Síntesis de especificaciones de programas XML para la construcción de oráculos de prueba

Dae S. Kim-Park, Claudio de la Riva, Javier Tuya

Departamento de Informática
Campus de Viesques
Universidad de Oviedo
kim_park@lsi.uniovi.es, claudio@uniovi.es, tuya@uniovi.es

Resumen. La automatización de pruebas de software está limitada por el problema del oráculo, el cual establece que bajo ciertas circunstancias, es difícil o imposible obtener un mecanismo denominado oráculo de prueba, capaz de comprobar la corrección de las salidas de prueba sin intervención humana. Se aborda este problema en el contexto de prueba de programas destinados al procesamiento de datos XML, presentando una técnica para dar soporte automático a la obtención de oráculos de prueba. Dado un programa XML, la técnica sintetiza una especificación ejecutable del programa mediante análisis dinámico, ejercitando el programa sobre múltiples entradas generadas mediante perturbación de datos. La especificación ejecutable obtenida permite identificar defectos en el programa, y puede ser utilizada como elemento constituyente de un oráculo de prueba. Se incluye un caso de estudio para ilustrar la aplicación de la técnica.

Palabras clave. Pruebas de software, Oráculos de prueba, programas XML.

1 Introducción

El estándar XML [17] es empleado en una gran variedad de entornos como formato para representar e intercambiar información entre diferentes sistemas. Las ventajas que ofrece XML respecto a otros modelos de datos [1] han motivado que éste sea hoy el formato predominante en entornos cotidianos como Internet y en particular la Web.

El uso de datos XML hace necesario disponer de artefactos software capaces de procesar estos datos. En este trabajo nos referimos a dichos artefactos con el término de *programas XML*, englobando como tales todos los sistemas, componentes o partes de los mismos cuyo cometido es acceder a datos XML y llevar a cabo un procesamiento sobre los mismos para producir nuevos datos XML.

Las actividades de prueba sobre programas XML plantean dos problemas generales. Por un lado, la selección de las entradas de prueba para estos programas debe abordar la complejidad de las estructuras XML, determinando qué características han de tener estas estructuras para que la prueba revele la mayor cantidad de defectos posible. Por otro lado, los programas XML pueden producir salidas de gran volumen y complejidad cuya evaluación, si se realiza de forma manual, es tediosa, propensa a errores y costosa. Por tanto, resulta útil disponer de mecanismos automatizados, denominados *oráculos de prueba* [14][16] que permitan evaluar las salidas sin intervención humana. Para abordar el primer problema existen diversas técnicas que pueden dar soporte a la selección y generación de entradas XML para la prueba [3][4][10][11]. Sin embargo, los trabajos relacionados con la obtención de oráculos de prueba para programas XML son muy escasos.

Los trabajos más directamente relacionados con la obtención de oráculos para programas XML [12][13] se basan en construir estos oráculos a partir de especificaciones parciales de los programas, que deben ser suministradas por el ingeniero de pruebas en un lenguaje de especificación ejecutable (un lenguaje que permite especificar comportamientos del software con alto nivel de abstracción y, a su vez, puede ser ejecutado para obtener datos de interés para la prueba [9]). Este enfoque hace posible reducir el coste de obtención de oráculos frente a otros métodos más comunes como la implementación de pseudo-oráculos [16], donde es necesario desarrollar una o varias implementaciones de los requisitos del programa con el coste que ello conlleva.

Para dar continuidad a los trabajos previos, en este trabajo se aborda la reducción del coste de especificación manual, automatizando la obtención de especificaciones de programas XML mediante análisis dinámico. La técnica aplica perturbación de datos [20] sobre una entrada XML de ejemplo, produciendo así diversas instancias de entradas alternativas destinadas a ejercitar un determinado programa XML. Observando las salidas que el programa produce sobre dichas entradas, la técnica infiere el comportamiento metamórfico [5] del programa y sintetiza su posible especificación. Como resultado, se produce una especificación ejecutable que se aproxima a los requisitos del programa.

Como principal contribución, la técnica permite suprimir tareas de especificación manual que habitualmente debe realizar el ingeniero de pruebas, tales como identificar especificaciones a partir de los requisitos de los programas, o suministrar las especificaciones en un determinado lenguaje. Con ello, el coste de especificación manual queda reducido a (1) suministrar una entrada XML a la técnica y a (2) validar la especificación ejecutable resultante, tarea que no puede ser automatizada por depender de un procedimiento no computable [15].

La especificación ejecutable producida por la técnica permite abordar el problema del oráculo desde dos perspectivas. En primer lugar, validar la especificación permite identificar fallos operacionales del programa del mismo modo que en la prueba de software; sin embargo, se trata de un análisis estático sobre la especificación que evita la necesidad de inspeccionar datos de prueba XML complejos y de gran volumen. En segundo lugar, al estar representadas en un lenguaje de especificación ejecutable, las especificaciones pueden integrarse en los mecanismos habituales de prueba (procedimientos de oráculo, arneses de prueba, etc.) para construir oráculos de prueba.

