Construcción de un Compilador de Asertos de Programación Metódica Utilizando El Método De Autómatas Adaptativos (28 de agosto de 2014)

D. Berolatti y C. Zapata, *Member IEEE*

***Abstract*— The project presented in this article carries on a study of the development about an asset's compiler using the adaptive automata technique. These automatons are designed trying to solve a pair of assets. These assets represent the initial and final states of executing a program. Being the program the solution of the assets, the compiler uses the methodic programing technique to solve the pair of assets. It is possible to execute formal tests to evaluate the program generated by the compiler and compare the result with others using the same method programming technique by hand.**

Keywords* — *compiler, assets, methodic programming technique, adaptive automata.

I. Nomenclatura

- AA: Autómata adaptativo.
- TPM: Técnica de programación metódica.

II. Introducción

Siempre ha existido la necesidad de validar la codificación de un programa. Este proyecto tiene como objetivo la implementación de un compilador que, mediante notaciones matemáticas que especifican un programa, genere las instrucciones de manera automática. El resultado de la compilación tiene como principal característica que sea formalmente correcto. Esto se da debido a la utilización de una metodología llamada derivación de programas la cual garantiza esa característica.

Este trabajo ha sido apoyado por la Pontificia Universidad Católica del Perú a través del curso de Tesis que se imparte en la especialidad de Ingeniería Informática de la Facultad de Ciencias e Ingeniería.

Claudia Zapata imparte docencia en el Departamento de Ingeniería, Sección de Informática de la Pontificia Universidad Católica del Perú, Av. Universitaria 1801, San Miguel, Lima 32, Perú (correo e.: zapata.cmp@pucp.edu.pe).

Diego Berolatti es alumno de pregrado en la Especialidad de Ingeniería Informática de la Facultad de Ciencias e Ingeniería de la Pontificia Universidad Católica del Perú, Av. Universitaria 1801, San Miguel, Lima 32, Perú (correo e.: berolatti.diego@pucp.edu.pe).

La implementación de esta metodología se da mediante la estructura formal de un compilador y la inclusión de un autómata adaptativo capaz de aplicar las reglas de programación metódica.

El proyecto tiene como limitación, no aplicar ninguna regla que implique resolver un problema de complejidad *np*. Debido a esto la expresividad del lenguaje y su capacidad de generación automática se encuentra limitada. El resultado es un compilador capaz de generar código de manera automática en base a las especificaciones que es capaz de compilar. Este proyecto es la base de los compiladores de programación automática.

III. Marco conceptual

Existen metodologías, técnicas y herramientas cuya función es la de corregir errores cometidos durante la codificación de un programa de computadora como lo hecho por Dijkstra [7] y Rosenblum[10]. Sin embargo, éstos no garantizan que el software producido contenga la menor cantidad de imperfecciones posible, excepto aquellos que se basan en las leyes de Charles Richard Hoare [1]. La lógica de Hoare es un sistema formal que proporciona una serie de reglas de inferencia para razonar sobre la corrección de programas imperativos con el rigor de la lógica matemática y su aplicación garantiza que el programa que las utilice cumpla con las especificaciones que este tenga a priori [2].

Para entender estas reglas es importante explicar antes lo que es el triple de Hoare. El triple de Hoare tiene la forma {P} C {Q} y es una estructura formada por dos expresiones matemáticas con un resultado de tipo booleano llamadas asertos (P y Q) y un comando o instrucción (C). El aserto P, llamado precondición, es una expresión que representa el estado del programa antes de ejecutar la instrucción C. Mientras que el aserto Q, llamado postcondición, representa el estado del programa después de ejecutar la instrucción C.

El conjunto de reglas busca en cada momento que la precondición y la postcondición sean verdaderos, en consecuencia el conjunto de instrucciones formadas en C van a cumplir las especificaciones. Esto debido a que las

condiciones iniciales de las especificaciones de C son representadas en la precondición y el resultado de estas en la postcondición. Es por eso que todo programa, que sea capaz de seguir estas reglas o cualquier conjunto de reglas derivadas, es considerado "formalmente correcto". Según [2] estas reglas son tan importantes en el desarrollo de programas como lo son las leyes físicas al momento de construir un edificio o hacer un circuito electrónico.

Una adaptación de estas reglas, llamada programación metódica, permite mediante la inclusión de métodos formales deducir el conjunto de instrucciones C. Esto se logra a partir de definir la precondición y la postcondición en función de las especificaciones deseadas para las instrucciones. Esta forma de construir código a partir de asertos, llamada derivación, conserva las mismas cualidades que la lógica de Hoare debido a que siguen sus reglas. La programación metódica permite entonces construir instrucciones formalmente correctas, siendo su aplicación una forma de garantizar que el software producido tenga una mínima cantidad de errores.

Sin embargo, de acuerdo a [3], debido al corto tiempo del que ahora disponen los programadores para realizar esta tarea y a que el dominio de las nuevas tecnologías se ha convertido en lo más importante durante el desarrollo de un producto de software es que ha existido siempre la necesidad de automatizar técnicas que permiten eliminar errores. Además afirma que la enseñanza en el pregrado de los métodos formales aplicados en la programación metódica ha disminuido su relevancia frente a otros temas.

Por otro lado los compiladores de los lenguajes más usados actualmente no incluyen alguna técnica automatizada que permita corregir errores en los programas [3].

