ENFOQUTE 3: 29-42

ISSN: 1390-6542

Traducción automatizada de programas entre lenguajes de operaciones

Dr. Diego Ordóñez Camacho, Ph.D.[1]

Resumen

Los lenguajes de operaciones se usan para programar, en procedimientos organizados, las instrucciones a ser ejecutadas por una máquina, para realizar distintos tipos de operaciones. Para poder importar procedimientos existentes realizados en un lenguaje de operaciones dado, en frameworks de programación específicos para el diseño de operaciones, es necesario contar con traductores entre el lenguaje en que ha sido programado el procedimiento, y el lenguaje interno utilizado por el framework de diseño. La construcción de estos traductores puede ser automatizada si se establece un mapeo entre producciones equivalentes de las gramáticas de los lenguajes fuente y destino. Las producciones se pueden considerar equivalentes cuando a pesar de tener una sintaxis diferente, tienen la misma intención semántica y pueden ser equiparadas con un árbol de sintaxis abstracta en común. Debido a ciertas diferencias sintácticas, en ocasiones este árbol sintáctico común no puede ser hallado directamente. Esto se vuelve posible, sin embargo, al realizar en primer lugar ciertas transformaciones a los programas. Este artículo presenta un acercamiento específico a la construcción de reglas de transformación para resolver estas diferencias.

Palabras clave

Lenguajes de programación, lenguajes de operaciones, transformaciones de código, traductores de lenguajes

Abstract

Operations languages are used to program the procedures that instruct machinery to perform specific operations. To import procedures in a given operations language, into a dedicated tool for designing operations, language translators are needed to translate those procedures to the design tool's internal operations language. The construction of these translators can be automated by mapping equivalent productions in the grammar of source and target language. Productions are considered equivalent when, in spite of a differing syntax, they have the same intended semantics and can be matched to a same abstract syntax tree. For some corresponding productions, due to syntactic differences, such a common abstract syntax tree cannot be found directly. However, they can be made to match by performing some specific program transformations first. This paper presents a specific approach to write transformation rules to resolve such mismatches.

Keywords

Programming languages, operations languages, code trabsformations, languages translators

[1] Universidad Tecnológica Equinoccial, Facultad de Ciencias de la Ingeniería, Quito – Ecuador (*ocda88010@ute.edu.ec*).

1. Introducción

Los lenguajes de operaciones son lenguajes de programación diseñados para un entorno específico dentro del campo del control de operaciones. Su propósito es agrupar las instrucciones enviadas a los equipos o máquinas, dentro de un flujo de control organizado llamado un procedimiento. A pesar de que muchos lenguajes de operaciones pueden variar significativamente a nivel sintáctico, proporcionan en gran medida los mismos constructos semánticos, y comparten la misma estructura imperativa y de control de flujo.

Herramientas especializadas para diseñar procedimientos, como el framework Manufacturing and Operations Information System, MOIS (Quigley, Cater, 2006), abstraen las diferencias sintácticas y permiten a los diseñadores de procedimientos trabajar a un nivel conceptual, enfocándose sobre todo en la lógica del control de flujo, más que en particularidades sintácticas.

Para poder importar procedimientos existentes, escritos en un lenguaje de operaciones dado, dentro de una herramienta como MOIS, un traductor automático entre el lenguaje original del procedimiento y el lenguaje interno de MOIS debe ser desarrollado. Considerando que una gran cantidad de lenguajes de operaciones existen, muchos de ellos con varias versiones y dialectos, la tarea de desarrollar los mencionados traductores debe ser automatizada lo más posible.

En experimentos previos desarrollando traductores entre lenguajes de operaciones, ha sido posible alcanzar un nivel satisfactorio de automatización mediante el uso de la técnica de gramáticas anotadas (Ordóñez et al., 2006, 2007). Esta técnica permite a un ingeniero de software anotar explícitamente producciones equivalentes en las gramáticas de los lenguajes fuente y destino. A partir de estas gramáticas anotadas, es posible generar automáticamente las reglas de transformación apropiadas para las producciones que sean puestas en equivalencia con un árbol de sintaxis abstracta, AST (Abstract Syntax Tree), común. Un traductor para programas desde el lenguaje fuente hacia el lenguaje destino, se obtiene finalmente combinando todas las reglas de transformación.

