Novática, revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de ATI (Asociación de Técnicos de Informática). _Novática_ edita también _Upgrade_, revista digital de CEPIS (_Council of European Professional Informatics Societies_), en lengua inglesa.

<http://www.ati.es/novatica/>
<http://www.upgrade-cepis.org/>

ATI es miembro de CEPIS (*Council of European Professional Informatics Societies*) y tiene un acuerdo de colaboración con ACM (*Association for Computing Machinery*). Tiene asimismo acuerdos de vinculación o colaboración con AdaSpain, AI2 y ASTIC

Coordinación Editorial y Redacción Central (ATI Madrid)
Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <*novatica@ati.es*>
Composición, Edición y Redacción ATI Valencia
Reino de Valencia 23, 46005 Valencia
Tlf./fax 963330392 <*secreval@ati.es*>
Administración y Redacción ATI Cataluña
Via Laietana 41, 1º, 1ª, 08003 Barcelona
Tlf.934125235; fax 934127713 <*secregen@ati.es*>
Redacción ATI Andalucía
Isaac Newton, s/n, Ed. Sadiel, Isla Cartuja 41092 Sevilla
Tlf./fax 954460779 <*secreand@ati.es*>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza
Tlf./fax 976235181 <*secreara@ati.es*>
Redacción ATI Asturias-Cantabria <*gp-astucant@ati.es*>
Redacción ATI Castilla-La Mancha <*gp-clmancha@ati.es*>
Redacción ATI Galicia
Recinto Ferial s/n, 36540 Silleda (Pontevedra)
Tlf.986581413; fax 986580162 <*secregal@ati.es*>
Suscripción y Ventas: <*http://www.ati.es/novatica/interes.html*>, o en ATI Cataluña y ATI Madrid
Publicidad: Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <*novatica.publicidad@ati.es*>

Imprenta: 9·Impressió S.A., Juan de Austria 66, 08005 Barcelona.
Depósito Legal: B 15.154-1975
ISSN: 0211-2124; CODEN NOVAEC

Portada: Antonio Crespo Foix / © ATI 2003

novática
UPGRADE
Revista de la Asociación de Técnicos de Informática

MAYO · JUNIO 2003
163

SUMARIO

Monografía del próximo número:
«Ingeniería del Software: estado de un arte»

Lenguajes informáticos

COMPAS: un compilador para un lenguaje imperativo con aserciones embebidas

Joaquín Ezpeleta Mateo, Pedro Gascón Campos, Natividad Porta Royo
Depto. de Informática e Ingeniería de Sistemas, Centro Politécnico Superior, Universidad de Zaragoza

<ezpeleta@posta.unizar.es>
<pedrogascon74@yahoo.es>
<nporta@aragob.es>

Este trabajo ha sido financiado por el proyecto TIC2000-1568-C03-01

Resumen: *en este artículo se presenta un compilador para un lenguaje de programación imperativo con aserciones embebidas. El código generado por el compilador comprueba, en tiempo de ejecución, si las aserciones son satisfechas por los datos correspondientes a dicha ejecución. Como lenguaje de especificación se usa la Lógica de Primer Orden. El lenguaje de especificación dispone, además de aserciones genéricas, de elementos especializados: aserciones Pre/Post para la especificación de procedimientos y funciones y aserciones de invarianza y expresiones de cota para bucles. Dado que uno de los objetivos buscados era que la herramienta fuera utilizable en diferentes plataformas, el compilador ha sido desarrollado en Java, y el código generado son bytecodes de Java.*

Palabras clave: *aserciones, compiladores, enseñanza de la programación, especificación y verificación formal de programas.*