Es entonces que ante esta problemática, se desarrollará un compilador de asertos de programación metódica cuyo fin sea el uso de estos métodos formales de manera automática y cuyo resultado se encuentre en un lenguaje de alto nivel. De esta forma se obtendrá un programa formalmente correcto con menor esfuerzo de parte del programador.

El proyecto utilizará la estructura general de un compilador convencional adaptando cada una de sus partes en función a lo requerido. La primera parte de este proyecto se hizó el análisis de viabilidad y documentación del conjunto de reglas a aplicar. Luego y en función a ellas, se desarrolló un compilador, siguiendo las etapas de análisis léxico, sintáctico y semántico. Y finalmente, se implementó el traductor automático.

IV. Estado Del Arte

A continuación se mencionaran proyectos ya existentes relacionados:

a) Forma computarizada aproximada de resolver el problema
   - The KeY System: Software de verificación de programas, soporta un subconjunto de estructuras del lenguaje Java llamado JavaCard y cuya verificación está basada en lógica dinámica, una generalización de la lógica de Hoare. Utiliza además para la especificación de objetos Uml y Ocl, No es lo suficientemente expresiva como para especificar el comportamiento de un programa. Además la verificación se realiza después de construir el programa.

b) Procedimientos aproximados para resolver el problema:
   - Guarded Command Language : es un lenguaje definido por Esdger Dijkstra que incorpora la lógica de Hoare mediante una estructura llamado "Guarded Command" o instrucciones protegidas. Tienen el mismo funcionamiento que las instrucciones protegidas de las instrucciones alternativas de la programación metódica pero se extienden a cualquier condicional del lenguaje y ofrecen una variante que ayuda a garantizar algoritmos determinísticos. Solo integra (instrucciones protegidas) de manera parcial las reglas de la lógica de Hoare.

   - Diseño por contrato: es una forma de diseñar software, se utiliza la lógica de Hoare en forma de metáfora afirmando que para uno existen obligaciones y beneficios a los cuales uno está ligado mediante un contrato. Mediante esto contrato uno asume roles de proveedor o de cliente buscando en cualquiera de los casos encontrar todas las formas de las cuales las obligaciones y los beneficios de un contrato se garanticen. Se encarga de garantizar el diseño del software pero no garantiza la correctitud formal del programa generado.

   - Asertos embebidos: herramienta desarrollada con el fin de detectar automáticamente errores de ejecución en distintas versiones de un sistema de software. Utiliza asertos los cuales especifican lo que el sistema debería hacer más que él como lo hace.

   - Análisis de programas estáticos: es el análisis de software sin ejecutarlo, se analiza mayormente el código fuente buscando probar que cumpla con los objetivos que propone el software

c) Productos comerciales para resolver aproximadamente el problema:
   - Perfect Developer: Es un software de verificación

de programas que utiliza un lenguaje llamado Perfect, cuyas características incluyen una herramienta para demostrar automáticamente un teorema y un traductor de Perfect a Java, Ada y C++. Sin embargo al ser este un proyecto muy avanzado se requiere de muchos conocimientos previos relacionados a la verificación de programas para poder utilizarlo.

- Prototype Verification System: Es un verificador automático de teoremas que no genera código de programa verificado, sino que prueba automáticamente las propiedades de los algoritmos. Este es muy versátil y necesita de un profundo conocimiento en lógica formal para poder ser utilizado.

d) Productos no comerciales (de investigación) para resolver aproximadamente el problema
   - Fredge Program Prover: [Feinerer, 2005] Software de verificación de programas, también conocido como FPP incluye estructuras de programas imperativos típicos (como el while, if y case). Solo permite enteros como variables y el lenguaje es muy restrictivo a la hora de enunciar las precondiciones y postcondiciones.

Las formas y productos que buscan resolver el problema establecen la teoría de la lógica de Hoare de manera implícita en su solución como metodología o lenguaje. Es decir, solo utilizan los conceptos pero no aplican de manera automática estos. Además ninguno incluye la capacidad de derivación automática de programas a partir de asertos. Sino la verificación de estos a través de las reglas.

Es entonces que solo el especialista de estas metodologías o lenguajes puede aplicar los beneficios de las soluciones fuera de los límites de aplicabilidad de las mismas. En cambio la solución propuesta ofrece un mecanismo que, mediante la ejecución automática de las reglas teóricas de la programación metódica, provee una alternativa capaz de aplicar de manera directa estos conceptos sin tener conocimientos profundos sobre la programación metódica. Desde el momento en que uno aprende a construir asertos matemáticos es suficiente como para desarrollar aplicaciones que tengan todos los beneficios que tienen aplicar la metodología.

La importancia de poder aplicar estas reglas en cuestión es tan importante como tener presente la ley de la gravedad para un ingeniero civil; es muy claro hoy en día que la mayoría de desarrolladores tienen la noción que no existen programas sin errores. Siendo una analogía la de afirmar que siempre una casa se caerá, siempre una pista terminara rajada, siempre se caerá el servidor; esta forma de pensar no es más que una consecuencia de trabajo mal hecho; y que, sin embargo, existen formas de garantizar la calidad de lo que se produce y no solo mediante pruebas. La principal razón por la cual la construcción de software es tan costosa es que la mayoría de involucrados desconocen o ignoran la lógica de Hoare o no ven facilidad al aplicarlas. Es entonces que se propone esta herramienta como un esfuerzo para la difusión de las leyes de la lógica de Hoare en forma de compilador de aplicación de reglas de programación metódica de manera automática.