Incluso si para una mayoría de las producciones en el lenguaje fuente, existe una producción equivalente en el lenguaje destino, experimentos preliminares muestran que no siempre es el caso. Para ciertas producciones, a pesar de existir una correspondencia semántica, debido a diferencias sintácticas no es posible derivar el mapeo apropiado directamente. En dichos casos, el programa fuente debe ser en primer lugar re-estructurado, antes de que el mapeo correspondiente pueda ser establecido, y las reglas de traducción correspondientes puedan ser derivadas automáticamente. Las transformaciones necesarias para la re-estructuración de programas, generalmente deben ser definidas a mano por el ingeniero de software que diseña el traductor.

Para facilitar la tarea de estos ingenieros, se ha desarrollado una librería de funciones en Java en la cual se define un conjunto de transformaciones de base (Ordóñez et al., 2007), en términos de

las cuales una gran variedad de re-estructuraciones pueden ser expresadas. El presente artículo ilustra el uso de esta librería de transformaciones para resolver diferencias sintácticas, mediante la presentación de un caso de estudio entre dos lenguajes operacionales

El resto de este artículo está estructurado de la siguiente manera. En la Sección 2 se presenta el uso de la técnica de gramáticas anotadas para construir traductores de programas, y se introduce el caso de estudio. La Sección 3 analiza algunas de las diferencias sintácticas encontradas durante el estudio, y explica cómo se lidió con las mismas gracias al uso de la librería de transformaciones en Java. La Sección 4 discute otros trabajos relacionados con el tema, y la Sección 5 extrae conclusiones y presenta posibles alternativas de trabajo a futuro.

2. Mapeo de gramáticas

Derivación automatizada de traductores a partir de gramáticas anotadas

Como se explicó en la Sección 1, una posibilidad para automatizar el desarrollo de traductores de programas es el uso de gramáticas anotadas para mapear producciones correspondientes en las gramáticas fuente y destino, aprovechando esto para derivar automáticamente las reglas de transformación para dichos casos (Ordóñez et al., 2006). Una condición necesaria para que este enfoque funcione es que los nodos de los árboles sintácticos de las producciones correspondientes se encuentren en una relación de mapeo uno a uno con un AST común.

La Figura 1 muestra un ejemplo de tal mapeo: los círculos representan los nodos significativos en las producciones de los lenguajes fuente y destino que queremos relacionar. Los diamantes representan las anotaciones que ligan estos nodos relacionados en ambas producciones, y dichos diamantes están organizados en un AST que emula la sintaxis concreta de las producciones en ambos lenguajes, fuente y destino. Los nodos presentados con rectángulos, representan símbolos (palabras reservadas del lenguaje) en ambas gramáticas, los cuales son irrelevantes dado que pueden ser inferidos automáticamente durante el proceso de generación de código fuente. Esencialmente el AST oculta los nodos rectangulares de las gramáticas fuente y destino, y unifica los nombres de los nodos en ambas gramáticas.