1. Introducción

Todos los curricula de estudios universitarios de informática contienen asignaturas que, bajo la denominación de Programación Metódica o Metodología de la Programación, por ejemplo, inciden en los «aspectos formales» de la programación secuencial. Debemos entender aquí aspectos formales como los relativos a las especificación, verificación y derivación de algoritmos usando técnicas formales ([1] [4 [7] son excelentes referencias introductorias). La orientación elegida se basa en una especificación, formal, Pre/Post del programa a desarrollar: Pre (abreviación de pre-condición) representa las propiedades que deben cumplir los datos de entrada al programa, mientras que Post (abreviación de post-condición) representa las propiedades que deben cumplir los resultados del programa (es de esperar que éstos contengan la solución al problema planteado). Este enfoque clásico también suele denominarse «diseño por contrato» [6], entendiendo que la Pre es lo que el usuario se compromete a suministrar, mientras que la Post es lo que el programa entregará a cambio.

El adjetivo formal hace referencia a que las propiedades que queremos establecer relativas a los datos estarán expresadas mediante un lenguaje formal que, en este caso, es la Lógica de Primer Orden. El enfoque se basa también en el concepto de estado del programa. Un estado está determinado por la tupla de valores de sus variables. Así, una acción `A` se especificará mediante una terna `{ Pre} A{ Post}`. Tanto en la precondición como en la postcondición las variables libres (en el sentido de predicado de la Lógica) corresponden a variables y/o parámetros del programa `A`. La interpretación de la terna es la siguiente: la terna es totalmente correcta si, haciendo la invocación a la acción `A` desde un estado que verifica[1] `Pre`, se puede asegurar que su ejecución termina y el estado en que termina verifica la aserción `Post`. De acuerdo con esta perspectiva, un programa es un transformador de predicados. El razonamiento sobre la corrección se apoya en la definición de una semántica formal del lenguaje de programación: cada instrucción del lenguaje se define en términos de la manera en que transforma predicados. Como paso metodológico previo a su verificación, el programa suele estar anotado: se añaden aserciones (normalmente en forma de comentarios especiales) en aquellos puntos del código que el programador considera oportuno, de manera que éstas serían suficientes para construir la demostración de su corrección. Un valor añadido de un programa correctamente anotado es que las aserciones sirven también como una documentación precisa, compacta y no ambigua.

Desde el punto de vista de su corrección, se pueden adoptar tres estrategias fundamentales de tratamiento de un programa anotado.

1.1. Verificación estática completa

Habitualmente se lleva a cabo «a mano» o asistida por herramientas (probadores de teoremas, por ejemplo). Es la propuesta en [1] [4] [7]. Su principal ventaja: es el método que permite tener la certeza de la corrección del software, pues está basado en su verificación matemática. Su principal inconveniente es que requiere conocimientos profundos de Lógica y manipulación simbólica, además de ser costosa en tiempo.

1.2. Verificación estática parcial

Dentro de un enfoque estático, pero sin llegar a probar la corrección del software, se han desarrollado entornos capaces de detectar algunas propiedades relativas a la corrección. En [2] se describe un entorno, denominado SPARK, muy interesante para el trabajo con un subconjunto de Ada, mientras que [9] lo hace para Modula-3 (actualmente los autores de [9] están desarrollando un versión para Java, denominado ESC/Java). Básicamente, el entorno de desarrollo contiene, además del compilador del lenguaje, herramientas que son capaces de procesar las aserciones insertas en el código, informando al programador de problemas encontra-

dos o causas de posibles problemas. Dado que se trata de un análisis estático, la información dada gira en torno a dependencias entre los datos, a uso de variables no inicializadas, a declaración de variables que no se utilizan, a guardas de bucle cuyo valor es constante, etc. Se trataría de herramientas que suponen una evolución y enriquecimiento de otras bien conocidas, como lint de Unix, por ejemplo. Respecto a la opción anterior, presentan la ventaja de que las comprobaciones se llevan a cabo de manera automática. Su principal inconveniente es que sólo analizan algunos aspectos de la corrección. Los entornos no suelen adoptar esta opción como única, sino que, como ocurre en [2] y [9], suelen ir acompañados de probadores de teoremas para adoptar complementariamente el enfoque anterior siendo, además, capaces de generar código para algunas de las aserciones consideradas, pudiendo, por tanto, ser usadas también para un enfoque dinámico (que se describe a continuación).

