UN ALGORITMO PARA REESTRUCTURAR PROGRAMAS PROCEDURALES

Cobo, Hernán Mauco, María Virginia

Instituto de Sistemas de Tandil

Facultad de Ciencias Exactas

Universidad Nacional del Centro de la Pcia. de Buenos Aires

San Martín 57 - (7000) Tandil - Bs. As. - Argentina

e-mail: hcobo@tandil.edu.ar vmauco@tandil.edu.ar

Resumen

Un aspecto importante en el mantenimiento de software es la existencia de programas viejos que son difíciles de comprender y mantener porque son grandes y están desestructurados.

En este artículo se presenta la primera parte de un proyecto para reestructurar programas escritos en algún lenguaje procedural (Cobol, Pascal, C, etc). El objetivo de esta etapa ha sido, además de la reestructuración sintáctica de los mismos, la definición de una representación que no sólo permitiera almacenar toda la información contenida en el código fuente de un programa sino que también contribuyera a la simplicidad de los algoritmos definidos para transformarla.

El artículo describe el algoritmo de reestructuración desarrollado y muestra que el grafo de dependencias del programa extendido, que es la representación interna definida, es una estructura adecuada para resolver este problema.

Palabras claves: reestructuración de código, dependencias de control, mantenimiento de software, grafo de dependencias del programa, reingeniería.

1. INTRODUCCION

Una gran cantidad del software existente tiene una media de edad de entre 10 y 15 años (Pressman, 1993). Aunque esos programas se hubieran desarrollado con las mejores técnicas de diseño y codificación existentes en su momento, luego migraron a plataformas nuevas, se ajustaron a cambios en el hardware y en los sistemas operativos y se mejoraron para satisfacer las necesidades de nuevos usuarios. El resultado de esto es la existencia de sistemas de software con estructuras de datos pobremente diseñadas, una codificación, una lógica y una documentación pobres, que tienen que seguir funcionando en la actualidad.

El mantenimiento de software se lleva gran parte del presupuesto destinado a un proyecto de desarrollo de software (Pressman, 1993). Por esto, en los últimos años las herramientas de ingeniería reversa y reingeniería están empezando a ocupar un lugar fundamental en el proceso de mantenimiento de software. La ingeniería reversa extrae información de diseño utilizando solamente el código fuente de un programa. La reingeniería toma la información obtenida y reestructura el programa para mejorar su calidad, y por lo tanto simplificar su mantenimiento. El software resultante de la reingeniería reimplementa la función del sistema existente, aunque también se pueden añadir nuevas funciones.

Los cambios continuos tienden a degradar la estructura del software generando costos de mantenimiento más elevados (Griswold et al., 1992). En algunos casos entonces, vale la pena manipular la estructura de un sistema para hacer que los cambios sean más fáciles. Dentro de estas manipulaciones estructurales, las técnicas de reestructuración de código ocupan un lugar muy importante.

El objetivo de la reestructuración es transformar programas no estructurados en programas estructurados equivalentes que se puedan expresar en términos de secuencias y anidamientos de programas más pequeños con una entrada y una salida, y que se puedan leer y entender sistemáticamente (Arnold, 1993).

Sin embargo, la reestructuración manual es una actividad cara y propensa a errores. Por eso, el gran potencial de la reestructuración automática de programas está en manejar la estructura de programas grandes, en la escala de millones de líneas de código, donde hay un gran impacto económico.

En este trabajo se presenta un algoritmo para reestructurar programas escritos en un lenguaje procedural como Cobol, Pascal, C, etc. La reestructuración sintáctica constituye la primera etapa de un proyecto cuyo objetivo final es lograr agrupar todo el conocimiento necesario para obtener, desde un programa fuente, una estructura de más alto

nivel que se pueda analizar y manipular de forma tal que el nuevo código generado a partir de esta estructura, además de preservar la funcionalidad original, sea estructurado, más legible, modular, flexible, reusable, etc.

