Verificación de Programas no Determinísticos

» Ricardo Rosenfeld

Centro de Altos Estudios en Tecnología Informática, Universidad Abierta Interamericana

Resumen

Continuando con nuestra serie de artículos intoductorios sobre la verificación axiomática de programas, en este segundo trabajo nos enfocamos en el paradigma secuencial no determinístico, siempre en el marco de los programas imperativos de entrada/salida. Como el no determinismo se manifiesta en la concurrencia, el artículo sirve también como introducción a la verificación de programas concurrentes, en los que más se justifica por su complejidad un tratamiento formal de las pruebas de correctitud. Trabajamos con un clásico lenguaje de programación, con selección condicional y repetición no determinísticas, al que luego se incorporan asignaciones aleatorias. Para las pruebas de los programas planteamos una adaptación del método axiomático de verificación descripto en la publicación previa, limitado a la programación secuencial determinística. Presentamos ejemplos de aplicación del método e incluimos un desarrollo sistemático de programa, volviendo a destacar el *approach* de utilizar los axiomas y reglas para programar al mismo tiempo que verificar, con el objeto de obtener programas correctos por construcción. Finalmente introducimos el concepto de *fairness*, cuyo efecto es reducir el grado de no determinismo de un programa en base a determinados criterios de equidad en el entorno de ejecución, y describimos un par de adaptaciones en las reglas de prueba para contemplar este aspecto.

PALABRAS CLAVE: PROGRAMA, NO DETERMINISMO, VERIFICACIÓN, AXIOMÁTICA

Verification of Nondeterministic Programs

Abstract

We continue with our series of introductory articles on the axiomatic verification of programs. In this second work, we focus on the nondeterministic sequential paradigm, always within the framework of imperative input/output programs. As nondeterminism is manifested in concurrency, the article also serves as an introduction to the verification of concurrent programs, in which a formal treatment of correctness verification is more justified due to their complexity. We work with a classic programming language, with nondeterministic conditional selections and repetitions,

Revista Abierta de Informática Aplicada/vol 6 Nº2 (2022): 46-71 *Recibido: Junio 2022 / Aceptado: Noviembre 2022 / Publicado: Diciembre 2022*
ISSN 2591-5320

and then incorporating random assignments. For the verification of the programs we propose an adaptation of the verification axiomatic method described in the previous publication, limited to deterministic sequential programming. We present examples of the application of the method and a systematic program development is also included, emphasizing again the approach of using the axioms and rules for programming as well as verifying, in order to obtain correct programs by construction. Finally, we introduce the concept of fairness, which effect is to reduce the degree of nondeterminism of a program based on certain equity criteria in the execution environment, and we describe a couple of adaptations in the verification rules to contemplate this aspect.

KEYWORDS: PROGRAM, NONDETERMINISM, VERIFICATION, AXIOMATICS

1. Introducción

El no determinismo es un concepto que aparece en distintas áreas de la computación, como la teoría de autómatas y la algorítmica, además de la teoría de la programación en la que nos concentramos en este artículo. En [AB09] se resume la historia del concepto, y para su estudio se presenta una taxonomía con distintos ejes (dominio, semántica, nivel de abstracción, método de ejecución, etc). Dicho trabajo destaca [RS59] como una de las primeras referencias a las *máquinas de Turing no determinísticas*, en las que el no determinismo es *existencial*, una máquina acepta una cadena de entrada si al menos lo hace en una de sus computaciones. Por su parte establece [Flo67] como punto de partida de los *algoritmos no determinísticos*, introducidos para expresar concisamente soluciones computacionales, implementadas con *backtracking*, abstracción del recorrido exhaustivo de todas las computaciones. Y entre los hitos iniciales en el marco de la programación menciona las publicaciones de E. Dijkstra sobre concurrencia [Dij68] y programación no determinística [Dij76]. En este último trabajo se describe el lenguaje GCL (por *Guarded Commands Language* o *Lenguaje de Comandos Guardados*), que empleamos en las próximas secciones.

En el área de la programación el no determinismo es útil fundamentalmente por dos motivos:

1. *Abstracción*. Se simplifica la construcción de programas, evitando determinismo innecesario. Los eventuales refinamientos para lograr la eficiencia esperada en la máquina objeto se posponen hasta etapas posteriores apropiadamente escogidas.

2. Algunos sistemas son inherentemente de naturaleza no determinística (por ejemplo, los sistemas operativos), por lo que indefectiblemente los lenguajes de programación deben contar con estructuras de dicha naturaleza.

En lo que sigue describimos el clásico método axiomático para probar programas secuenciales no determinísticos, adaptación del método para los programas secuenciales determinísticos presentado en el artículo anterior (recomendamos su lectura previa por plantear los pilares de la verificación axiomática de programas). Cabe remarcar:

- A diferencia de la teoría de autómatas, en el área de la programación la interpretación del no determinismo es *universal*: en la ejecución de un programa, representada por un árbol de computaciones posibles, se debe cumplir que *toda* computación sea correcta (de acuerdo a la semántica definida), por lo que verificar el programa consiste en probar que las propiedades

especificadas se cumplan en cada una de sus computaciones. Para la implementación se deja un rango de opciones, no se fuerza a que se reproduzcan todas las posibilidades, logrando más eficiencia que en la interpretación existencial.

- El no determinismo se manifiesta en la programación concurrente, por lo que tratar la verificación de programas secuenciales no determinísticos sirve como introducción a la problemática de la correctitud en la concurrencia, evitando su mayor complejidad. De hecho, en algunos textos se trata la verificación de un programa concurrente transformándolo primero en uno secuencial no determinístico y luego probándolo con su nueva estructura.

[Lau71] es la primera referencia a la correctitud de los programas no determinísticos, que se profundiza en [dBa80]. La idea de transformar programas concurrentes en programas secuenciales no determinísticos para su verificación se plantea en [AM71, FS78, FS81], mientras que en [Dij76] primero y [Gri81] después, el foco se pone en el desarrollo sistemático.

El artículo continúa de la siguiente manera. En la sección 2 se introducen los lenguajes utilizados, se muestran ejemplos de programas no determinísticos y se especifican las propiedades a probar. En las secciones 3 y 4 se describen los axiomas y reglas del método de verificación y se ejemplifica su aplicación. En la sección 5 se desarrolla un ejemplo de programación sistemática basada en los axiomas y reglas estudiados. En la sección 6 se amplía el lenguaje de programación con *asignaciones aleatorias*, que provocan un nuevo tipo de no determinismo con respecto al de las construcciones no determinísticas definidas antes, y se adapta el método de prueba para contemplarlas. En la sección 7 se introduce el concepto de *fairness*, íntimamente ligado al no determinismo, que consiste en establecer determinados criterios de equidad sobre la ejecución de las posibles computaciones de los programas, y así también se adapta el método de verificación para considerarlo. La sección 8 cierra el artículo con observaciones finales. Parte del material en que nos basamos aparece en [Apt84, Fra86, Fra92, AO97, RI10, PRS17]. Al igual que en el artículo anterior, por fines didácticos describimos el método como si se aplicara *a posteriori* (dados una especificación y un programa, probar que el programa es correcto con respecto a la especificación), pero la idea central sigue siendo valerse de sus conceptos fundamentales para desarrollar programas correctos por construcción, en el sentido de partir de una especificación y derivar un programa que la satisfaga en base a un cálculo bien definido.

2. Características generales del método de verificación

Verificar un programa consiste en probar formalmente sus propiedades con respecto a su especificación. En la verificación, entonces, intervienen dos artefactos formales, una especificación y un programa, expresados mediante lenguajes con sintaxis y semántica precisas. A continuación describimos los lenguajes con los que vamos a trabajar y las propiedades de programas que consideraremos.

2.1. Lenguaje de programación

La sintaxis en notación BNF (*Backus-Naur Form*) del lenguaje de programación que utilizaremos es la siguiente:

$$S :: skip \mid x := e \mid S_1 ; S_2 \mid \text{if } CG \text{ fi} \mid \text{do } CG \text{ od}$$
$$CG :: B \rightarrow S \mid B \rightarrow S \text{ or } CG$$

Los subíndices no son parte del lenguaje, se utilizan para facilitar su definición. La expresión *e* es de tipo entero (consideraremos en general variables simples de este tipo, sólo excepcionalmente variables simples booleanas y arreglos de enteros) y la expresión B es de tipo booleano. Las instrucciones *if CG fi* y *do CG od* son las instrucciones no determinísticas del lenguaje, respectivamente la *selección condicional* y *la repetición*. En general las denotaremos, la primera con *if* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *fi*, abreviándola con *IF*, y la segunda con *do* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *od*, abreviándola con *DO*. El componente GC, de forma $B \rightarrow S$, se conoce como *comando guardado* o *comando con guardia.* Las expresiones tienen la siguiente sintaxis:

$$e :: n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 . e_2 \mid \ldots \mid \text{if B then } e_1 \text{ else } e_2 \text{ fi}$$
$$B :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid \ldots$$

n es una constante entera, x es una variable entera, y *true* (por verdadero) y *false* (por falso) son las constantes booleanas. Completamos la descripción del lenguaje presentando, informalmente, la semántica de sus instrucciones:

1. La instrucción *skip* es atómica (se consume en un paso), y no tiene ningún efecto sobre las variables. Se puede usar, por ejemplo, en comandos guardados en que no deba efectuarse ninguna acción.