1.3. Comprobación dinámica

Las aserciones insertas en el código pueden, con un compilador adecuado, ser transformadas en código ejecutable, de manera que en tiempo de ejecución se convierten en un mecanismo para asegurar la corrección de la ejecución. Lenguajes como C o C++ incluyen instrucciones del tipo `assert` con esta finalidad. Eiffel va un poco más allá, incluyendo aserciones del tipo requiere/promise para métodos (equivalentes a nuestras Pre/Post) `e invariant` para invariantes para una clase de objetos. Una característica fundamental de este enfoque reside en su simplicidad, ya que, con un lenguaje de aserciones adecuado, permite que cualquier programador especializado pueda adoptarla. Tiene dos inconvenientes fundamentales. El primero es claro: las aserciones se chequean en tiempo de ejecución con los datos correspondientes a dicha ejecución; por lo tanto, no se prueba que el programa en sí sea correcto. «Sólo» se está estableciendo un mecanismo para asegurar que la ejecución no viola ninguna de las propiedades descritas en las aserciones. El segundo inconveniente viene determinado porque cuanto más potente es el lenguaje usado para especificar (como se ha comentado, aquí proponemos la Lógica de Primer Orden, que es mucho más potente que las simples expresiones booleanas permitidas en instrucciones `assert` o `requiere/promise`) más costoso será ejecutar el código que las aserciones generen. Esto se puede aliviar estableciendo niveles de severidad en las aserciones, de manera que algunas aserciones que han sido utilizadas en las etapas de desarrollo del software no generan código para la versión de explotación.

En este artículo presentamos un compilador que hemos desarrollado, y que permite trabajar de acuerdo con el enfoque dinámico, siendo la Lógica de Primer Orden el lenguaje usado para las aserciones. Este enfoque ya fue usado en [8] con C como lenguaje soporte. A diferencia de la propuesta de [8] o de la que siguen las aserciones en Eiffel [6], en nuestro caso hemos impuesto que las aserciones sean entes «observadores» del programa, de manera que la verificación de una aserción nunca pueda llevar a cabo un efecto lateral que modifique el propio estado en el que la aserción se está comprobando (excepto en el caso de que la aserción no sea verificada, en que se aborta la ejecución del programa). Hemos tratado también, dentro de lo posible, de independizar el lenguaje de especificación del lenguaje de programación, buscando que el primero fuera lo más parecido posible al lenguaje de la Lógica que manejan los libros de texto utilizados en nuestras asignaturas.

En la siguiente sección hacemos una breve descripción tanto del lenguaje de programación como del de especificación. Posteriormente mostramos la arquitectura del compilador desarrollado y hacemos algunos comentarios respecto a su desarrollo. La sección 4 muestra un ejemplo de uso del entorno desde el punto de vista del usuario.

2. Breve descripción del lenguaje de programación enriquecido

Nuestro compilador corresponde a un lenguaje imperativo tipo Pascal. Se trata del lenguaje algorítmico que utilizamos en los cursos de introducción a la programación, y está muy próximo al utilizado en [1]. El fuente contiene dos tipos de elementos. Por un lado, instrucciones en el sentido de transformaciones de estados (como en cualquier lenguaje imperativo); por otro, aserciones referentes al estado del programa. Posteriormente ejemplificaremos las características del lenguaje. De momento, vamos a hacer una breve enumeración de los principales elementos.

2.1. Elementos del lenguaje imperativo

Dada su similitud con Pascal (por ello, precisamente, hemos denominado Pascual al lenguaje), no vamos a entrar en detalles de cuáles son estos elementos. La siguiente enumeración, junto con el ejemplo en el anexo, debe ser suficiente para obtener una idea de cómo es el lenguaje. La diferencia fundamental con Pascal está en las clases de parámetros. Para ellas se ha elegido las que implanta Ada, por lo que los parámetros pueden ser de entrada, de salida, o de entrada/salida, en el caso de los procedimientos, mientras que en el caso de las funciones sólo pueden ser de entrada.