La técnica propuesta presenta similitudes con herramientas orientadas a la obtención automática de invariantes, como Daikon [8], DIDUCE [7] o DySy [6], capaces de detectar potenciales invariantes a partir de información extraída de la ejecución del programa objetivo. Sin embargo, las herramientas existentes no son factibles para la inferencia de especificaciones de programas XML debido a dos razones principales: (1) estas herramientas no proporcionan mecanismos para manejar tipos de datos semiestructurados [1] como XML cuya estructura varía en función de los datos de entrada suministrados; y (2) dado que estas herramientas emplean enfoques de caja blanca, son dependientes del lenguaje de implementación. Esto resulta poco adecuado en programas XML, debido a la gran variedad de tecnologías con las que pueden estar implementados este tipo de programas. La técnica propuesta, en cambio, sigue un enfoque de caja negra, siendo independiente de la tecnología de implementación utilizada.

Para ilustrar la aplicación de la técnica, se incluye un caso de estudio donde se ejemplifica su uso y se discuten sus características más destacables.

El resto del trabajo se estructura de la forma siguiente. En la Sección 2 se detalla la técnica y los principales pasos que la definen. La Sección 3 describe la construcción de oráculos a partir de las especificaciones ejecutables producidas por la técnica. La Sección 4 contiene el caso de estudio y su discusión. Finalmente, la Sección 5 incluye las conclusiones y líneas de trabajo futuro.

2 Síntesis de especificaciones

La técnica para la síntesis de especificaciones parte de un ejemplar de entrada para el programa XML bajo prueba, sin importar su medio de obtención (mediante un criterio de selección de entradas, arbitrariamente, de forma aleatoria, etc.). Con ello procede según los pasos descritos a continuación, ilustrados en la Fig. 1.

INFERENCIA Y SÍNTESIS DE LA ESPECIFICACIÓN DEL PROGRAMA XML

- *Entrada*: programa XML bajo prueba y un ejemplar de entrada para el mismo.
- *Salida*: especificación ejecutable del programa XML, aproximada a los requisitos del programa.
- *Proceso*:

1. Se aplican diversas modificaciones sobre el ejemplar de entrada mediante perturbación de datos, lo cual produce nuevas instancias de entradas alternativas, denominadas *entradas perturbadas* (Sección 2.1).
2. El programa XML se ejecuta sobre el ejemplar de entrada y sobre cada una de las entradas perturbadas, obteniendo las salidas correspondientes.
3. Tomando como referencia el ejemplar de entrada y la salida que el programa produce sobre el mismo, se identifican los cambios en las salidas del programa resultantes de las perturbaciones de la entrada. Con ello se infiere el comportamiento del programa en forma de relaciones metamórficas [5] (Sección 2.2).
4. Finalmente, se sintetiza una especificación ejecutable del programa, transformando el comportamiento inferido en predicados expresados en un lenguaje de especificación ejecutable (Sección 2.3). La especificación ejecutable resultante constituye la salida de la técnica.

2.1 Perturbación de datos para la generación de entradas

El proceso de perturbación aplica reiteradamente un operador de eliminación de subárboles XML (una estructura formada por un nodo XML junto con todos sus descendientes) sobre el ejemplar de entrada inicial. Se generan así tantas instancias de entradas perturbadas como subárboles sea posible eliminar en la entrada inicial, de modo que cada entrada perturbada equivale a la entrada inicial salvo por un subárbol XML que es eliminado.

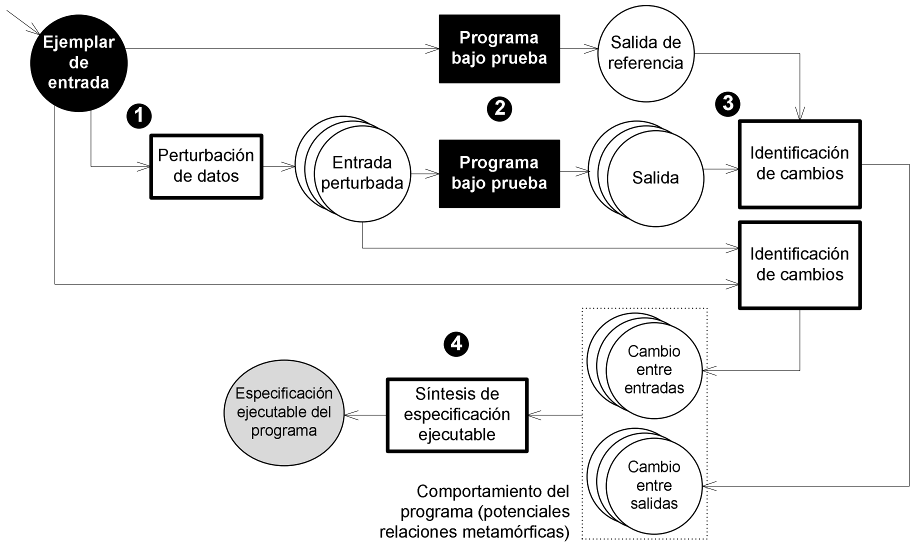

Fig. 1. Técnica para inferir la especificación del programa bajo prueba.

Detallando el proceso de perturbación, se denota cualquier estructura XML como una tupla $S = \langle N, C, A \rangle$, donde N es el conjunto de nodos de tipo elemento, atributo o texto [20] que forman la estructura. C es el conjunto de relaciones padre-hijo que definen las jerarquías entre nodos. Está compuesto por relaciones de la forma $x(c_1, c_2, ..., c_n)$, donde x es un nodo padre de tipo elemento definido en N, y $(c_1, c_2, ..., c_n)$ es una secuencia de nodos hijo de tipo elemento o texto definidos en N. Por último, el conjunto A se compone de relaciones de la forma $x\{a_1, a_2, ..., a_m\}$, donde x es un elemento de N y $\{a_1, a_2, ..., a_m\}$ es el conjunto de atributos de x.