2. La *asignación x := e* también es atómica. Asigna el valor de *e* a la variable x.

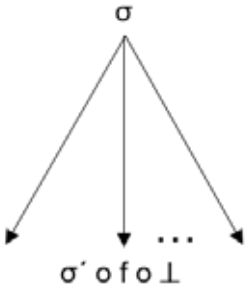
3. La *secuencia* S_1 *;* S_2 ejecuta S_1 y luego S_2.

4. La *selección condicional no determinística if* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *fi* se comporta de la siguiente manera: Si algunas de las guardias B_i son verdaderas, se opta no determinísticamente por una, se ejecuta la S_i asociada, y después el programa continúa con la instrucción siguiente a la selección. Y si en cambio ninguna de las guardias es verdadera, la selección (y el programa) *falla,* termina incorrectamente.

5. Finalmente, la *repetición no determinística do* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *od* actúa así: Si algunas de las guardias B_i son verdaderas, se opta no determinísticamente por una, se ejecuta la S_i asociada, se repite el mismo ciclo mientras existan guardias verdaderas, y si en algún momento todas las guardias son falsas el programa continúa con la instrucción siguiente a la repetición (no falla como la selección condicional). En caso de que siempre exista alguna guardia verdadera, la repetición (y el programa) *diverge*, se ejecuta infinitamente. La repetición es la única instrucción del lenguaje que puede provocar divergencia.

Cuando una guardia B_i de una selección condicional o una repetición es verdadera, se dice que la *dirección i* de la instrucción está *habilitada*. La elección entre las direcciones habilitadas es no determinística, sin ningún tipo de asunción de probabilidades. Por la semántica de las instrucciones no determinísticas, a partir del estado inicial de un programa (contenidos iniciales de las variables de programa) pueden haber varias computaciones, conformando un árbol de computaciones, y por lo tanto varios estados finales (contenidos finales de las variables de

programa). El caso más general es el de un árbol que incluye al mismo tiempo computaciones que *terminan,* computaciones que *fallan* y computaciones que *divergen*, como lo muestra la figura siguiente:



Es decir, a partir de un estado inicial σ, una computación de un programa no determinístico puede terminar en un estado final σ´ (conocido como estado *propio*), terminar en el estado de falla (denotado con f) o divergir (el símbolo ⊥ denota el estado *indefinido*, que representa la infinitud de la computación asociada).

Completamos la descripción del lenguaje de programación con ejemplos de programas. Los siguientes dos programas simulan, respectivamente, la selección condicional determinística *if B then* S_1 *else* S_2 *fi* y la repetición determinística *while B do S od*. En el primer caso corresponde:

$$\text{if } B \rightarrow S_1 \text{ or } \neg B \rightarrow S_2 \text{ fi}$$

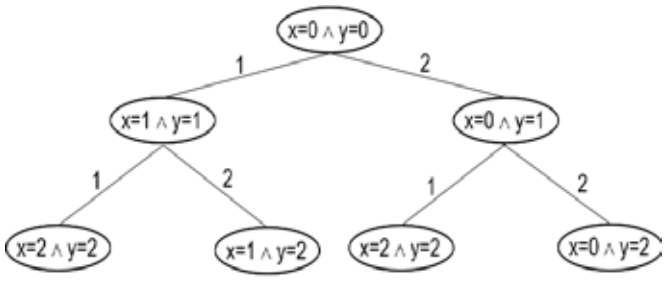
y en el segundo:

$$\text{do } B \rightarrow S \text{ od}$$

El siguiente programa obtiene en la variable x algún número natural del intervalo [0, N], con $N \geq 0$:

S_{sel} :: x := 0 ; y := 0 ;
do y < N → y := y + 1 ;
if true → x := y
or true → skip
fi
od

El árbol de computaciones asociado al programa S_{sel} cuando N = 2 es:



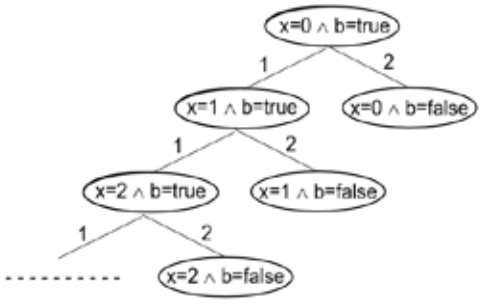
Cada nivel del árbol, desde el nodo raíz hasta las hojas, representa una iteración del DO. Los nodos contienen los valores que van adquiriendo las variables x e *y*. Las hojas representan los

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

estados finales del programa, asociados a sus posibles cuatro computaciones (a lo largo del DO, las dos direcciones del IF, etiquetadas con los números 1 y 2, están siempre habilitadas). Al no haber ninguna asunción de probabilidades, no puede asegurarse que luego de varias ejecuciones del programa se vayan a obtener en x los números naturales 0, 1 y 2. En este otro programa se obtiene en x *cualquier* número natural:

$$\begin{aligned} S_{nat} :: \; & x := 0 \; ; \; b := true \; ; \\ & \text{do } b \rightarrow \; x := x + 1 \\ & \text{or } b \rightarrow \; b := false \\ & \text{od} \end{aligned}$$

b es una variable booleana. Notar que en este caso el árbol de computaciones tiene una rama infinita, correspondiente a la computación en la que siempre se elige la primera dirección del DO:



Es decir, el programa S_{nat} cumple con su cometido, pero puede divergir, lo que se justifica por el *Lema de König* de la teoría de grafos: en un árbol infinito con grado finito alguna rama es infinita (el árbol de computaciones asociado al programa tiene infinitos estados finales, todos los números naturales). En efecto, las instrucciones IF y DO manejan finitos comandos guardados, por lo que todo programa tendrá finitos estados finales a menos que alguna de sus computaciones sea infinita. Este no determinismo se conoce como *no determinismo acotado*.

Los ejemplos que siguen sirven para destacar distintas características del lenguaje de programación no determinístico descripto. El siguiente programa calcula el máximo común divisor de dos números naturales mayores que cero mediante el *Algoritmo de Euclides*:

$$\begin{aligned} S_{mcd} :: \; & \text{do } x > y \rightarrow \; x := x - y \\ & \text{or } x < y \rightarrow \; y := y - x \\ & \text{od} \end{aligned}$$

El programa S_{mcd} termina cuando x = y (el máximo común divisor se obtiene en las dos variables). La simetría del programa permite entender fácilmente la idea del algoritmo. Lo mismo sucede con este otro programa, que ordena ascendentemente cuatro números:

$$\begin{aligned} S_{ord} :: \; & \text{do } n_1 > n_2 \rightarrow \; aux := n_1 \; ; \; n_1 := n_2 \; ; \; n_2 := aux \\ & \text{or } n_2 > n_3 \rightarrow \; aux := n_2 \; ; \; n_2 := n_3 \; ; \; n_3 := aux \\ & \text{or } n_3 > n_4 \rightarrow \; aux := n_3 \; ; \; n_3 := n_4 \; ; \; n_4 := aux \\ & \text{od} \end{aligned}$$

El programa termina cuando $n_1 \leq n_2 \leq n_3 \leq n_4$. Otra ventaja del lenguaje es que facilita la detección de fallas. Por ejemplo, para evitar una división indebida podemos escribir:

$$\text{if } y \neq 0 \rightarrow x := x / y \text{ fi}$$

Si y = 0, el programa falla. Otro ejemplo en este sentido se relaciona con el control de rango de índices, cuando se usan arreglos. Para evitar un índice fuera de rango se puede hacer:

$$\text{if } 0 \leq i \wedge i < n \rightarrow x := a[i] \text{ fi}$$

de modo tal que se produzca una excepción si se pretende acceder al arreglo *a* fuera del rango [0, n – 1]. Como último ejemplo mostramos cómo puede derivarse un programa secuencial no determinístico equivalente a uno concurrente. Dado:

$$S_{con} :: [x := 0 \,||\, x := 1 \,||\, x := 2]$$

podemos transformarlo de la siguiente manera:

$$\begin{aligned} S_{sec} :: {} & b_1 := \text{true} ; b_2 := \text{true} ; b_3 := \text{true} ; \\ & \text{do } b_1 \rightarrow x := 0 ; b_1 := \text{false} \\ & \text{or } b_2 \rightarrow x := 1 ; b_2 := \text{false} \\ & \text{or } b_3 \rightarrow x := 2 ; b_3 := \text{false} \\ & \text{od} \end{aligned}$$

Las variables booleanas b_i inicialmente verdaderas del programa secuencial S_{sec} dejan de valer una a una en un orden no determinístico, emulando el no determinismo de la secuencia de ejecución de los procesos del programa concurrente S_{con} (propio de la semántica de *interleaving*). La verificación del programa concurrente podría hacerse directamente sobre el programa secuencial, se supone de una manera más sencilla.