Se describe además el grafo de dependencias del programa extendido que es la representación interna definida para almacenar y manipular la información extraída del código fuente, y se muestra que mediante transformaciones simples es posible convertir un programa no estructurado en su equivalente estructurado, conservando la funcionalidad del programa original.

2. BENEFICIOS DE LA REESTRUCTURACION

La complejidad de un programa se puede correlacionar, en parte, con la complejidad topológica de su grafo de flujo de control (Bush, 1985). Los programas no estructurados tienen grafos cuyos nodos están tan conectados uno con otro que el grafo no se puede particionar en regiones independientes. Esto implica que los programas se pueden entender solamente como un todo. Por esto, el primer paso para contribuir a la comprensión de los mismos es transformarlos de modo que la lógica del programa y el texto se correspondan precisamente.

El objetivo de la reestructuración es enmendar los programas que han sufrido las cicatrices de decenas de cambios apresurados realizados por imperio de las circunstancias, y que los han convertido en componentes inestables, poco fiables y plagadas de defectos sutiles que sólo aparecen en los momentos menos deseables (Boria, 1992). La reestructuración comprende el proceso de reorganización de la lógica procedural de un programa de modo que cumpla las reglas de la programación estructurada; la reestructuración automática es la traducción mecánica de un programa (no estructurado) en un programa estructurado funcionalmente equivalente.

La programación estructurada es una forma normal sobre los grafos de flujo de control de los programas (Bush, 1985). Un grafo de flujo de control está en forma normal estructurada, si está compuesto solamente de tres tipos de subgrafos simples: la secuencia, el condicional y la iteración, cada uno de los cuales tiene un solo punto de entrada y un solo punto de salida. Por eso, un grafo construido a partir de ellos se puede factorizar en una jerarquía top-down de muchos grafos más pequeños.

Entre los beneficios de la forma normal estructurada (Bush, 1985) quizás el más obvio sea mejorar la comprensión del programa. Dividiendo un grafo en lotes de pequeñas partes auto-contenidas, el código fuente del programa resultante se divide en una cantidad de pequeños módulos auto-contenidos, dando lugar a un programa que tiene

muchas vistas pequeñas, en oposición a un programa no estructurado que tiene sólo una vista grande. Otro beneficio surge cuando se necesita reemplazar o mejorar alguna sección del programa, ya que frecuentemente es posible localizar el cambio a un módulo en particular. También el testing, el debugging y el control de efectos laterales se simplifican considerablemente al realizarse sobre programas estructurados.

3. ALGORITMOS DE REESTRUCTURACION

Los algoritmos de reestructuración de código se pueden dividir en dos grupos: a)- el proceso de reestructuración implica modificar el programa original agregando una o más variables de control que permiten simular estructuración; b)- el proceso de reestructuración involucra el reconocimiento y reemplazo de porciones de código no estructurado por clichés (porciones de código que implementan una estructura de datos o un algoritmo en particular).

Entre los algoritmos del grupo a) se pueden mencionar el de Böhm y Jacopini (Böhm et al., 1966) y el de Linger, Mills y Witt (Linger et al., 1979). Böhm y Jacopini mostraron que todo diagrama de flujo de programa se puede traducir en un programa while equivalente (con una sentencia while), introduciendo nuevas variables lógicas, nuevos predicados para evaluar estas variables, junto con asignaciones para hacer que su valor sea verdadero o falso. Esta transformación es indeseable ya que cambia totalmente la topología del programa, aún cuando el mismo estuviera ya estructurado, dando como resultado un programa que es más difícil de entender. El algoritmo propuesto por Linger, Mills y Witt, agrega una sola variable que actúa como contador de programa, y aunque preserva las partes que tienen estructura aceptable, modifica la estructura general del programa.