Este proyecto propone implementación de un compilador convencional especificado en [8] de asertos de programación metódica utilizando el método de autómatas adaptativo

V. Descripción de la solución

Este proyecto es la implementación de un compilador convencional especificado en [8] de asertos de programación metódica utilizando el método de autómatas adaptativos en la representación de las propuestas de derivación aplicadas en la derivación de asertos. Cada una de ellas en función a las condiciones que existen para poder utilizarlas. La evaluación se hizo en función a resultados de los mismos par de asertos aplicando las reglas de la programación metódica a mano.

El proyecto se encuentra dividido en tres fases: la investigación de las reglas de la programación metódica a aplicar en el compilador, el diseño de cada autómata que represente a cada propuesta de derivación a aplicar y la implementación del compilador. Los objetivos establecidos son: establecer las reglas que el compilador va a ejecutar de manera automática para garantizar la aplicación correcta de la programación metódica, implementar el analizador léxico, sintáctico y semántico de la notación matemática utilizada en la programación metódica para satisfacer las especificaciones que tenga cada programa generado por el compilador e implementar un traductor de código intermedio que sea capaz de generar programas en el lenguaje de alto nivel para garantizar la correctitud formal de los programas y su aplicación en el desarrollo de software. Actualmente el proyecto se encuentra finalizado.

A. Sustento de la solución

La finalidad de este proyecto es la de obtener un producto que sirva como herramienta de desarrollo de software el cual sea formalmente correcto. Para que pueda ser usado por cualquier usuario con conocimientos básicos de programación y matemática, sin necesidad de tener conocimientos avanzados de programación metódica. Los programas generados son considerados formalmente correctos y se reduce con esto en gran medida la cantidad de errores que este pueda contener.

B. Resultados obtenidos

A continuación los resultados obtenidos:

- Mapeo de reglas de la programación metódica existentes.
- Conjunto de casos que verifiquen la implementación las reglas definidas en el mapeo.
- Análisis, clasificación y descripción de la aplicación de cada regla de la programación metódica.
- Análisis y diseño de los autómatas adaptativos para las propuestas de derivación seleccionadas.
- Implementación del analizador léxico.

- Implementación del analizador sintáctico.
- Implementación del analizador semántico.
- Implementación del traductor de código intermedio.
- Pruebas de un conjunto de casos que verifiquen la implantación correcta de las reglas en el compilador.

VI. Apoyo teórico

La identificación de distintos pares de asertos hace posible que, de acuerdo al contexto en el que se encuentran, la aplicación de distintas propuestas de derivación tengan un resultado distinto.

Para ello, es necesaria la evaluación del contexto en el que se encuentran los pares de asertos para poder determinar qué propuesta de derivación se usará y, a partir de esto, seleccionar la propuesta adecuada. Debido a que este modelo se usa normalmente para la lectura de lenguaje natural se tuvo que adaptar el algoritmo para cumplir con las necesidades planteadas. Para este proyecto a cada submaquina se le fue asignada una propuesta de derivación. El trabajo de cada submaquina es que, en función a los asertos generen propuestas de derivación en forma de estados. Luego a partir de cada uno de estos estados se generan nuevos asertos (debilitando la postcondición) llegando a una solución parcial. Esto genera por cada estado una lista de parejas de asertos (o duplas) los cuales son registrados en una tabla. Cada una de estas duplas son comparadas, si estas son equivalentes se devuelve el conjunto de estados (instrucciones), en otro caso se vuelve a aplicar el análisis de las submaquinas siempre y cuando esta dupla no haya sido analizada antes. En resumen:

1. Se crea una dupla en función de dos asertos.

2. Se valida la equivalencia de los dos asertos.

3. Si son equivalentes se devuelve la lista de estados involucrados en la transformación de los asertos.

4. Si no son equivalentes y no han sido analizados antes se aplica el análisis de las submaquinas.

5. Se registra la dupla en la lista de duplas ya analizadas.

6. Se recibe cada resultado parcial en forma de lista de estados de cada submaquina.

7. Por cada uno de estos estados y en función de la dupla actual se genera una nueva dupla.

8. Por cada nueva dupla formada que no haya sido analizada se aplica el paso 2.

9. El algoritmo termina si no se cuenta con duplas que no hayan sido analizadas con anterioridad con resultado de error de capacidad de derivación.

VII. Construcción del Compilador

La construcción de un compilador conforma dos partes, la primera parte dedicada al análisis del leguaje base y la segunda parte dedicada a la síntesis de este y la posterior transformación al lenguaje final. La adaptación del autómata adaptativo se encuentra en la segunda parte. De la primera parte:

- Análisis Léxico: La función específica del analizador léxico es la de convertir las líneas de texto de entrada en una lista de tokens. Para lograr esto es necesario profundizar en la estructura general del lenguaje fuente. Esta estructura se encuentra formada por 3 elementos:
  1. (definición de variables no ligadas)
  2. {estructura del aserto correspondiente a la precondición}
  3. {estructura de aserto correspondiente a la postcondición}
  4. …

  La primera parte (1) de la estructura se encuentra formada por dos paréntesis y entre ellos la definición de variables no ligadas separadas por comas. Ejemplo:

  (a,b,c)

  En este ejemplo las letras “a”, “b” y “c” representan las variables no ligadas.
  La segunda (2) parte de la estructura esta formada por una dupla de notaciones matemáticas cada una entre dos llaves. Ejemplo:

  { a= 2*b*c+r AND r< 2*b}
  { a= b*c+r AND r< b}

  La estructura de las notaciones matemáticas es la estructura clásica de formación de notaciones matemáticas a excepción de los símbolos relacionales “∧” y “∨”. Estos símbolos han sido reemplazados debido a la dificultad de escribirlos en un editor de textos convencional.
  La tercera (3) parte se encuentra formada por un número finito de duplas de asertos de la forma definida ya en la segunda (2) parte.

