Matching de Signatura y su Uso en el Desarrollo Basado en Componentes

Laura Kees Elsa Estévez Pablo Fillottrani

Departamento de Ciencias de Computación
Universidad Nacional del Sur
Av. Alem 1253 – (8000) Bahía Blanca
Argentina

e-mail: {ece,prf}@cs.uns.edu.ar

Resumen

El reuso de componentes de software promete ventajas como la reducción de tiempo y costo de programación, incrementando la productividad de los programadores y la confiabilidad de los programas. Sin embargo, existen razones por las cuales su uso no está generalizado en la práctica: por una parte no se encuentran fácilmente componentes hechas que satisfagan nuestros requerimientos, y además a medida que las librerías de componentes de software se hacen más grandes, es más complicado el trabajo necesario para ubicar la componente adecuada. El proceso de recuperación de componentes se realiza a través de un conjunto de pasos que filtran los elementos hasta lograr aquellos que satisfagan los requerimientos. En este trabajo se presenta al proceso de matching de signatura como parte del mecanismo, con el objetivo de optimizar la búsqueda. Se analizan sus distintas definiciones y se presentan ejemplos usando descripciones formales en RSL (RAISE Specification Language).

Palabras Claves: ingeniería de software, recuperación de componentes, matching de signatura

1 Introducción

La mayor parte de los trabajos sobre el reuso de software [3, 6] mencionan el problema de recuperar una componente de una librería: no podemos reusar una pieza de software si no la encontramos. Existen distintas posibilidades para ubicar una componente, ya sea describiéndola mediante una consulta y recuperando las componentes que satisfagan esos requerimientos o navegando a través de la librería hasta encontrar la componente que buscamos. Estas actividades de *recuperación*, *visualización* y *comparación* tienen varias aplicaciones. Por ejemplo, la recuperación de una componente soluciona el problema de un desarrollador que especifica el comportamiento de un módulo o función e intenta buscar si

ya existe en una librería una pieza de software que satisfaga sus requerimientos. Por otro lado, las librerías son una fuente de ejemplos para el uso de un lenguaje de programación, un programador puede aprender como usar constructores de un lenguaje en particular ya sea visualizando los elementos de la librería o recuperando componentes en particular. Finalmente la comparación de dos componentes responde la pregunta general de cuanto están relacionadas, por ejemplo, podemos determinar si una componente es un subtipo de otra.

Una de las soluciones para la recuperación, visualización y comparación de componentes es agregar un índice a la librería. Esto nos permite navegar a través de ella e incrementar la eficiencia con la cual podemos almacenar y recuperar componentes. Las ventajas de índices jerárquicos son bien conocidas en el dominio orientado a objetos, donde los usuarios pueden navegar la estructura jerárquica de clases con un *browser*, por ejemplo en Smalltalk y C++. Actualmente muchas librerías usan el sistema de archivos para su organización. Por ejemplo, las librerías *ML* están organizadas mediante categorías de componentes a través de directorios. Más allá de cómo la librería esté organizada, la tarea de encontrar algo en ellas recae en los nombres de las componentes y, compartir las componentes con otros, requiere un mutuo acuerdo en el esquema de nombres y estructuras de directorios.

Una vez que recuperamos una componente desde una librería, nos queda decidir cómo utilizarla en nuestro diseño. Podemos sustituirla directamente donde la necesitamos, o de lo contrario modificarla para poder ensamblarla. En el primer caso debemos conocer que el comportamiento de la componente es equivalente al que esperamos. Si no podemos sustituir la componente, necesitamos saber cómo adaptarla para poder reusarla en el contexto actual.

Los puntos mencionados tienen varios aspectos en común. En la *recuperación* de componentes, buscamos todas las componentes que satisfagan una determinada consulta. En la *construcción de un índice* jerárquico para una librería, relacionamos cada par de componentes. En la *navegación*, vamos de una componente a otra que está más alto (o más bajo) en la jerarquía. En la *sustitución* esperamos que el comportamiento de una componente sea equivalente desde el punto de vista de la observación al de otra. En la *modificación*, adaptamos una componente en base a nuestros requerimientos. La respuesta común a todas estas cuestiones es decidir cuando una componente *matchea* con otra. Existen definiciones de distintas clases de matching de componentes. En general, una función de match de componentes, toma dos componentes y retorna un valor booleano indicando si se verifica una determinada relación entre las dos componentes. En la próxima sección se plantean las representaciones de signaturas utilizando módulos escritos en el lenguaje RSL [1, 2]. En la sección tres se presentan las definiciones de matching de componentes. En la cuarta sección se clasifican y definen los distintos matching de signatura para funciones y se ejemplifican sobre la librería presentada en la sección anterior. En la sección cinco se analiza el matching de signaturta para módulos en forma análoga al de funciones. A continuación se presenta el uso de matching de signatura para recuperación de componentes y finalmente se emiten las conclusiones.