2.2. Lenguaje de especificación

Especificaremos los programas utilizando la *lógica de predicados* de la misma manera que lo hicimos en el artículo anterior, mediante un par de predicados (p, q), la *precondición* y la *postcondición*, que establecen la relación que debe existir entre los estados inicial y final. La sintaxis de un predicado p es la misma que la de una expresión booleana, salvo que también puede incluir los cuantificadores existencial y universal:

$$p :: \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \ldots \mid \exists x\colon p \mid \forall x\colon p$$

Por ejemplo, la especificación:

$$(x = X \wedge X > 0 \wedge y = Y \wedge Y > 0, x = \text{mcd}(X, Y))$$

es satisfecha por el programa del máximo común divisor presentado previamente:

$$S_{mcd} :: \text{ do } x > y \rightarrow x := x - y \text{ or } x < y \rightarrow y := y - x \text{ od}$$

En el ejemplo, x e *y* son variables de programa, y X e Y son variables de especificación, que sirven para resguardar el contenido inicial de las variables de entrada. La función mcd(X, Y)

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

abrevia la expresión que especifica el máximo común divisor de X e Y. En esta otra especificación ejemplificamos el uso del predicado *true*:

$$(\text{true}, x = 0 \vee x = 1 \vee x = 2)$$

true representa cualquier estado. Esta especificación establece que cualesquiera sean los contenidos de las variables de entrada, el programa requerido debe obtener en x el valor 0, 1 o 2 (la postcondición tiene la forma típica en el caso de los programas no determinísticos, con varios resultados posibles).

Los predicados denotan conjuntos de estados. Por ejemplo, la precondición del programa del máximo común divisor denota el conjunto de todos los estados con $x > 0$ e $y > 0$. Dicho conjunto define el alcance de aplicación del programa: a partir de un estado del conjunto, calcula el máximo común divisor. El predicado *true* denota el conjunto de todos los estados, y por oposición, el predicado *false* denota el conjunto vacío de estados. La expresión:

$$\sigma \models p$$

significa que el estado σ pertenece al conjunto de estados denotado por el predicado p. Otra manera de expresar lo mismo es que σ satisface p, o que p evaluado en σ es verdadero. Por ejemplo, si en un estado σ se cumple que $x = 0$, entonces:

$$\sigma \models (x = 0 \vee x = 1 \vee x = 2)$$

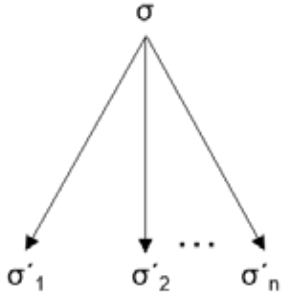
Cuantos más estados denote la precondición de un programa (en términos lógicos, cuanto más *débil* sea la precondición), mayor será el alcance de aplicación del programa. De acá surge el concepto de *precondición más débil*.

2.3. Propiedades a probar

La verificación de un programa no determinístico S, dada una especificación (p, q), requiere probar que a partir de todo estado σ que satisface la precondición p, toda computación de S:

1. Si termina (si no falla ni diverge), lo hace en un estado σ´ que satisface la postcondición q.
2. No falla.
3. No diverge.

En otras palabras, a partir de todo estado σ que satisface p, toda computación de S debe terminar en un estado σ´ que satisface q:



La propiedad (1) se conoce como *correctitud parcial.* El nombre proviene del concepto de función parcial. Se prueba *inductivamente*, lo mismo que la propiedad (2), la ausencia de fallas (para simplificar consideramos que la única falla posible es por la ejecución de una instrucción IF sin guardias verdaderas). Ambas propiedades pertenecen a la familia de propiedades *safety*, relacionadas con *eventos que no pueden ocurrir*. Por su parte, la propiedad (3), la no divergencia, no se prueba por inducción, y pertenece a la familia de propiedades *liveness*, relacionadas con *eventos que deben ocurrir*. La tres propiedades en conjunto constituyen la *correctitud total*, en definitiva lo que se busca probar. Algunos autores distinguen una categoría intermedia de *correctitud total débil*, que no incluye la propiedad de ausencia de fallas. Es habitual probar la correctitud total mediante pruebas separadas por propiedad, en cada una asumiendo que las otras propiedades se cumplen.

Las propiedades se representan mediante *fórmulas de correctitud* (también se conocen como *ternas de Hoare*), integradas por componentes de los lenguajes de programación y especificación:

- La fórmula {p} S {q} expresa la correctitud parcial: a partir de la precondición p, si el programa S termina lo hace cumpliéndose la postcondición q.

- La fórmula ⟨p⟩ S ⟨true⟩ expresa que a partir de la precondición p el programa S termina.

- Consistentemente, la correctitud total de un programa S con respecto a una especificación (p, q) se expresa con ⟨p⟩ S ⟨q⟩.

Por ejemplo, se cumple:

$$\langle x = 10 \rangle \text{ do } x > 1 \rightarrow x := x - 2 \text{ or } x > 0 \rightarrow x := x - 1 \text{ od } \langle x = 0 \rangle$$

También se cumple {true} S {true}, porque independientemente de la forma del programa S, si no termina a partir de un determinado estado inicial no se viola la correctitud parcial. En cambio, no se cumple ⟨true⟩ S ⟨true⟩, el siguiente programa falla si al comienzo $x = 0$:

$$\text{if } x > 0 \rightarrow y := x \text{ or } x < 0 \rightarrow y := -x \text{ fi}$$

Otro contraejemplo de ⟨true⟩ S ⟨true⟩, ahora por divergencia en caso de que al comienzo valga $x < 0$, es:

$$\text{do } x > 1 \rightarrow x := x - 2 \text{ or } x \neq 0 \rightarrow x := x - 1 \text{ od}$$

Si existe algún estado que satisface la precondición de un programa a partir del cual alguna computación: (a) termina en un estado que no satisface la postcondición, (b) falla, o (c) diverge, entonces el programa no es totalmente correcto con respecto a su especificación. Nada se puede decir en cambio si el estado inicial del programa no satisface la precondición, al estar fuera de su alcance de aplicación.

3. Verificación de la correctitud parcial

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

Para probar axiomáticamente los programas no determinísticos presentados, se mantienen naturalmente las reglas descriptas en el artículo anterior correspondientes a las instrucciones determinísticas:

1. Axioma del skip (SKIP): {p} skip {p}

El *skip* consume un paso y termina en el mismo estado en el que empieza. De esta manera, si un predicado se cumple antes de su ejecución, sigue valiendo después.

2. Axioma de la asignación (ASI): {p[x|e]} x := e {p}

Si se cumple el predicado p en términos de la variable x después de la asignación $x := e$, significa que antes se cumplía p en términos de la expresión e. p[x|e] denota el reemplazo de toda ocurrencia libre (no ligada a un cuantificador) de x en p por e.

$$\textit{3. Regla de la secuencia (SEC):} \quad \frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

Si se cumple$\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$, entonces se cumple $\{p\}\ S_1\ ;\ S_2\ \{q\}$. El predicado r actúa como nexo y luego se descarta. La regla se puede generalizar a cualquier número de premisas.

La novedad en el método de verificación está en las reglas asociadas a las instrucciones de selección condicional y repetición no determinísticas, que de todos modos conservan las mismas ideas del caso determinístico:

$$\textit{4. Regla del condicional (NCOND):} \quad \frac{\{p \wedge B_i\}\ S_i\ \{q\}\ ,\ \ i = 1, \ldots, n}{\{p\}\ \text{if}\ B_1 \rightarrow S_1\ \text{or}\ldots\text{or}\ B_n \rightarrow S_n\ \text{fi}\ \{q\}}$$

Si para toda dirección i, cuando S_i termina a partir de $p \wedge B_i$ se cumple q, entonces cuando la selección condicional *if* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *fi* termina a partir de p se cumple q. La regla establece una manera de probar una selección condicional con un único punto de entrada y un único punto de salida, correspondientes a la precondición p y la postcondición q, respectivamente.

$$\textit{5. Regla de la repetición (NREP):} \quad \frac{\{p \wedge B_i\}\ S_i\ \{p\}\ ,\ \ i = 1, \ldots, n}{\{p\}\ \text{do}\ B_1 \rightarrow S_1\ \text{or}\ldots\text{or}\ B_n \rightarrow S_n\ \text{od}\ \{p \bigwedge_i \neg B_i\}}$$

Si para toda dirección i, cuando S_i termina a partir de $p \wedge B_i$ se cumple p (es decir, preserva p), entonces cuando la repetición *do* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *od* termina a partir de p se cumple $p \bigwedge_i \neg B_i$. La regla se basa en un predicado *invariante* p, que debe cumplirse al comienzo de la repetición y luego de toda iteración de su cuerpo, compuesto por uno o más comandos guardados. Mientras alguna guardia B_i sea verdadera, S_i debe preservar el predicado p, y por eso si termina la repetición se va a cumplir $p \bigwedge_i \neg B_i$. La regla NREP no asegura la no divergencia de la repetición, y su forma muestra a las claras que la correctitud parcial se prueba inductivamente.