  A continuación el analizador se va a encargar de convertir las líneas de texto de entrada en una secuencia de tokens. Los cuales cada uno corresponde a un elemento de la construcción de asertos formales. Entre los tokens que existen se encuentran: VARIABLE, CONSTANTE, SUMA, RESTA, MULT, DIVI, AND, OR, MAYORIGUAL, MENORIGUAL, MENOR, MAYOR, IGUAL, CIERTO, ENTERO, COMA, LLAVEIZQ, LLAVEDER, PARIZQ, PARDER, CUANT_SUMA,CUANT_MULT,CUAN_MAX,CUANT_MIN,CUANT_TODO,CUANT_ALME.

- Analizador Sintáctico: El siguiente paso en el proceso de compilación es determinar si la secuencia de tokens formada es sintácticamente correcta. Para esto es necesaria la definición de una gramática libre de contexto.
  Esta gramática sirve para establecer las especificaciones necesarias para cumplir con las reglas de formación de asertos de la programación metódica. Para la

realización del analizador sintáctico se va a definir la siguiente gramática libre de contexto.

Terminales:

{AND,OR,"{","}","(",")",MAYORIGUAL,MENORIGUAL, MAYOR,MENOR,IGUAL,SUMA,RESTA,MULT,DIVI,CIERTO , PARIZQ, PARDER,CONSTANTE, ENTERO,VARIABLE,CUANT_SUMA,CUANT_MULT,CUAN_ MAX,CUANT_MIN,CUANT_TODO,CUANT_ALME }

No-Terminales:

{PROGRAM, DEFVAR, DUPLAS, ASERTO, LISTAVAR, ECUACION, RELACION, EXPRESION, COMPA, MODIFIC, VALOR}

Reglas:

(1) PROGRAM → DUPLAS
(2) PROGRAM → DEFVAR DUPLAS
(3) DUPLAS → LLAVEIZQ ASERTO LLAVEDER LLAVEIZQ ASERTO LLAVEDER DUPLAS
(4) DUPLAS → LLAVEIZQ ASERTO LLAVEDER LLAVEIZQ ASERTO LLAVEDER
(5) DEFVAR → PARIZQ LISTAVAR PARDER
(6) LISTAVAR → VARIABLE COMA LISTAVAR
(7) LISTAVAR → VARIABLE
(8) ASERTO → ECUACION RELACION ASERTO
(9) ASERTO → ECUACION
(10) RELACION → AND
(11) RELACION → OR
(12) ECUACION → EXPRESION COMPA ECUACION
(13) ECUACION → EXPRESION
(14) ECUACION → PARIZQ ECUACION PARDER
(15) ECUACION → CIERTO
(16) ECUACION → CUANTIFICADOR
(17) CUANTIFICADOR → VALOR IGUAL CUANT_TRIP
(18) CUANTIFICADOR → VALOR IGUAL CUANT_CUAT
(19) CUANT_TRIP → CUANT_SUMA (VALOR,VALOR,VALOR)
(20) CUANT_TRIP → CUANT_MULT (VALOR,VALOR,VALOR)
(21) CUANT_TRIP → CUANT_CONT (VALOR,VALOR,VALOR)
(22) CUANT_TRIP → CUANT_MAX (VALOR,VALOR,VALOR)
(23) CUANT_TRIP → CUANT_MIN (VALOR,VALOR,VALOR)
(24) CUANT_CUAT → CUANT_TODO (VALOR,VALOR,COMPA,VALOR)
(25) CUANT_CUAT → CUANT_ALME (VALOR,VALOR,COMPA,VALOR)
(26) COMPA → MAYORIGUAL
(27) COMPA → MENORIGUAL
(28) COMPA → MAYOR
(29) COMPA → MENOR
(30) COMPA → IGUAL
(31) EXPRESION → VALOR MODIFIC EXPRESION
(32) EXPRESION → VALOR
(33) EXPRESION → PARIZQ EXPRESION PARDER
(34) MODIFIC → SUMA
(35) MODIFIC → RESTA
(36) MODIFIC → MULT
(37) MODIFIC → DIVI
(38) VALOR → CONSTANTE
(39) VALOR → ENTERO
(40) VALOR → VARIABLE

Símbolo inicial PROGRAM

Para la formación de este analizador en Java se utilizó BYACC y se especificó cada terminal y no terminal para cumplir con las características de este proyecto.

- Analizador semántico: El analizador semántico se encarga de formar el árbol de traducción y la tabla de traducción a partir de la gramática libre de contexto y los tokens definidos anteriormente. Son el paso previo a la traducción en si del código fuente.

Árbol de Traducción: Para formar este árbol se crearon las clases necesarias que soporten esta estructura. Por ejemplo:

- Para los valores se diseñó una clase llamada Valor la cual tiene como atributos el tipo de valor y su valor en forma de variable o constante según sea el caso.
- Para las ecuaciones se diseñó una clase llamada Ecuación la cual está formada por dos clases llamadas Expresion (que forman una expresión) y un enumerador de tipo Comparador que representa el comparador de la ecuación.

