Aportes para la verificación eficiente de sistemas utilizando el cálculo relacional*

Gabriel Baum
LIFIA
Facultad de Informática - Univ. Nac. de La Plata
gbaum@info.unlp.edu.ar

Ricardo H. Medel†
Departamento de Computación
Facultad de Cs. Exactas, Fco-Qcas y Naturales
Univ. Nac. de Río Cuarto
rhmedel@dc.exa.unrc.edu.ar

Resumen

En este trabajo se propone aplicar el cálculo de relaciones binarias a fin de mejorar la eficiencia del método de verificación de sistemas de software propuesto en [14] y [15]. En dicho método el diseño arquitectónico del sistema es especificado gráficamente, mientras que las propiedades a verificar son expresadas como fórmulas de una lógica modal híbrida o de la lógica Fork. La aplicación de reglas de reducción a dichas fórmulas Fork permite la implementación de algoritmos eficientes para su verificación.

Palabras clave: Ingeniería de software, Métodos formales, Cálculo relacional, Álgebras Fork

*El presente trabajo ha sido subsidiado por la Secretaría de Ciencia y Técnica - UNRC.

†Actualmente es alumno de posgrado en el Stevens Institute of Technology, Hoboken, EE.UU. - rmedel@cs.stevens-tech.edu

1 Introducción

Los métodos formales de desarrollo de software intentan mejorar la confiabilidad de los sistemas informáticos abstrayéndolos en modelos matemáticos, proveyendo un formalismo para la descripción de las propiedades deseadas y permitiendo la demostración formal del cumplimiento de dichas propiedades por parte del modelo.

El limitado impacto que estos métodos han tenido en la industria puede ser superado mediante la integración con métodos clásicos. Esta integración requiere del uso de lenguajes en común, siendo las notaciones gráficas las más utilizadas en la industria [12], y el desarrollo herramientas automatizadas de verificación de propiedades [5].

El método propuesto por Baum y Medel ([14], [15]) permite diseñar un sistema mediante la formalización de su *diseño arquitectónico* utilizando un grafo dirigido, con nodos y arcos rotulados:

$$G = \langle N, E, LN, LE, R_{LN}, R_{LE} \rangle$$

donde N es el conjunto de nodos o módulos del sistema, $E \subseteq N \times N$ es el conjunto de arcos entre nodos, LN son los rótulos de nodo, LE los rótulos de arco y las relaciones $R_{LN} \subseteq N \times LN$ y $R_{LE} \subseteq E \times LE$ asignan rótulos a los nodos y a los arcos, respectivamente.

En esta formalización cada arco y cada nodo tendrán al menos un rótulo asignado, y además todo nodo tiene un rótulo que lo identifica unívocamente, es decir le dá un "nombre". Otros rótulos de nodo pueden referirse a propiedades que se verifican en un conjunto de módulos, por ejemplo pertenecer a cierta librería o ser de cierto *tipo* [7].

Las propiedades a verificar en el diseño pueden ser descriptas utilizando una lógica modal híbrida [1] o la lógica Fork [11], y su verificación puede automatizarse completamente.

La lógica modal híbrida mencionada es la lógica polimodal temporal y con nominales utilizada en el método de verificación propuesto por Areces, Felder, Hirsch y Yankelevich ([2], [3]). Al ser polimodal pueden utilizarse las modalidades de *necesidad* y *posibilidad* combinadas con los rótulos de arcos ($\langle E \rangle$ y $[E]$, con $E \in R_{LE}$). Por ser temporal pueden usarse también sus *inversas* ($\langle E \rangle_i$ y $[E]_i$). Mientras que los símbolos proposicionales nominales representan los nombres unívocos de los nodos del grafo.

Por su parte, la lógica Fork es definida por Frias y Orlowska en base a las álgebras Fork desarrolladas por Baum, Frias, Haeberer y Veloso ([10], [13]). El alfabeto de esta lógica está formado por el conjunto de variables relacionales (en esta aplicación corresponden a los rótulos de nodos y arcos), las constantes 1 (relación universal), 0 (relación vacía) y $1'$ (identidad), así como las operaciones $+$ (unión), $\bullet$ (intersección), ; (producto), ∇ (fork), $^-$ (complemento) y $\breve{}$ (transpuesta).

