Verificación de Programas Paralelos

» **Ricardo Rosenfeld**
Centro de Altos Estudios en Tecnología Informática (CAETI) - Universidad Abierta Interamericana (UAI)

Resumen

En este tercer artículo de la serie que venimos presentando sobre la verificación axiomática de programas (en su variante de la Lógica de Hoare), tarea emprendida en el marco del proyecto del CAETI de construcción de un ambiente para asistir en el desarrollo de software, comenzamos a tratar el paradigma concurrente, para el que más se justifica el empleo de un método de prueba riguroso. Trabajamos con el modelo de los programas paralelos, programas concurrentes con procesos que comparten variables y se comunican a través de ellas (a pesar de la problemática común, por razones de espacio dejamos para el cuarto y último artículo el modelo de los programas distribuidos, programas concurrentes cuyos procesos son disjuntos y se comunican por medio de mensajes). Como en las publicaciones anteriores, remarcamos la idea de utilizar las axiomáticas descriptas como guías para obtener programas correctos por construcción. Destacamos además que las nociones fundamentales de la verificación axiomática observadas cuando analizamos los programas secuenciales, principalmente los predicados invariantes y las funciones variantes. se preservan en el contexto concurrente, a pesar de tener que considerarse una mayor variedad de clases de programas, propiedades y aspectos (metodológicos, semánticos, metateóricos).

PALABRAS CLAVE: PROGRAMA PARALELO, VERIFICACIÓN, AXIOMÁTICA

Abstract

In this third article of the series that we have been presenting on the axiomatic verification of programs (in its variant of Hoare´s Logic), a task undertaken within the framework of the CAETI project for the construction of an environment to assist in software development, we begin to deal with the concurrent paradigm, for which the use of a rigorous proof method is most justified. We work with the model of parallel programs, concurrent programs with processes that share variables and communicate through them (despite the common issue, for space reasons we leave for the fourth and last article the model of distributed programas, concurrent programs whose processes are disjoint and communicate through messages). As in previous publications, we stress the idea of using the described axiomatics as guides to obtain correct programs by construction. We also highlight that the fundamental notions of axiomatic verification observed

Recibido: Marzo 2023 / Aceptado: Mayo 2023 / Publicado: Junio 2023

when we analyzed sequential programs, mainly invariant predicates and variant functions, are preserved in the concurrent context, despite having to consider a greater variety of program classes, properties and aspects (methodological, semantic, metatheoretical).

KEYWORDS: PARALLEL PROGRAM, VERIFICATION, AXIOMATICS

1. Introducción

Este tercer artículo y el de la próxima edición de la revista cierran una primera serie de cuatro publicaciones introductorias sobre la *verificación axiomática de programas* (en su variante de la *Lógica de Hoare*), tarea emprendida en el marco del proyecto de construcción de un ambiente para asistir en el desarrollo de software que se viene llevando a cabo en el CAETI. Pasamos a tratar el paradigma *concurrente*, para el que más se justifica el empleo de un método de prueba riguroso. Tal como planteamos antes la verificación de los programas secuenciales no determinísticos por medio de variantes axiomáticas de las utilizadas para los programas secuenciales determinísticos, así también planteamos ahora la verificación de los programas concurrentes con axiomáticas que extienden las anteriores, para introducir de manera gradual las características de la metodología, y sobre todo para mostrar que nociones tales como los *predicados invariantes* y las *funciones variantes* son esenciales en el enfoque axiomático cualquiera sea su alcance de aplicación (recomendamos al lector leer o releer, según el caso, los dos trabajos anteriores, [Ros22a, Ros22b]).