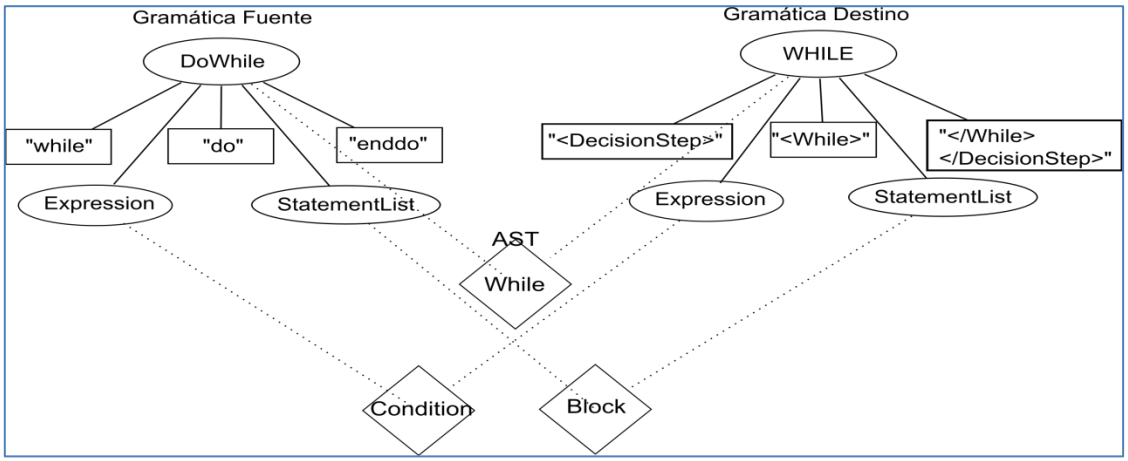

Figura 1. Mapeo de producciones equivalentes en dos gramáticas.

En la Figura 2 se muestra la representación anotada en el formato Syntax Directed Formalism, SDF (Heerings et al., 1989), de las dos producciones mapeadas en la Figura 1. Gracias a la anotación común “While”, al extremo derecho de las producciones, el ingeniero de software indica considerar que la producción fuente “DoWhile” es equivalente a la producción destino “WHILE”. De forma similar se utilizan anotaciones para declarar la equivalencia entre las producciones fuente “Expression” y “StatementList”, con las producciones destino “MoisExpression” y “StepList” (no presentadas en la Figura 2 para simplificarla).

```
                    Source production
"while" Expression
"do" StatementList
"enddo"                -> DoWhile        {cons("While")}

                    Target production
"<DecisionStep>"
      MoisExpression
      "<While>"
         StepList
      "</While>"
"</DecisionStep>"      -> WHILE          {cons("While")}
```

Figura 2. Dos producciones correspondientes, en formato SDF, pertenecientes a los lenguajes STOL y MOIS

La manera como la técnica de gramáticas anotadas aprovecha esta información para generar un traductor entre dos lenguajes se describe a continuación:

i) Por cada producción en la gramática fuente, las anotaciones señalan los símbolos relevantes del árbol, especificando de esta manera cómo generar el AST: los símbolos anotados permanecerán en el árbol, mientras los restantes serán descartados.

ii) Los nombres de los símbolos en el AST corresponderán a las anotaciones, en lugar que a los símbolos originales de la gramática fuente.

iii) Asumiendo que no existan diferencias estructurales entre la producción fuente y su correspondiente producción destino, el proceso de traducción inverso hacia la gramática destino se puede efectuar directamente. Caso contrario las diferencias

deben resolverse aplicando transformaciones al AST para que pueda ser mapeado al lenguaje destino.

iv) Los símbolos vuelven a ser modificados cambiando el nombre abstracto por el concreto, mediante una búsqueda de correspondencia en la gramática destino, basada en las anotaciones.

v) La generación de código fuente se consigue inyectando las palabras clave o reservadas, así como los símbolos adicionales del AST, en la posición esperada de acuerdo a la gramática destino.

Este proceso se desarrolla automáticamente, salvo por el paso iii), donde se requiere una cierta dosis de intervención por parte del ingeniero, para aquellos casos donde diferencias estructurales impiden que el AST derivado a partir de la gramática fuente, pueda ser mapeado a la gramática destino.

Debido a la gran similitud existente entre muchos lenguajes de operaciones, para muchas de las producciones en el lenguaje fuente, se puede encontrar una producción semánticamente equivalente en el lenguaje destino. Inclusive, en caso de no existir diferencias estructurales, el proceso detallado puede ser aplicado de manera automática en su totalidad. El presente estudio, sin embargo, se concentra más bien en aquellos casos donde sí existen diferencias estructurales entre producciones correspondientes. Estos casos requieren de la intervención del ingeniero de software para que proporcione las reglas que trasformen el AST de manera que la traducción a la gramática destino sea posible.

Caso de estudio: traduciendo desde STOL hacia MOIS

Para explorar los diversos tipos de diferencias estructurales que pueden presentarse en la práctica, se presenta un caso de estudio en el cual se construye un traductor desde el lenguaje Spacecraft Test and Operations Language, STOL (Integral Systems, 2000), hacia el lenguaje MOIS (Quigley, Cater, 2006). Aunque en el caso de estudio se trabaja con los lenguajes completos, para facilitar la explicación en este artículo se han simplificado algunos de los detalles de las gramáticas y procedimientos.