En la interpretación usual de las álgebras Fork las operaciones y constantes tienen el significado obvio, mientras que el operador *fork* denota una operación que refleja la estructuración del dominio de esta clase de álgebras, incorporando estructuras del tipo de árboles binarios sobre los elementos

básicos. En este trabajo haremos uso, además, de algunas definiciones adicionales que serán útiles en lo que sigue:

- Conjuntos: serán internalizados como partes de la relación identidad $1'$. El conjunto A será denotado por la relación $1'_A$, esto es la relación $\{(x, x) \mid x \in A\}$. En particular, el conjunto de valores básicos, aquellos que no son "árboles", se denotará $1'_U$ y sus elementos de denominarán *urelementos*.

- Dominios y Rangos de relaciones: serán conjuntos denotados $\mathsf{dom}(R)$ y $\mathsf{ran}(R)$, para cualquier relación R.

A los fines de esta presentación se innecesario utilizar explicitamente elementos estructurados, de modo que en lo que sigue todas las relaciones utilizadas solamente involucran urelementos.

La lógica Fork fue desarrollada como una lógica a la cual traducir varias lógicas no clásicas, incluyendo lógicas modales, por lo que el método de Baum y Medel traduce las fórmulas modales a fórmulas Fork antes de realizar la verificación.

En dicho método el grafo de diseño es transformado en una estructura Fork, la cual es un par $F = (A, v)$ donde A es un álgebra Fork y v es una función que asigna relaciones en A a cada una de las variables relacionales.

En particular cada módulo y cada propiedad compartida por varios módulos son representados como relaciones ideales derecho (R es ideal derecho si $R = R; 1$).

Un modelo con estas características puede ser ingresado en el sistema de manipulación de estructuras relacionales RELVIEW [4], el cual es entonces utilizado para realizar la verificación automática de las propiedades deseadas.

Una propiedad establecida por una fórmula α es válida en todo el sistema si se verifica $v(\alpha) = 1$. También puede verificarse si la propiedad es válida al menos en algunos módulos del sistema chequeando que $v(\alpha) \neq 0$.

En general, dada una relación R que representa un conjunto de módulos, se puede establecer si una propiedad expresada como una fórmula α es válida en dicho conjunto de módulos verificando $R \bullet v(\alpha) = R$. En particular, si R representa a un único módulo (R es el nombre del módulo) es suficiente con chequear la validez de la ecuación $R \bullet v(\alpha) \neq 0$.

Los algoritmos implicados en estas verificaciones tienen una complejidad computacional similar a la de los algoritmos tradicionales de *model checking*, basados en lógicas modales [8].

En el presente trabajo extendemos este método mostrando cómo las fórmulas Fork utilizadas para expresar las propiedades pueden ser manipuladas ecuacionalmente utilizando el cálculo relacional, de modo de obtener fórmulas equivalentes cuya evaluación puede ser implementada usando algoritmos eficientes.

En la siguiente sección explicamos cómo aplicar el razonamiento ecuacional sobre términos relacionales a fin de mejorar la eficiencia del método de verificación. En la tercera sección mostramos la aplicación de estas ideas

en un ejemplo de diseño orientado a objetos. Las conclusiones y propuestas de trabajos futuros se dan en la cuarta sección.

2 Aplicación del cálculo relacional

Las fórmulas Fork que representan propiedades deseadas de los sistemas a verificar y/o desarrollar pueden ser transformadas aplicando un conjunto de identidades muy simples, algunas de las cuales se muestran en la tabla 1. Un listado completo se encuentra en [6].

En lo que sigue utilizaremos la notación $\neg A$ para expresar el *complemento en* $1'_U$ de una relación identidad A, por lo cual la expresión $\neg\mathsf{dom}(R)$ denota la relación $1'_U - \mathsf{dom}(R)$.

2.1 Implementación de relaciones

Dado un grafo de diseño, las relaciones que representan arcos del grafo son implementadas como matrices *booleanas*, las relaciones que representan módulos o conjuntos de módulos pueden ser implementadas como vectores de tamaño $|N|$, donde en la posición i habrá un 1 si y sólo si el módulo n_i pertenece al dominio de la relación.