Los algoritmos clasificados dentro del grupo b), como el algoritmo propuesto por Bush (Bush, ,1985), trabajan en forma bottom-up y, a partir de una biblioteca de clichés, van armando jerarquías hasta que se ha reestructurado todo el programa o no hay clichés definidos para las porciones de código que quedan. Algunos de los problemas de este enfoque (Rich et al., 1990), son la definición y actualización de una biblioteca de clichés que contenga la mayor cantidad de casos posibles, la existencia de código no reconocible y la limitación de su aplicación a programas de un determinado dominio. Por lo tanto, la solución que proponen este tipo de algoritmos no es completa.

4. DEFINICION DE UNA ESTRUCTURA PARA REPRESENTAR LA INFORMACION DEL PROGRAMA

La definición de una representación interna para los programas constituyó uno de los aspectos más importantes de este trabajo. La relevancia de esta decisión se fundamenta en que dicha representación no sólo tendría que almacenar toda la información posible extraída del código fuente de un programa, en lo que respecta a flujo de datos y de control, sino también en que la complejidad de los algoritmos definidos para transformarla estaría afectada en gran medida por la facilidad o no de acceso y manipulación de la información almacenada en la estructura. Además, el éxito del método de reestructuración dependería directamente de que la información obtenida y almacenada fuera correcta y completa.

La estructura definida es el grafo de dependencias del programa extendido que permite representar las dependencias de control y de datos para cada operación del programa, y que corresponde a una extensión del grafo de dependencias del programa (GDP) propuesto en (Ferrante et al., 1987). Aunque el desarrollo del GDP tuvo su origen en aplicaciones relacionadas con optimización de compiladores, trabajos posteriores demostraron las ventajas del uso de esta representación en ingeniería de software (Horwitz et al., 1988; Horwitz et al., 1992).

4.1. GRAFO DE DEPENDENCIAS DEL PROGRAMA (GDP)

La representación correspondiente al GDP (Ferrante et al., 1987) es un grafo orientado en el cual los nodos son sentencias y expresiones de predicados y los arcos incidentes a un nodo representan los valores de datos y las condiciones de control de las cuales dependen las operaciones del nodo. El conjunto de todas las dependencias de un programa induce un orden parcial sobre las sentencias y predicados en el programa; dicho orden se debe seguir para preservar la funcionalidad del programa original.

Las dependencias son de dos tipos: **dependencias de datos**, que surgen entre dos sentencias cuando una variable que aparece en una de ellas podría tener un valor incorrecto si las dos sentencias se invirtieran, y **dependencias de control**, que se dan entre una sentencia y el predicado cuyo valor controla inmediatamente la ejecución de esa sentencia.

Los arcos del GDP representan dependencias de control o de datos entre componentes del programa. Los arcos de dependencias de control están rotulados T o F, y la fuente de un arco de este tipo es siempre el nodo Entry o un nodo predicado. Un arco de dependencia de control de un nodo n_1 a un nodo n_2, $n_1 \rightarrow_R n_2$, indica que durante la ejecución,

siempre que se evalúa el predicado representado por n_1 y su valor coincide con el rótulo R sobre el arco a n_2, se ejecutará la componente del programa representada por n_2. Existen dos clases de arcos de dependencias de datos (Binkley et al., 1995): arcos de dependencia por flujo y arcos de dependencia def-order. Un arco de dependencia por flujo $v \rightarrow_f w$, va de un nodo v, que representa una asignación a una variable x, a un nodo w que representa un uso de x alcanzado por esa asignación. Un arco def-order va desde un nodo v a un nodo w si ambos contienen una definición de la misma variable x, las dos definiciones alcanzan un uso común de x en un nodo u, es decir $v \rightarrow_f u$ y $w \rightarrow_f u$, y v precede léxicamente a w.

4.2. GRAFO DE DEPENDENCIAS DEL PROGRAMA EXTENDIDO (GDPE)

Como la primera parte de este proyecto apunta a lograr la reestructuración de programas en lo que respecta a su estructura sintáctica, sólo será necesario tener en cuenta las dependencias de control.