En la Figura 3 se presenta un procedimiento (simplificado) de ejemplo escrito en STOL, y en la Figura 4 se puede ver su equivalente en MOIS, hacia el cual el procedimiento en STOL deberá ser traducido. La instrucción en la línea 1 del procedimiento en STOL presenta una instrucción *Step* consistente en una instrucción *Declaration* con dos variables, y una instrucción asociada *Comment*. Esta instrucción *Step* se traduce a MOIS en las líneas 3 a 10. La segunda instrucción *Step* (línea 2) en el procedimiento STOL consiste de una instrucción *Assignment* con una instrucción *Comment* asociada. Se traduce en las líneas 17 a 25 del procedimiento en MOIS. La instrucción *Telemetry Declaration* junto a la instrucción asociada *Comment*, en la siguiente

instrucción *Step*, en la línea 3 del procedimiento STOL, corresponde a las líneas 11 a 14 del procedimiento en MOIS.

```
1 local vOne, vTwo #variables sencillas
2 vOne = 1 #asignamiento
3 point tOne #telemetría
4 write tOne
```

Figura 3. Un procedimiento STOL, simplificado.

```
1 <Proc>
2   <Header>
3     <Declare>
4       <Identifier>vOne</Identifier>
5       <Comment>#simple variables</Comment>
6     </Declare>
7     <Declare>
8       <Identifier>vTwo</Identifier>
9       <Comment>#simple variables</Comment>
10    </Declare>
11    <Declare>
12      <Identifier>tOne</Identifier>
13      <Comment>#telemetry variable</Comment>
14    </Declare>
15  </Header>
16  <Body>
17    <Step>
18      <Comment>#assignment</Comment>
19    </Step>
20    <Step>
21      <Assign>
22        <Variable><Identifier>vOne</Identifier></Variable>
23        <Value><Identifier>1</Identifier></Value>
24      </Assign>
25    </Step>
26    <Step>
27      <GetTelemetry>
28        <Variable><Identifier>tOne</Identifier></Variable>
29        <Telemetry><Identifier>tOne</Identifier></Telemetry>
30      </GetTelemetry>
31    </Step>
32    <Step>
33      <Print><Identifier>tOne</Identifier></Print>
34    </Step>
35  </Body>
36 </Proc>
```

Figura 4. Un procedimiento MOIS, simplificado.

Diferencias estructurales entre STOL y MOIS

Ahora se verán más detenidamente las gramáticas de los lenguajes de operaciones STOL y MOIS, para cada uno de los cuales se dio un ejemplo de código en la sección anterior. La Figura 5 muestra parte de la gramática (simplificada), en formato SDF, del lenguaje STOL, al cual consideramos nuestro lenguaje fuente. La Figura 6 presenta una versión simplificada de una parte de la gramática SDF del lenguaje MOIS, al cual se considera el lenguaje destino. Para facilitar la explicación se ha retirado previamente de las gramáticas los símbolos terminales irrelevantes, y solo se ha conservado los símbolos no terminales significativos para el mapeo. Las dos

gramáticas han sido decoradas con las anotaciones necesarias para establecer el mapeo entre las producciones equivalentes en los dos lenguajes.

Si se revisa cuidadosamente estas gramáticas se puede observar lo siguiente:

- Las cuatro producciones en las líneas 6 a 9 de la gramática de STOL pueden ser mapeadas directamente a las últimas cuatro producciones en MOIS.
- Para la producción en la línea 5 de la gramática de STOL, no hay producción equivalente en MOIS. En efecto, en MOIS las telemetrías no necesitan ser declaradas antes de usarlas.
- De forma similar, para cada una de las producciones en las líneas 2, 3 y 7 de la gramática de MOIS, no hay producciones equivalentes en STOL. STOL no maneja la noción de encabezado y cuerpo. Un procedimiento en STOL es solo una secuencia de instrucciones consecutivas. Además STOL no maneja tampoco la noción de asignación a telemetría, dado que las telemetrías, una vez declaradas, se tratan como cualquier otra variable.
- Para cada una de las tres producciones en las líneas 1 a 4 de STOL, sí existe una correspondencia con las producciones en las líneas 1, 4 y 6 de la gramática de MOIS. Sin embargo existen diferencias estructurales que impiden un mapeo directo.

```
1 Statement+                    -> StolProcedure        {cons("Procedure")}
2 (Declaration | TlmDeclaration | Assignment | Print) Comment?
3                               -> Statement            {cons("Step")}
4 Identifier+                   -> LocalDeclaration     {cons("Declaration")}
5 Identifier+                   -> PointDeclaration     {cons("Tlm Declaration")}
6 Identifier Identifier         -> Assignment           {cons("Assignment")}
7 Identifier                    -> Write                {cons("Print")}
8 <lexical definition>          -> Identifier           {cons("Identifier")}
9 <lexical definition>          -> Comment              {cons("Comment")}
```