Durante nuestra investigación hemos aplicado este método a varios ejemplos: KWIC (*Key Word In Context*, [16]), acceso a módulos de una librería, una rutina de CRC (), un sistema de control de trenes basado en [9] y la jerarquía de clases que desarrollamos en este artículo.

Según muestran los resultados obtenidos con los ejemplos mencionados, puede obtenerse una importante mejora en la eficiencia general del método

Sean R, S y T relaciones cualesquiera:

$$(R+S)+T = R+(S+T) \quad (1)$$
$$R+S = S+R \quad (2)$$
$$R = \overline{\overline{R}+S}+\overline{\overline{R}+\overline{S}} \quad (3)$$
$$R;(S;T) = (R;S);T \quad (4)$$
$$R;1' = R \quad (5)$$
$$(R+S);T = R;T+S;T \quad (6)$$
$$(\breve{R})^{\smile} = R \quad (7)$$
$$(R+S)^{\smile} = \breve{R}+\breve{S} \quad (8)$$
$$(R;S)^{\smile} = \breve{S};\breve{R} \quad (9)$$
$$\breve{R};\overline{R;S}+\overline{S} = \overline{S} \quad (10)$$
$$\mathsf{dom}(R) = R;\breve{R}\bullet 1' \quad (11)$$
$$\mathsf{dom}(R) = \breve{R};R\bullet 1' \quad (12)$$
$$R\to S = \overline{R;\overline{\breve{S}}} \quad (13)$$

Tabla 1: Algunas identidades y definiciones útiles.

si la herramienta calcula previamente el dominio, rango y transpuesta ($\breve{R}$) de las relaciones que representan arcos en el grafo.

Debemos también contar con la operación $V[e \leftarrow x]$, que dados un vector V, un conjunto de urelementos e y un valor *booleano* x, almacena el valor x en las posiciones correspondientes a los valores de e en el vector V y deja el resto de las posiciones intactas.

Con estas facilidades, hemos manipulado las fórmulas Fork obteniendo en casi todos los casos términos relacionales que pueden ser calculados en tiempo lineal respecto de la cantidad de módulos en el sistema.

Sin embargo, no todas las fórmulas que denotan propiedades interesantes de los sistemas pueden ser llevadas a expresiones resolubles en tiempo lineal. La operación de producto entre relaciones, con complejidad $O(|N|^3)$, aparece bajo la forma del dominio de un producto ($\mathsf{dom}(R;S)$) o como la expresión $i;S;j = 0$, con i y j representando conjuntos de urelementos. En las siguientes secciones explicaremos las técnicas desarrolladas para resolver estas situaciones.

2.2 Relación de dominación

Podemos evitar calcular el producto contenido en un término de la forma $\mathsf{dom}(R;S)$, si tenemos en cuenta que:

$$\begin{aligned} \mathsf{dom}(R;S) &= \mathsf{dom}(R;\mathsf{dom}(S)) \\ &= \mathsf{dom}(R;\mathsf{ran}(R) \bullet \mathsf{dom}(S)) \end{aligned}$$

La última línea indica que necesitamos calcular solamente todos aquellos elementos que son pre-imágenes de los elementos del rango de R y que a su vez están en el dominio de S. Entonces la operación $\mathsf{dom}(R;S)$ puede implementarse como la unión de los vectores columna de R para cada elemento que esté en la intersección del rango de R con el dominio de S.

Aunque más eficiente en promedio, esta operación aún tiene un peor caso de $O(|N|^2)$, ya que es posible que todas las columnas de la relación R tengan que ser obtenidas y agregadas a la unión. Pero podemos utilizar el concepto de "dominación", que introduciremos en las siguientes definiciones, para disminuir aún más su complejidad.

En lo que sigue utilizaremos la notación xR para indicar la imagen de un elemento x por la relación R, esto es el conjunto $\{y \mid xRy\}$.

Definición 2.1 *Sea R una relación y sean x e y elementos del rango de R. Se dice que "y* domina a *x" (en símbolos $x < y$) si y sólo si toda pre-imagen de x en R es también pre-imagen de y.*

En base a esta *dominación* entre elementos del rango de una relación podemos definir una relación entre ellos.