El método también mantiene las reglas semánticas que definimos en el artículo anterior, justamente porque son independientes del lenguaje de programación, se relacionan con el dominio de aplicación de los programas, en este caso los números enteros:

$$\textit{6. Regla de consecuencia (CONS):} \quad \frac{p \to p_1 \, , \, \{p_1\} \; S \; \{q_1\} \, , \, q_1 \to q}{\{p\} \; S \; \{q\}}$$

La regla CONS permite, en términos lógicos, reforzar precondiciones y debilitar postcondiciones. Por ejemplo, a partir de $r \to p$ y de $\{p\}$ S $\{q\}$ permite derivar $\{r\}$ S $\{q\}$. Por la forma de CONS, algunos pasos de una prueba son directamente enunciados verdaderos del dominio semántico.

$$\textit{7. Regla de instanciación (INST):} \quad \frac{f(X)}{f(c)}$$

La regla INST permite instanciar fórmulas de correctitud. f es una fórmula de correctitud, X es una variable de especificación, representa cualquier valor del dominio, y c es algún elemento del dominio de X. Por ejemplo, INST permite instanciar con X = 5 la fórmula {x = X} y := x {y = X}, obteniendo {x = 5} y := x {y = 5}. Al igual que la regla CONS, la regla INST es una regla universal cuya utilización en las pruebas se asume implícitamente.

El método es *sensato*, sólo obtiene fórmulas de correctitud verdaderas. También tiene la propiedad inversa, es *completo*, sus axiomas y reglas alcanzan para demostrar todas las fórmulas verdaderas, si bien se lo suele ampliar con reglas adicionales para acortar las pruebas, como se acostumbra en la lógica.

Completamos la sección con un ejemplo de aplicación del método:

Ejemplo 1. Correctitud parcial de un programa que obtiene un divisor de un número

El programa siguiente obtiene en *y* algún divisor de $x \geq 1$ (la expresión $z \mid x$ que se utiliza en el programa significa que *z* divide a *x*):

S_{div} :: z := 1 ; y := 1 ;
do z < x → z := z + 1 ;
if z | x → if true → y := z
or true → skip
fi
or ¬ z | x → skip
fi
od

Vamos a probar $\{x \geq 1\}$ S_{div} $\{y \mid x\}$. Como invariante de la instrucción DO usaremos el predicado $y \mid x$ (en la visión metodológica de obtención de programas correctos por construcción que destacamos en la introducción, en contraposición a las pruebas *a posteriori* de las que nos valdremos sólo por fines didáctivos, el invariante está primero, luego se escribe el DO para plasmarlo en términos del lenguaje de programación utilizado). Claramente, $y \mid x$ refleja la esencia del algoritmo planteado: la variable *y* irá recibiendo no determinísticamente a partir de su valor

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

inicial 1, a lo largo de la repetición, cero o más divisores de x, quedándose con alguno de ellos. El plan de prueba es el siguiente:

a) $\{x \geq 1\}\ z := 1\ ;\ y := 1\ \{y \mid x\}$. A partir del fragmento inicial se llega al invariante.
b) $\{y \mid x\}$ DO $\{y \mid x\}$. Si el DO termina, se alcanza la postcondición.
c) $\{x \geq 1\}\ S_{div}\ \{y \mid x\}$, por la aplicación de la regla SEC sobre (a) y (b).

Prueba de (a).

1. $\{1 \mid x\}\ y := 1\ \{y \mid x\}$ (ASI)
2. $\{1 \mid x\}\ z := 1\ \{1 \mid x\}$ (ASI)
3. $\{1 \mid x\}\ z := 1\ ;\ y := 1\ \{y \mid x\}$ (1, 2, SEC)
4. $x \geq 1 \rightarrow 1 \mid x$ (MAT)
5. $\{x \geq 1\}\ z := 1\ ;\ y := 1\ \{y \mid x\}$ (3, 4, CONS)

Como en la lógica, enumeramos los pasos e indicamos en cada uno qué axioma o regla se aplica y sobre qué pasos anteriores. En el paso 4 introdujimos directamente un predicado verdadero del dominio de los números enteros (en las siguientes aplicaciones de CONS daremos por sobrentendido el enunciado utilizado).

Prueba de (b).

6. $\{y \mid x \wedge z < x\}\ z := z + 1\ \{y \mid x\}$ (ASI, CONS)
7. $\{y \mid x \wedge z \mid x \wedge \text{true}\}\ y := z\ \{y \mid x\}$ (ASI, CONS)
8. $\{y \mid x \wedge z \mid x \wedge \text{true}\}$ skip $\{y \mid x\}$ (SKIP, CONS)
9. $\{y \mid x \wedge z \mid x\}$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi $\{y \mid x\}$ (7, 8, NCOND)
10. $\{y \mid x \wedge \neg z \mid x\}$ skip $\{y \mid x\}$ (SKIP, CONS)
11. $\{y \mid x\}$ if $z \mid x \rightarrow$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi or $\neg z \mid x \rightarrow$ skip fi $\{y \mid x\}$ (9, 10, NCOND)
12. $\{y \mid x \wedge z < x\}$
    $z := z + 1$; if $z \mid x \rightarrow$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi or $\neg z \mid x \rightarrow$ skip fi
    $\{y \mid x\}$ (6, 11, SEC)
13. $\{y \mid x\}$
    do $z < x \rightarrow z := z + 1$; if $z \mid x \rightarrow$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi or $\neg z \mid x \rightarrow$ skip fi od
    $\{y \mid x\}$ (12, NREP, CONS)

Los pasos 7 a 9 corresponden a la prueba del IF más interno, y los pasos 9 a 11 a la del otro IF. El paso 12 permite probar el DO, establece que su cuerpo preserva el invariante $y \mid x$ cuando se cumple la guardia $z < x$.

Prueba de (c).
14. $\{x \geq 1\}\ S_{div}\ \{y \mid x\}$ (5, 13, SEC)

El paso 14 concluye la verificación del programa considerando las fórmulas finales obtenidas en las pruebas de (a) y (b). La prueba no establece nada acerca de una posible falla o divergencia de S_{div}. Una presentación alternativa, ya insinuada en los últimos pasos de la prueba, es la *proof outline* (esquema de prueba), que consiste en intercalar algunos o todos los pasos de la demostración entre las instrucciones. Se obtiene una presentación más estructurada, muy útil para documentar programas. Por ejemplo, una *proof outline*

de correctitud parcial del programa S_{div} podría ser:

$\{x \geq 1\}$
$z := 1 ; y := 1 ;$
$\{y \mid x\}$
do $z < x \rightarrow \{y \mid x\}$
$z := z + 1 ; \{y \mid x\}$
if $z \mid x \rightarrow \{y \mid x \wedge z \mid x\}$
if true $\rightarrow y := z$ or true $\rightarrow$ skip fi $\{y \mid x\}$
or $\neg z \mid x \rightarrow \{y \mid x \wedge \neg z \mid x\}$
skip $\{y \mid x\}$
fi
$\{y \mid x\}$
od
$\{y \mid x\}$

La *proof outline* muestra qué predicados se cumplen en qué lugares del programa (en los que se comportan como invariantes, porque siempre se cumplen en esos lugares, independientemente del estado inicial). Es común insertar sólo los predicados relevantes, mínimamente la precondición, los invariantes de las repeticiones y la postcondición.

4. Verificación de la correctitud total

La instrucción IF es la única que puede provocar una falla en un programa, y la instrucción DO es la única fuente posible de divergencia. Las reglas NCOND y NREP para la prueba de correctitud parcial no contemplan estos casos, por lo que el método de verificación tiene un par de reglas más, que son en realidad extensiones de las reglas mencionadas, con más premisas, para cubrir la correctitud total.

Siguiendo con la convención de nomenclatura del artículo anterior, las nuevas reglas se identifican con NCOND* y NREP*, respectivamente, y los delimitadores $\langle\ \rangle$ reemplazan a $\{\ \}$. Las reglas son:

8. Regla del condicional sin falla (NCOND):*
$$\frac{p \rightarrow \bigvee_i B_i\ ,\ \langle p \wedge B_i \rangle\ S_i\ \langle q \rangle\ ,\ \ i = 1, \ldots, n}{\langle p \rangle\ \text{if}\ B_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n \rightarrow S_n\ \text{fi}\ \langle q \rangle}$$

Se le agrega a la regla NCOND una premisa para asegurar que la selección condicional no determinística no falle: si vale la precondición p, debe existir alguna guardia B_i que sea verdadera.

9. Regla de la no divergencia (NREP):*
$$\frac{\begin{array}{l} 1)\ \langle p \wedge B_i \rangle\ S_i\ \langle p \rangle\ ,\ i = 1, \ldots, n \\ 2)\ \langle p \wedge B_i \wedge t = Z \rangle\ S_i\ \langle t < Z \rangle\ ,\ i = 1, \ldots, n \\ 3)\ \ p \rightarrow t \geq 0 \end{array}}{\langle p \rangle\ \text{do}\ B_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n \rightarrow S_n\ \text{od}\ \langle p \bigwedge_i \neg B_i \rangle}$$

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

Se mantiene la idea de un invariante p, y se agrega un *variante* t, que es una función entera definida en términos de las variables de programa. La primera premisa es la misma que la de la regla NREP. Las otras dos premisas son las que aseguran la no divergencia de la repetición:

- Por la segunda premisa, la función t se decrementa en cada iteración, cualquiera sea la dirección tomada. Z es una variable de especificación que no ocurre en p ni t, y su objetivo es resguardar el valor de t previo a la ejecución de S_i.