El punto de partida para la construcción del GDPE correspondiente a las dependencias de control, es el grafo de flujo de control de un programa, aumentado con un nodo Inicio (representa la entrada al programa) y un nodo Fin (en él confluyen todas las salidas del programa). El grafo de flujo de control es un grafo dirigido en el cual cada nodo representa una sentencia y cada arco representa el flujo de control entre sentencias. Los arcos que representan transferencia de control condicional tienen rótulo T (true) o F (falso); los demás arcos son no rotulados. En (Ferrante et al. 1987) se puede encontrar un método para determinar dependencias de control en programas arbitrarios. En este trabajo se realizó una extensión a la estructura construida por este algoritmo, que consiste en considerar el nivel de cada uno de los nodos .

En general, se define nivel de un nodo n como la longitud del camino que comienza en el nodo Entry y termina en el nodo n. Cuando un nodo n depende de varios nodos, y por lo tanto existen varios caminos hasta él desde el nodo Entry, se define su nivel como la longitud del camino más largo desde el nodo Entry al nodo n, tal que n no es un nodo interior en ese camino. Se considera que el nivel 0, correspondiente al nodo Entry, es el nivel superior.

La importancia de la incorporación del nivel para cada uno de los nodos del GDPE está en que permite determinar rápida y simplemente si un programa está o no estructurado, evitando tener que recorrer el GDPE para detectar ciclos. Además, contribuye a simplificar las transformaciones definidas para convertir un programa no estructurado en uno estructurado.

Por ejemplo, en el GDPE de la Figura 5, el nivel de los nodos 2 y 12 es 4 y 3, respectivamente.

Una de las diferencias más importantes del GDPE con el grafo de flujo de control es que expone paralelismo potencial. Esto significa que dos nodos n_1 y n_2 se podrían ejecutar simultáneamente a menos que existiera una dependencia de datos entre ellos; más aún, se podría intercambiar el orden de ejecución de los nodos sin alterar la funcionalidad del programa original.

Como en esta etapa no se tienen en cuenta las dependencias de datos, se agregan temporariamente arcos que indican el orden de ejecución de los nodos que tienen el mismo nivel, para asegurar la preservación de la funcionalidad del programa original. Este orden se corresponde con el orden impuesto por el grafo de flujo de control.

5. PROGRAMA ESTRUCTURADO Y GDPE ESTRUCTURADO

Tal como se estableció en la sección 2, un programa está estructurado si se construye únicamente a partir de las estructuras de control: secuencia, selección (IF THEN ELSE, IF THEN) e iteración (WHILE DO). En las Figuras 1, 2 y 3 se muestran el grafo de flujo de control aumentado y el GDPE asociado a cada una de estas estructuras.