Definición 2.2 *Dada una relación R, la relación de* dominación sobre *R, denominada $\delta(R)$, es el preorder definido por:*

$$x\delta(R)y \quad \textit{si y sólo si} \quad x\breve{R} \subseteq y\breve{R}.$$

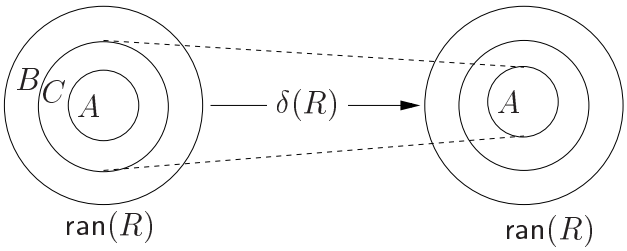

Figura 1: Relación de dominación de R.

En el álgebra relacional esto puede expresarse como:

$$\begin{aligned} \delta(R) &= \mathsf{ran}(R); (\breve{R} \rightarrow \breve{R}) \\ &= \mathsf{ran}(R); \overline{\breve{R}; \overline{R}} \end{aligned}$$

Esto nos permite utilizar solamente el conjunto de maximales en $\delta(R)$ y sus pre-imágenes por R en vez de calcular las pre-imágenes de todos los elementos del rango de R.

El caso general de una relación de dominación se muestra en la figura 1. El conjunto A representa los maximales, C es el conjunto de los elementos dominados por ellos y B los elementos no dominados. La imagen de $A + B$ por $\breve{R}$ debe ser igual al dominio de R, tal como lo establecen el siguiente teorema, que presentamos sin demostración, y su corolario:

Teorema 2.3 *Sean* $A = (\delta(R) \rightarrow 1') \bullet \mathsf{dom}(\delta(R))$ *y* $B = \neg\mathsf{dom}(\delta(R); A)$, *entonces* $\mathsf{ran}((A + B); \breve{R}) = \mathsf{dom}(R)$.

Corolario 2.4 $B = 0 \Rightarrow \mathsf{ran}(A; \breve{R}) = \mathsf{dom}(R)$.

Puede verse que nuestro método obtendrá la mayor eficiencia cuando A sea relativamente pequeño y $B = 0$.

Si el *forest* que representa a $\delta(R)$ es computado al comienzo de cada sesión, de la misma forma que hacemos con dominio, rango y transpuesta de las relaciones, entonces podemos desarrollar un algoritmo que calcule $\mathsf{dom}(R; S)$ eficientemente.

El algoritmo que damos a continuación realiza la unión de las columnas en R que corresponden a los elementos que están en el conjunto formado por la intersección del rango de R y el dominio de S, teniendo en cuenta sólo a aquellos elementos que no son "dominados" por otros del mismo conjunto.

Como lenguaje de programación utilizamos una versión modificada del lenguaje provisto por RELVIEW, agregando el tipo *booleano* y manteniendo nuestra notación para operaciones y constantes relacionales.

El algoritmo utiliza tres funciones auxiliares: `roots(F)`, que retorna todas las raíces de los árboles del *forest* `F`, `children(S,F)`, que retorna todos los hijos que los nodos de `S` tienen en el *forest* `F`, y `take_one(S)`, que obtiene y elimina de `S` un elemento de su dominio.

Su complejidad para el peor caso aún es de $O(|N|^2)$, pero en promedio la cantidad de cálculos necesarios es mucho menor.

```
domprod(R,S)       {retorna el dom(R;S)}
  DECL dom_rs,current_roots,rts,rest_roots,r_s,nrs,i
  BEG  current_roots = 0;
       rts = roots(δ(R));
       r_s = ran(R)•dom(S);
       nrs = -r_s;
       WHILE -empty(rts) DO
         current_roots = current_roots + (r_s • rts);
         rest_roots = rts • nrs;
         rts = children(rest_roots, δ(R));
         DO;
       dom_rs = ∅;
       WHILE -empty(current_roots) DO
         i = take_one(current_roots);
         dom_rs = dom_rs + col(i,R)
         DO
       RETURN dom_rs
  END.
```

2.3 La ecuación $i; S; j = 0$

En algunos de nuestros experimentos encontramos expresiones de la forma $i; S; j = 0$, con S una relación cualquiera, pero con i y j representando conjuntos de urelementos ($i, j \preceq 1'_U$). Su evaluación puede simplificarse analizando las distintas formas que pueden adoptar i y j.