Conforme a esta notación, el proceso de perturbación se define sobre el ejemplar de entrada $I = \langle N, C, A \rangle$ como la aplicación del operador de eliminación de subárboles, $DeleteTree(\bullet)$, sobre todo nodo $x_i \in N$. Como resultado, se obtienen instancias de entradas perturbadas $I_i' = \langle N_i', C_i', A_i' \rangle$, cada una incluyendo una perturbación única que satisface la siguiente condición

$$N_i' = N - \{x_i\} - Descendants(x_i) \wedge$$
$$C_i' = C - \{y(c_1, c_2, ..., c_n) \in C \mid y \in (Descendants(x_i) \cup \{x_i\})\} \wedge$$
$$A_i' = A - \{y\{a_1, a_2, ..., a_m\} \in A \mid y \in (Descendants(x_i) \cup \{x_i\})\}.$$

El conjunto $Descendants(x_i)$ definido en el contexto de I representa todo nodo $z \in N$ tal que para algún k (1) existe alguna relación de x_i, $x_i(c_1, c_2, ..., c_n)$ en C o $x_i\{a_1, a_2, ..., a_m\}$ en A, con $c_k = z$ o $a_k = z$; o (2) $z \in Descendants(w)$, existiendo una relación $x_i(c_1, c_2, ..., c_n)$ en C con $c_k = w$.

Aunque se podrían emplear otros operadores de perturbación, el motivo por el cual se emplea sólo el operador $DeleteTree(\bullet)$ en el proceso de perturbación, radica en su simplicidad. Este operador actúa sobre el dominio del ejemplar de entrada, produciendo nuevos ejemplares en el mismo dominio (cada I_i' cumple que $N_i' \subset N$, $C_i' \subset C$ y $A_i' \subseteq A$), lo cual supone dos ventajas. Por un lado permite conocer el dominio de las entradas perturbadas, pues equivale al dominio del ejemplar de entrada inicial. Y por otro lado, como la operación de eliminación mantiene la consistencia de los conjuntos N, C y A, las entradas perturbadas forman estructuras XML válidas, lo que es de interés para que se ejercite el funcionamiento normal del programa.

2.2 Identificación del comportamiento del programa

La ejecución del programa XML sobre las entradas perturbadas (Sección 2.1) da lugar a un conjunto de salidas que ejemplifican el comportamiento particular del programa. La técnica identifica este comportamiento construyendo reglas cuyo antecedente representa cambios en las entradas producidos por la perturbaciones, y cuyo consecuente indica el cambio derivado en la salida del programa.

Dadas I e I_k' dos estructuras XML de acuerdo a la notación de la Sección 2.1, si I es el ejemplar de entrada inicial e I_k' es una entrada modificada mediante la perturbación de I, el cambio en la entrada estará definido por las diferencias entre ambas estructuras, es decir $\Delta(I_k', I)$. Del mismo modo, si denotamos p al programa XML considerado, los cambios en la salida están definidos por $\Delta(p(I_k'), p(I))$, donde $p(I_k')$ es la salida derivada de la entrada perturbada, y $p(I)$ es la salida derivada de la entrada inicial. Con cada entrada perturbada y su salida (I_k' y $p(I_k')$) se construye una regla de la forma $\Delta(I_k', I) \to \Delta(p(I_k'), p(I))$, la cual establece que cuando existe el cambio $\Delta(I_k', I)$ en la entrada, en la salida existe el cambio $\Delta(p(I_k'), p(I))$. Estas reglas son potenciales relaciones metamórficas [5], pues describen posibles relaciones de comportamiento entre entradas y salidas observadas en múltiples ejecuciones del programa.

La expresión $\Delta(\bullet,\bullet)$ es una función de diferenciación [2] que recibe como parámetros dos estructuras XML. La función identifica la secuencia mínima de *operaciones de edición* que al ser aplicadas sobre la estructura XML del primer parámetro, producen su transformación en la estructura XML del segundo parámetro. Esta secuencia puede estar formada por las siguientes operaciones de edición.

- $\textit{DeleteTree}\{x'\}$: Representa la eliminación del subárbol cuya raíz es el nodo x' definido en I_k'. Describe la misma acción realizada por el operador de eliminación en la perturbación de datos (Sección 2.1).
- $\textit{InsertTree}\{x\}$: Denota la inserción del subárbol cuya raíz es el nodo x de I.
- $\textit{ChangeValue}\{x', x\}$: Representa un cambio de valor del nodo de texto o atributo x' definido en I_k'. El nuevo valor del nodo es del nodo x definido en I.

Como ejemplo, considérese dos estructuras XML S_1 y S_2 como las siguientes.