Figura 5. Gramática STOL, simplificada.

```
1 Header Body                   -> Proc                 {cons("Procedure")}
2 (Variable | Comment)*         -> Header               {cons("Header")}
3 MoisStmt+                     -> Body                 {cons("Body")}
4 Comment | Assignment | TlmAssignment | Print
5                               -> MoisStmt             {cons("Step")}
6 Identifier Comment?           -> Variable             {cons("Declaration")}
7 Identifier Identifier         -> SetTelemetry         {cons("Tlm Assignment")}
8 Identifier Identifier         -> SetVariable          {cons("Assignment")}
9 Identifier                    -> Print                {cons("Print")}
10 <lexical definition>         -> Identifier           {cons("Identifier")}
11 <lexical definition>         -> Comment              {cons("Comment")}
```

Figura 6. Gramática MOIS, simplificada.

Para aquellos casos en los cuales existe un mapeo directo entre las producciones equivalentes, las reglas de traducción pueden ser derivadas automáticamente por la técnica de gramáticas anotadas (Ordóñez et al., 2008). Para los otros casos es necesario que se intervenga antes de la generación.

En el caso del estudio completo, un total de 150 producciones en STOL necesitaban relacionarse con 62 producciones en MOIS. Para 109 de las producciones de STOL fue posible encontrar un mapeo directo entre los AST fuente y destino. Las 41 producciones restantes de STOL presentaron diferencias estructurales que requirieron cierto tipo de intervención. Los diferentes tipos y cantidad de intervenciones realizadas en el AST para resolver las diferencias se resume en la Figura 7.

Categoría	Frecuencia
Mover nodos a un lugar diferente del AST	7
Remplazar parcial/completamente el contenido de un nodo	28
Transformar el tipo de un nodo	8
Crear nodos adicionales	13
Eliminar nodos del AST	3

Figura 7. Tipos de transformaciones utilizadas para resolver diferencias estructurales

3. Resolución de diferencias estructurales

Esta sección presenta cómo, haciendo uso de una librería de transformaciones desarrollada con anterioridad en Java, se logran superar las diferencias estructurales entre los dos lenguajes. Esencialmente se toma el AST derivado a partir de la gramática de STOL y se lo transforma gracias a la librería de tal manera que corresponda con un AST válido derivado de MOIS, el cual pueda usarse para generar un programa fuente válido en MOIS.

El enfoque utilizado se ilustra gracias a tres ejemplos representativos extraídos del ejemplo que se presentó (simplificado) en la sección anterior. Para cada uno de los ejemplos se explicará:

- **Diferencia:** por qué existía una diferencia estructural entre las producciones correspondientes de STOL y MOIS.
- **Transformación:** cómo se resolvió la diferencia transformando el AST del programa fuente hasta convertirlo en un AST compatible con el lenguaje destino.
- **Implementación:** cómo se uso la librería de transformaciones en Java para efectivizar la transformación.

Telemetrías

En los lenguajes de operaciones, las telemetrías son instrucciones especiales enviadas al equipo para recolectar la información proveniente del estado de los sensores.

Diferencia: en STOL, las telemetrías se declaran mediante una instrucción específica *Telemetry Declaration*, la cual difiere de la instrucción *Declaration* usada para otros tipos de variables. En MOIS no hace falta declarar explícitamente las telemetrías. Adicionalmente, luego de la declaración, cuando es necesario referenciar una telemetría en STOL se lo hace como si se tratara de una variable cualquiera, permitiéndoles participar directamente en expresiones más complejas. En MOIS, las telemetrías solo pueden ser usadas a través de la instrucción *Telemetry Assignment*.