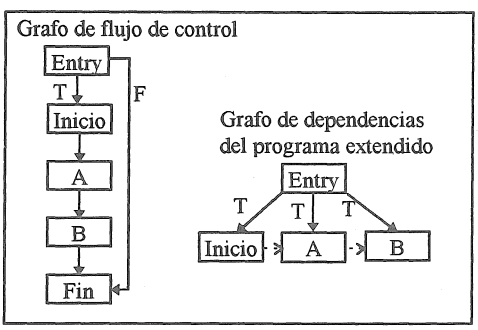

Figura 1: Secuencia

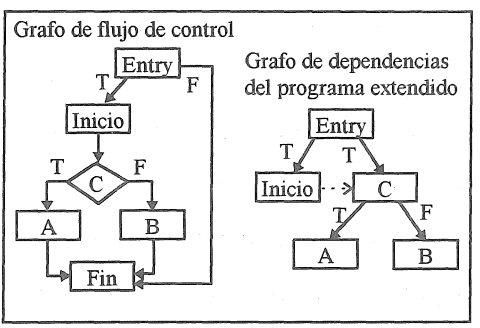

Figura 2: Selección

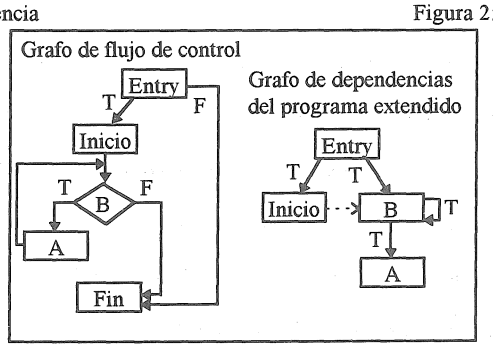

Figura 3: Iteración

Se puede decir entonces que un GDPE, y por lo tanto el programa asociado al mismo, está estructurado si se construye a partir de la composición de los GDPE's correspondientes a las estructuras de control permitidas.

En un GDPE estructurado cada nodo depende por control de solamente uno de los nodos del nivel inmediato superior. Esto implica que el subgrafo inducido por las dependencias de control es un árbol.

Por lo tanto, será necesario y suficiente encontrar un arco de control que destruya la estructura de árbol para determinar que un GDPE, y en consecuencia el programa asociado al mismo, no está estructurado. Esto es equivalente a encontrar un nodo que depende de dos o más nodos.

Se debe notar que para determinar si un GDPE está o no estructurado, no es necesario tener en cuenta los arcos de un nodo a sí mismo, como el arco con rótulo T sobre el nodo B en la Figura 3, ya que la presencia o ausencia de este tipo de arcos sobre un nodo predicado sólo permite determinar si el subgrafo que depende de ese nodo corresponde a una iteración o a una selección respectivamente.

6. ALGORITMO DE REESTRUCTURACION PROPUESTO

Como se mencionó en la sección anterior, es suficiente encontrar un nodo que depende de dos ó más nodos para determinar que un GDPE no está estructurado. Pero haciendo un análisis más detallado de las dependencias de control entre los nodos, se puede establecer que existen dos clases de dependencias, claramente diferenciadas entre sí, que implican desestructuración en el GDPE, dando lugar a los siguientes tipos de desestructuración:

Tipo A) al menos un nodo depende de dos o más nodos de niveles superiores;

Tipo B) al menos un nodo depende de nodos de niveles inferiores, determinando la aparición de un ciclo en el GDPE.

Por eso, las transformaciones posibles a aplicar al GDPE serán dos, una correspondiente a cada tipo. Sin embargo, el objetivo de ambas es el mismo: convertir el subgrafo inducido por las dependencias de control del GDPE en un árbol.

Es importante destacar que los subgrafos del GDPE que no contengan dependencias del **tipo A** ó **B**, es decir que ya estén estructurados, no sufrirán ninguna modificación en el proceso de reestructuración. Por lo tanto, la aplicación del algoritmo de reestructuración a un programa ya estructurado dará como resultado el programa original.

El proceso de reestructuración elimina primero las dependencias de **tipo A**. Comenzando con los nodos de nivel inferior y para cada nodo n que depende de dos o más nodos de niveles superiores n_1, n_2, ..., n_k, se hacen k copias del

subgrafo que tiene a n como nodo inicial y que está constituido por nodos de nivel inferior a n, cada uno con todos sus arcos emergentes; cada copia se hace depender de n_1, n_2, ..., n_k, con el rótulo correspondiente. Esto significa que para aplicar esta transformación a un nodo se deben eliminar, previamente, las dependencias de tipo A para todos los nodos de nivel inferior que dependen de él. Se debe destacar que, aún cuando los nodos a duplicar se eligieran en orden arbitrario, el resultado final de estas transformaciones es el mismo. La razón por la cual se aplican desde los nodos de nivel inferior del GDPE es que de esta forma se puede asegurar la consistencia de los GDPE's intermedios.