S_1 =`<vector><int>1</int><int>2</int></vector>`
S_2 =`<vector><string>ABCD</string><int>3</int></vector>`

Las diferencias entre ambas estructuras vienen dadas por $\Delta(S_1, S_2)$ = *(ChangeValue{*S_1`/vector[1]/int[1]/text()[.='1']`, S_2`/vector[1]/int[1]/text()[.='3']`*}, DeleteTree{*S_1`/vector[1]/int[2]`*}, InsertTree{*S_2`/vector[1]/string[1]`*})*. Esta secuencia indica que para transformar S_1 en S_2 es necesario cambiar el valor del primer elemento *int* (con valor 1) a 3, después eliminar el segundo subárbol con raíz en el elemento *int*, y por último insertar el subárbol de S_2 con raíz en el elemento *string*. Aquí, las referencias a nodos (x y x') de las operaciones de edición se representan como expresiones de trayecto XPath [19].

Nótese que existe una equivalencia en la transformación realizada por la operación de perturbación $DeleteTree(\bullet)$ y la operación de edición $DeleteTree\{\bullet\}$. Se trata de una circunstancia casual, pues cada operador está definido en un contexto diferente: el primero en la generación de entradas del análisis dinámico y el segundo en la identificación de comportamientos del programa.

Es importante apreciar que debido a que la perturbación aplica una sola transformación en cada entrada perturbada, $DeleteTree(\bullet)$, las reglas construidas tienen la forma $r_i \equiv (InsertTree\{x_i\}) \rightarrow (e_{i,1}, e_{i,2}, ..., e_{i,l(i)})$ para algún x_i de N, donde el antecedente $\Delta(I_i', I) = (InsertTree\{x_i\})$ es la secuencia de operaciones que describen la perturbación sobre el subárbol en x_i, y el consecuente $\Delta(p(I_i'), p(I))$ = $(e_{i,1}, e_{i,2}, ..., e_{i,l(i)})$ es la secuencia de operaciones que describen el cambio derivado en la salida. La expresión $l(i)$ denota el número de operaciones de edición en el consecuente de la regla i (es decir, $|\Delta(p(I_i'), p(I))|$). Con esta disposición, cada regla indica que "*la presencia del subárbol* x_i *en la entrada, determina el cambio* $(e_{i,1}, e_{i,2}, ..., e_{i,l(i)})$ *en la salida*".

2.3 Síntesis de la especificación ejecutable

El paso de síntesis de la especificación ejecutable consiste en obtener una especificación ejecutable a partir de la información proporcionada por las reglas obtenidas (Sección 2.2). Como lenguaje de especificación ejecutable se emplea un subconjunto de XQuery [13][20], aprovechando construcciones específicas para el manejo de datos XML que proporciona este lenguaje.

La especificación ejecutable está compuesta por una serie de predicados XQuery (expresiones de valor booleano) que establecen condiciones de comportamiento entre los datos de entrada y salida del programa. Los predicados expresan condiciones estructurales o de valores entre nodos XML. A modo de patrón, las condiciones estructurales se expresan como relaciones de cardinalidad de nodos de la salida en relación a la cardinalidad de ciertos nodos de la entrada. Por otro lado, las condiciones sobre valores se expresan como relaciones de totalidad, estableciendo qué relación algebraica cumplen ciertos valores de nodos de la salida en relación a los valores de determinados nodos de la entrada.

El proceso completo de síntesis de la especificación ejecutable es el siguiente.

1. Se descartan las reglas cuyo consecuente sea vacío, es decir, aquéllas que describen cambios en entradas que no tienen efecto en las salida. Como resultado se obtiene conjunto de reglas $\{r_1, r_2, ..., r_n\}$, donde cada r_i tiene la forma $r_i \equiv (InsertTree\{x_i\}) \rightarrow (e_{i,1}, e_{i,2}, ..., e_{i,l(i)})$, con $1 \le i \le n$ y $l(i) > 0$.
2. Se particiona cada regla separando las operaciones del consecuente. Cada r_i queda particionada en reglas $r_{i,1}, r_{i,2}, ..., r_{i,l(i)}$, donde $r_{i,j} \equiv (InsertTree\{x_i\}) \rightarrow (e_{i,j})$.

3. Se clasifican las reglas que afectan a similares estructuras XML de entrada y salida. Para ello, tomando el conjunto de todas las reglas particionadas, se forman grupos $G_1, G_2, ..., G_m$, cada uno agrupando reglas que sean similares. Dos reglas particionadas se consideran similares si sus antecedentes y consecuentes son iguales exceptuando los valores de nodos de texto y atributos referenciados en las operaciones de edición.
4. Por último, se transforma cada grupo de $G_1, G_2, ..., G_m$ en un predicado XQuery de acuerdo a los patrones de código de la Tabla 1.

Las referencias a nodos de las operaciones de edición, denotadas x_i e $x_{i,j}$ en la Tabla 1, se representan en la especificación XQuery como trayectos XPath [19] abstraídos omitiendo los índices de filtrado. Y si el trayecto hace referencia a un nodo de texto o un atributo, se indica el filtrado de aquellos rangos de valores identificados en reglas pertenecientes a un mismo grupo. De este modo los trayectos de la especificación ejecutable son más generales que los expresados por las reglas. Por ejemplo, considérese el grupo G_k compuesto por las reglas siguientes.