Transformación: para resolver la diferencia estructural se procede como sigue.

i) Se atraviesa el AST del procedimiento STOL para encontrar todos los nodos *Statement*, dentro de los cuales se encuentre un nodo *Identifier* que referencie una telemetría.

ii) Luego de recolectar los nodos con telemetrías, se reemplaza el nodo *Identifier* por uno representando una variable normal, que pueda ser utilizado en cualquier expresión de MOIS.

iii) Adicionalmente, dado que el valor de la telemetría debe asignarse a esta nueva variable, antes de cada instrucción *Statement* conteniendo un identificador de telemetría, es necesario añadir un nuevo nodo *Telemetry Assignment*, donde la nueva variable *Identifier* reciba el valor de la telemetría original.

iv) Finalmente, dado que MOIS no utiliza declaraciones específicas para las telemetrías, remplazamos todos los nodos *Telemetry Declaration*, por nodos *Declaration* que contengan las variables regulares.

Implementación: el uso de la librería para realizar la transformación se presenta en la Figura 8.

```
/*iterar por los identificadores de telemetrías*/
for(Node node :
fun.getNodes("//Identifier[not(ancestor::TlmDeclaration)]" +
  "[string/text()=//TlmDeclaration/list/Identifier/string/text()]")) {
  /*crear nodo de asignamiento de telemetría*/
  Node stat = fun.getNode(node, "ancestor::Statement[1]");
  Node newN = fun.createBefore(stat, "Statement/TlmAssignment")
    .getFirstChild();
  newN.appendChild(fun.clone(node));
  newN.appendChild(fun.clone(node));
}
/*cambiar tipo de declaración de telemetría*/
fun.rename("//TlmDeclaration", "Declaration");
```

Figura 8. Transformando telemetrías con la librería Java.

Encabezado y cuerpo del procedimiento

Diferencia: al comparar las definiciones de la producción *Procedure*, en STOL y MOIS, se puede ver que los procedimientos en MOIS constan de un *Header* y un *Body*, los cuales no están presentes en procedimientos de STOL

Transformación: para adaptar esta sección del AST de STOL, haciéndolo compatible con MOIS, se requieren los siguientes pasos.

i) En STOL, un procedimiento tiene una sola lista de nodos *Statement* donde se ubican todas las instrucciones a ser ejecutadas. Es necesario recubrir esta lista con un nuevo nodo *Body*.

ii) Justo antes del recientemente creado nodo *Body* (que ahora contiene todas las instrucciones), es necesario crear un nuevo nodo *Header*.

iii) En MOIS solo es posible declarar variables al interior del *Header*, de tal manera que es necesario recolectar todos los nodos *Declaration* y moverlos desde *Body* hacia *Header*.

iv) En STOL es permitido declarar más de una variable en una sola instrucción. En MOIS esto se prohíbe, siendo necesario una instrucción *Declaration*, con una sola variable, por instrucción. Es necesario entonces crear un nuevo nodo *Declaration* por cada nodo *Identifier* dentro de un nodo *Declaration* en STOL.

Implementación: en la Figura 9 se puede observar el código Java para este caso.

```
/*recubrir instrucciones con un nodo Body*/
fun.wrap("/Procedure/list", "Body");
/*crear la estructura Header*/
fun.createInsideFirst("/Procedure", "Header/at:list");
/*mover declaracines dentro del encabezado*/
fun.moveInside("//Statement[Declaration]", "/Procedure/Header/list");
/*dividir declaraciones múltiples de variables*/
fun.removePreserveContent("//Declaration/list");
fun.splitChilds("//Declaration");
```

Figura 9. Transformando encabezado y cuerpo

Comentarios

Finalmente, en el tercer caso analizado, se revisa cómo adaptar los comentarios. Uno de los requerimientos de este experimento era mantener los comentarios, respetando al máximo su posición relativa en el código fuente.