2 Representación de Signatura

Para realizar nuestros ejemplos, elegimos *RSL, RAISE Specification Lanaguage* para las descripciones de las componentes. Presentamos una librería con tres módulos CONJUNTO, PILA Y COLA. Los módulos tienen definiciones de tipos, encabezados por la palabra *type*, y definiciones de funciones que están precedidas por la palabra *value*. Las definiciones de las funciones constan de una definición de signatura más una definición del comportamiento. Por ejemplo la función *agregar* del módulo *PILA* toma como argumentos un dato de tipo *Elem* y uno de tipo *Pila* y devuelve un valor de tipo *Pila*, esto es lo que se conoce como signatura de la función. En RSL el comportamiento de una función se puede definir de tres maneras distintas: mediante axiomas que indican las propiedades que debe satisfacer, de manera explícita indicando precisamente cómo se calcula la función o de manera implícita mediante post-condiciones. En los ejemplos se presentan definiciones explícitas e implícitas. Por ejemplo en el caso de *Longitud* del módulo *PILA* se provee una definición explícita ya que se especifica que el resultado es el cardinal de los elementos de la lista. En el caso de *eliminar* del módulo *CONJUNTO* se provee una definición implícita ya que se dice que la post condición es devolver un conjunto $c1$ tal que el elemento e no pertenezca al mismo.

Asumimos que un valor definido en RSL como por ejemplo *vacio* de tipo αC, operador de tipo definido en el módulo como en el caso de los módulos $PILA$ y $COLA$, tiene una signatura $Unit \rightarrow \alpha C$.

La expresividad de un sistema de tipos, y por lo tanto la efectividad del matching de signatura, varía enormente de un lenguaje de programación a otro. En lenguajes similares a C, las funciones operan sobre unos pocos tipos propios del lenguaje y entonces, los tipos tienen limitada expresividad. Por el contrario, los lenguajes de programación más avanzados cuentan con tipos abstractos definidos por el usuario, tipos funcionales y tipos polimórficos, y por lo tanto podemos derivar más información acerca del comportamiento de una componente. Por ejemplo, la función *eliminar* del módulo *COLA* tiene signatura $Cola \rightarrow (Elem,Cola)$, pero, exactamente ¿que elemento de la cola retorna esta función?. Para responder a esta pregunta necesitamos la definición del comportamiento de la función: las definiciones axiomáticas, implícitas o explícitas nos permiten resolverlo, y el matching de especificación es el encargado de la decisión.

3 Matching de Componentes

El concepto de matching de componentes está parametrizado por tres elementos:

- *La clase de información usada para describir las componentes*: se comparan abstracciones de componentes. Ejemplos de abstracciones son descripciones textuales, información estructural, signaturas y especificaciones formales. Las descripciones textuales presentan el problema del uso de sinónimos. Las abstracciones estructurales incluyen diagramas de flujos de datos y diagramas de control. Aunque estos grafos son precisos, la desventaja de su utilización radica en que presentan 'cómo trabaja' en lugar de describir 'qué hace'. Las abstracciones semánticas nos permiten describir el comportamiento y destacar 'qué hace'. Dos ejemplos de abstracciones semánticas son las signaturas, las cuales describen la información de tipos con los

```
scheme PILA = class
 type
   Elem,
   Pila = Elem*

 value
   vacia : Pila = ⟨⟩,

   agregar : Elem × Pila → Pila
   agregar (e, p) ≡ ⟨e⟩ ⌢ p,

   eliminar : Pila ~→ Elem × Pila
   eliminar (p) ≡
    let ⟨e⟩ ⌢ p1 = p in (e, p1) end
    pre p ≠ ⟨⟩,

   longitud : Pila → Nat
   longitud (p) ≡ card elems p
end
```

```
scheme COLA = class
 type
   Elem,
   Cola = Elem*

 value
   vacia : Cola = ⟨⟩,

   agregar : Cola × Elem → Cola
   agregar (c, e) ≡ c ⌢ ⟨e⟩,

   eliminar : Cola ~→ Elem × Cola
   eliminar (c) ≡
    let ⟨e⟩ ⌢ c1 = c in (e, c1) end
    pre c ≠ ⟨⟩,

   longitud : Cola → Nat
   longitud (c) ≡ card elems c
end
```

```
scheme CONJUNTO = class
    type
        Elem,
        Conjunto = Elem-set

    value
        vacio : Conjunto = {},

        agregar : Elem × Conjunto → Conjunto
        agregar (e, c) as c1
          post e ∈ c1,

        eliminar : Elem × Conjunto → Conjunto
        eliminar (e, c) as c1
          post e ∉ c1,

        esta_en : Elem × Conjunto → Bool
        esta_en (e, c) ≡ e ∈ c,

        nro_elems : Conjunto → Nat
        nro_elems (c) ≡ card c
end
```