$r_{k,1} \equiv$ *InsertTree{*`$IN/vector[1]/int[2]/text()[. = '1']`*}*
$\rightarrow$ *DeleteTree{*`$OUT/result[1]`*}*
$r_{k,2} \equiv$ *InsertTree{*`$IN/vector[1]/int[2]/text()[. = '4']`*}*
$\rightarrow$ *DeleteTree{*`$OUT/result[1]`*}*
$r_{k,3} \equiv$ *InsertTree{*`$IN/vector[1]/int[2]/text()[. = '5']`*}*
$\rightarrow$ *DeleteTree{*`$OUT/result[1]`*}*

Tabla 1. Patrones de código para obtener la especificación ejecutable a partir de un grupo G_k de reglas particionadas. Nótese que las expresiones x_i se refieren a trayectos XPath de la operación *InsertTree* de los antecedentes, mientras que los del tipo $x_{i,j}$ son trayectos de las operaciones de edición de los consecuentes ($e_{i,j}$).

Operador de edición en los consecuentes ($e_{i,j}$)	Procedimiento para obtener la especificación ejecutable	
InsertTree$\{x_{i,j}\}$	Si x_i o $x_{i,j}$ referencia elementos	Producir el predicado: `count(` x_i `)` *op* `count(` $x_{i.j}$ `)`
	Si x_i y $x_{i,j}$ referencian atributos o nodos de texto	Producir el predicado: `every $x in` $x_{i.j}$ `satisfies ($x` *op* x_i `)`
DeleteTree$\{x_{i.j}'\}$ o *ChangeValue*$\{x_{i,j}', x_{i,j}\}$	Si x_i o $x_{i,j}'$ referencia elementos	Producir el predicado: `count(` x_i `)` *op* `count(` $x_{i.j}'$ `)`
	Si x_i y $x_{i,j}'$ referencian atributos o nodos de texto	Producir el predicado: `every $x in` $x_{i,j}'$ `satisfies ($x` *op* x_i `)`

La conversión del grupo de reglas según la Tabla 1 daría lugar al predicado siguiente.

```
count($IN/vector/int/text()[.>='1' and .<= '5']) op count($OUT/result)
```

Nótese que los trayectos XPath de este predicado abstraen los índices de filtrado de las reglas (índices entre corchetes), lo cual permite que el predicado exprese el comportamiento del programa de un modo más general.

Por último, para completar la síntesis de la especificación, la técnica determina el operador relacional *op* adecuado en cada predicado XQuery, siendo *op* del conjunto *{*>, <, >=, <=, =, `!=`*}*. El operador seleccionado en cada caso es aquel que describe con mayor precisión lo observado en los datos del análisis dinámico. Esto se realiza automáticamente evaluando los predicados XQuery sobre las entradas perturbadas y salidas producidas. Continuando con el ejemplo, la técnica procedería ejecutando, por un lado `count($IN/vector/int/text()[.>='1' and .<= '5'])` sobre las entradas perturbadas, y por otro `count($OUT/result)` sobre las salidas, determinando en función de los resultados el operador *op* más adecuado.

3 Construcción de oráculos de prueba

La construcción de un oráculo de prueba involucra la obtención de sus dos componentes constituyentes: la *información del oráculo* y el *procedimiento del oráculo* [14]. La *información del oráculo* es toda aquella información de que dispone el oráculo acerca del comportamiento esperado de un software bajo prueba concreto. Por otro lado, el *procedimiento del oráculo* es el mecanismo encargado de comprobar que los datos resultantes de las pruebas satisfacen lo indicado por la información del oráculo, emitiendo en consecuencia un veredicto que permite valorar la corrección (*correctness*) del software. El veredicto indicará que el software *pasa* la prueba si los resultados satisfacen lo establecido por la información del oráculo, o en caso contrario indicará que el software *falla* la prueba.

La información del oráculo no puede obtenerse por medios totalmente automáticos debido a que determinar el comportamiento esperado de un programa es un procedimiento no computable (equivale a comprobar la verdad o falsedad de una sentencia algebraica [15]). No obstante, es posible facilitar la tarea bajo el soporte automático de la técnica si el oráculo se construye empleando la especificación ejecutable sintetizada (Sección 2) a modo de información del oráculo. Para ello es conveniente que la especificación sea validada mediante inspección manual con el fin de garantizar que se trata de una aproximación correcta de los requisitos del programa. Esto implica que la validación del comportamiento dinámico del programa se convierte en una tarea de prueba estática sobre la especificación ejecutable, sin necesidad de un tratamiento manual de datos de ejecución (entradas y salidas) de gran volumen o complejidad.

Una vez validada, la especificación ejecutable puede integrarse en un procedimiento de oráculo para construir el oráculo de prueba. Al estar la especificación ejecutable expresada en forma de secuencia de predicados XQuery (Sección 2.3), es suficiente que su correspondiente procedimiento de oráculo ejecute la especificación sobre un intérprete XQuery con las entradas y salidas de cada prueba, emitiendo el veredicto de paso (el programa pasa la prueba) si todos los predicados se satisfacen, o el veredicto de fallo en otro caso. El procedimiento puede además proporcionar veredictos de fallo detallados si indica qué predicados violan los datos de la prueba. Nótese que el procedimiento definido de esta forma efectúa un proceso invariante, y por tanto es reutilizable en la construcción de otros oráculos. En [13] se emplea un procedimiento con un enfoque similar.

Es importante destacar que validar la especificación ejecutable supone validar a su vez comportamientos observados del programa, pero ello no implica que el programa se considere probado, ya que el comportamiento inferido es generalmente incompleto y se basa en entradas perturbadas que no son explícitamente representativas del dominio de aplicación.

