 end-if
end-for
```

Algoritmo 3. Algoritmo para la generación de restricciones.

La variable *tos* son todos los objetivos de prueba, *vs* son todas las variables identificadas y *m* es el mapa de dependencias generado aplicando el Algoritmo 2.

El método *getVariablesInTO* devuelve como salida el subconjunto de variables de *vs* (en la variable *svs*) que están contenidas en el objetivo. La función *getDepVarsFor* devuelve todas las variables que dependen de la variable indicada como primer parámetro. La función *addWithoutVal* añade una restricción por la cuál no es necesario calcular valor para la variable indicada como segundo parámetro.

Al igual que en los pasos anteriores, es posible complementar estas restricciones con nuevas restricciones añadidas a mano por los ingenieros de prueba.

3.3. Generación de combinaciones de valores

A partir de los resultados de los pasos anteriores, es posible aplicar distintas técnicas de combinaciones para generar conjuntos de valores de prueba. Con la ayuda de los algoritmos anteriores, la herramienta de soporte ha generado un script en lenguaje BeanShell (tabla 5) para calcular todas las posibles combinaciones del caso práctico que cumplan las restricciones generadas en la sección anterior. Se ha omitido la definición de variables y funciones auxiliares del script.

El código de este script funciona de manera similar al lenguaje TSL definido para el método CPM [1]. El resultado de la ejecución de este script se muestra en la captura de pantalla de la figura 2. El número total de combinaciones para cuatro variables con dos particiones cada una es de 16. Sin embargo, aplicando las dos restricciones detectadas automáticamente e indicadas en el script de la tabla 5, dichas combinaciones se reducen a 7.

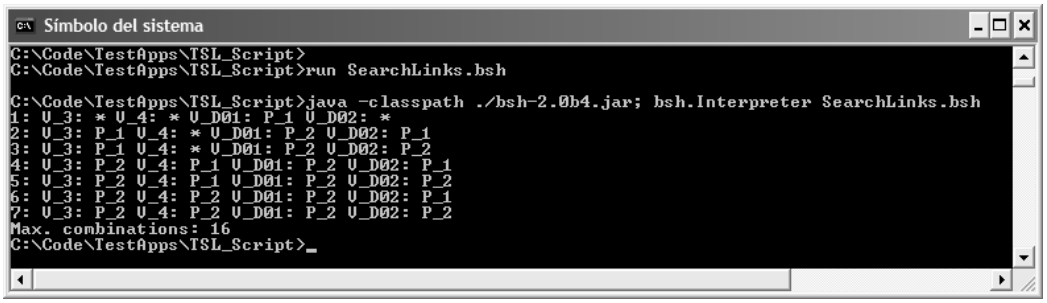

Figura 2. Cálculo de las combinaciones.

Tabla 5. Script para la generación de combinaciones de prueba para el caso práctico.

```
String[] d_V_3 = { "P_1" , "P_2" };
String[] d_V_4 = { "P_1" , "P_2" };
String[] d_V_D01 = { "P_1" , "P_2" };
String[] d_V_D02 = { "P_1" , "P_2" };

for (c_V_3=0; c_V_3<2;c_V_3++ ) {
  for (c_V_4=0; c_V_4<2;c_V_4++ ) {
    for (c_V_D01=0; c_V_D01<2;c_V_D01++ ) {
      for (c_V_D02=0; c_V_D02<2;c_V_D02++ ) {
        V_3=d_V_3[c_V_3];
        V_4=d_V_4[c_V_4];
        V_D01=d_V_D01[c_V_D01];
        V_D02=d_V_D02[c_V_D02];

        // Properties
        boolean P_V_D01 = false;
        boolean P_V_3 = false;
        if (V_D01 == "P_1")
          P_V_D01 = true;
        if (V_3 == "P_1")
          P_V_3 = true;