Figura 1: Ejemplos en RSL.

que trabaja la componente, y especificaciones formales que describen el comportamiento dinámico. El inconveniente de las especificaciones formales es que pueden no estar disponibles para todo componente.

- *Granularidad de la componente*: Las componentes varían en tamaño, desde los constructores individules del lenguaje, pasando por bloques de código de tamaño moderado, hasta grandes sistemas de software. Las granularidades de las componentes referidas en este trabajo son *funciones* y *módulos*.

- *Grado de relajación de un match.* Es raro el caso en que se requiere que una componente matchee con otra 'exactamente'. En la recuperación buscamos un match cercano; con el uso de índices usamos un ordenamiento parcial sobre un conjunto de componentes, y en la determinación de subtipos no necesitamos sustituir una componente por otra que tenga un comportamiento exactamente igual. De aquí que además de la noción de match exacto tenemos la noción de match relajado.

Como ya se mencionó, las componentes sobre las cuáles estamos interesados son funciones y módulos y las abstracciones con las cuales trabajaremos son la signatura y la especificación formal de las mismas. Asumimos que cada componente C tiene asociada una signatura, C_{sig}, y una especificación formal de su comportamiento, C_{spec}. Dadas dos componentes, $C =< C_{sig}, C_{spec} >$ y $C' =< C'_{sig}, C'_{spec} >$, definimos un predicado de matching entre componentes en forma genérica, $Match$.

Definición 3.1 (Match [8])

Match: Componente,Componente $\rightarrow$ *Bool*
Match(C,C') = $match_{sig}(C_{sig},C'_{sig}) \wedge match_{spec}(C_{spec},C'_{spec})$.

Dos componentes C y C' matchean si (1) dada una definición de matching de signatura, $match_{sig}$, sus signaturas matchean y (2) dada una una definición de matching de especificación, $match_{spec}$, sus especificaciones formales matchean. Aunque definimos el match como una conjunción, podemos pensar en el match de signatura como un filtro que elimina elementos antes de efectuar el matching de especificación formal, que si bien es mucho mas preciso, también es más costoso.

El matching de signatura se basa en la información de signatura derivada de la componente. La signatura de una función simplemente es su tipo y la de un módulo es el conjunto de tipos definidos por el usuario mas las signaturas de las funciones pertenecientes al módulo. Para ilustrar la idea consideremos la librería de componentes de la sección previa. Si necesitamos hacer una búsqueda de una función específica, en lugar de realizar una consulta en base a su nombre, podemos consultar en base al tipo de la función, el cual es la lista de tipos de sus parámetros de entrada y salida. Por ejemplo, $Pila \rightarrow (Elem, Pila)$ es el tipo de la función *eliminar* del módulo $PILA$, la cual toma un objeto de tipo *Pila* retornando un objeto de tipo *Elem* y la nueva pila con el objeto eliminado. Si nuestro requerimiento es buscar un módulo, debemos realizar una consulta en base a su interface, la cual es un conjunto de tipos definidos por el usuario y tipos de funciones. Para satisfacer estas consultas es necesario definir el matching de signatura. Dada una consulta q, un predicado de match M y una librería de componentes C, el matching de signatura devuelve un conjunto de componentes, cada una de las cuales satisface el predicado de match [8].