Cuando i representa un único urelemento, la ecuación será válida si i no pertenece al dominio de S. Por lo tanto, podemos condicionar el chequeo del resto de la ecuación al resultado de checkear previamente si $i \in \mathsf{dom}(S)$.

Si j representa también a un único urelemento, entonces $i; S; j$ será vacía solamente si j no está entre los elementos que tienen pre-imagen i en la relación S. En la representación matricial de relaciones esto equivale a decir que j no está incluido en la `fila(i,S)`. Podemos utilizar entonces la siguiente función para determinar el resultado:

```
eq_cero1(i,S,j) = i ∉ dom(S) cor j ∉ fila(i,S).
```

Por otra parte, si j representa a varios urelementos entonces $i; S; j$ será vacía sólo si ninguno de los elementos de j pertenece al conjunto de elementos con pre-imagen i por S. Completamos entonces una nueva versión de la función que pueda ser aplicable a este caso:

```
eq_cero2(i,S,j) = i ∉ dom(S) cor fila(i,S)•j=0.
```

Algo equivalente sucede cuando i y j intercambian roles, representando j a un único elemento e i a uno o varios urelementos. Se pueden desarrollar dos funciones similares a las anteriores reemplazando i por j, dominio por rango de S y `fila` por `columna`.

Todas estas funciones tienen complejidad lineal respecto a $|N|$. Pero cuando ambos, i y j, representan a varios urelementos, debemos calcular la conjunción de todas las `fila(x,S)` para cada $x \in i$ y luego chequear si su intersección con j es vacía. Esto tiene una complejidad en el peor caso de $O(|N|^2)$.

Sin embargo, en este caso podemos utilizar nuevamente el concepto de *dominación*, ya que de los elementos en j sólo nos interesan los que están en el rango de S, en particular solamente los que sean "dominantes". De modo que no será necesario verificar todo $\mathsf{ran}(S) \bullet j$. Para cada elemento "dominante" en j debemos chequear si tiene una pre-imagen por S en i, en cuyo caso habremos encontrado una refutación de la ecuación $i; S; j = 0$. Si ninguno de los elementos revisados tiene esa característica entonces hemos determinado que $i; S; j$ es vacía.

El siguiente algoritmo, cuya función auxiliar `union(i,S)` retorna el conjunto $\cup_{x \in \mathtt{i}} \mathtt{col}(x, \mathtt{S})$, realiza los mencionados cálculos:

```
eq_cero5(i,S,j)
  DECL rts, ok, jroots
  BEG  rts = roots(δ(S));
       ok = true;
       WHILE -empty(rts) and ok DO
         jroots = rts • j;
         IF -empty(jroots) THEN ok = union(jroots,S)⊆ -i
         FI;
         IF ok THEN rts = children(rts • j, δ(S)) FI
       OD
       RETURN ok
  END.
```

Ecuaciones de la forma $X + \overline{S; X} = 1$ son la contrapartida algebraica de fórmulas multimodales del tipo $P \to [S]P$, útiles para expresar propiedades invariantes de sistemas. La siguiente proposición establece una equivalencia que permite utilizar las funciones anteriores para computar eficientemente ecuaciones de la forma mencionada.

Proposición 2.5 *Sean S una relación cualquiera y X una relación ideal derecho, entonces:*

$$X + \overline{S; X} = 1 \text{ si y sólo si } \neg \mathsf{dom}(X); S; \mathsf{dom}(X) = 0.$$

En la siguiente sección desarrollamos, a modo de ilustración, la aplicación de nuestro método a un diseño orientado a objetos.