Un programa concurrente está compuesto por subprogramas secuenciales, conocidos como *procesos*, que se ejecutan concurrentemente por medio de uno o más procesadores, en este último caso en general con velocidades que pueden diferir ampliamente. Así, durante su ejecución existen simultáneamente varios puntos o posiciones de control, uno por proceso. Los procesos se pueden *comunicar* y *sincronizar*. Estas y otras características de los programas concurrentes hacen que su verificación resulte mucho más compleja que la de los programas secuenciales. Consideramos una semántica muy común de la concurrencia, el *interleaving*, que consiste en la intercalación no determinística de las acciones atómicas (ininterrumpibles) de los procesos, por lo que, como en los programas no determinísticos, en este marco los programas concurrentes pueden producir a partir de un estado inicial varias computaciones y varios estados finales.

Por razones de espacio, dividimos el análisis en dos artículos. Es que la verificación de programas concurrentes amerita considerar una amplia variedad de programas (según la instrucción de sincronización empleada, el modelo de comunicación adoptado, si las comunicaciones son sincrónicas o asincrónicas, etc.), de propiedades (propiedades *safety* como la *correctitud parcial*, la *ausencia de deadlock* y la *exclusión mutua*, y propiedades *liveness* como la *no divergencia* y la *ausencia de inanición*) y de aspectos (metodológicos como la *composicionalidad*, semánticos como el *fairness*, metateóricos como la *sensatez* y la *completitud*). Así, más allá de la problemática común, en este artículo trabajamos con los programas *paralelos*, con procesos que comparten variables y se comunican a través de ellas, y en el siguiente con los programas *distribuidos*, con procesos disjuntos que se comunican a través de *pasajes de mensajes*.

Las axiomáticas que presentamos se basan en los trabajos de S. Owicki y D. Gries iniciados en 1976 [OG76a, OG76b]. Ya en versiones previas del artículo fundacional de C. Hoare de 1969 sobre la verificación axiomática de programas [Hoa69], el autor menciona su intención de extender

el método para los programas concurrentes, que da a conocer recién en [Hoa72] y amplía en [Hoa75]. Un método similar al de S. Owicki y D. Gries es [Lam77]. En [Lam15] se describen algunos hitos en la evolución del tratamiento de la correctitud de los programas concurrentes, en particular los trabajos iniciales de E. Dijkstra sobre la exclusión mutua y la sincronización entre procesos, en ocasión de la construcción del sistema operativo THE [Dij65, Dij68]. [Sch18] y [Jon92] también tratan la historia de la verificación de los programas concurrentes, el segundo en particular más enfocado en el desarrollo sistemático de programas guiado por el método de prueba. Para profundizar en técnicas de verificación de programas concurrentes se puede recurrir a [Bar85, SA86]. Otro libro muy recomendable para consultar, de reciente aparición, es [Mal19], que reúne numerosos artículos de distintos autores sobre el fundamental aporte de L. Lamport en distintos campos de la programación concurrente (algorítmica, especificación, verificación, etc).

La exposición de los temas sigue la estructura de las publicaciones anteriores. Primero introducimos los programas con los que trabajamos y las propiedades a verificar. Después presentamos las axiomáticas correspondientes, acompañadas por ejemplos de aplicación. Como consideramos programas de dos lenguajes de programación (difieren en la instrucción de sincronización empleada), presentamos en secciones separadas las axiomáticas para una y otra clase de programas (secciones 2 y 3). Continuamos con un ejemplo de desarrollo sistemático de programa basado en las axiomáticas descriptas, reforzando la idea sostenida a lo largo de toda la serie de utilizar las axiomáticas como guías para obtener programas correctos por construcción, construyéndolos y probándolos al mismo tiempo (lo hacemos en la sección 4, para una sola clase de programas). Cerramos el artículo con observaciones finales (sección 5).