- Por la tercera premisa, t arranca positiva y se mantiene así luego de cada iteración.

De esta manera, la instrucción DO necesariamente debe finalizar, porque los valores de t se reducen de iteración en iteración y nunca son negativos. La postcondición naturalmente es p $\bigwedge_i \neg B_i$ (o simplemente *true* cuando sólo interesa el hecho de la no divergencia). El invariante p se relaciona con el variante t de la siguiente manera: (a) Si vale p $\wedge$ B_i y se toma la dirección i, S_i decrementa t. (b) p asegura que la función t nunca se hace negativa. El valor inicial de t representa la cantidad máxima de iteraciones, y su decrecimiento se corresponde con el acercamiento al evento de finalización del DO. Queda claro que la prueba de no divergencia no es inductiva: no existe una noción de propiedad que se preserva a lo largo de una computación, sino de un evento que se producirá a futuro. La prueba se fundamenta en la inexistencia de cadenas descendentes infinitas en el dominio N de los números naturales con respecto a la relación < . El par (N, <) es un ejemplo de *orden parcial bien fundado*: conjunto con una relación antirreflexiva, antisimétrica y transitiva, sin cadenas que decrecen infinitamente.

Hemos notado que la regla NREP* permite derivar directamente la fórmula de correctitud $\langle p \rangle$ do $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ od $\langle p \bigwedge_i \neg B_i \rangle$, pero la aplicación de la regla implica siempre dos pruebas, una inductiva para la primera premisa y otra basada en un orden bien fundado para las dos restantes. Por eso la recomendación es probar separadamente {p} do $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ od {p $\bigwedge_i \neg B_i$} y $\langle p \rangle$ do $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ od $\langle true \rangle$, utilizando invariantes distintos (el invariante para la prueba de no divergencia es en general más simple, no todas las variables del invariante de la prueba de correctitud parcial deberían influir sobre la finalización del DO). Por su parte, la regla NCOND* sí podría sustituir directamente a la regla NCOND, para unir la prueba de correctitud parcial con la de ausencia de fallas, como lo plantean algunos autores. Las pruebas siguientes ejemplifican la aplicación de las reglas descriptas en esta sección:

Ejemplo 2. Ausencia de fallas en el programa que obtiene un divisor de un número

Antes probamos la fórmula de correctitud parcial $\{x \geq 1\}\ S_{div}\ \{y \mid x\}$, siendo S_{div}:

S_{div} :: z := 1 ; y := 1 ;
do z < x → z := z + 1 ;
if z | x → if true → y := z or true → skip fi or ¬ z | x → skip fi
od

En particular obtuvimos:

- $\{y \mid x \wedge z \mid x\}$ if true $\rightarrow y := z$ or true $\rightarrow$ skip fi $\{y \mid x\}$
- $\{y \mid x\}$ if $z \mid x \rightarrow$ if true $\rightarrow y := z$ or true $\rightarrow$ skip fi or $\neg z \mid x \rightarrow$ skip fi $\{y \mid x\}$

para el IF más interno y para el otro IF, respectivamente. En los dos casos, la precondición implica la disyunción de sus guardias (*true*), por lo que aplicando la regla NCOND* en lugar de NCOND logramos probar también que el programa no falla.

Ejemplo 3. Terminación del programa que calcula el máximo común divisor

Volviendo al programa que implementa el *Algoritmo de Euclides*, se puede probar sin dificultad su correctitud parcial a partir de un estado inicial con $x > 0$ e $y > 0$, es decir:

$$\{x = X \wedge X > 0 \wedge y = Y \wedge Y > 0\}$$
$$S_{mcd} :: \text{ do } x > y \rightarrow x := x - y \text{ or } x < y \rightarrow y := y - x \text{ od}$$
$$\{x = mcd(X, Y)\}$$

Por ejemplo, se puede utilizar el invariante $p = (mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0)$. El programa no puede fallar porque no tiene instrucciones IF. Para probar la no divergencia del programa debemos recurrir a la regla NREP*. Hay que definir un invariante p y un variante t. El invariante en este caso puede ser $p = (x > 0 \wedge y > 0)$. Como variante definimos $t = x + y$ (notar que la función t arranca positiva y se decrementa en cada iteración en *y*, si se cumple $x > y$, o en x, si se cumple $x < y$, por lo que siempre se mantiene positiva y así la repetición debe finalizar indefectiblemente, lo que se concreta cuando $x = y$). El plan de prueba es el siguiente:

a) $(x = X \wedge X > 0 \wedge y = Y \wedge Y > 0) \rightarrow (x > 0 \wedge y > 0)$
La precondición del programa implica el invariante del DO.

b) $\langle x > 0 \wedge y > 0 \wedge x > y \rangle\ x := x - y\ \langle x > 0 \wedge y > 0 \rangle$
$\langle x > 0 \wedge y > 0 \wedge x < y \rangle\ y := y - x\ \langle x > 0 \wedge y > 0 \rangle$
Primera premisa de NREP*.

c) $\langle x > 0 \wedge y > 0 \wedge x > y \wedge x + y = Z \rangle\ x := x - y\ \langle x + y < Z \rangle$
$\langle x > 0 \wedge y > 0 \wedge x < y \wedge x + y = Z \rangle\ y := y - x\ \langle x + y < Z \rangle$
Segunda premisa de NREP*.

d) $(x > 0 \wedge y > 0) \rightarrow (x + y \geq 0)$
Tercera premisa de NREP*.

También estas pruebas se pueden resolver sin mayores problemas, y así obtenemos:

$\langle x = X \wedge X > 0 \wedge y = Y \wedge Y > 0 \rangle$ do $x > y \rightarrow x := x - y$ or $x < y \rightarrow y := y - x$ od $\langle true \rangle$

Una *proof outline* de terminación del programa podría ser la siguiente (se usan los delimitadores $\langle\ \rangle$ y se deben especificar como mínimo, además de la pre y la postcondición, el invariante y el variante de la instrucción DO del programa):

Revista Abierta de Informática Aplicada/vol 6 Nº2 (2022): 46-71
ISSN 2591-5320

$\langle x = X \wedge X > 0 \wedge y = Y \wedge Y > 0 \rangle$
$\langle x > 0 \wedge y > 0 \rangle$
$\langle inv: x > 0 \wedge y > 0, var: x + y \rangle$
do
$x > y \rightarrow x := x - y$ or $x < y \rightarrow y := y - x$
$\langle x > 0 \wedge y > 0 \rangle$
od
$\langle x > 0 \wedge y > 0 \wedge x = y \rangle$
$\langle true \rangle$

Ejemplo 4. No divergencia del programa que obtiene un número natural entre 0 y N

Agregamos un ejemplo más de prueba de no divergencia para destacar la forma del variante utilizado, porque es típica de los programas no determinísticos. Vamos a probar la no divergencia del programa presentado previamente que devuelve un número natural entre 0 y N, pero escrito de otra manera. Demostraremos:

$\langle b \wedge x = 0 \wedge N \geq 0 \rangle$
do $b \wedge x < N \rightarrow x := x + 1$
or $b \rightarrow b := false$
od
$\langle true \rangle$

Los valores de x crecen en una unidad desde 0, y el programa finaliza con x = N si toma N veces la primera dirección, o con x < N si elige en alguna iteración anterior a la N-ésima la segunda dirección. Definimos $N \geq 0$ como invariante, y como variante:

$$t = \text{if } b \wedge x < N \text{ then } N - x + 1 \text{ else if } b \wedge x = N \text{ then } 1 \text{ else } 0 \text{ fi fi}$$

El máximo valor de t es N + 1, que representa la cantidad máxima de iteraciones posibles. Mientras se ejecute la primera dirección del DO, t se decrementa en 1. Cuando se pasa a la segunda dirección, indefectiblemente porque x = N o directamente por una selección no determinística, se ejecuta la asignación booleana para finalizar el DO, y t, que a lo sumo decreció hasta 1, se hace 0. Expresándolo de otra manera, t va decreciendo en escalones asociados a los números N + 1, N, N – 1, N – 2, …, llegando a 0 de esta manera uniforme o saltando de golpe desde un escalón intermedio. La prueba de no divergencia es la siguiente:

a) $(b \wedge x = 0 \wedge N \geq 0) \rightarrow N \geq 0$
La precondición del programa debe implicar el invariante del DO. Se cumple trivialmente.

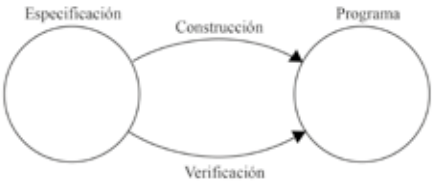
b) $\langle N \geq 0 \wedge b \wedge x < N \rangle\ x := x + 1 \langle N \geq 0 \rangle$
$\langle N \geq 0 \wedge b \rangle\ b := false\ \langle N \geq 0 \rangle$
Primera premisa de la regla NREP*. Ambas fórmulas se cumplen empleando ASI y CONS.

c) $\langle N \geq 0 \ \wedge b \wedge x < N \wedge t = Z \rangle \ x := x + 1 \ \langle t < Z \rangle$
$\langle N \geq 0 \ \wedge b \wedge t = Z \rangle \ b := false \ \langle t < Z \rangle$
Segunda premisa de la regla NREP*. La primera fórmula se cumple porque a partir de $t = N - x + 1$ (caso $b \wedge x < N$), luego de la instrucción $x := x + 1$ el valor de t se decrementa en 1. La segunda fórmula se cumple porque a partir de $t \geq 1$ (caso $b \wedge x \leq N$), luego de la instrucción $b := false$ el valor de t es 0.

d) $N \geq 0 \rightarrow t \geq 0$
Tercera premisa de la regla NREP*. Si se cumple $b \wedge x < N$, $t = N - x + 1$. Si se cumple $b \wedge x = N$, $t = 1$. Y si se cumple $\neg b$, $t = 0$. En los 3 casos vale $t \geq 0$.