- Tipos de datos: maneja tipos escalares primitivos (entero, real, booleano y carácter) y cadenas de hasta 255 caracteres.
- Dispone de los constructores vector, registro y fichero secuencial (de cualquier tipo previamente definido, excepto fichero).
- Estructuras de control: selección tipo «*If...Else...*», *selección múltiple* (varias alternativas, con semántica de una y sólo una activa), e iteración tipo «*While …*»
- La recursividad está también permitida.

2.2. Elementos del lenguaje de aserciones

Como se ha dicho, el lenguaje de aserciones, que hemos denominado LANOESPE (Lenguaje para ANOtación y ESPEcificación) implementa un lenguaje para la Lógica de Primer Orden, enriquecido a su vez con operadores de suma, producto y conteo sobre dominios finitos. Su características principales son las siguientes:

- Las fórmulas atómicas pueden ser cualquier expresión booleana aceptada por el lenguaje. Éstas hacen referencia a los valores de las variables usadas por el programa, pero no pueden invocar funciones del mismo.

· Maneja la implicación y doble implicación entre fórmulas.
· Maneja los cuantificadores «para todo» y «existe». Considerando un vector v de dimensión n, de enteros, por ejemplo, el siguiente predicado indica que dicho vector está ordenado en sentido estrictamente creciente (PT representa «Para Todo»):

```
PT alfa EN [ 1..n-1] . v[ alfa] < v[ alfa+1]
```

· Dispone de operadores para hablar acerca del estado de los ficheros secuenciales. Si bien desde el punto de vista del lenguaje Pascal se manejan únicamente ficheros secuenciales, desde el punto de vista de su especificación, Lanoespe dispone de operadores para hablar sobre su modo de apertura (lectura o escritura), el número de datos en el mismo, la posición de su *buffer* o alguna propiedad sobre un dato concreto. Así, por ejemplo, si f es una variable de tipo fichero de enteros, el siguiente aserto establece que el fichero está abierto para lectura, que el siguiente dato que se leería es el que ocupa la posición 27 y que todos sus elementos son positivos:

```
(f_MODO = LECT) AND (f_POS = 27) AND
    (PT alfa EN [ 1..f_NDATOS] . f_DATOS[ alfa] > 0)
```

· Tiene denifidos, para dominios discretos y finitos, los operadores de conteo (número de elementos de un conjunto), y la suma y el producto de expresiones indexadas por dichos dominios. Por ejemplo, las siguientes aserciones representan, respectivamente, que dos elementos del conjunto { 1.5,3.75,6.0} son mayores que el valor de la variable a, y que la suma de todos ellos es positiva:

```
CARD alfa EN (1.5, 3.75, 6.0) (alfa > a) = 2
SUM alfa EN (1.5, 3.75, 6.0) (alfa)) > 0.0
```

· Admite la definición de predicados funcionales parametrizados, como forma de notación compacta (una especie de macros enriquecidas). Lo que sigue define el predicado parametrizado que indica que las componentes { aa[ii] ,aa[ii+1] ,…,aa[dd] } están ordenadas:

```
Def: ordenado (aa, ii, dd) ==
      PT alfa en[ ii..dd-1] . aa[ alfa] <= aa[ alfa
+ 1]
```