        // Values
        if (P_V_D01)
          V_3 =  V_D02 =  V_4 =  "*";
        if (P_V_3)
          V_4 =  "*";

        tmp =  V_3  +   V_4  +   V_D01  +   V_D02   ;
        if ( isNew(tmp) ) {
          System.out.print((++id)+":");
          System.out.print(" V_3: "+V_3);
          System.out.print(" V_4: "+V_4);
          System.out.print(" V_D01: "+V_D01);
          System.out.print(" V_D02: "+V_D02);
          System.out.print("\n");
} } } } }
```

Estas combinaciones indican el estado en que debe estar el sistema y la información que debe introducir el usuario para poder probar un escenario del caso de uso.

Por ejemplo, para probar el escenario principal del caso de uso, que corresponde con la combinación 7, no se debe cancelar la búsqueda (V_01 toma valor en P_2), se debe introducir un texto no vacío (V_3 toma valor en P_2), el sistema no debe encontrar ningún error (V_02 toma valor en P_2), y el sistema debe encontrar algún resultado (V_4 toma valor en P_2).

4. Relación entre combinaciones de valores de prueba y objetivos de prueba

En nuestro trabajo anterior [14] se obtuvieron siete objetivos de prueba identificando siete objetivos de prueba a partir del caso de uso (tabla 1). Aplicando la técnica de categoría-partición también se han obtenido siete combinaciones distintas y válidas de valores de prueba.

Comparando los escenarios de la tabla 6 con las combinaciones de la figura 2 se aprecia como, por un lado, a cada objetivo de prueba le

corresponde una combinación de valores, la cuál indica la información de entrada, valores del sistema e información de salida necesaria para poder ejecutar dicho escenario.

Por el otro lado, cada objetivo de prueba indica qué acciones es necesario realizar sobre el sistema para poder darle la información que necesita y obtener un resultado.

A partir de las actividades y decisiones visitadas en cada objetivo de prueba, es posible relacionar cada objetivo de prueba con su combinación de variables.

5. Trabajos relacionados

La mayoría de trabajos sobre prueba de casos de uso [13] [8] [22] no se centran en variables operacionales sino en el análisis de caminos de ejecución o de escenarios de caso de uso.

Además de los trabajos citados en la introducción, la propuesta PLUTO [16] aplica el método CPM para generar pruebas a partir de los puntos de variabilidad de casos de uso para familias de productos.

No se ha encontrado ningún trabajo que proponga el uso de las dos técnicas de manera complementaria. Aunque existen trabajos que proponen procesos para la generación automática de objetivos de prueba a partir de casos de uso definidos como prosa, por ejemplo [4], no hemos encontrado ningún trabajo que automatice la aplicación de CPM o del patrón de Binder.

6. Conclusiones

En este trabajo se ha presentado un procesos sistemático y automático (mediante una herramienta de soporte de libre descarga y uso) que permite aplicar el método de categoría-partición y el patrón Extended Use Cases para obtener objetivos de prueba a partir de casos de uso escritos en lenguaje no formal.

Como se ha citado en la sección 4, la mayoría de propuestas para la obtención de objetivos de prueba a partir de casos de uso se centran en el análisis de escenarios de casos de uso. Sin embargo, en este trabajo hemos mostrado como el uso de la técnica CPM y de variables operacionales puede ser complementaria al análisis de escenarios, ya que permite definir futuros valores de prueba y resultados del sistema. Como hemos mencionado, no hemos tenido conocimiento de ningún trabajo hasta la fecha que proponga dicha relación.

Las dos herramientas de soporte utilizadas en este artículo son: ObjectGen para la generación de diagramas de actividades y objetivos de prueba, y ValueGen para la identificación de variables, identificación de restricciones y generación de combinaciones. Ambas pueden descargarse libremente en www.lsi.us.es/~javierj.

Tabla 6. Objetivos de prueba obtenidos mediante el análisis de escenarios