Definición 3.2 (Match de Signatura [8])
Match de Signatura: Consulta de Signatura, Predicado de Match,
Librería de Componentes $\rightarrow$ *Conjunto de Componentes*
Match de Signatura $(q, M, C) = \{c \in C : M(c, q)\}$

4 Matching de Signatura para Funciones

El matching de funciones basado en la información de signatura se reduce al matching de tipo. Según Field y Harrison [9] un *tipo* es una variable de tipo $\in TypeVar$, o un *operador* de tipo aplicado a otros tipos. Los operadores de tipo son internos: *BuiltInOp*, o definidos por el usuario: *UserOp*. Cada operador tiene una aridad que indica el número de argumentos. Los tipos base son operadores de aridad 0, por ejemplo en RSL, *Bool*,

Int, Nat, Real, Char, Text y Unit; el constructor de funciones '→' es binario, por ejemplo, *Conjunto→Nat*. Usamos notación infija para el constructor de tuplas, (,), y de funciones, (→), y en cualquier otro caso usamos notación postfija, por ejemplo, *Elem list* es la declaración para un objeto de tipo lista cuyos elementos son de tipo *Elem*. El tipo definido por el usuario: αT, representa un operador de tipo T con aridad 1, donde el tipo de los argumentos de T es α. Dos tipos, τ y τ', son iguales ($\tau = \tau'$) si son la misma variable de tipo o $\tau = typeOp(\tau_1, \ldots, \tau_n)$ y $\tau' = typeOp'(\tau'_1, \ldots, \tau'_n)$, $typeOp = typeOp'$, y $\forall i : 1 \leq i \leq n, \tau_i = \tau'_i$.

En algunos casos, para poder satisfacer el match es necesario realizar alguna sustitución. Para permitir la sustitución de tipos por variables de tipo introducimos la notación para *sustitución de variables*: $[\tau'/\alpha]\tau$, representa el tipo que resulta de reemplazar todas las ocurrencias de la variable de tipo α en τ con τ'', asumiendo que no hay variables en τ' que ocurran en τ. Por ejemplo:

$$[Nat/Elem]((Elem, Pila) \rightarrow Pila) = ((Nat, Pila) \rightarrow Pila)$$

Mediante esta sustitución reemplazamos el tipo de dato *Elem* por un tipo de dato *Nat*. Una secuencia de sustitución es asociativa a derecha, por ejemplo:

$$[\beta/\gamma][\alpha/\beta](\beta \rightarrow \gamma) = [\beta/\gamma](\alpha \rightarrow \gamma) = (\alpha \rightarrow \beta)$$

En el caso donde τ' es sólo una variable, $[\tau'/\alpha]\tau$ es simplemente un *renombramiento de variable*. Para el renombramiento de variables $\alpha, \tau' \in TypeVar$ o $\alpha, \tau' \in UserOp$.
A los efectos de la sustitución podemos asumir los operadores de tipo definidos por el usuario como variables, ya que distintos programadores pueden nombrar de manera diferente al mismo operador de tipo. Por lo tanto la secuencia de sustitución puede incluir tanto renombramiento de variables como renombramiento de tipos definidos por el usuario.

$$[\beta/\alpha][C/T](\alpha T \rightarrow \alpha) = (\beta C \rightarrow \beta)$$

Utilizaremos V para referirnos a una secuencia de renombramiento de variables y U para una secuencia de sustituciones más general. Se hace esta distinción debido a que se permite renombrar variables en un *match exacto* y sustitución de variables en *match relajado*. Dado el tipo de una función de una librería de componentes, τ_l, y el tipo de una consulta τ_q, definimos la forma genérica del match de función, $M(\tau_l, \tau_q)$ de la siguiente manera:

Definición 4.1 (Match Genérico [8])
M : Tipo de la librería, Tipo de Consulta → Bool
$M(\tau_l, \tau_q) := T_l(\tau_l)RT_q(\tau_q)$

Donde T_l y T_q son transformaciones (como por ejemplo un reordenamiento) y R es alguna relación entre tipos.

Cada vez que sea posible, se aplica la transformación al tipo de la librería τ_l y no se modifica el tipo de la consulta, de manera que T_q sea la función identidad. Por ejemplo si buscamos una función que permita agregar un nuevo elemento a una pila de números naturales, considerando la librería descripta en la sección anterior, podemos decidir que

T_l es la sustitución: $[Nat/Elem]((Elem, Pila) \rightarrow Pila) = ((Nat, Pila) \rightarrow Pila)$, que T_q es la función identidad y R es la relación de igualdad.

Dentro de esta definición de match genérico podemos clasificar aún en forma mas concreta los distintos tipos de match separando en *match exacto* y *match relajado*. A su vez, estos últimos pueden dividirse en *match parciales* o *match transformacionales*. En el primer caso lo que varía es R, y en los match transformacionales varía T_l o T_q, las transformaciones sobre los tipos.