Para eliminar los ciclos en el GDPE se tratan las dependencias de tipo B. La existencia de un ciclo en el GDPE significa que el control puede pasar más de una vez por el conjunto de nodos involucrados en el ciclo, y esto se reflejará en el código estructurado en la aparición de una estructura iterativa que no estaba presente en el código original. Por eso, la transformación correspondiente implicará agregar una variable lógica para controlar dicho ciclo.

Para cada ciclo los pasos a seguir son los siguientes:

- detectar el ciclo; esto implica encontrar en el GDPE un arco de control para el cual el nivel del nodo destino n_d es superior al nivel del nodo fuente n_f;
- insertar en el mismo nivel de n_d dos nuevos nodos: 1) un nodo con una asignación T a una variable lógica L, cuyo nombre puede ser determinado por el usuario; 2) un nodo predicado n_p , c}ya condición es que el valor de L sea T, con un arco de control a sí mismo rotulado T; los dos nodos agregados dependerán por control y con el mismo rótulo, del nodo de nivel superior del que dependía el nodo n_d ;
- dependiendo del nodo predicado n_p y con rótulo T, agregar un nodo con una asignación F a la variable lógica L;
- eliminar la dependencia de n_d del nodo de nivel superior, haciendo que n_d dependa por control y con rótulo T del nodo predicado n_p ;
- reemplazar cada dependencia por control con destino n_d y cuyos nodos fuente tengan nivel inferior a n_d , por una dependencia por control con el mismo rótulo que tenga como destino un nuevo nodo con una asignación T a la variable lógica L.

A diferencia de otros algoritmos analizados, el algoritmo desarrollado sólo agrega variables cuando las dependencias de control del programa determinan que el GDPE correspondiente contenga al menos un ciclo.

Las Figuras 4 y 5 contienen parte del texto de un programa escrito en Cobol y el grafo de dependencias del programa extendido correspondiente, respectivamente. Como puede observarse el GDPE no es un árbol; esto significa que el

programa asociado no está estructurado. La aplicación del algoritmo de reestructuración diseñado permite convertirlo en un árbol (Fig. 6), a partir del cual se deriva el código del programa estructurado correspondiente (Fig. 7).

Tanto en el GDPE desestructurado como en el estructurado, el contenido de cada nodo es el siguiente (teniendo en cuenta el tipo de cada uno, sentencia o predicado):

1: SECCION=3
2: READ EMP NEXT RECORD
3: CLOSE EMP
4: MOVE ZERO TO EXTRA
5: HORAS=160
6: CATEGORIA=1
7: CATEGORIA=0
8: EXTRA>0
9: COMPUTE SUEL=160*VALOR_HS+EXTRA
10: COMPUTE SUEL=160*VALOR_HS
11: MOVE SUEL TO SUELDO
12: HORAS>160
13: COMPUTE EXTRA= (HORAS-160)*VAL_EX

```
PROC_SUELDO_EMP.
      IF SECCION = 3 GOTO SALIDA
      ELSE GOTO EMPLEADOS.
HRS_EXTRAS.
      COMPUTE EXTRA=(HORAS-160)*VAL_EX.
      GOTO PROC_SUELDO.
EMPLEADOS.
       READ EMP NEXT RECORD.
       MOVE ZERO TO EXTRA.
       IF HORAS=160 GOTO PROC_SUELDO.
       IF HORAS>160 GOTO HRS_EXTRAS
       ELSE GOTO PROC_SUELDO_EMP.
PROC_SUELDO.
      IF CATEGORIA=1 GOTO EMPLEADOS
      ELSE IF CATEGORIA=0 GOTO SALIDA
      ELSE GOTO VALOR_SUELDO.
SALIDA.
     CLOSE EMP.
      STOP RUN.
VALOR_SUELDO.
      IF EXTRA>0
          COMPUTE SUEL=160*VALOR_HS+EXTRA
      ELSE
           COMPUTE SUEL=160*VALOR_HS.
      MOVE SUEL TO SUELDO.
      GOTO PROC_SUELDO_EMP.
```