Cuando se desee afirmar que el vector v de dimensión n está ordenado, se indicará mediante `ordenado(v,1,n)` en la aserción. Cuando, por el contrario, se necesite afirmar que el vector `w` tiene sus componentes 3 a 29 ordenadas, será suficiente con usar `ordenado(w,3,29)`. Como se puede apreciar en el ejemplo en el anexo esto da una gran flexibilidad al uso de aserciones.
· Distingue aserciones especiales de tipo `Pre` (precondición) y `Post` (postcondición). Éstas se usan para especificar procedimientos y funciones y son chequeadas, respectivamente, en el momento de la invocación y el abandono del procedimiento o función ({ −4−} y { −5−} en el anexo, por ejemplo).
· Dispone también de aserciones especiales `Inv` (invariante para un bucle) y expresiones `Var` (variante para un bucle). Una aserción de tipo invariante, ligada a un bucle, es chequeada antes de cada iteración y después de la última. Una expresión `Var` (que debe ser entera) representa una expresión de cota para el bucle. Se chequea que cuando la guarda del bucle está abierta la expresión sea no negativa. También se chequea que sus valores correspondientes a dos iteraciones consecutivas decrezcan estrictamente ({ −8−} en el anexo, por ejemplo).
· Por último, admite también aserciones en cualquier parte del código donde una instrucción tenga sentido (téngase presente que las aserciones generarán código para su verificación).

3. Estructura del compilador

La **figura 1** muestra un esquema de la arquitectura de la aplicación. Básicamente, además de mostrar su viabilidad, el objeto de la herramienta es que sea útil para el aprendizaje de la programación (a pequeña escala) que se lleva a cabo en los cursos introductorios de esta materia. Por ello, uno de los requisitos que establecimos inicialmente fue que el compilador fuera multi-plataforma desde el punto de vista de su ejecución y que, por otra parte, el código generado fuera a su vez ejecutable en múltiples plataformas. Para cumplir el primer requisito optamos por desarrollar la herramienta en Java. Para cubrir el segundo, optamos porque el código generado por el compilador fueran bytecodes de Java, ejecutable en cualquier máquina virtual Java (JVM).