4 Caso de estudio

En este caso de estudio se ejemplifica la aplicación de la técnica sobre un programa de procesamiento XML con defectos. La técnica produce automáticamente una especificación casi completa de los requisitos del programa que puede ser validada con poco esfuerzo, y es posible construir con ella un oráculo capaz de detectar los defectos del programa.

Entrada (*bib.xml*):

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content
           for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
  <book year="1989">
    <title>Managing the software process</title>
    <author>Wats S. Humphrey</author>
    <publisher>Addison-Wesley</publisher>
  </book>
</bib>
```

Salida esperada:

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
    </book>
    <book year="1992">
        <title>Advanced Programming in the
               Unix environment</title>
    </book>
</bib>
```

Fig. 2. Ejemplo de entrada para el programa de estudio y su salida esperada.

4.1 Programa bajo prueba con defectos

Considérese que se desea someter a pruebas un programa de procesamiento XML cuyo objetivo es obtener de un fichero con información bibliográfica, aquellos libros publicados por la editorial *Addison-Wesley* después del año 1991. Este programa, incluido en [18], pertenece a una batería de programas que ejemplifican situaciones típicas de procesamiento XML del ámbito de los documentos y las bases de datos. Como ejemplo de ejecución, en la Fig. 2 se ilustra una entrada para el programa junto con su salida esperada.

Supongamos que el programa no ha sido probado y contiene dos defectos. En lugar de cumplirse la condición $(publisher = "Addison - Wesley") \wedge (year > 1991)$ sobre cada libro de la salida, el programa defectuoso produce libros que cumplen la condición $(publisher = "Addison - Wesley") \vee (year \geq 1991)$. El defecto sobre el operador relacional ($\geq$) provoca que el programa filtre incorrectamente los libros por el año de publicación, permitiendo que libros de 1991 sean mostrados en la salida del programa. El otro defecto, sobre el operador lógico ($\vee$), elimina la necesidad de que se cumplan simultáneamente la condición sobre la editorial y el año de publicación en cada libro de la salida.

Si se ejecuta el programa con la entrada de la Fig. 2, el defecto del filtrado por año no se ejercita al no haber ningún libro de 1991. En cambio, el defecto sobre el operador lógico sí se ejercita y como consecuencia, la salida incluye todos los libros de la entrada, pues todos son posteriores a 1991 o de la editorial Addison-Wesley.

4.2 Síntesis de la especificación ejecutable

Suministrando el fichero *bib.xml* de la Fig. 2 como ejemplar de entrada arbitrario, la técnica genera 72 entradas diferentes mediante perturbación (Sección 2.1), con las cuales identifica las 72 reglas de comportamiento correspondientes (Sección 2.2). Tras ser éstas particionadas y clasificadas automáticamente (Sección 2.3) quedan reducidas a las 11 reglas esbozadas en la Tabla 2.

A continuación, el proceso de síntesis de la especificación ejecutable (Sección 2.3) produce los predicados XQuery siguientes.

```
count($IN/bib/book) = count($OUT/bib/book),

count($IN/bib/book/title) = count($OUT/bib/book/title),

count($IN/bib/book/publisher) = count($OUT/bib/book),

every $x in $OUT/bib/book/@year
     satisfies ($x = $IN/bib/book/@year),

every $x in $OUT/bib/book/title/text()
     satisfies ($x = $OUT/bib/book/title/text()),

count($IN/bib/book/publisher/text()[. = 'Addison-Wesley']) <
count($OUT/bib/book),

count($IN/bib/book/@year[. >= 1999 and . <= 2000]) =
     count($OUT/bib/book)
```

Tabla 2. Estado de los grupos de reglas particionadas. Por motivos de legibilidad se han omitido de antemano los índices en los trayectos XPath. La variable *$IN* representa la entrada del programa (*bib.xml*), y la variable *$OUT* representa la salida obtenida, ambas referidas a una ejecución del programa sobre una entrada perturbada.

G_i	Reglas obtenidas	
G_1	$r_{1,1}$	*(InsertTree{*`$IN/bib/book`*})* → *(InsertTree{*`$OUT/bib/book`*})*
G_2	$r_{2,1}$	*(InsertTree{*`$IN/bib/book/title`*})* → *(InsertTree{*`$OUT/bib/book/title`*})*
G_3	$r_{3,1}$	*(InsertTree{*`$IN/bib/book/publisher`*})* → *(InsertTree{*`$OUT/bib/book`*})*
G_4	$r_{4,1}$	*(InsertTree{*`$IN/bib/book/publisher/text()[. = 'Addison-Wesley']`*})* → *(InsertTree{*`$OUT/bib/book`*})*
G_5	$r_{5,1}$	*(InsertTree{*`$IN/bib/book/@year[. >= 1999 and . <= 2000]`*})* → *(InsertTree{*`$OUT/bib/book`*})*
G_6	$r_{6,1}$	*(InsertTree{*`$IN/bib/book/@year[. >= 1989 and . <= 1994]`*})* → *[ChangeValue{*`$OUT/bib/book/@year[. = '']`, `$OUT/bib/book/@year[. >= 1989 and . <= 1994]`*})*
G_7	$r_{7,1}$	*(InsertTree{*`$IN/bib/book/title/text()[.= 'TCP/IP Illustrated']`*})* → *(InsertTree{*`$OUT/bib/book/title/text()[.= 'TCP/IP Illustrated']`*}*
	… El grupo incluye reglas hasta $r_{7,5}$, cada una referida a uno de los títulos de la entrada.	