Id	Objetivos de prueba
1	1 -> 2 -> 3 -> D1(No) -> D2(No)-> 4 -> D3(Sin error) -> D4(Resultados) -> 5
2	1 -> 2 -> 3 -> D1(No) -> D2(No)-> 4 -> D3(Sin error) -> D4(Sin resultados) -> 5.1
3	1 -> 2 -> 3 -> D1(No) -> D2(No)-> 4 -> D3(Error) -> 4.2
4	1 -> 2 -> 3 -> D1(No) -> D2(Sí)-> 4.1 -> D3(Sin error) -> D4(Resultados) -> 5
5	1 -> 2 -> 3 -> D1(No) -> D2(Sí)-> 4.1 -> D3(Sin error) -> D4(Sin Resultados) -> 5.1
6	1 -> 2 -> 3 -> D1(No) -> D2(Sí)-> 04.1 -> D3(Error) -> 4.2
7	1 -> 2 -> 3 -> D1(Sí)

Referencias

[1] Balcer M. Hasling W. Ostrand T. 1989. Automatic Generation of Test Scripts from Formal Test Specifications. ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3), ACM Press, pp. 257-71.

[2] Bertolino, A., Gnesi, S. 2004. PLUTO: A Test Methodology for Product Families. Lecture Notes in Computer Science. Springer-Verlag Heidelberg. 3014 / 2004. pp 181-197.

[3] Binder, R.V. 1999. Testing Object-Oriented Systems. Addison Wesley.

[4] Boddu R., Guo L., Mukhopadhyay S. 2004. RETNA: From Requirements to Testing in Natural Way. 12th IEEE International Requirements Engineering RE'04.

[5] Burnstein, I. 2003. Practical software Testing. Springer Professional Computing. USA.

[6] Cockburn, A. 2000. Writing Effective Use Cases. Addison-Wesley 1st edition. USA.

[7] De la Riva C. García-Fanjul J. Tuya J. 2006. Diseño de Pruebas para Consultas XPath Utilizando Técnicas de Partición. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD. Sitges. Spain.

[8] Denger, C. Medina M. 2003. Test Case Derived from Requirement Specifications. Fraunhofer IESE Report. Germany.

[9] Escalona M.J. 2004. Models and Techniques for the Specification and Analysis of Navigation in Software Systems. Ph. European Thesis. University of Seville. Seville, Spain.

[10] Escalona M.J. Martín-Pradas A., De Juan L.F, Villadiego D., Gutiérrez J.J. 2005 El Sistema de Información de Autoridades del Patrimonio Histórico Andaluz. V Jornadas de Bibliotecas Digitales. Granada, Spain.

[11] Escalona M.J. Gutiérrez J.J. Villadiego D. León A. Torres A.H. 2006. Practical Experiences in Web Engineering. 15th International Conference On Information Systems Development. Budapest, Hungary, 31 August – 2 September.

[12] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2006. Generating Test Cases from Sequences Of Use Cases. International Conference on Web Information Systems. Setúbal. Portugal.

[13] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2006. Generation of test cases from functional requirements. A survey. 4º Workshop on System Testing and Validation. Potsdam. Germany.

[14] Gutiérrez J.J. Escalona M.J. Mejías M. Torres J. 2006. Modelos Y Algoritmos Para La Generación De Objetivos De Prueba. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD. Sitges. Spain.

[15] Gutiérrez J.J. Escalona M.J. Mejías M. Reina A.M. 2006. Modelos de pruebas para pruebas del sistema. Taller de Desarrollo de Software Dirigido por Modelos. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD. Sitges. Spain.

[16] Bertolino, A., Gnesi, S. 2004. PLUTO: A Test Methodology for Product Families. *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg. 3014 / 2004. pp 181-197.

[17] Koch N. Zhang G. Escalona M. J. 2006. Model Transformations from Requirements to Web System Design. Webist 06. Portugal.

[18] Myers G. 2004. The art of software testing. Second edition. Addison-Wesley. USA.

[19] Object Management Group. 2003. The UML Testing Profile. www.omg.org.

[20] Offutt J., Irvine A. 1999. Testing Object-Oriented Software Using the Category-Partition Method. 17th International Conference on Technology of Object Oriented Languages on Systems. Pp 104-214. Santa Barbara. USA.

[21] Ostrand T. J., Balcer M. J. 1988. Category-Partition Method. Communications of the ACM. 676-686.

[22] Roubtsov S. Heck P. 2006. Use Case-Based Acceptance Testing of a Large Industrial System: Approach and Experience Report. TAIC-PART. Windsor, UK.