5. Desarrollo sistemático de programas

En esta sección ejemplificamos el desarrollo sistemático de un programa guiado por el método de verificación descripto, idea fuerza que venimos sosteniendo en nuestros artículos:



Como lo indica la figura, la idea es construir y probar simultáneamente un programa, teniendo en cuenta los principios plasmados en los axiomas y reglas del método de verificación. Para simplificar, nos centraremos en una estructura muy sencilla de programa, con un fragmento inicial y una instrucción DO. Dados un invariante p y un variante t, supongamos que se quiere construir un programa:

$$P :: T \ ; \text{do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od}$$

tal que se cumpla la fórmula de correctitud total:

$$\langle r \rangle \ P \ \langle q \rangle$$

Según el método de verificación que planteamos, tienen que satisfacerse los siguientes requerimientos:

1. A partir de la precondición r, T establece el invariante p: $\langle r \rangle \ T \ \langle p \rangle$.
2. El predicado p es efectivamente un invariante del DO: $\langle p \wedge B_i \rangle \ S_i \ \langle p \rangle$, $i = 1, \ldots, n$.
3. El variante t decrece con cada iteración: $\langle p \wedge B_i \wedge t = Z \rangle \ S_i \ \langle t < Z \rangle$, $i = 1, \ldots, n$.
4. Mientras se cumpla el invariante p, el variante t no es negativo: $p \rightarrow t \geq 0$.
5. Al terminar el DO, se cumple la postcondición q. Es decir, $(p \bigwedge_i \neg B_i) \rightarrow q$.

La siguiente *proof outline* de correctitud total resume los requerimientos:

$$\langle r \rangle\ T\ ;\ \langle \text{inv: } p\ ,\ \text{var: } t \rangle\ \text{do}\ B_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n \rightarrow S_n\ \text{od}\ \langle p \wedge_i \neg B_i \rangle\ \langle q \rangle$$

Siguiendo esta guía, construiremos un programa que, dados tres arreglos de números enteros a, b y c, ordenados crecientemente, sin repeticiones y con uno o más elementos en común, devuelve las posiciones del menor de ellos. Especificamos la postcondición q de la siguiente manera (no vamos a expresar formalmente la precondición r):

$$q = (i = im \wedge j = jm \wedge k = km)$$

siendo i, j y k variables de programa que referencian genéricamente las posiciones de a, b y c, y siendo $im \geq 0$, $jm \geq 0$ y $km \geq 0$ variables de especificación que referencian las posiciones del menor elemento común en los tres arreglos, respectivamente. Considerando un algoritmo de búsqueda que comience desde el inicio de los arreglos, resulta natural plantear el siguiente invariante, que acota los valores de i, j y k:

$$p = (r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km)$$

y el siguiente variante:

$$t = (im - i) + (jm - j) + (km - k)$$

que irá decreciendo a medida que se incrementen las variables i, j y k con las asignaciones respectivas $i := i + 1$, $j := j + 1$ y $k := k + 1$ para recorrer los arreglos. Para que valga el invariante inicialmente (requerimiento 1), hacemos:

$$T :: i := 0\ ;\ j := 0\ ;\ k := 0$$

Es decir, inicialmente i, j y k apuntan a a[0], b[0] y c[0], respectivamente. Por ASI, SEC y CONS, se cumple efectivamente:

$$\langle r \rangle\ i := 0\ ;\ j := 0\ ;\ k := 0\ \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \rangle$$

Notar que podemos plantear la siguiente *proof outline* parcial hasta este momento:

$\langle r \rangle$
$i := 0\ ;\ j := 0\ ;\ k := 0$
$\langle \text{inv: } p\ ,\ \text{var: } t \rangle$
$\text{do}\ B_1 \rightarrow \langle p \wedge B_1 \rangle\ i := i + 1$
$\text{or}\ B_2 \rightarrow \langle p \wedge B_2 \rangle\ j := j + 1$
$\text{or}\ B_3 \rightarrow \langle p \wedge B_3 \rangle\ k := k + 1$
od
$\langle p \wedge \neg B_1 \wedge \neg B_2 \wedge \neg B_3 \rangle$
$\langle q \rangle$

quedando sólo por encontrar B_1, B_2 y B_3. Lo lógico sería utilizar i ≠ im, j ≠ jm y k ≠ km, pero las variables de especificación no pueden aparecer en el programa. Veamos cómo llegamos a otras guardias booleanas que representen la misma idea. Se cumple:

- i ≠ im es equivalente a i < im, que establece que a[i] no es el menor número común.
- a[i] < b[j] implica a[i] < b[jm] = a[im], que a su vez implica i < im.

De esta manera, se puede elegir como B_1 la expresión a[i] < b[j]. De manera similar se llega a b[j] < c[k] para B_2 y c[k] < a[i] para B_3. Y así llegamos a la siguiente *proof outline* final:

⟨r⟩
i := 0 ; j := 0 ; k := 0
⟨inv: p , var: t⟩
do a[i] < b[j] → ⟨p ∧ a[i] < b[j]⟩ ⟨p ∧ i < im⟩ i := i + 1
or b[j] < c[k] → ⟨p ∧ b[j] < c[k]⟩ ⟨p ∧ j < jm⟩ j := j + 1
or c[k] < a[i] → ⟨p ∧ c[k] < a[i]⟩ ⟨p ∧ k < km⟩ k := k + 1
od
⟨p ∧ ¬(a[i] < b[j]) ∧ ¬(b[j] < c[k]) ∧ ¬(c[k] < a[i])⟩
⟨q⟩

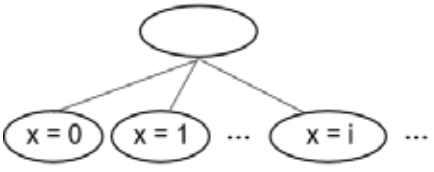
Se comprueba fácilmente el cumplimiento de los requerimientos 2, 3 y 4. El requerimiento 5 se cumple por lo siguiente: a la salida del DO vale a[i] = b[j] = c[k], que en conjunción con el invariante implica la postcondición. Para la construcción del programa hemos tenido que partir naturalmente de un solo invariante que sustente la correctitud parcial y la no divergencia.

6. Asignaciones aleatorias

El no determinismo del lenguaje de programación con el que venimos trabajando proviene de sus estructuras de control. Una instrucción no determinística adicional que consideraremos en lo que sigue se centra en los datos, es la *asignación aleatoria.* La forma de la instrucción es:

x := ?

que asigna algún número natural a la variable entera x. Notar que a diferencia de lo observado con la selección condicional y la repetición no determinísticas, la asignación aleatoria introduce un *no determinismo no acotado*: un programa aún sin computaciones infinitas puede tener infinitos estados finales:



Es decir, como lo muestra la figura, el árbol de computaciones asociado a *x := ?* tiene infinitas hojas, si bien todas sus ramas son finitas. No se asume ninguna hipótesis de probabilidad, y así por ejemplo, el siguiente programa puede divergir:

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

$$\text{do } x \neq n \rightarrow x := ? \text{ od}$$

En la literatura también aparece una variante de la forma $x := ? \leq y$, con la que se asigna a x un número del intervalo natural [0, y]. En este caso el no determinismo es acotado, se lo conoce como *finito* para diferenciarlo del que ya vimos, porque no depende de la sintaxis del programa. No consideraremos esta variante, ya sabemos cómo simularla sin recurrir a asignaciones aleatorias, por ejemplo de la siguiente manera:

$$\begin{array}{l} b := \text{true} \ ; \ x := 0 \ ; \\ \text{do } b \wedge x < y \rightarrow x := x + 1 \\ \text{or } b \wedge x < y \rightarrow b := \text{false} \\ \text{od} \end{array}$$

Para la prueba de correctitud parcial de programas con asignaciones aleatorias agregamos al método de verificación un axioma que captura la semántica de la nueva instrucción:

10. Axioma de la asignación aleatoria (NASI): $\{\forall x \geq 0: p\}\ x := ?\ \{p\}$

Es decir, si se cumple el predicado p en términos de x después de la asignación, significa que antes de la asignación p se cumplía en términos de cualquier valor entero $x \geq 0$. Para la prueba de no divergencia se mantiene la idea establecida por la regla NREP*, pero debemos modificarla en algún aspecto. Sucede que no alcanza con el dominio de los números naturales para definir en todos los casos los variantes de las repeticiones. Por ejemplo, observemos el siguiente programa:

$$\begin{array}{l} S_{nasi} :: \text{do } b \wedge x > 0 \rightarrow x := x - 1 \\ \quad \text{or } b \wedge x < 0 \rightarrow x := x + 1 \\ \quad \text{or } \neg b \rightarrow x := ? \ ; \ b := \text{true} \\ \quad \text{od} \end{array}$$