Figura 4: Texto Cobol desestructurado

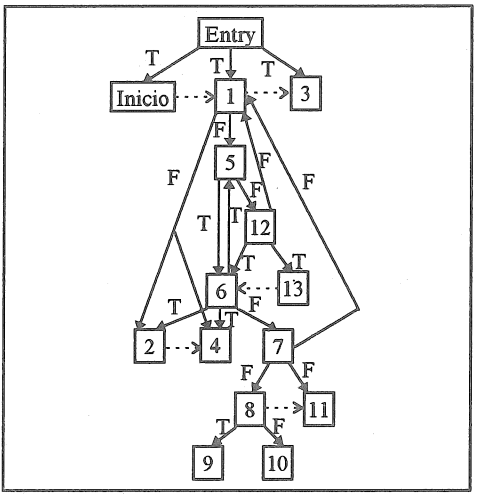

Figura 5: Grafo de dependencias del programa extendido, desestructurado

Los dos tipos de desestructuración planteados en el punto anterior, se corresponden con determinadas características del código fuente del programa. El **tipo A** corresponde a sentencias GOTO "hacia adelante". La solución a este tipo de desestructuración implica duplicación de código. El algoritmo detecta esta duplicación y codifica el conjunto de sentencias duplicadas como un párrafo, permitiendo que el usuario le asigne un nombre. El **tipo B** corresponde a sentencias GOTO "hacia atrás", que determinan la aparición de ciclos en el GDPE. Por eso, la solución implica el agregado de variables, para controlar esos ciclos. Pero como estas variables tienen normalmente una semántica

dentro del programa, se da al usuario la posibilidad de ponerles un nombre para que el programa estructurado resultante sea más legible. Por ejemplo, a las variables L1 y L2 de la Fig. 6 agregadas por el algoritmo de reestructuración, se las puede reemplazar en el código final por CATEG_NO_CERO y CATEG_UNO respectivamente.

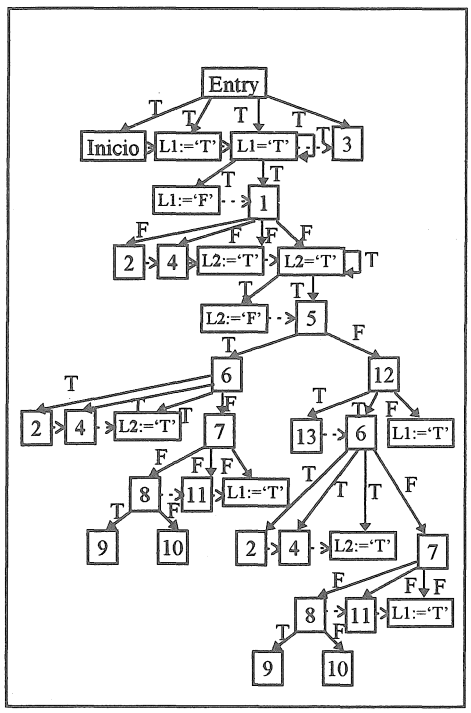

Figura 6: Grafo de dependencias del programa extendido, estructurado

```
PRINCIPAL.
      MOVE 'T' TO CATEG_NO_CERO.
      PERFORM UNTIL CATEG_NO_CERO='F'
        MOVE 'F' TO CATEG_NO_CERO
        IF NOT (SECCION=3)
         READ EMP NEXT RECORD
         MOVE ZERO TO EXTRA
         MOVE 'T' TO CATEG_UNO
         PERFORM UNTIL CATEG_UNO='F'
          MOVE 'F' TO CATEG_UNO
          IF HORAS=160
           PERFORM CALC_SUELDO
          ELSE
          IF HORAS>160
           COMPUTE EXTRA=(HORAS-160)*VAL_EX
           PERFORM CALC_SUELDO
          ELSE
           MOVE 'T' TO CATEG_NO_CERO
          END-IF
          END-IF
         END-PERFORM
        END-IF
      END-PERFORM.
      CLOSE EMP.
      STOP RUN.

CALC_SUELDO.
      IF CATEGORIA=1
        READ EMP NEXT RECORD
        MOVE ZERO TO EXTRA
        MOVE 'T' TO CATEG_UNO
      ELSE
       IF NOT (CATEGORIA=0)
        IF EXTRA>0
         COMPUTE SUEL=160*VALOR_HS+EXTRA
        ELSE
         COMPUTE SUEL=160*VALOR_HS
        END-IF
        MOVE SUEL TO SUELDO
        MOVE 'T' TO CATEG_NO_CERO
       END-IF
      END-IF.
```