Se presentan a continuación las definiciones de cada uno de estos tipos de match.

4.1 Match Exacto

Dos tipos de funciones matchean exactamente si matchean mediante renombramiento de variables.

Definición 4.2 (Match Exacto [8])
$match_E(\tau_l, \tau_q)=$
$\exists$ *una secuencia de renombramientos de variable,* V*, tal que* $V\tau_l = \tau_q$.

Veamos un ejemplo donde el match exacto puede ser útil en la práctica: supongamos el ejemplo que se presentó anteriormente de agregar un número natural a una lista de números, donde el tipo de la consulta es: $(Nat, Lista) \rightarrow Lista$. En este caso, la consulta devuelve la función *agregar* del módulo $PILA$, con el siguiente renombramiento:

$$[Nat/Elem][Lista/Pila]((Elem, Pila) \rightarrow Pila) = ((Nat, Lista) \rightarrow Lista)$$

Observar que no se devuelve la función *agregar* del módulo $COLA$.

4.2 Match Parcial

Utilizando únicamente el match exacto para recuperar componentes de una librería se corre el riesgo de no seleccionar componentes, que si bien no satisfacen este tipo de match, pueden resultar útiles para la aplicación. Supongamos que en el módulo $PILA$ el tipo de dato abstracto $Elem$ fuera un tipo de dato concreto como por ejemplo $Elem = Int$, es decir definido de tipo entero. Supongamos que el usuario está realizano una búsqueda de una función de tipo $((Real, Pila) \rightarrow Pila)$, en este caso el tipo de la función de la librería es mas general que el de la consulta. Aún más, podría suceder que la persona que realiza la búsqueda tenga dificultades para determinar un tipo de dato más general y por lo tanto se pierden funciones que podrían ser aplicables. Recíprocamente, existen casos donde consultamos por un tipo general que no matchea exactamente con nada en la librería pero puede haber una función útil en la librería cuyo tipo sea más específico y pueda ser generalizado fácilmente, como por ejemplo si el tipo de la consulta fuera $((Nat, Pila) \rightarrow Pila)$. En orden a resolver esta cuestión se define match *generalizado* y *especializado* para direccionar ambos casos. En la definición de match exacto la relación entre los tipos es la igualdad. Para el match parcial relajaremos la relación a un orden parcial sobre los tipos que definimos usando sustitución de variables. Por ejemplo $\alpha \rightarrow \alpha$ es una generalización de infinitos tipos, incluyendo $Nat \rightarrow Nat$ y $(Bool,\beta) \rightarrow (Bool,\beta)$, usando las sustituciones $[Nat/\alpha]$ y $[(Bool,\beta)/\alpha]$ respectivamente. Se dice que τ es más general que τ'($\tau \geq \tau$') si el tipo de τ' es el resultado de una secuencia de sustituciones de

variable, posiblemente vacía, aplicadas al tipo τ. Recíprocamente, decimos que τ' es una instancia de τ.

Definición 4.3 (Match Generalizado [8])
$match_{gen}(\tau_l,\tau_q) = \tau_l \geq \tau_q$.

El match exacto con renombramiento de variables, es un caso especial del match generalizado, donde todas las sustituciones se hacen mediante renombramiento de variables, es decir,

$$match_E(\tau_l, \tau_q) \Rightarrow match_{gen}(\tau_l, \tau_q)$$

Definición 4.4 (Match Especializado [8])
$match_{espec}(\tau_l,\tau_q) = \tau_l \leq \tau_q$.

El match especializado es la recíproca del match generalizado:

$$match_{espec}(\tau_l, \tau_q) = match_{gen}(\tau_q, \tau_l)$$

Surge de aquí que el match exacto es un caso especial del match especializado:

$$match_E(\tau_l, \tau_q) \;\Rightarrow\; match_{espec}(\tau_l, \tau_q)$$

Aunque mostramos el match *generalizado* y *especializado* en término de cambios sobre la relación R entre τ_l y τ_q, también podemos definirlos como *match transformacionales*, puesto que la relación $\leq$ sobre tipos se define en términos de sustitución de variables.

4.3 Match Transformacionales

Otra clase de match relajado consiste en transformar el orden o la forma de las partes de una expresión de tipo para alcanzar un match. Por ejemplo, podemos cambiar el orden de los tipos en una tupla o cambiar el orden de los argumentos de una función. Supongamos el caso de la consulta $((Nat, Lista) \rightarrow Lista)$, si permitimos match transformacional, la consulta devolverá las funciones *agregar* de los módulos $PILA$ y $COLA$, donde en este último módulo la función matchea con la consulta mediante match transformacional.