Diferencia: dos diferencias significativas existen entre ambos lenguajes. Primero, STOL acepta un nodo *Comment* al final de la línea, formando parte de la instrucción que inicia la línea. En MOIS, los nodos *Comment* son siempre una instrucción independiente. Segundo, STOL no establece ninguna diferencia en cuanto al contexto en el cual un comentario aparece. Para MOIS, si un comentario está atado a un nodo *Declaration*, debe ir al interior del mismo, y no antes, como ocurre en el caso general.

Transformación: los pasos a seguir vienen a continuación.

i) Para cada nodo *Comment*, al interior de nodos *Statement* que declaren variables, mover el nodo *Comment* al interior del nodo *Declaration* relacionado. Un comentario puede ser compartido por más de una declaración de variable, en cuyo caso deberá ser replicado para cada una de ellas.

ii) Tomar los nodos *Comment* de fin de línea, y moverlos justo antes del nodo *Statement* al cual pertenecen, recubriéndolos de un nodo *Statement* independiente.

iii) Eliminar los nodos *Comment* vacíos.

Implementación: la implementación de esta transformación se presenta en la Figura 10.

```
/*replica comentarios en declaraciones*/
for (Node node : fun.getNodes("//Statement/Declaration")) {
node.appendChild(fun.clone(node, "../Some | ../None"));
}
/*extraer comentarios e independizarlos*/
for (Node node : fun.moveBefore(
"//Statement[not(Declaration)]/Some[Comment]", "..")) {
fun.rename(node, "Statement");
}
/*eliminar comentarios vacíos*/
fun.remove("//Statement/None | //Statement/Some");
fun.removePreserveContent("//Header/list/Statement");
```

Figura 10. Transformando comentarios.

4. Discusión

La librería de transformaciones utilizada se compone de una serie de funciones de transformación primitivas, las cuales pueden ser combinadas para obtener transformaciones más complejas. Estas funciones primitivas fueron reutilizadas en diversas situaciones y combinaciones (ver la Figura 7) para resolver las diferencias estructurales encontradas. Esta reusabilidad permitió a los programadores escribir transformaciones bastante complejas, mientras que solo un número relativamente pequeño de funciones dentro de la interfaz de programación debía dominarse.

Analizando en retrospectiva el experimento completo de construir un traductor desde STOL hacia MOIS, se nota que al menos para una tercera parte de las producciones cierto tipo de intervención fue necesaria. Se ve así que la cantidad de trabajo adicional para completar el traductor es significativa. Afortunadamente es notorio que la implementación de las intervenciones presenta un alto nivel de repetitividad, lo cual pone de manifiesto que es muy probable incrementar el grado de automatización, inclusive generando un lenguaje dedicado para la definición de transformaciones.

5. Trabajos relacionados

La técnica de gramáticas anotadas, sobre la que este trabajo se fundamenta, fue presentada por Ordóñez et al. (2010). Dicha técnica aplica principios de la transducción dirigida por la sintaxis de Lewis y Stearns (1968), de la grammarware de Klint et al. (2005) y de la técnica de inversión de gramáticas de Yellin (1988), principalmente.

Varios problemas específicos de traducción de programas han sido sujeto de estudio, como aquellos de Lämmel (2001) y Terekhov (2000, 2001), muchos de los cuales sirvieron de inspiración también para el presente enfoque. Visser (2001) describe de manera apropiada las alternativas existentes.

Entre las técnicas relacionadas existentes, merece la pena mencionar las siguientes: Wyk (2002) estudia un mecanismo de expansión para añadir modularmente nuevas características a un lenguaje mediante el uso de gramáticas de atributos. Schürr (1994) estudia traductores de grafos, en los cuales las relaciones se describen a través de reglas de correspondencia. Wile (1991) proporciona una forma alternativa de generar traductores basada en sets de reglas dirigidas por la

sintaxis. Similar a nuestro enfoque, Cleenewerck (2005) utiliza linglets, que dividen cada problema de traducción en los componentes que lo constituyen para ejecutarlos en pasos lógicos separados. Finalmente, un enfoque muy cercano a la librería descrita en este artículo lo presenta Moreau (2003), con su framework TOM, el cual muestra cómo extender un programa como Java o C con el soporte necesario para facilitar la manipulación de sus propios AST.