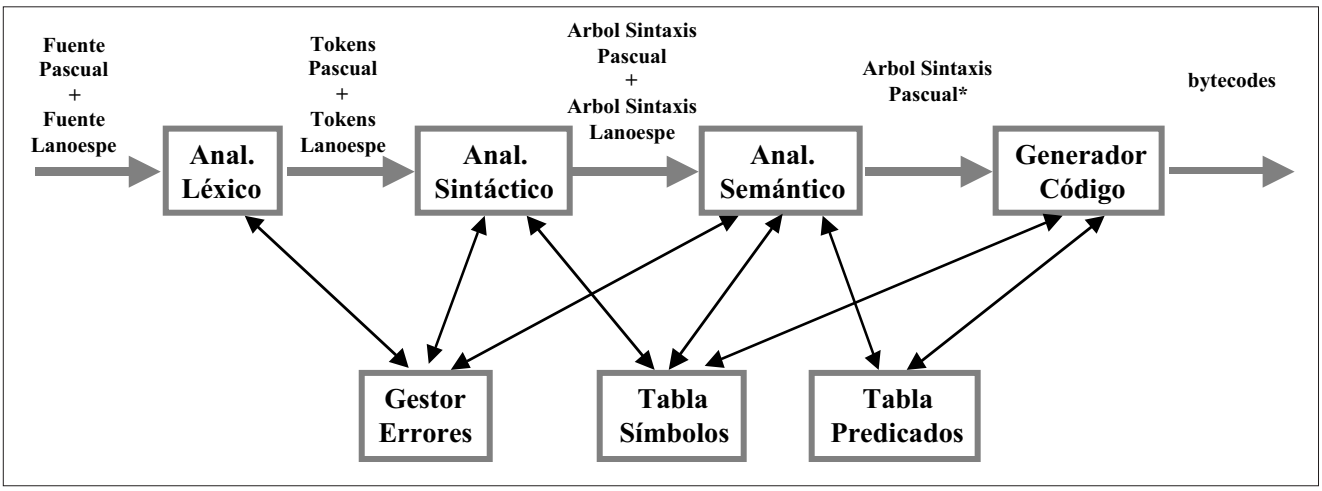

Figura 1. Esquema de la arquitectura del compilador

El compilador funciona como sigue. El fuente de entrada contiene Pascual y, opcionalmente, también aserciones escritas en Lanoespe (una opción en la invocación al compilador determina si las aserciones han de ser consideradas como tales o como simples comentarios). Al terminar correctamente las fases de análisis léxico y sintáctico, se han generado dos árboles de sintaxis, uno correspondiente al fuente Pascual y otro al fuente Lanoespe. Si la fase de análisis semántico se lleva a cabo satisfactoriamente, ésta traduce el fuente Lanoespe a Pascual, integrando el código correspondiente a las aserciones más el código Pascual en un único árbol de sintaxis decorado. Este árbol integrado, que es correcto, es pasado al generador de código, que genera un programa en bytecodes para la JVM. El código generado, dependiendo de las estructuras de datos manejadas en el programa fuente, estará formado por uno o varios ficheros «`.class`» para la JVM.

Para la implementación del analizador léxico se ha utilizado Jlex [5], un generador de analizadores léxicos en Java (análogo a Lex de Unix o Flex de GNU). En cuanto al analizador sintáctico, se ha usado CUP [3], un generador de analizadores sintácticos en Java para gramáticas de la clase LALR (análogo a Yacc de Unix o Bison de GNU). El resto de compilador se ha implementado directamente en Java.

4. Ejemplo de uso del entorno

El **anexo 1** muestra un ejemplo de programa anotado. Una sesión típica de uso del compilador con dicho programa sería la que sigue (aquí usamos «$» para identificar las instrucciones del usuario. Las líneas que no van precedidas por dicho símbolo correspoden a información dada por el entorno):

```
$java compas mezclar.pas -a
Se ha generado el fichero: mezclar.class.
$java mezclar
V ANTES de ser ordenado
v[ 1] =10  v[ 2] =9  v[ 3] =8  v[ 4] =7  v[ 5] =6  v[ 6] =5
v[ 7] =4  v[ 8] =3  v[ 9] =2  v[ 10] =1
V DESPUES de ser ordenado
v[ 1] =1 v[ 2] =2 v[ 3] =3 v[ 4] =4 v[ 5] =5 v[ 6] =6 v[ 7] =7
v[ 8] =8  v[ 9] =9  v[ 10] =10
```

Supongamos ahora el caso de un error sintáctico: hemos olvidado el punto y coma de la última instrucción de la acción `escribeVector`:

```
$java compas mezclar.pas -a
_____________________________________________
ERROR SINTACTICO: 130, 1 -> Fin
Se espera el simbolo ; para finalizar instruccion.
_____________________________________________
```

Veamos por último el caso de una ejecución que viola una aserción. Supongamos que en el punto `{ -8-}` del fuente el invariante estuviera escrito como

`{ # INV: PT x en [ I..otroI] .vAux[ x] =v[ x]  #}`

que presupone que las componentes desde `I` hasta `otroI` ya han sido copiadas, cuando en realidad sólo han sido copiadas hasta la `otroI-1`. La ejecución es la siguiente:

```
$java mezclar
V ANTES de ser ordenado
v[ 1] =10  v[ 2] =9  v[ 3] =8  v[ 4] =7  v[ 5] =6  v[ 6] =5
v[ 7] =4

No se cumple la asercion. Linea: 73. Columna:
13.
```

5. Conclusiones

En este artículo hemos presentado un prototipo de compilador desarrollado para facilitar el aprendizaje del uso de técnicas de especificación formal de programas. A diferencia de otras herramientas existentes, el lenguaje utilizado para la especificación es el de la Lógica de Primer Orden, que es el mismo utilizado en los textos usados para nuestras asignaturas de programación metódica. Esto extiende la potencia del lenguaje de especificación permitiendo, además de lo asumible en instrucciones del tipo assert de otros lenguajes, el manejo de cuantificadores. Desde el punto de vista del lenguaje imperativo, éste es suficientemente amplio como para cubrir los aspectos tratados en las asignaturas básicas de programación. Actualmente estamos evaluando dos posibles líneas alternativas (aunque no excluyentes) de continuación del trabajo. Por un lado, mejorar el prototipo, extendiendo el entorno, tanto desde el punto de vista del lenguaje (introducir compilación separada, por ejemplo) como desde el punto vista de la edición (desarrollar un editor específico que permita usar la simbología matemática en el fuente). Una segunda línea se encamina hacia la inclusión del lenguaje de especificación manejado, Lanoespe, en un compilador existente de (un subconjunto de) algún lenguaje estándar, como puede ser Pascal o Ada.