El *match con reordenamiento* permite el matching sobre tipos que difieren solo en el orden de sus argumentos y es un caso particular de match transformacional. Se define match con reordenamiento en término de permutaciones. Dado un tipo de función cuyo primer argumento es una tupla, sea $\tau = (\tau_l,\ldots,\tau_{n-1}) \rightarrow\tau_n$, una permutación σ es un mapeo 1-a-1 con dominio y rango 1...n-1 tal que $\sigma(\mathrm{r}) = (r_{\sigma(1)},\ldots,r_{\sigma(n-1)}) \rightarrow\tau_n$.

Definición 4.5 (Match con Reordenamiento [8])
$match_{reorder}\ (\tau_l,\tau_q) = \exists$ *una permutación,* σ*, tal que* $match_E(\sigma(\tau_l),\tau_q)$

4.4 Combinaciones

Cada una de las definiciones de match relajados pueden utilizarse en la forma general:

$$\exists \text{un par de transformación}, T = (T_l, T_q), \text{tal que } match_E(T_l(\tau_l), T_q(\tau_q))$$

T es una tupla de transformaciones, (T_l, T_q), T_l es una transformación sobre el tipo de la librería y T_q es una transformación sobre el tipo de la consulta. Para $match_{gen}$ y $match_{reorder}$, T_q es la función identidad, y en $match_{spec}$, T_l es la función identidad y T_q obviamente es una secuencia de sustituciones de variables.

5 Matching de Signatura para Módulos

El matching de funciones soluciona una parte del problema, sin embargo posiblemente se necesite comparar un conjunto de funciones, como por ejemplo la colección de funciones provistas por un tipo de dato abstracto. La mayoría de los lenguajes de programación modernos soportan la definición de tipos de datos abstractos a través de módulos independientes [6], por ejemplo: módulos *RSL*, *packages de ADA* o clases C++ . Un match entre dos módulos requiere un match entre pares de funciones de los módulos.

Como ya se mencionó la signatura de una función es simplemente su tipo, en cambio la signatura de un módulo es una interface, Υ. Una interface de módulo es [6], $\Upsilon = \langle \Upsilon_T, \Upsilon_F \rangle$ donde Υ_T es un conjunto de tipos definidos por el usuario y Υ_F es un conjunto de signaturas de funciones. Para que una interface de librería, $\Upsilon_L = \langle \Upsilon_{LT}, \Upsilon_{LF} \rangle$, matchee con una interface de consulta, $\Upsilon_Q = \langle \Upsilon_{QT}, \Upsilon_{QF} \rangle$, debe haber una correspondencia entre Υ_L y Υ_Q [7].

5.1 Match Exacto de Módulos

Dos módulos tienen match exacto si verifican la siguiente definición:

Definición 5.1 (Match Exacto de Módulo [6])
M-match$_E$*(*Υ_L*,*Υ_Q*) =*
$\exists$ *una función total* U_F*:* $\Upsilon_{QF} \rightarrow \Upsilon_{LF}$ *tal que* U_F *es una-a-una y total,*
y $\forall\ \tau_q \in \Upsilon_{QF}$*,* $match_E$ *(*$U_F(\tau_q)$*,*τ_q*)*

U_F mapea el tipo de cada función de la consulta τ_q a la signatura de la correspondiente función en la librería. Puesto que U_F es una-a-una el número de funciones en las dos interfaces debe ser el mismo. La correspondencia entre cada τ_q y $U_F(\tau_q)$ es tal que satisface el match exacto, $match_E$. También podemos requerir un mapeo entre tipos definidos por el usuario, pero generalmente el matching de funciones es suficiente, puesto que para que $U_F(\tau_q)$ y τ_q matcheen todo tipo definido por el usuario debe matchear. El siguiente ejemplo describe un módulo que contiene la definición de un tipo abstracto y su interface matchea con la del módulo $CONJUNTO$, mediante el obvio mapeo de los tipos de las funciones en Υ_{QF} a tipos de funciones en $CONJUNTO$; en cada uno de los matchs renombramos el operador de tipo definido por el usuario *Conjunto* a C.

$$
\begin{aligned}
\Upsilon_{QF} &= \{\alpha\mathrm{C}\} \\
\Upsilon_{LF} &= \{Unit \rightarrow Conjunto, \\
&\qquad (Elem, Conjunto) \rightarrow Conjunto \\
&\qquad (Elem, Conjunto) \rightarrow Conjunto \\
&\qquad (Elem, Conjunto) \rightarrow Bool \\
&\qquad (Conjunto) \rightarrow Nat\}
\end{aligned}
$$

El match exacto es bastante restrictivo. Definimos dos formas de match de módulo (1) modificando U_F y (2) reemplazando la definción de match de función, $match_E$.