VIII. Metodología aplicada

El proyecto de investigación se encuentra guiado por la metodología propuesta en **Error! Reference source not found.** para proyectos de investigación. La adaptación de las líneas generales descritas en **Error! Reference source not found.** para este proyecto dieron como resultado las siguientes etapas:

1. Identificación de las reglas de la programación metódica existentes a utilizar en el proyecto.
2. Análisis y clasificación de la participación de cada regla en cada parte del compilador.
3. Desarrollo de un conjunto de casos de prueba que verifiquen la implementación de las reglas definidas anteriormente.
4. Construcción del autómata adaptativo que implementa las propuestas de derivación.
5. Implementación de los analizadores léxico, sintáctico y semántico del compilador.
6. Implementación del traductor de código intermedio el cual incluye el autómata adaptativo.
7. Validación mediante el conjunto de pruebas de la correcta aplicación de las reglas de la programación metódica.
8. Análisis de resultados.
9. Elaboración de conclusiones.

IX. Diseño e implementación del Autómata Adaptativo

Para identificar las propuestas de derivación del autómata adaptativo se tuvo que investigar cada regla y clasificar en función a su aplicación dentro del compilador siendo la clasificación:

• Reglas para el Lenguaje Fuente y Final: Relacionados al análisis del conjunto de palabras en un lenguaje, a las características de cada palabra en particular y a la funcionalidad de cada palabra en el lenguaje. Estas reglas se aplicaran como características del lenguaje y están basadas en las descritas en [9].

• Reglas para el Compilador:

- Reglas de Aplicación Matemática:

- Despeje Ecuacional: Se utilizaran las reglas relacionadas al despeje ecuacional de una variable en función a las otras, así como el remplazar una variable por otra. Ejemplo:

a+4d-2f/c=K +4a (despejar a en función de d,f,c y k)
a+4d-2f/c -a=K +4a –a (igualdad en los lados)
4d-2f/c=K+3a (simplificación)
4d-2f/c-K=3a (igualdad y simplificación)
(4d-2f/c-K)/3=a (cambio de factor)

- Lógica Matemática: Se utilizaran las reglas relacionadas al análisis lógico de una sentencia lógico-matemática. Ejemplo:

p AND q = p
p AND (p AND q) (reemplazo)
p AND p (simplificación)
p

- Aritmética Matemática: Se utilizaran las reglas de la suma, resta, multiplicación de naturales y enteros.

- Reglas de Aplicación de la Metodología:

  - Aplicación de la Metodología: Son reglas fijas de la metodología para su aplicación. Ejemplo:

"El aserto generado por la derivación de una postcondición ha de ser suficientemente fuerte para garantizar el cumplimiento de esta postcondicion."

  - Propuestas de Derivación: Este tipo de reglas son propuestas que intentan, pero no garantizan, llegar a una solución que lleve a una derivación adecuada. Ejemplo:

"Cuando en la postcondicion tenemos una conjunción de igualdad entre solo dos variables podemos asignar una variable a otra."

Es en este conjunto de reglas en los cuales se va a aplicar el autómata adaptativo. Para poder comprender el funcionamiento de cada propuesta es necesario tener en cuenta algunos conceptos relacionados a la programación metódica:

- Conjunción: Parte de un cuantificador que se encuentra relacionado a un valor booleano.
- Postcondición: Cuantificador que se encarga de explicar la relación con los resultados.
- Variable Ligada: Variable vinculada siempre a un cuantificador.
- Variable Libre y Constantes: Variable inicial o constante del programa.
- Cuantificador: Son formados por variables ligadas y un dominio.
- Precondicion: Cuantificador que se encarga de imponer las condiciones a los datos.
- La instrucción de asignación simple: Corresponde a la modificación de una determinada variable, y por tanto conlleva a una modificación del estado. Se denota con ":=".
- La instrucción de asignación múltiple : Corresponde a la modificación de una dos o más variables, y por tanto conlleva a una modificación del estado. Se denota con "<a,b>:=<c,d>" siendo asignado a igual a b y c a d respectivamente.
- La instrucción alternativa: Corresponde a la ejecución de una instrucción dependiendo de si su protección se encuentra validada.
- Función de cota: También llamada función limitadora se encarga de validar el razonamiento por inducción en los casos de programación recursiva.

A continuación se listan y explican las propuestas que se pueden aplicar en la resolución de problemas relacionados a la programación metódica. Estos no garantizan la resolución del problema pero proponen una alternativa de solución. Estas reglas también son las que se aplicarán directamente en el autómata adaptativo para la toma de decisiones de aplicación de reglas:

1. "Cuando en la postcondición tenemos una conjunción de igualdad entre solo dos variables podemos asignar una variable a otra". Si en la postcondición nos encontramos que una ecuación, dentro de un conjunto de ecuaciones, es la igualdad entre dos variables entonces son propuestas de estado la asignación de una variable a otra. Cualquiera de ellas puede ser la asignada siempre y cuando se cumplan con las reglas de aplicación de la metodología.

   Ejemplo:

   {Post: … AND x=z}

   Siendo propuestas validas z:=x y x:=z

2. "Si en la postcondición existe una única variable ligada, siendo las demás libres o constantes, habremos de efectuar una asignación sobre esta variable en función a todas las variables (variables de la precondicion y la postcondición)". Si en la postcondición dentro de cada ecuación y conjunción existe solo una variable ligada, es entonces la asignación a esta variable ligada una propuesta de estado de derivación.