3 Aplicación del método

En esta sección aplicamos nuestro método a la verificación de propiedades del diseño de una jerarquía de clases en el paradigma de la programación

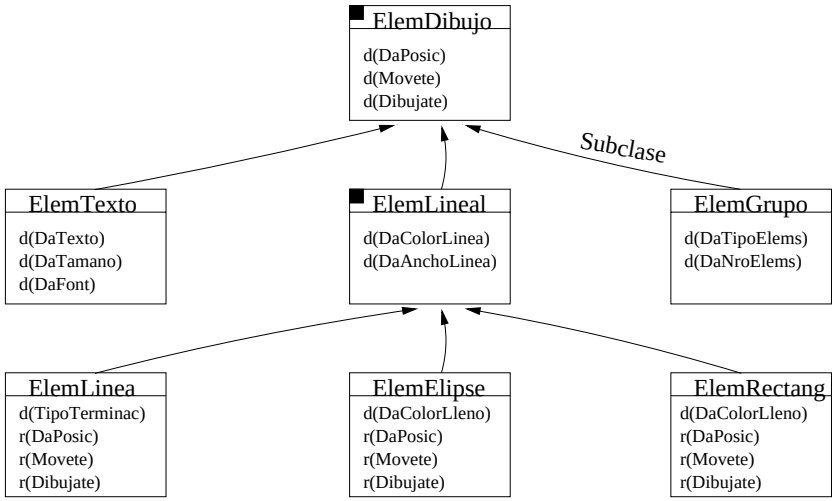

Figura 2: Diseño de la clase *ElemDibujo*.

orientada a objetos. El diseño de la clase *ElemDibujo* de un editor gráfico está basado en un ejemplo dado en [17] y las propiedades a verificar son tomadas de [2] y [3].

La única relación en el diseño de la clase *ElemDibujo* es la relación *Subclase* (ver figura 2). Las clases abstractas son $ElemDibujo$ y $ElemLineal$. Además, para cada método se indica si es definido (d) o redefinido (r) en esa clase.

La traducción del diseño a nuestra notación gráfica (ver figura 3) introduce nuevos símbolos relacionales: $EsClase$ para indicar cuáles nodos son clases, $EsMetodo$ para indicar cuáles son métodos y $EsAbstracta$ para indicar cuáles clases son abstractas. Las relaciones D y R permiten establecer qué métodos son definidos o redefinidos en qué clase.

En lo que sigue, cuando tratemos con los dominios de relaciones ideales derechos, tales como $EsAbstracta$ y $EsClase$, utilizaremos repetidamente la siguiente equivalencia:

$$\text{Si } R \text{ es ideal derecho, entonces } \mathsf{dom}(\overline{R}) = \neg\mathsf{dom}(R).$$

Las propiedades a verificar en este diseño serán dos: *ninguna clase abstracta hereda de una clase concreta* y *toda clase debe tener responsabilidades.* Pero además podemos utilizar el método para detectar automáticamente qué clases abstractas redefinen responsabilidades, de modo que el programador pueda chequear que el código de la implementación de esta responsabilidad no afecte a la condición de abstracta de la clase.

La primera propiedad, *ninguna clase abstracta hereda de una clase concreta*, puede redefinirse diciendo que *todas las superclases de una clase abstracta son también abstractas.* Lo cual puede formalizarse mediante la siguiente fórmula modal o su traducción a ecuación relacional equivalente:

$$EsAbstracta \rightarrow [Subclase]EsAbstracta$$

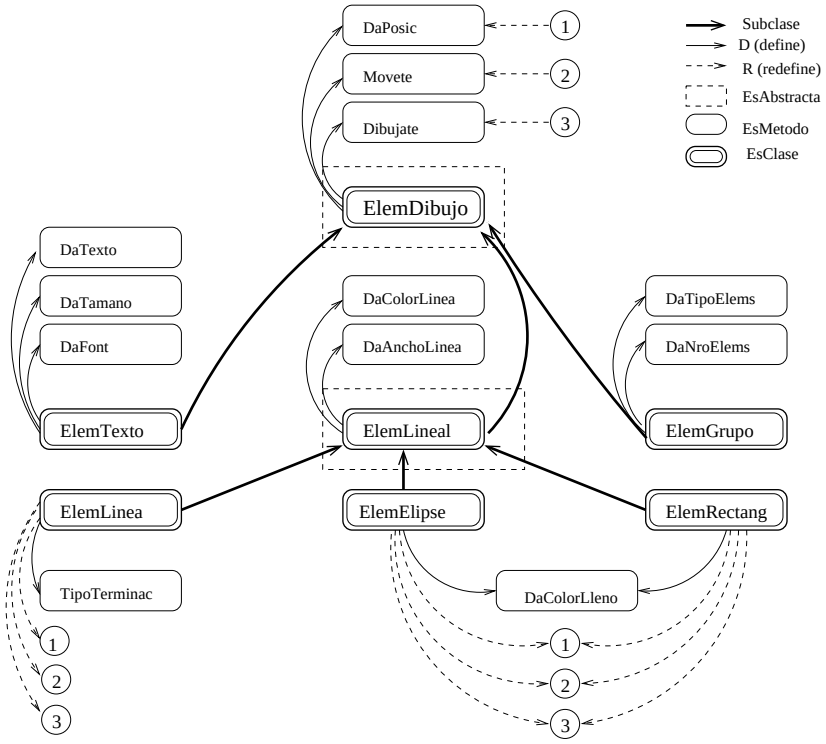

Figura 3: Diseño relacional de la clase *ElemDibujo*.

$$\overline{EsAbstracta} + \overline{Subclase; \overline{EsAbstracta}} = 1$$

Utilizamos la proposición 2.5 para determinar que la ecuación previa es equivalente a:

$$\neg\mathsf{dom}(\overline{EsAbstracta}); Subclase; \mathsf{dom}(\overline{EsAbstracta}) = 0$$

Esta ecuación puede ser eficientemente verificada aplicando la función:

```
eq_cero5(dom(EsAbstracta),Subclase,-dom(EsAbstracta))
```

La segunda propiedad a verificar es que *toda clase debe tener responsabilidades*. En caso contrario, se deberían detectar y eliminar las clases que no agregan funcionalidad. Esta propiedad puede ser expresada por la siguiente fórmula modal o su traducción a ecuación relacional equivalente:

$$EsClase \rightarrow (\langle D\rangle EsMetodo \vee \langle R\rangle EsMetodo)$$

$$\overline{EsClase} + (D; EsMetodo + R; EsMetodo) = 1$$

Mediante la siguiente derivación obtenemos una ecuación que puede ser evaluada más eficientemente.

$$\begin{aligned}
&\overline{EsClase} + (D; EsMetodo + R; EsMetodo) = 1 \\
&\quad \text{ssi} \quad \overline{EsClase} + (D + R); EsMetodo = 1 \\
&\quad \text{ssi} \quad \mathsf{dom}(\overline{EsClase}) + \mathsf{dom}((D + R); EsMetodo) = 1' \\
&\quad \text{ssi} \quad \neg\mathsf{dom}(EsClase) + \mathsf{dom}((D + R); EsMetodo) = 1'
\end{aligned}$$

Utilizamos el programa `domprod` para realizar el chequeo de esta ecuación, mediante el siguiente programa:

```
prop2()
  DECL dominio, nodom
  BEG  nodom = -dom(EsClase);
       dominio = domprop(D + R, EsMetodo, nodom)
       RETURN (nodom + dominio = 1')
  END.
```

Finalmente, podemos detectar automáticamente qué clases abstractas redefinen responsabilidades, indicando cuáles están relacionadas con al menos un método a través de R. Es decir, aquellos nodos que verifican la siguiente fórmula modal, con T una tautología, o su traducción relacional:

$$EsAbstracta \wedge \langle R \rangle \mathsf{T}$$

$$EsAbstracta \bullet R;1$$

Ambos miembros de la conjunción son relaciones ideales derecho y sólo estamos interesados en identificar qué modulos verifican la propiedad, por lo que podemos simplemente resolver la siguiente fórmula:

$$\mathsf{dom}(EsAbstracta) \bullet \mathsf{dom}(R)$$

Con los dominios de cada relación ya calculados, este término puede evaluarse en tiempo lineal respecto de la cantidad de nodos en el grafo de diseño.

4 Conclusiones y trabajo futuro

En este trabajo presentamos la aplicación del cálculo relacional a un método de verificación de sistemas, lo cual permite el desarrollo de algoritmos más eficientes en promedio que el *model checking* tradicional, basado en lógicas modales, y podría permitir su integración a entornos industriales de desarrollo de software. Una herramienta que permite aprovechar estas características está siendo desarrollada en la Universidad Nacional de Río Cuarto.