5.2 Match Parcial de Módulos

Debido a que en el match exacto de módulos la función U_F es de cardinalidad 1-a-1, se deben testear todas las funciones del módulo, aún cuando existan algunas que no son de interés. Una alternativa más razonable es permitir que se especifique un subconjunto de las funciones y permitir matchear con un módulo que es más general en el sentido que puede contener otras funciones que las especificadas en la consulta. A estos efectos se introduce la definición de *match generalizado y especializado de módulos*

Definición 5.2 (Match Generalizado de Módulos [8])
M-match$_{gen}$*(*Υ_L*,*Υ_Q*) es el mismo que M-match*$_E$ *excepto que mientras con M-match*$_E$*(*Υ_L*,*Υ_Q*),*$|\Upsilon_{LF}| = |\Upsilon_{QF}|$*, con M-match*$_{gen}$*(*$\Upsilon_L$*,*$\Upsilon_Q$*),*$|\Upsilon_{LF}| \geq \Upsilon_{QF}|$ *y* $\Upsilon_{LF} \supseteq \Upsilon_{LF}$ *bajo el renombramiento adecuado.*

Definición 5.3 (Match Especializado de Módulos [8])
M-match$_{spec}$*(*Υ_L*,*Υ_Q*) = M-match*$_{gen}$*(*Υ_Q*,*Υ_L*)*

Con el match especializado, una librería no necesita tener todas las funciones definidas en la consulta. De la misma manera que para el matching de funciones, el matching especializado de módulos es la recíproca del matching generalizado de módulos.

5.3 Match *Relax**

En la definición de match de módulo exacto usamos el predicado de match exacto, $match_E$, para determinar si una función en la interface de la consulta matchea con otra en la interface de la librería. Un relajamiento obvio es el uso de match relajado en lugar de match exacto.

Definición 5.4 (Match *Relax** [8])
M-match$_{Relax^*}$*(*Υ_L*,* Υ_Q*,* M_F*) =*
$\exists$ *una función* U_F*:* $\Upsilon_{QF} \rightarrow \Upsilon_{LF}$ *tal que* U_F *es una-a-una, y*
$\forall \tau_q$ Υ_{QF}*,* M_F*(*U_F*(*τ_q*),*τ_q*)*

La única diferencia entre M-$match_E$ y M-$match_{Relax^*}$ es que este último utiliza el parámetro M_F. Notemos que M-$match_E$(Υ_L,Υ_Q) = M-$match_{Relax^*}$(Υ_L,Υ_Q,$match_E$). El parámetro nos da flexibiliddad, permitiéndonos usar cualquiera de las funciones de matching definidas para matching de funciones individuales.

5.4 Composición de Match de Módulos

Análogo al match de funciones, podemos componer matchs de módulos. Puesto que el match especializado puede definirse en términos del generalizado, sólo necesitamos considerar la composición de match generalizado y $relax^*$. El orden de la composición no tiene importancia ya que el match generalizado afecta al maping U_F mientras que el $relax^*$ sólo cambia la función de matching usada.

El siguiente match combinado es el más usado en la práctica:

Definición 5.5 (Match $relax^*$ Generalizado [8])
M-match$_{gen-relax^*}$(Υ_L,Υ_Q,M_F) es igual que M-match$_{relax^*}$(Υ_L,Υ_Q,M_F) excepto que U_F no es total.

El siguiente ejemplo de consulta de módulo contiene sólo dos tipos de funciones:

$$\begin{aligned}\Upsilon_{QT} &= \{\alpha\text{C}\}\\ \Upsilon_{QF} &= \{\text{unit}\rightarrow\alpha\text{C}\\ &\quad\ (\alpha\text{C},\alpha)\rightarrow\alpha\text{C}\}\end{aligned}$$

Bajo el match generalizado, esta consulta sólo matchea con el módulo $COLA$, con las funciones *vacía* y *agregar*. Bajo el match $relax^*$ generalizado, usando la función de match de reordenamiento, matchea también con el módulo $PILA$, con las funciones *vacía* y *agregar* y con el módulo $CONJUNTO$ con las funciones *vacio*, *agregar* y *eliminar* reordenando los argumentos de entrada.