2. Lenguaje de variables compartidas

Los programas del primer lenguaje paralelo que consideramos tienen la siguiente forma:

$$S :: S_0 ; [S_1 \,||\, \ldots \,||\, S_n]$$

S_0 es un subprograma secuencial con instrucciones del lenguaje de los *while* utilizado en el primer artículo (puede no existir, su función es inicializar variables). $[S_1 \,||\, \ldots \,||\, S_n]$, que se puede abreviar con $[\,||_{i=1,n} S_i\,]$, es la composición paralela de un conjunto de procesos $S_1, \ldots, S_n$ (se pueden etiquetar), los cuales contienen instrucciones del lenguaje de los *while* y eventualmente instrucciones de sincronización *await*, que describimos enseguida. Los procesos pueden compartir variables. Para no complicar las descripciones, no se permite concurrencia anidada ni procesos dinámicos.

La ejecución de un programa $S_0 ; [\,||_{i=1,n} S_i\,]$ consiste en la ejecución de S_0 seguida de la ejecución concurrente de $S_1, \ldots, S_n$. El programa puede tener varias computaciones, producto de la semántica de *interleaving* asumida, y por lo tanto, a partir de un estado inicial puede obtener varios estados finales. Mientras haya una instrucción posible para ejecutar, el programa la ejecuta. Esta es la única hipótesis de progreso que manejamos para todas las clases de programas del artículo, conocida como *propiedad de progreso fundamental* (salvo mención explícita, no consideramos ninguna asunción de *fairness*).

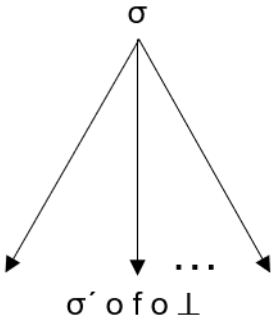
Los procesos se pueden comunicar a través de sus variables compartidas. De esta manera, el contenido de una variable compartida en un momento dado depende de las asignaciones que

se le hayan aplicado desde distintos procesos. Las ejecuciones de los *skip* y las asignaciones y las evaluaciones de las expresiones booleanas son atómicas (que no es lo que ofrecen los lenguajes concurrentes reales, que sólo garantizan la atomicidad de una *referencia crítica simple* - lectura o escritura exclusiva de una variable compartida -, pero optamos por este nivel de abstracción para simplificar la presentación, en lugar de complicar los programas con variables auxiliares o instrucciones de sincronización para lograr la atomicidad requerida). Y para ejecutar fragmentos de programa con variables compartidas más amplios que requieran ser ejecutados atómicamente, conocidos como *secciones críticas*, el lenguaje provee la instrucción de sincronización *await* que ya anticipamos, cuya sintaxis es:

$$\text{await } B \rightarrow S \text{ end}$$

B es una expresión booleana, y S es un subprograma que no contiene otras instrucciones *await* ni instrucciones *while*, lo que hace que la ejecución de un *await* siempre se complete. La ejecución consiste en evaluar B, si B resulta verdadera se ejecuta el *await* completo atómicamente, y si en cambio B resulta falsa, el proceso queda *bloqueado* hasta que en una instancia posterior progresa si es que el escenario se modifica.

Al igual que lo que observamos en los programas secuenciales no determinísticos, el *árbol de computaciones* de un programa del lenguaje de variables compartidas tiene en el caso más general computaciones que *terminan* (terminan sin falla), computaciones que *fallan* (terminan con falla) y computaciones que *divergen* (no terminan):



σ´ es un estado final sin falla, f es un estado de falla, y el símbolo ⊥ representa una computación infinita. La novedad es que las fallas son casos de *deadlock*, situaciones que se producen cuando uno o más procesos quedan bloqueados indefinidamente (para simplificar, no consideramos otras fallas).