La especificación sintetizada está formada por una secuencia de 7 predicados XQuery (separados con comas). De los primeros tres predicados se deduce que el programa se comporta de forma incorrecta, puesto que las cardinalidades de las entidades de información de la entrada coinciden con las de la salida. Existe el mismo número de libros en la entrada y la salida, y cada título y editorial de la entrada se refiere a información de algún libro de la salida. Con esto se puede conjeturar que el programa no realiza un filtrado sobre la entrada, sino un cambio de formato.

Los dos predicados siguientes, el cuarto y el quinto, son condiciones sobre valores, expresando que tanto el año de publicación como el título de los libros de la salida son consultados directamente de la entrada. Esto sugiere que aunque el programa no realiza el filtrado de datos, construye correctamente salidas, con libros, títulos y años de publicación pertenecientes a la entrada.

Los dos últimos predicados describen características próximas a los requisitos del programa. El primero expresa una restricción sobre los libros de Addison-Wesley, pero debido al defecto sobre el operador lógico, no expresa que la salida del programa resulte del filtrado por la editorial. Sin embargo, sugiere con precisión el fallo en la salida, indicando que el programa produce libros de más editoriales además de aquéllos de Addison-Wesley. El último predicado expresa un filtrado por el año de publicación pero no está acotado por los valores límite esperados debido a que el ejemplar de entrada utilizado no presenta una gran diversidad de datos. Por este motivo tampoco se ejercita el defecto del operador relacional. A pesar de ello, al validar la especificación ejecutable es posible corregir con poca intervención manual tanto las imprecisiones debidas a defectos, como las debidas a la falta de diversidad en la entrada (Sección 4.3).

4.3 Validación de la especificación ejecutable

En este punto se puede proceder a validar la especificación con el objetivo de construir un oráculo (de acuerdo a lo descrito en la Sección 3) para confirmar la presencia de defectos mediante pruebas. La validación de este caso de estudio consiste en ajustar los operadores relacionales para reflejar el filtrado de datos, y ajustar los valores límite para expresar las restricciones del filtrado. La especificación validada es la siguiente (los ajustes están resaltados).

```
count($IN/bib/book) >= count($OUT/bib/book),

count($IN/bib/book/title) >= count($OUT/bib/book/title),

count($IN/bib/book/publisher) >= count($OUT/bib/book),

every $x in $OUT/bib/book/@year
      satisfies ($x = $IN/bib/book/@year),

every $x in $OUT/bib/book/title/text()
      satisfies ($x = $OUT/bib/book/title/text()),

count($IN/bib/book[publisher = 'Addison-Wesley' and @year > 1991]) =
      count($OUT/bib/book)
```

Finalmente, empleando esta especificación como información de un oráculo sobre un procedimiento como el descrito en la Sección 3, es posible construir un oráculo automatizado capaz de revelar los dos defectos considerados en el programa de estudio, así como otros posibles fallos sobre las estructuras XML de la salida y sobre los valores en nodos de texto y atributos.

4.4 Discusión del resultado

La entrada utilizada en el caso de estudio no ha sido construida de forma inteligente para revelar defectos específicos. De hecho, sólo ejercita uno de los dos defectos del caso de estudio. Sin embargo, al aplicar la técnica con dicha entrada y validar la especificación con pequeños cambios manuales, se puede obtener un oráculo de prueba capaz de detectar ambos defectos. Desde el punto de vista de la prueba de software, esto implica que con cualquier entrada para el programa la técnica puede generar una especificación ejecutable útil, incluso si esta entrada se genera de forma automática con un criterio de selección poco efectivo ante algunos defectos.

Puesto que el comportamiento del programa se extrae en forma de relaciones metamórficas (relaciones entre entradas y salidas observadas en múltiples ejecuciones reales del programa [5]), es posible que los predicados de la especificación ejecutable derivada identifiquen comportamientos no evidentes, difíciles de capturar como requisitos. Por ejemplo, el tercer predicado (Sección 4.3) es una condición estructural poco intuitiva que relaciona editoriales de la entrada con libros de la salida.

Las especificaciones ejecutables obtenidas no requieren un alto coste de validación, principalmente debido a que la técnica produce predicados XQuery siguiendo patrones de código fijos cuya interpretación es sencilla. Además, dado que los predicados están basados en cambios relativos observados (cambios respecto a la ejecución del programa sobre el ejemplar de entrada inicial), sólo describen comportamientos reflejados en puntos muy concretos de la entrada o la salida del programa. Por este motivo, la legibilidad o utilidad de los predicados no se ve afectada por salidas complejas o de gran volumen, y tampoco por salidas con fallos como ocurre en el caso de estudio.

5 Conclusiones y trabajo futuro

La técnica propuesta permite sintetizar una especificación ejecutable aproximada de los requisitos de un programa XML, infiriendo el comportamiento del programa mediante análisis dinámico automático. La especificación sintetizada puede revelar la presencia de defectos en el programa, y tras ser validada puede integrarse a un procedimiento de oráculo para construir un oráculo de prueba automatizado capaz de dar soporte a la prueba del programa.