Referencias

[1] J.L. Balcázar, *Programación metódica*, McGraw-Hill, 1993.
[2] J. Barnes, *High Integity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
[3] CUP. LALR Parser Generator for Javaä. <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>.
[4] D. Gries, *The science of programming*, Texts and Monographs in Computer Science, Springer-Verlag, 1981.
[5] J.Lex, *A lexical analyzer generator for Javaä.* <http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>.
[6] B. Meyer, Applying «Desing by Contract», *IEEE Computer*, Vol. 25, Octubre 1992, pp. 40-51.
[7] R. Peña, *Diseño de programas. Formalismo y abstracción*, Prentice-Hall, 1998.
[8] D.S. Rosenblum, «A Practical Approach to Programming With Assertions», *IEEE Transactions on Software Engineering*, Vol. 21, nº 1, Enero 1995, pp. 19-31.
[9] K. Rustan, M. Leino, G. Nelson, «An Extended Static Checker for Modula-3», en Kai Koskimies, Editor. *Compiler Construction: 7th International Conference, CC'98, volume 1383 of Lecture Notes in Computer Science*. Springer, April 1998, pp. 302-305.

Nota

[1]El compilador Compas, que implementa el entorno descrito, se puede descargar de la siguiente dirección URL: <http://www.cps.unizar.es/~ezpeleta/MP/MP.htm>.