6. Conclusiones y trabajo a futuro

En trabajos preliminares se estudió cómo construir automáticamente reglas de transformación para traducción de programas, mediante el uso de anotaciones de mapeo incluidas en las gramáticas de los lenguajes. Para ser efectiva, esta técnica requiere de una equivalencia directa entre los AST correspondientes a las producciones mapeadas, y generalmente era posible encontrar muchas diferencias estructurales entre dos gramáticas.

En este trabajo se ha intentado resolver las diferencias estructurales entre los AST de diferentes lenguajes, mediante el uso de una técnica operacional. En primer lugar se diseñó una librería de funciones específicas de alto nivel para ejecutar transformaciones de programas. Luego se utilizó dicha librería para definir las transformaciones apropiadas que permitan equiparar los AST de las gramáticas fuente y destino.

A futuro es de interés estudiar alternativas para incrustar en las gramáticas de los lenguajes anotaciones simples pero significativas, que permitan describir cómo un AST proveniente de un lenguaje fuente, debe ser transformado para adaptarlo a un lenguaje destino, en casos más complejos que la equivalencia uno a uno.

Bibliografía

Cleenewerck, T., & D'Hondt, T. (2005). Disentangling the implementation of local to global transformations in a rewrite rule transformation system, *Proceedings of the 2005 ACM symposium on Applied computing*. pp. 1398–1403.

Integral Systems (2000). EPOCH T & C Directives and STOL Functions Reference Manual. *Kratos Integral Systems International, Inc.* 5200 Philadelphia Way - Lanham MD. 20706 - U.S.A. http://www.integ.com/.

Klint, P., Lämmel, R., & Verhoef, C. (2005). Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM*), 14, 3 (July 2005), pp. 331-380.

Lämmel, R., & Verhoef, C. (2001) Cracking the 500-Language Problem, *IEEE Software*, pp. 78–88.

Lewis, P. M., & Stearns, R. E., (1968). Syntax-Directed Transduction. *Journal of the ACM,* 15, 3 (July 1968), pp. 465-488.

Moreau, P-E., Ringeissen, C., & Vittek. M. (2003). A pattern matching compiler for multiple target languages. *Proceedings of the 12th international conference on Compiler construction (CC'03),* Springer-Verlag, Berlin, Heidelberg, pp. 61-76.

Ordóñez Camacho, D., Mens, K., van den Brand, M., & Vinju, J. (2006) Automated derivation of translators from annotated grammars. *Electronic Notes in Theoretical Computer Science* 164, Issue 2, pp. 121–137.

Ordóñez Camacho, D., & Mens, K. (2007). Using annotated grammars for the automated generation of program transformers*. Ingénierie Dirigée par les Modéles, IDM 2007*, pp. 7 – 24.

Ordóñez Camacho, D., & Mens, K. (2008). APPAREIL: A Tool for Building Automated Program Translators Using Annotated Grammars. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08),* pp. 489-490.

Ordóñez Camacho, D., Mens, K., van den Brand, M., & Vinju, J. (2010). Automated generation of program translation and verification tools using annotated grammars. *Science of Computer Programming,* 75, 1-2 (January 2010), pp. 3-20.

Quigley, D., & Cater, S. J. (2006). Satellite test and operation procedures cost reduction through standardization. *IEEE Aerospace Conference*., p. 10.

Schürr, A. (1994). Specification of Graph Translators with Triple Graph Grammars. *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94),* Springer-Verlag, London, UK, pp. 151-163.

Terekhov, A., (2001). Automating Language Conversion: A Case Study, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pp. 654–658.

Terekhov, A., (2001); Automating language conversion: a case study, *IEEE International Conference on Software Maintenance*, pp. 654–658.

Terekhov, A., & Verhoef, C. (2000). The realities of language conversions, *IEEE Software*, 17, pp. 111–124.

Visser, E. (2001). A survey of strategies in program transformation systems*. Electronic Notes in Theoretical Computer Science.* 57, pp. 109 – 143.

Wile, D. S. (1991). Popart Manual. *USC/Information Sciences Institute*.

Wyk, E. V., de Moor, O., Backhouse, K., & Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. *Computational Complexity*, pp. 128–142.

Yellin. D. M. (1988). *Attribute Grammar Inversion and Source-To-Source Translation*. Springer-Verlag New York, Inc., New York, NY, USA.