El uso de esta técnica supone una reducción del coste en la obtención de oráculos, ya que automatiza las tareas de identificar y suministrar la especificación de los programas bajo prueba. Sólo requiere que el ingeniero de pruebas valide la especificación inferida, inspeccionándola para detectar fallos del programa, y ajustándola con poco esfuerzo para adecuarla con precisión a los requisitos. De este modo, mediante tareas de prueba estática el ingeniero de pruebas puede obtener la especificación del comportamiento dinámico del programa sin necesidad de tratar manualmente datos de entrada y salida de gran volumen o complejidad.

Existen varias líneas de trabajo futuro orientadas a mejorar la técnica. En principio, se plantea mejorar la precisión de las especificaciones resultantes aprovechando las reglas descartadas en la síntesis de la especificación o identificando relaciones no metamórficas entre reglas. También es necesario estudiar en profundidad cómo afecta el tipo de entrada suministrada a la precisión de las reglas obtenidas, y cómo diferentes operadores de perturbación pueden aportar más información sin cambiar la entrada del proceso. Por último, otra línea de trabajo futuro consiste en utilizar las especificaciones generadas como criterio de selección de entradas de prueba.

Agradecimientos

Este trabajo ha sido financiado por el Gobierno del Principado de Asturias con la beca PCTI-FICYT (Ref. BP09080), y ha sido parcialmente financiado por el Ministerio de Educación y Ciencia de España dentro del Plan Nacional I+D+i, a través del proyecto Test4DBS (TIN2010-20057-C03-01).

Referencias

1. Abiteboul, S.: Querying Semi-Structured Data. Proc. of the 6th Int. Conf. on Database Theory, pp. 1--18, (1997).
2. Barnard, D., Clarke, G. y Duncan, N.: Tree-to-tree Correction for Document Trees. Technical Report 95-372. Dep. of Comput. and Inf. Sci., Queen's Univ., Canada. 1--44 (1995).
3. Bertolino, A., Gao, J., Marchetti, E., Polini, A., “TAXI—A tool for XML-Based Testing”. Proc. of the 29th Int. Conf. on Softw. Eng. pp. 53--54 (2007).
4. De la Riva, C., García-Fanjul, J. y Tuya, J., “A Partition-Based Approach for XPath Testing”, Proc. of the Int. Conf. on Softw. Eng. Adv., pp. 17-22, (2006).
5. Chen, T.Y., Cheung, S.C. y Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01. Dep. of Comput. Sci., Hong Kong Univ. of Sci. and Technol. (1998).
6. Csallner, C., Tillman, N., Smaragdakis, Y., “DySy: dynamic symbolic execution for invariant inference”, Proc. of the 30th Int. Conf. on Soft. Eng. ACM, New York, NY, pp. 281--290, (2008).
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S. and Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. of Comput. Program., vol. 69, nº 1--3, pp. 35--45 (2007).
8. Hangal, S., Lam, M.S., “Tracking down software bugs using automatic anomaly detection”, Proc. of the 24th Int. Conf. on Soft. Eng. ACM, New York, NY, pp. 291--301, (2002).
9. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., y Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv., vol. 41, nº 2, pp. 1--76 (2009).
10. Hoffman, D., Wang, H.Y., Chang, M., Ly-Gagnon, D., Sobotkiewicz, L., y Strooper, P.: Two case studies in grammar-based test generation. J. of Syst. Softw., vol. 83, nº 12, pp. 2369--2378 (2000).
11. Kim-Park, D.S., De la Riva, C., Tuya, J. y García-Fanjul, J.: Generating Input Documents for Testing XML Queries with ToXgene. Test.: Acad. and Ind. Conf. - Pract. and Res. Techniques (2008).
12. Kim-Park, D.S., de la Riva, C., and Tuya, J.: A Partial Test Oracle for XML Query Testing. In Proc. of the Test.: Acad. and Ind. Conf. on Pract. And Res. Techniques (2009).
13. Kim-Park, D.S., De la Riva, C. y Tuya, J., “An Automated Test Oracle for XML Processing Programs”. Proc. of the 1st Int. Workshop on Softw. Test Output Valid., pp. 5-12 (2010).
14. Richardson, D. J., Aha, S. L., y O'Malley, T. O.: Specification-based test oracles for reactive systems. Proc. of the 14th Int. Conf. on Softw. Eng. ACM, New York, NY, 105--118, (1992).
15. Sipser, M.: Introduction to the Theory of Computation. Course Technology (2005).
16. Weyuker, E. J.: On Testing Non-testable Programs. The Comput. J., vol. 25, nº 4, pp. 465--470 (1982).
17. World Wide Web Consortium, Extensible Markup Language (XML) 1.0 (Fifth Edition), 28 de noviembre de 2008. http://www.w3.org/TR/xml/ (último acceso: 2011).
18. World Wide Web Consortium, XML Query Use Cases, 23 de marzo de 2007, http://www.w3.org/TR/xquery-use-cases/ (último acceso: 2011).
19. World Wide Web Consortium, XQuery 1.0: An XML Query Language, 23 de enero de 2007. http://www.w3.org/TR/xquery/ (último acceso: 2011).
20. Xu, W., Offut, J., y Luo J., “Testing Web Services by XML Perturbation”. En: Kawada, S. (ed.), Proc. of the 16th IEEE Int. Symp. on Softw. Reliability Eng., pp. 257-266 (2005).