El programa S_{nasi} termina cualquiera sea su estado inicial. Si al comienzo b es verdadero, S_{nasi} termina al cabo de $|x|$ iteraciones. Si es falso también termina, pero no se puede predecir el número de iteraciones porque no depende del estado inicial, se conoce recién después de la ejecución de la asignación aleatoria. Así, el valor inicial del variante t debe cumplir:

$$t \geq x, \text{ para todo } x \geq 0$$

lo que resulta imposible para una expresión t de tipo entero. Hay que recurrir a otro orden parcial bien fundado para el variante. Se utiliza $W = N \cup \{\omega\}$, siendo ω el primer ordinal infinito, que cumple que es mayor que todos los números naturales. Llamaremos NREP** a la nueva regla, cuya forma entonces es:

*11. Regla de la no divergencia con ordinales infinitos (NREP**):*

$$\frac{\begin{array}{l} 1)\ \langle p \wedge B_i \rangle\ S_i\ \langle p \rangle\ ,\ i = 1, \ldots, n \\ 2)\ \langle p \wedge B_i \wedge t = \alpha \rangle\ S_i\ \langle t < \alpha \rangle\ ,\ i = 1, \ldots, n \\ 3)\ \ p \rightarrow t \in W \end{array}}{\langle p \rangle\ \text{do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \langle p \wedge_i \neg B_i \rangle}$$

La función t varía en el orden parcial bien fundado ($W = N \cup \{\omega\}, <$), siendo $<$ la relación habitual, y la variable α toma valores de W y no ocurre en p ni t.

Como ejemplo de aplicación de la regla NREP** vamos a probar la no divergencia del programa S_{nasi}. Definimos como invariante el predicado *true*, y como variante:

$$t = \text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}$$

La premisa (1) de NREP** se cumple trivialmente, lo mismo que la premisa (3). Con respecto a la premisa (2), hay que probar:

a) $\langle true \wedge b \wedge x > 0 \wedge t = \alpha \rangle\ x := x - 1\ \langle t < \alpha \rangle$
b) $\langle true \wedge b \wedge x < 0 \wedge t = \alpha \rangle\ x := x + 1\ \langle t < \alpha \rangle$
c) $\langle true \wedge \neg b \wedge t = \alpha \rangle\ x := ?\ ;\ b := true\ \langle t < \alpha \rangle$

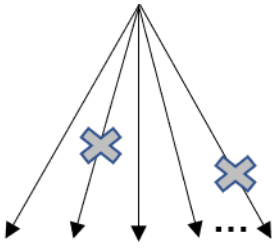
Las pruebas de (a) y (b) se resuelven fácilmente aplicando ASI y CONS. La prueba de (c) es como sigue:

1.	$\langle \lvert x \rvert < \alpha \rangle\ b := true\ \langle t < \alpha \rangle$	(ASI, CONS)
2.	$\{\forall x \geq 0: \lvert x \rvert < \alpha\}\ x := ?\ \{\lvert x \rvert < \alpha\}$	(NASI)
3.	$\{\alpha = \omega\}\ x := ?\ ;\ b := true\ \langle t < \alpha \rangle$	(1, 2, SEC, CONS)
4.	$\langle \neg b \wedge t = \alpha \rangle\ x := ?\ ;\ b := true\ \langle t < \alpha \rangle$	(3, CONS)

Si bien las reglas NREP* y NREP** se basan específicamente en $(N, <)$ y $(N \cup \{\omega\}, <)$ para la definición del variante, en la prueba de no divergencia (en realidad de cualquier propiedad de tipo *liveness*) se podría recurrir a cualquier orden parcial bien fundado apropiado.

7. Fairness

Completamos este artículo con una sección introductoria sobre el *fairness*, hipótesis del entorno de ejecución de los programas no determinísticos con distintas variantes presentes en muchos lenguajes de programación. El *fairness* establece que teniendo que elegir repetidamente entre distintas alternativas, todas serán escogidas en algún momento, ninguna quedará postergada de manera perpetua. Así, reduce el grado de no determinismo de un programa. Su efecto puede representarse como una poda en el árbol de computaciones de un programa, la poda de las ramas *unfair* según la semántica asumida:



En el contexto particular de la concurrencia, el *fairness* establece que todos los procesos deben

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

progresar (con velocidad desconocida pero en definitiva positiva), todos deben ejecutar alguna vez su próxima instrucción atómica. Como ya vimos, los programas concurrentes pueden transformarse en programas secuenciales no determinísticos, y así estudiar el *fairness* sobre estos últimos, menos complejos, constituye una apropiada incursión inicial en el tema. En la bibliografía existente figuran distintos tipos de *fairness*. Nos concentraremos en dos, y los definiremos directamente a partir de ejemplos. Comenzamos con el programa ya visto que genera cualquier número natural:

S_{nat} :: x := 0 ; b := true ;
do b → x := x + 1
or b → b := false
od

Hemos indicado que el programa puede divergir, una posible computación de S_{nat} es la que siempre elige la primera dirección del DO. Si bien la segunda dirección está siempre habilitada, sin *fairness* puede suceder que nunca sea elegida. Si la semántica del lenguaje de programación asegurara que una dirección de una repetición, habilitada permanentemente a lo largo de una computación infinita, no puede ser postergada indefinidamente, entonces S_{nat} nunca divergiría, porque alguna vez inexorablemente elegirá la segunda dirección. Este tipo de *fairness* se denomina *fairness* débil (el por qué del nombre se explicará enseguida). Notar en cambio que en este segundo programa, aún con *fairness* débil hay posibilidad de divergencia:

S_{par} :: x := 0 ; b := true ;
do b → x := x + 1
or b ∧ par(x) → b := false
od

par(x) es verdadero si x es par. El programa genera cualquier número natural par, pero puede divergir si siempre elige la primera dirección del DO, lo que no viola el *fairness* débil porque la segunda dirección no está permanentemente habilitada, sino de manera intermitente. Si la semántica del lenguaje de programación asegurara que una dirección de una repetición, habilitada infinitas veces a lo largo de una computación infinita, no puede ser postergada indefinidamente, entonces S_{par} nunca divergiría, alguna vez optará por la segunda dirección. En este caso el *fairness* es *fuerte*. Si hay *fairness* fuerte entonces naturalmente hay *fairness* débil, esto explica la nomenclatura (algunos autores denominan *fairness* al *fairness* fuerte y *justice* al *fairness* débil). Completamos los ejemplos con este tercer programa, que puede divergir aún con *fairness* fuerte:

S_{syb} :: x := 1 ;
do x > 0 → x := x + 1
or x > 0 → x := x – 1
od

Por ejemplo, la computación de S_{syb} que elige de manera alternada la primera y la segunda dirección del DO no viola la hipótesis de *fairness* fuerte. Enfocándonos en los programas S_{nat} y S_{par}, observamos asumiendo *fairness* que el no determinismo que se manifiesta es no acotado. En efecto, ambos programas no divergen y a pesar de ello tienen un conjunto infinito de estados finales. Que el no determinismo no acotado se manifieste también con las asignaciones aleatorias

no es casual, existe una relación entre ellas y el *fairness* que analizaremos más adelante. Con hipótesis de *fairness*, la verificación se restringirá a las computaciones finitas y las computaciones infinitas *fair*, las que no contradigan la semántica asumida. Las reglas que hemos presentado para la correctitud parcial y la ausencia de fallas se mantienen, porque con el *fairness* sólo se descartan computaciones infinitas. Más en general, el *fairness* afecta solamente las propiedades de tipo *liveness*. Para la prueba de no divergencia con *fairness* débil *y con fairness* fuerte se usan comúnmente dos *approaches*, cuyas características principales pasamos a describir.

Para la prueba con *fairness* débil consideraremos una adaptación de la regla con la que trabajamos en las secciones precedentes. Ahora no se requerirá que el variante se decremente cualquiera sea la dirección elegida, sino que alcanzará con que lo haga sólo en determinadas direcciones *útiles* (por eso el método se conoce como *método de las direcciones útiles*). Concretamente, para probar la no divergencia de una instrucción DO se utiliza un invariante p, un variante t, y además:

- Debe cumplirse $\langle p \wedge B_i \rangle\ S_i\ \langle p \rangle$, para i = 1, …, n.

- Se elige un orden parcial bien fundado (W, <), con minimal w_0 (no existen elementos menores que él), en el que estará definido el variante.

- Por cada valor w del variante, distinto del minimal w_0, se define un conjunto no vacío de direcciones útiles D_w, llamadas así porque acortan la *distancia* a la finalización del DO, es decir, al escogerlas se decrementa el valor del variante:

  $$\langle p \wedge B_i \wedge t = w \rangle\ S_i\ \langle t < w \rangle, \text{ para } i \in D_w$$

  El conjunto restante de direcciones asociado a w, identificado con D'_w, puede ser vacío. Ninguna de estas direcciones alarga la distancia a la finalización del DO, al elegirlas el valor del variante se mantiene o se decrementa:

  $$\langle p \wedge B_i \wedge t = w \rangle\ S_i\ \langle t \leq w \rangle, \text{ para } i \in D'_w$$

- Debe cumplirse $p \rightarrow t \in W$, y además que $t = w_0$ implique $\bigwedge_i \neg B_i$.