   Ejemplo:
   {Pre: x+y=T}
   {Post: z+y=T}

   Siendo x una variable que no se encuentra en la postcondición, z una variable libre y T una constante entonces:

y:= E(x,y,z) es una propuesta.

3. "Si en la postcondición tenemos una conjunción de igualdad entre una variable ligada y una constante y esta constante se encuentra asignada a un variable en la precondición entonces se puede efectuar una asignación sobre esta variable con la variable de la precondición la cual tiene a la constante asignada". Si en la postcondición existe dentro de una ecuación una igualdad entre una variable ligada y una constante, es una propuesta la asignación a esa variable del valor de la constante, el cual sea igual al valor de otra variable en la precondición.

Ejemplo:

{Pre: x=X}

{Post: y=X}

Debido a que la variable y es equivalente a la constante X en la postcondición y a que existe en la precondición una variable que es equivalente a esta constante, entonces y:=x seria la propuesta generada por esta regla.

4. "Si la regla 1 no se puede aplicar en ninguna de las conjunciones se puede asignar la variable a una nueva variable ". Si por alguna razón la regla 1 o la regla de igualdad entre dos variables no se puede aplicar o no lleva a ningún resultado, se puede aplicar asignar la variable a una nueva variable creada.

Ejemplo:

{Post: x=y}

Asumiendo que no se puede aplicar la regla uno entonces se propone hacer x:= z siendo z una variable nueva.

5. "Si en la postcondición existe disyunción entre dos conjunciones se sugiere entonces por cada una establecer un conjunto de instrucciones alternativas". Si la postcondición está conformada por disyunciones de conjunciones entonces es una propuesta un conjunto de instrucciones alternativas donde para cada una se le proponga cada disyunción por separado.

Ejemplo:

Si la postcondición es:

{Post: (z = x OR z = y) AND…}

Entonces se propone,

[… →… para el aserto {z = x AND …} y

[]…→… para el aserto {z = y AND …}

6. "Si en la postcondición existen conjunciones y ninguna es de igualdad entonces se puede establecer una asignación múltiple de todas las variables ligadas". Si la postcondición está conformada por conjunciones y ninguna ecuación es de igualdad entonces se propone una asignación múltiple a cada variable ligada.

Ejemplo:

Si a y b son variables iniciales:

{Pre: a= 2*b*c + r AND r< 2*b}

{Post: a= b*c + r AND r< b}

Entonces se puede intentar hacer una asignación múltiple con ambos:

{Pre: a= 2*b*c+r AND r< 2*b}

<c,r>:=<E1,E2>

{Post: a= b*c+r AND r< b}

7. "Si al generar un nuevo aserto este contiene conjunciones que no pueden ser garantizadas por la precondición entonces se puede utilizar la condición (la conjunción que no puede ser garantizada) como protección". Es una forma de establecer las protecciones de las instrucciones alternativas, se establecen como condiciones para la aplicación de la instrucción, lo cual es a su vez la nueva postcondición. La disyunción de todas representan el nuevo aserto generado por el conjunto de instrucciones alternativas.

Ejemplo:

{Pre: a= 2*b*c+r AND r< 2*b}

[r<b → c := 2*c

[] ? → …

{Post: a= b*c+r AND r< b}

Dado que el nuevo aserto de c:= 2*c es r<b este actúa también como protección de la instrucción. Siendo a un vector, x un entero y k un numero natural.

Para seguir con la notación formal descrita en [6], la cual define al autómata adaptativo encargado de la construcción de palabras fonológicas, se va a utilizar la siguiente analogía:

- Una frase es un aserto.
- Una palabra es una ecuación, siendo la frase y el aserto un conjunto de estos.
- Una letra es una variable, constante o relación dentro de una ecuación.

X. Implementación

A continuación se va a definir la implementación de cada uno de los autómatas diseñados para este proyecto. En primer lugar, para la aplicación de las propuestas de derivación se utilizaron tres tipos de autómatas. Para todos estos autómatas el estado de entrada es el a0:

- Como se muestra en la Fig. 1 la configuración del autómata tiene 3 salidas.

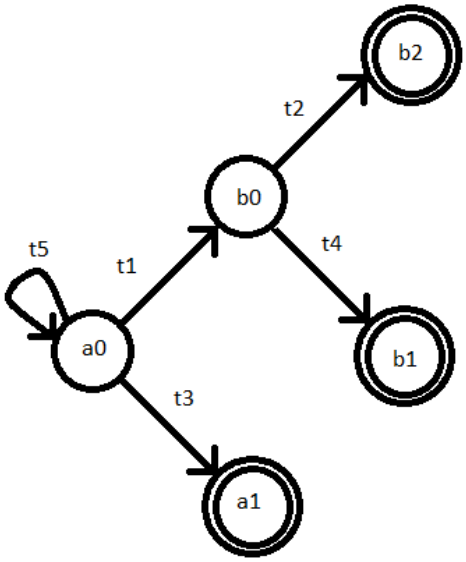


Fig. 1. Configuración del autómata del tipo 1.

Esta se aplica en la regla 1 la cual consta de los siguientes estados. El estado inicial (a0) es el encargado de determinar si existe una conjunción de igualdad entre solo dos variables; si existe se utiliza la transición t1 al estado b0, sino se toma la transición t3 al estado de salida a1. Luego en el estado b0 se verifica si ambas variables son ligadas, de serlas se va al estado final b2 mediante la transición t2. Si solo una es ligada entonces se va al estado b1 por el estado t5. Debido a que este análisis se hace por cada ecuación, la transición t5 representa la transición de análisis entre ecuaciones de un aserto.