Aún se requiere de un análisis más profundo para obtener datos cuantitativos de la mejora en eficiencia producida por el aprovechamiento de la relación de *dominación* presentada en este texto.

Técnicas de transformación de grafos pueden aplicarse a fin de obtener nuevos diseños que mantengan algunas propiedades del sistema original mientras presentan algunas nuevas.

Hasta el momento sólo hemos aplicado el método para la verificación de propiedades estáticas o estructurales de los diseños de sistemas de software. Resultados recientes permiten avizorar la posibilidad de su aplicación a propiedades dinámicas.

References

[1] Areces, C. E., "Logic Engineering: The Case of Description and Hybrid Logics", PhD Thesis, ILLC DS-2000-05, Institute for Logic, Language and Computation, University of Amsterdam, 2000.

[2] Areces, C. E., Felder, M., Hirsch, D. F. and Yankelevich, D., "Modal Logic as a Design Notation", en *Proceedings of the 1st K1T125 Workshop*, Como, Italy, 1997.

[3] Areces, C. E. y Hirsch, D. F., "La Lógica Modal como Herramienta de Ingeniería de Software", Seminario de Tesis, UBA, 1996.

[4] Behnke, R., Berghammer, R. and Schneider, P., "Machine Support of Relational Computations: The Kiel RELVIEW* System", Technical Report Nr. 9711, Institut für Informatik und Praktische Mathematik, Christian-Albrechts Universität Kiel, June 1997.

[5] Braun, P., Lötzbeyer, H., Schätz, B. and Slotosh, O., "Consistent Integration of Formal Methods", en Graf, S. and Schwartzbach, M. (editores), *Tools and Algorithms for the Construction and Analysis of Systems*, *Proc. of the 6th Conference TACAS 2000*, Berlin, Germany, March/April 2000, LNCS 1785, págs. 48-62, Springer-Verlag, 2000.

[6] Brink, C., Kahl, W. and Schmidt, G. (editores), "Relational Methods in Computer Science", Springer-Verlag, 1997.

[7] Dean, T. R. and Cordy, J. R., "A Syntactic Theory of Software Architecture", en *IEEE Transactions on Software Engineering*, 21(4):302-313, April 1995.

[8] Emerson, E. A., "Temporal and Modal Logic", en van Leeuwen, J. (editor), "Handbook of Theoretical Computer Science", Elsevier Science Publishers, 1990.

[9] Fradet, P., Le Métayer, D. and Périn, M., "Consistency Checking for Multiple View Software Architectures", en Nierstrasz, O. and Lemoine, M. (editores), *Proceedings of 7th European Software Engineering Conference*, LNCS 1687, Springer-Verlag, 1999.

[10] Frias, M., Baum, G. and Haeberer, A., "Fork Algebras in Algebra, Logic and Computer Science", en *Fundamenta Informaticae*, 32(1):1-25, IOS Press, October 1997.

[11] Frias, M. and Orlowska, E., "Equational Reasoning in Non-Classical Logics", en *Journal of Applied Non Classical Logic*, 8(1-2), 1998.

[12] Ghezzi, C., Jazayeri, M. and Mandrioli, D., "Fundamentals of Software Engineering", Prentice Hall, 1991.

[13] Haeberer, A., Frias, M., Baum, G. and Veloso, P. A. S., "Fork Algebras", en [6], págs. 54-69.

[14] Medel, R. H., "Utilizando RELVIEW para la verificación de especificaciones en Lógica Modal", en *Anales del WAIT'99 - Workshop Argentino de Informática Teórica*, págs. 93-106, Buenos Aires, Setiembre de 1999.

[15] Medel, R. H. y Baum, G. A., "Aplicación de Algebras Fork en la verificación automática de sistemas especificados en Lógica Modal", en *Anales del CACIC'99*, Tandil, Argentina, Octubre de 1999.

[16] Parnas, D. L., "On the Criteria to be used in Decomposing Systems in Modules", en *Communications of the ACM*, December 1972.

[17] Wirfs-Brock, R., Wilkerson, B. and Wiener, L., "Designing Object-Oriented Software", Prentice Hall, 1990.