6 Uso del Matching de Signatura

Como se ha presentado, los distintos matching de signatura nos permiten recuperar componentes de una librería. En la práctica, si bien es un paso importante en la selección, se necesita complementarlo con el matching de especificación. Por ejemplo ante una consulta de tipo $(\alpha, \alpha C \rightarrow \alpha C)$ con el match de funciones podríamos obtener como resultado *agregar* y *eliminar* del módulo $CONJUNTO$ cuando en realidad el comportamiento de ambas es diferente. Para solucionar esto se requiere una especificación formal de cada una de las funciones y ejecutar un match de especificación. Para poder usar toda la potencialidad del matching de signatura se requiere que el ingeniero de software tenga el conocimiento suficiente en la aplicación de tal manera que pueda generalizar y especificar tipos de datos.

Una posible mejora al matching de signatura podría darse en considerar a la estructura de datos administrada por el módulo como una caja negra llamada *Estado*, y en clasificar las funciones dependiendo de la acción que tengan sobre el estado, ya sea lectura, escritura o lectura y escritura. En este caso si la función es de lectura sobre el estado su signatura sería $Estado \rightarrow T$, donde T es una tupla de tipos. Si la función fuera de escritura o de lectura-escritura sería en forma general $(Estado, T_i) \rightarrow (Estado, T_j)$, donde T_i y T_j son tuplas de tipos. De esta manera se simplificaría en el matching de signatura la complicación introducida por los distintos tipos de datos administrados por el módulo, ya que quedarían encapsulados en el mismo. En el caso de nuestro ejemplo en lugar de definir los tipos de datos *Pila*, *Cola* y *Conjunto* se definiría en todos los casos un tipo de dato *Estado*. De esta manera las signaturas de la función *agregar* sería $(Elem, Estado) \rightarrow Estado$ para el caso de $PILA$ y $CONJUNTO$ simplificándose así la sustitución de variables con la consiguiente mejora en los match. A partir de esto, el match de especificación debiera limitarse a una especificación del comportamiento observable del módulo, ya que se ocultaría totalmente la estructura interna de datos.

7 Conclusiones

Se ha presentado el uso de matching de signatura aplicado al problema de recuperación de componentes. Otros enfoques se pueden dividir en dos clases: búsquedas basadas en texto y búsquedas basadas en esquemas de clasificación. Las búsquedas basadas en texto utilizan información textual como el nombre de la componente, autor, lenguaje o comentarios. Los métodos basados en clasificación permiten acotar el espacio de búsqueda facilitando la navegación, y el principal objetivo es suministrar una estructura a partir de la cual el usuario pueda navegar hasta encontrar la componente deseada. El matching de signatura puede considerarse como un enfoque complementario a estos métodos tradicionales. Por ejemplo podría combinarse un esquema de clasificación en conjunción con el matching de signatura. Si a esto se le suma una normalización en la forma de escribir las funciones de las componentes, como se describió en la sección anterior, se optimizarían el tiempo de búsqueda y la cantidad de componentes encontradas. El matching de signatura ofrece la mayor cantidad de información acerca de los módulos y no requiere un trabajo adicional ya que se puede generar a partir de la información escrita para el compilador. Este matching devuelve un primer conjunto de componentes que deben volver a filtrarse por medio del match de especificación. Nuestro trabajo a futuro se basa en investigar las distintas alternativas de match de especificación para la selección de componentes y cómo a partir del enfoque descripto en la sección anterior debe especificarse el comportamiento de las mismas.

Referencias

[1] The RAISE Method Group. *The RAISE Specification Language*. Pr. Hall, 1992.

[2] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall, 1995.

[3] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6), 1995.

[4] R. Mili, A. Mili, and R. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7), 1997.

[5] M. Nielsen and C. Clausen. Bisimulation, Games and Logic. In *CONCUR94*, volume 836 of *LNCS*, pages 385–400. Springer-Verlag, 1994.

[6] A. Zaremski and J. Wing. Signature Matching: A Key to Reuse. *Proceedings of SIGSOFT'95 First ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes*, Diciembre 1993.

[7] A. Zaremski. Signature and Specification Matching. *School of Computer Science, Carnegie Mellon University*, Enero 1996.

[8] A. Zaremski and J. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4),1997.

[9] A.J. Field and P.G. Harrison. Functional Programming. Addison-Wesley,1988.