- Como se muestra en la Fig. 2 es una configuración de análisis simple

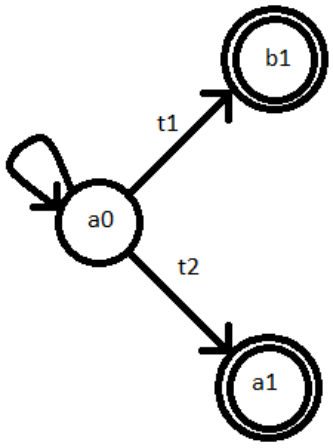


Fig. 2. Configuración del autómata del tipo 2.

Esta configuración se utiliza para aplicar las propuestas de derivación 2,4 y 6 debido a que estas comparten la misma naturaleza. El estado inicial a0 se encarga de validar que se cumpla su condición la cual involucra todo el aserto. De no cumplirlo se va por la transición t2 al estado a1, en otro caso se traslada por la transición t1 al estado b1.

- Como se muestra en la Fig. 3 es una configuración de análisis doble.

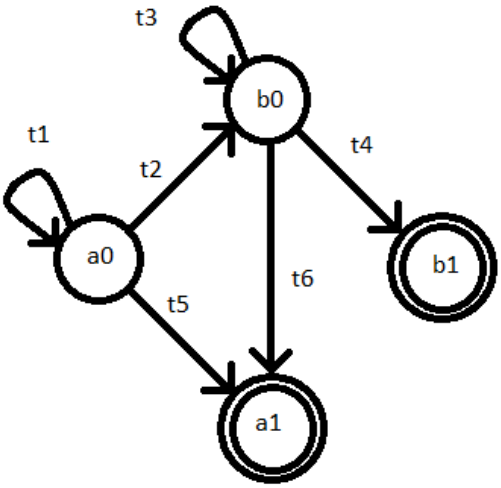
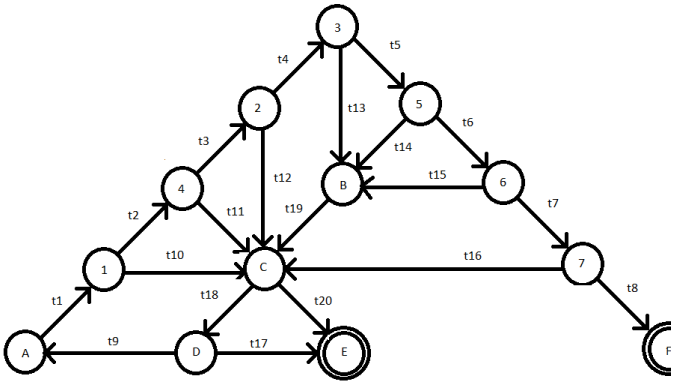


Fig. 3. Configuración del autómata del tipo 3.

Esta configuración se utiliza para aplicar las propuestas de derivación 3,5 y 7 debido a que estas comparten la misma naturaleza. El estado inicial a0 se encarga de validar que se cumpla con la existencia de una característica del conjunto de ecuaciones. Cada una de ellas sin excepción debe cumplirla, la transición t1



representa entonces el análisis de cada uno de las ecuaciones. De no cumplirlo se va por la transición t5 al estado a1, en otro caso se traslada por la transición t2 al estado b0. Este estado valida que se cumpla con una condición en el aserto inicial o precondición. La transición t3 representa e análisis de cada uno de las ecuaciones de la precondición. Si la condición se cumple entonces se procede al estado b1 por la transición t4; sino se va al estado a1 por la transición t6.

Además para el funcionamiento de derivación se utilizó un autómata. El cual se encarga de la aplicación de las reglas de derivación. La configuración de este autómata es como se muestra en la Fig. 4.:

Fig. 3. Configuración del autómata de derivación.

El estado A, representa el estado inicial en el cual nos encontramos cuando tenemos un par de asertos los cuales no son equivalentes. Para los autómatas que representan las propuestas de derivación, los estados finales del tipo a representan a la no aplicabilidad de la propuesta y los estados de tipo b a su aplicabilidad. A continuación el estado A aplica el autómata de la regla 1(representado por el estado 1) por la transición t1. Si el resultado de esta propuesta no es aplicable se procede a evaluar a la siguiente regla siguiendo un orden específico. El orden de evaluación de las reglas permite dar prioridad a determinadas propuestas.

Si no se puede aplicar ninguna regla o el par de asertos ya ha sido validado se procede al estado final F. Este estado representa la posibilidad de un resultado nulo, es decir, que excede la capacidad del compilador. Para las reglas 5,6 y 7 se tiene que hacer ejecutar el estado B para generar el conjunto de adaptaciones adecuadas para poder ir al estado C. Si el resultado es positivo para los autómatas se procede al estado C. En el estado C se convierten estas propuestas en un nuevo par de asertos. Si el nuevo par es equivalente se va al estado E dando por terminado el autómata, siendo el resultado, las instrucciones generadas por el conjunto de propuestas aplicadas para la transformación de asertos.

XI. Conclusiones

Durante el desarrollo del proyecto se encontraron diversas dificultades relacionadas a la aplicación automática de procesos que para el ser humano con entrenamiento adecuado puede aplicar con facilidad pero que son difíciles de aplicar para un computador; en la mayoría de estos casos se optó por diseñar un algoritmo capaz de aplicar estos procesos, sin embargo, la limitación de estos algoritmos reducen el alcance del proyecto.