Intuitivamente, p asegura que t varía en (W, <). Mientras $t = w > w_0$, siempre hay una dirección que lo decrementa y no puede ser postergada indefinidamente por haber f*airness* débil, y el resto de las direcciones no lo incrementan. Por lo tanto, en algún momento $t = w_0$ (el programa termina). Por ejemplo, sea el siguiente programa, que a partir de $b_1 \wedge b_2 \wedge x = 0 \wedge y = 0$ devuelve algún par de números naturales x e *y*:

S_{xy} :: do $b_1 \wedge b_2 \rightarrow x := x + 1$
or $b_1 \wedge b_2 \rightarrow b_1 :=$ false
or $\neg b_1 \wedge b_2 \rightarrow y := y + 1$
or $\neg b_1 \wedge b_2 \rightarrow b_2 :=$ false
od

Asumiendo fairness débil, se puede probar del siguiente modo que S_{xy} no diverge:

Revista Abierta de Informática Aplicada/vol 6 N°2 (2022): 46-71
ISSN 2591-5320

- Definimos el orden ($W = \{0, 1, 2\}, <$) con la relación $<$ habitual.
- Usamos el invariante $p = (b_1 \rightarrow b_2)$.
- Usamos el variante $t = \text{if } b_1 \wedge b_2 \text{ then 2 else if } \neg b_1 \wedge b_2 \text{ then 1 else 0 fi fi}$.
- Definimos los conjuntos de direcciones $D_2 = \{2\}$ y $D'_2 = \{1, 3, 4\}$ para $t = 2$, y los conjuntos de direcciones $D_1 = \{4\}$ y $D'_1 = \{1, 2, 3\}$ para $t = 1$.

El valor inicial de t es 2, una abstracción que representa la distancia máxima a la finalización del DO. Al comienzo valen permanentemente las primeras dos guardias. Por el *fairness* débil, en algún momento se elegirá la segunda dirección, t pasará a valer 1 y las últimas dos guardias serán ahora las verdaderas permanentemente. Finalmente, otra vez por el *fairness* débil, en algún momento se elegirá la cuarta dirección, t alcanzará el minimal 0 y el programa habrá culminado. En síntesis, el variante habrá descendido dos escalones, primero del valor 2 al valor 1, y luego del valor 1 al valor 0. Se puede comprobar fácilmente que se cumplen los requerimientos establecidos por el método.

El método de las direcciones útiles también se puede aplicar cuando se asume *fairness* fuerte. De todos modos, para este caso vamos a describir los aspectos salientes de otro *approach*, que incluye un *scheduler* (planificador) *fair* en el programa considerado, para eliminar sus computaciones *unfair* (en otras palabras, implementa el *fairness*). El *approach* se conoce como *método del scheduler explícito*, y consiste en:

- Primero se agregan al programa asignaciones aleatorias, con el objeto de anular las computaciones *unfair*.
- Luego se prueba que el programa transformado no diverge ya sin asunción de *fairness*, recurriendo a las reglas de verificación con asignaciones aleatorias.

En lugar de considerar un programa en particular, vamos a ejemplificar la aplicación de este *approach* sobre un esquema de programa muy simple, con dos direcciones, que se puede generalizar a varias (etiquetamos las direcciones para facilitar la descripción):

$$S_{sch} :: \text{do } 1\text{: } B_1 \rightarrow S_1 \text{ or } 2\text{: } B_2 \rightarrow S_2 \text{ od}$$

Primero entonces, introducimos asignaciones aleatorias para implementar una política de *fairness* fuerte. El programa queda de la siguiente manera:

$$\begin{aligned} S'_{sch} :: & \; z_1 := ? \; ; \; z_2 := ? \; ; \\ & \text{do } 1\text{: } B_1 \wedge z_1 \leq z_2 \rightarrow S_1 \; ; \; z_1 := ? \; ; \\ & \qquad \text{if } B_2 \rightarrow z_2 := z_2 - 1 \text{ or } \neg B_2 \rightarrow \text{skip fi} \\ & \text{or } 2\text{: } B_2 \wedge z_2 < z_1 \rightarrow S_2 \; ; \; z_2 := ? \; ; \\ & \qquad \text{if } B_1 \rightarrow z_1 := z_1 - 1 \text{ or } \neg B_1 \rightarrow \text{skip fi} \\ & \text{od} \end{aligned}$$

La transformación del programa permite efectivamente suprimir las computaciones *unfair*:

- Se introducen dos variables, z_1 y z_2, que representan las prioridades asignadas a las direcciones 1 y 2, respectivamente. El decremento de una variable implica el aumento de la prioridad de la dirección asociada.

- Al elegirse una dirección, la prioridad de la otra aumenta, siempre que esté habilitada, al tiempo que la prioridad de la primera se reinicializa. El decremento gradual de z_i asegura que la dirección i no será postergada indefinidamente.

La prueba se completa sin asunción de *fairness*, empleando el axioma de la asignación aleatoria (NASI) y la regla de la no divergencia con ordinales infinitos (NREP**).

8. Observaciones finales

La cuestión de si la programación no determinística es la natural, como predicara E. Dijkstra, es tan polémica como la planteada en un marco más amplio entre la programación imperativa y la programación funcional. Al margen de esta discusión, queda claro que el no determinismo es un concepto muy importante en el ámbito no sólo de la construcción sino también de la especificación de programas: además de la necesidad insoslayable del paradigma por la existencia de sistemas intrínsecamente no determinísticos, el no determinismo facilita el manejo adecuado del nivel de abstracción en cada etapa del desarrollo de software, lo que acompañado de técnicas rigurosas de refinamientos sucesivos contribuye sobremanera a la derivación de artefactos correctos y eficientes.

En el plano particular de la verificación de programas, comprobamos que el método axiomático para los programas no determinísticos mantiene los mismos conceptos básicos del que presentamos para el paradigma determinístico. Las reglas de verificación son adaptaciones de las que estudiamos antes, siguen siendo sensatas y completas, y constituyen una guía metodológica para el desarrollo de los programas en simultáneo con sus pruebas de correctitud, aspecto en que hemos puesto especial interés.

El estudio de la verificación de los programas no determinísticos resulta muy útil como incursión inicial en la problemática de la correctitud de los programas concurrentes, que trataremos en un siguiente artículo, permitiendo su análisis por medio de artefactos de software menos complejos. En este marco se destaca el *fairness*, presente en muchos lenguajes de programación, con la novedad semántica del no determinismo no acotado, y así de la insuficiencia de los números naturales como dominio universal para los variantes bien fundados en las pruebas de no divergencia de los programas.

Referencias

» [AB09] Armoni, M. & Ben-Hari, M. *The concept of nondeterminism: its development and implications for teaching.* ACM SIGCSE Bulletin, 41, 2, 2009.

» [AM71] Ashcroft, E. & Manna, Z. *Formalization of properties of parallel programs*. Machine Intelligence, 6, 17-41, 1971.

» [AO97] Apt, K. & Olderog, E. *Verification of secuencial and concurrent programs, second edition*. Springer-Verlag, 1997.

» [Apt84] Apt, K. Ten years of Hoare's logic, a survey, part 2: nondeterminism. Theoretical Computer Science, 28, 83-109, 1984.

» [dBa80] de Bakker, J. Mathematical theory of programm correctness. Prentice Hall International, Englewood Cliffs, N. J., 1980.

» [Dij68] Dijkstra, E. Co-operating sequential processes. Programming Languages, F. Genuys, Ed., Academic Press, New York, 43-112, 1968.

» [Dij76] Dijkstra, E. A discipline of programming. Prentice-Hall, 1976.

» [Flo67] Floyd, R. Nondeterministic algorithms. JACM, 14, 4, 636-644, 1967.

» [Fra86] Francez, N. Fairness. Springer-Verlag, 1986.

» [Fra92] Francez, N. Program verification. Addison-Wesley, 1992.

» [FS78] Flon, L. & Suzuki, N. Nondeterminism and the correctness of parallel programs. Formal Description of Programming Concepts, E. J. Neuhold, ed., North-Holland, Amsterdam, 598-608, 1978.

» [FS81] Flon, L. & Suzuki, N. The total correctness of parallel programs. SIAM J. Comput., 227-246, 1981.

» [Gri81] Gries, D. The science of programming. Springer-Verlag, 1981.

» [Lau71] Lauer, P. Consistent formal theories of the semantics of programming languages. Tech. Rep. 25, 121, IBM Laboratory Vienna, 1971.

» [PRS17] Pons, C., Rosenfeld, R. & Smith, C. Lógica para informática. EDULP, 2017.

» [RI10] Rosenfeld, R. & Irazábal, J. Teoría de la computación y verificación de programas. McGraw-Hill y EDULP, 2010.

» [RS59] Rabin, M. & Scott, D. Finite automata and their decision problems. IBM Journal of Research and Development, 3, 114-125, 1959.