Anexo 1. Un ejemplo de fuente con aserciones

El siguiente programa en Pascual implementa la ordenación por mezcla

de un vector de enteros. El objetivo es mostrar muchos de los aspectos de Pascual y de Lanoespe.

```
{ ————————————————————————————————
 - Fichero: mezclar.pas
 -                          . . . .
 ————————————————————————————————}
Programa  mezclar;
Constantes
          n = 10;
Tipos
          vect = vector[ 1..n] de entero;
Variables
          i: entero;
          v: vect;
{ ——— Definicion de predicados ———————————}
  { # Def: ordenado (aa, ii, dd) ==
     PT x en [ ii..dd-1] . aa[ x] <= aa[ x+1] #} { —1—}

  { # Def: mezcla (v1, i1, d1, v2, i2, d2, v3, i3, d3) ==
                (d3 - i3 + 1 = d2 - i2 + 1 + d1 - i1 + 1)
            AND
               (PT ind en [ i3..d3] .
                CARD j3 en [ i3..d3] (v3[ j3] = v3[ ind] ) =
                 CARD j1 en [ i1..d1] (v1[ j1] = v3[ ind] ) +
                  CARD j2 en [ i2..d2] (v2[ j2] = v3[ ind] ))
            #}                                          { —2—}

  { # Def: permutacion (a, b, izq, dch) ==
           PT ind en [ izq..dch] .
                (EX j en [ izq..dch] . a[ ind] = b[ j] )
              AND
               (CARD k en [ izq..dch] (a[ k] = a[ ind] ) =
                CARD l en [ izq..dch] (b[ l] = a[ ind] ))
    #}                                                  { —3—}
{ ————————————————————————————————}
Accion merge (ES v: vect; E I, M, D: entero);
{ # Pre: (1 <= I) AND (I <= M) AND (M <= D) AND (D <= n) AND
  ordenado (v!, I, M) AND ordenado (v!, M + 1, D) #} { —4—}
{ # Post: mezcla (v!, I, M, v!, M + 1, D, v, I, D)
                          AND ordenado (v, I, D) #} { —5—}
{ Com: Estando ordenados v[ I] ,v[ I+1] ,...,v[ M]
     y v[ M+1] ,v[ M+2] ,...,v[ D] ,
     los mezcla, dejando ordenado v[ I] ,v[ I+1] ,...,v[ D] }
Variables
     vAux: vect;
     iI, iD, iNuevo, otroI: entero;
     { # DefConst: v! #}{ v!: valor de v en entrada} { —6—}
Principio
    iI := I;
    iD := M + 1;
    iNuevo := I;
       { # VAR: D+1-iNuevo #}                          { —7—}
      { # INV: ordenado(vAux,I,iNuevo-1) AND
       mezcla(v!,I,iI-1,v!,M+1,iD-1,vAux,I,iNuevo-1) #}
   Mq (iI <= M) AND (iD <= D)
       Si v[ iI] < v[ iD] Ent
            vAux[ iNuevo] := v[ iI] ;
             iI := iI + 1;
        Si_No
            vAux[ iNuevo] := v[ iD] ;
            iD := iD + 1;
        FSi
            iNuevo := iNuevo + 1;
    FMq
    otroI := iI - 1;
   Mq otroI < M { copia lo que quede de v[ I] ,...,v[ M] }
        otroI := otroI + 1;
       vAux[ iNuevo] := v[ otroI] ;
        iNuevo := iNuevo + 1;
    FMq
    otroI := iD - 1;
   Mq otroI < D { copia lo que quede de v[ M+1] ,...,v[ D] }
        otroI := otroI + 1;
      vAux[ iNuevo] := v[ otroI] ;
       iNuevo := iNuevo + 1;
   FMq
   otroI := I;          { Copiar sol. de vAux a v}
        { # VAR: D-otroI #}
     { # INV: PT x en [ I..otroI-1] .vAux[ x] =v[ x] #}
                                                 { —8—}
   Mq otroI <= D
      v[ otroI] := vAux[ otroI] ;
       otroI := otroI + 1;
   FMq
Fin
{ ————————————————————————————————}
```

```
Accion mergeSort (ES v: vect; E pI, pF: entero);
{ # Pre: (1 <= pI) AND (pI <= pF) AND (pF <= n) #}
{ # Post: permutacion (v, v!, pI, pF) AND (ordenado(v,pI,pF)) #}
{ Com: ordena por mezcla en v las componentes de pI a pF}
Variables
    medio: entero; { # DefConst: v! #}
Principio
     Si pI < pF Ent
           medio := (pF + pI) div 2;
            mergeSort (v, pI, medio);
            mergeSort (v, medio + 1, pF);
            merge (v, pI, medio, pF);
     FSi
Fin
{ ————————————————————————————————}
Accion ordenMezcla (ES v: vect);
{ # Pre: verdad #}
{ # Post: permutacion (v, v!, 1, n) AND (ordenado(v,1,n)) #}
{ Com: ordena el vector v por mezcla}
                          { # DefConst: v!#}
Principio
      mergeSort (v, 1, n);
Fin
{ ————————————————————————————————}
Accion escribeVector(E v: vect);
{ # Pre: verdad #}
{ # Post: verdad #}
{ Com: escribe v por la salida estándar}
Variables i: entero;
Principio
     i := 1;
     Mq i <= n
    escribir('v[ ',i,'] =' ,v[ i] );
     i := i+1;
     FMq
    escribirLinea;
Fin
{ ————————————————————————————————}
Principio
    i := 1;
    Mq i <= n              { inicializa el vector}
            v[ i] := n - i + 1;
            i := i + 1;
    Fmq
   escribirLinea('V ANTES de ser ordenado' );
   escribeVector(v);
   escribirLinea;
   ordenMezcla (v);
   escribirLinea('V DESPUES de ser ordenado' );
   escribeVector(v);
Fin
```