Durante el análisis y síntesis de las reglas a aplicar por el compilador se tuvieron que descartar propuestas de derivación debido a que su aplicación requería de algoritmos especializados como la generación de programas recursivos, aplicaciones en función a una estructura abstracta de datos específica y algoritmos de transformación por inducción. En otros casos las reglas no podían ser generalizadas y eran aplicadas en función a sugerencias arbitrarias del autor. Sin embargo se incluyeron suficientes reglas como para satisfacer los objetivos del proyecto y el alcance del mismo, debido a que se incluyeron todas las reglas relacionadas a la mecánica de aplicación de propuestas de derivación. Es decir, se incluyeron todas las reglas que forman la base de la aplicación de esta metodología.

Para el análisis léxico, semántico y sintáctico se diseñó un lenguaje capaz de expresar los asertos que el compilador puede resolver. Se incluyeron variables, enteros, constantes, ecuaciones y cuantificadores. Estos últimos son los únicos capaces de representar un vector y su transformación se hace de manera automática. Era necesario incluir los vectores para enriquecer el lenguaje, pero debido a que las propuestas de derivación relacionadas a los vectores implicaban en su gran mayoría la inclusión de algoritmos np se opto por derivar de manera automática cada cuantificador sin utilizar las propuestas no implementadas. De esta manera se logró incluir la representación de vectores en el lenguaje cumpliendo con la expresividad necesaria para satisfacer los objetivos del lenguaje.

En la implementación del traductor de código intermedio se aplicaron y adaptaron los conceptos relacionados a un autómata adaptativo. El algoritmo no solo es capaz de utilizar las propuestas de derivación de programas con éxito, sino que también soporta la inclusión de nuevas reglas ya que la formación de cada submaquina es independiente del funcionamiento del algoritmo. La expresividad del lenguaje, en algunos casos, excede la capacidad de traducción del compilador sin embargo, en estos casos el compilador no resuelve dar ningún resultado a dar uno erróneo. As así que el traductor es capaz de generar instrucciones en un lenguaje de alto nivel, y garantizar la correctitud formal de los programas que pueden ser generados por el compilador.

De esta manera se logró cumplir con el objetivo del proyecto estableciendo además la base para la derivación automática de programas y la construcción de un compilador de programación automática. Los programas obtenidos por este compilador son regidos por las reglas de programación metódica y al cumplir estas son formalmente correctos lográndose cumplir con el objetivo. Este proyecto es la base de muchos proyectos que se pueden generar en pro de la programación automática.

XII. TRABAJOS FUTUROS

Existen varios trabajos futuros que se pueden generar de este proyecto:

- Debido a que el resultado del traductor intermedio es una representación de instrucciones universal es posible traducir el lenguaje a cualquier lenguaje de alto nivel. Para la traducción solo sería necesaria la adaptación adecuada de las estructuras

esenciales de todo lenguaje como son las variables, constantes, asignación de vectores, las condicionales y los bucles.

- El aumentar en el lenguaje la capacidad de incluir especificaciones de algebraicas de tipos abstractos de datos se podría incluir más reglas relacionadas a estas estructuras.

- Se puede implementar un algoritmo de deducción por inducción e incluir más reglas de programación metódica.

- Si se implementa un mecanismo de aprendizaje se podría entrenar al compilador para resolver los problemas cuya solución tienen un origen arbitrario o son tomados a partir de juicio experto.

- Durante la presentación del conjunto de estados que forman parte de la solución de cada derivación se tiene como información la regla aplicada por cada estado. Es decir que si se deseara aplicar este proyecto como herramienta de aprendizaje de la metodología, es posible adaptar el compilador para que este explique paso a paso la derivación de cada par de asertos.

REFERENCIAS

[1] C. Hoare "An Axiomatic Basis For Computer Programming, New York, Communications of the ACM", *Communications of the ACM*, vol. 12, n° 19, pp. 576-580, 1969.

[2] J.L. Balcázar "Programación Metódica", Madrid, McGraw-Hill/Interamericana de España S. A, 1993.

[3] I. Feinerer "Formal Aspects of Computing, Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations" , Viena 2005, vol 21 pp 293-301, 2005.

[4] Project Management Institute "Guia de los Fundamentos para la Dirección de Proyectos (Guia de PMBOK)", 4ta Edición, Pennsylvania.

[5] J.J. Neto, "Adaptive Automata for Context-Sensitive Languages", *ACM Sigplan Notices*, vol. 29, n° 9, pp. 115-124, 1994.

[6] H. Pistori, "Tecnologia adaptativa en Engenharia de computação: estado da arte e aplicações", Dissertação de doutorado, orientada por J. J. Neto, Departamento de Engenharia de Comparação e Sistemas Digitais, Escola Politecnica da Universidad de Sao Paulo, 2003.

[7] Esdger Dijkstra, "Applying Design By Contract", USA, IEEE-Computer, USA, vol 25, n° 10, pp40-51, 1992.

[8] Ravi Sethi, "Programming Languages Concepts & Constructs" , New Jersey, Pearson, 2001.

[9] Anne Kaldewaij, Programming: The Derivation of Algorithms, Inglaterra, Prentice Hall International, 1990.

[10] David Rosemblum, "IEEE Transactions on Software Engineering, "A Practical Aproach to Programming With Assertions", New Jersey, 1995, vol 21 -1 ,pp19-31, 1990.