Figura 7: Texto Cobol estructurado

7. FUTUROS DESARROLLOS

Como se explicó anteriormente, este trabajo constituye la primera etapa de un proyecto para reestructurar programas escritos en un lenguaje procedural. La próxima etapa del proyecto apunta a transformar un programa sintácticamente estructurado, en un programa modular.

En un programa bien diseñado y poco mantenido, cada funcionalidad del usuario se puede ubicar en un solo módulo. Sin embargo, en código viejo es más probable que la misma esté desparramada por varios módulos o programas, complicando el mantenimiento del mismo.

Por eso, se estudiarán las estructuras y los flujos de datos del programa para desarrollar una técnica que permita reconocer y extraer porciones de código funcionalmente relacionadas para agruparlas en un módulo.

8. CONCLUSIONES

El mantenimiento es la etapa más cara del proceso de desarrollo de software. La estructura de un programa ejerce una gran influencia sobre el costo de mantenimiento.

En este trabajo se presentó un algoritmo para reestructurar sintácticamente programas, que corresponde a la primera etapa de un proyecto cuyo objetivo es mejorar la calidad de los programas haciéndolos estructurados, modulares, reusables, etc. Se mostró además que la simplicidad del algoritmo se debe a la elección del grafo de dependencias del programa extendido como estructura interna para representar la información contenida en un programa.

REFERENCIAS

- Arnold, R., Software Reengineering, 2nd edition, IEEE Computer Society Press, 1994.
- Binkley, D., Horwitz, S., Reps, T., Program Integration for Languages with Procedure Calls, ACM Transactions on Software Engineering and Methodology, Vol. 4, N° 1, 3-35, 1995.
- Böhm, C., Jacopini, G., Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, Communications of the ACM, Vol. 9, N° 5, 366-371, 1966.
- Boria, J., Mauco, M., Moreno, M., Un Prototipo de Ambiente para la Reestructuración de Programas Cobol, Anales de USUARIA '92, Bs. As., Argentina, 133-143, mayo 1992.
- Bush, E., The Automatic Restructuring of Cobol, Proc. of the Conference on Software Maintenance, 35-41, 1985.
- Ferrante, J., Ottenstein, K., Warren, J., The Program Dependence Graph and its Use in Optimization, ACM Transactions on Programming Languages and Systems, Vol. 9, N° 3, 319-349, 1987.
- Griswold, W., Notkin, D., Automated Assistance for Program Restructuring, Technical Report Number CS92-221, University of California, 35 pages, 1992.
- Horwitz, S., Prins, J, Reps, T., On the Adequacy of Program Dependence Graphs for Representing Programs, 15th ACM Symp. on Principles of Programming Languages, 12 pages, Jan 1988.
- Horwitz, S., Reps, T., The Use of Program Dependence Graphs in Software Engineering, 14th International Conference on Software Engineering, 20 pages, 1992.
- Linger, R., Mills, H., Witt B., Structured Programming: Theory and Practice, Addison-Wesley Publishing Company, Cambridge, Mass., 1979.
- Pressman, R., Ingeniería del Software: Un Enfoque Práctico, 3° edición, Mc Graw Hill, 1993.
- Rich, C., Wills, L., Recognizing a Program's Design, IEEE Software, Jan 1990.