Por ejemplo, el siguiente programa devuelve en la variable n el factorial de $N \geq 1$:

S_{fac} :: c_1 := true ; c_2 := true ; i := 1 ; j := N ; n := N ;
[S_1 :: while c_1 do await true → if i + 1 < j then i := i + 1 ; n := n . i else c_1 := false fi end od || S_2 :: while c_2 do await true → if j − 1 > i then j := j − 1 ; n := n . j else c_2 := false fi end od]

El programa tiene dos procesos que colaboran en el cálculo. Como n se inicializa con N, un proceso multiplica iterativamente n por 2, 3, …, y el otro multiplica iterativamente n por (N – 1), (N – 2), …. Los procesos terminan cuando sus respectivos índices i y j cumplen i + 1 = j. Los valores finales de i y j dependen del *interleaving* ejecutado (al no haber *fairness*, incluso puede darse el caso de que un solo proceso lleve a cabo todo el cálculo). Los *await* se utilizan sólo para lograr la *exclusión mutua* entre los procesos: cada cálculo parcial realizado por un proceso utiliza las variables compartidas n, i y j, por lo que debe efectuarse sin *interferencias* del otro proceso.

Este otro programa, en cambio, utiliza los *await* para sincronizar los procesos, y aprovecha mejor la concurrencia. El programa resuelve el clásico problema del productor-consumidor:

```
S_pc :: in := 0 ; out := 0 ; i := 1 ; j := 1 ;
    [prod :: while i ≤ n do                          cons :: while j ≤ n do
                x := a[i] ;                                     await in – out > 0 → skip end
                await in – out < b → skip end ;  ||             y := buffer[out mod b] ;
                buffer[in mod b] := x ;                         out := out + 1 ;
                in := in + 1 ;                                  b[j] := y ;
                i := i + 1                                      j := j + 1
                od                                              od]
```

El programa copia el arreglo *a*, recorrido por el índice i, en el arreglo b, recorrido por el índice j. Los arreglos, de números enteros, tienen n elementos. Para la copia se utiliza un tercer arreglo, *buffer*, de tamaño b, recorrido por los índices in y out, cuyos valores representan la cantidad de elementos que se agregan y se extraen de *buffer*, respectivamente. De esta manera, *buffer* tiene en todo momento in – out elementos, e in y out deben evaluarse módulo b. El proceso *prod* escribe en *buffer* siempre que haya espacio disponible, es decir si se cumple in – out < b, y el proceso *cons* lee de *buffer* siempre que éste tenga elementos, es decir si se cumple in – out > 0. Cuando alguna de estas condiciones no se cumple, el proceso respectivo queda bloqueado hasta que la evaluación de la condición resulta verdadera.

En lo que sigue, presentamos una axiomática para verificar programas del lenguaje de variables compartidas. Describimos axiomas y reglas para probar las propiedades que constituyen su *correctitud total*, que en este contexto se define de la siguiente manera: dado un programa S, una especificación (p, q) de S y cualquier estado inicial de S que satisfaga la precondición p, todas las computaciones de S finitas y sin *deadlock* deben terminar en un estado final que satisfaga la postcondición q (*correctitud parcial*), todas las computaciones de S deben ser finitas (*no divergencia*) y ninguna computación de S debe tener *deadlock* (*ausencia de deadlock*). Postergamos hasta las secciones dedicadas a la segunda clase de programas paralelos que tratamos referencias a otras dos propiedades, comúnmente consideradas en el paradigma concurrente: la *exclusión mutua* (o *ausencia de interferencia*) y la *ausencia de inanición*, que establecen la ejecución sin interferencias de las secciones críticas definidas, y el acceso a las mismas por parte de todos los procesos que las requieran, respectivamente.

2.1. Axiomática para las pruebas de correctitud parcial

La axiomática para las pruebas de correctitud parcial de programas paralelos del lenguaje de variables compartidas incluye los axiomas y reglas que presentamos para las pruebas de correctitud parcial de programas secuenciales determinísticos, e incorpora tres reglas más, dos correspondientes a las instrucciones novedosas del lenguaje, el *await* y la composición paralela, y una tercera, cuya necesidad explicamos más adelante. Describimos a continuación las dos primeras:

1. Regla del await (AWAIT)

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await } B \rightarrow S \text{ end } \{q\}}$$

Por la forma del *await*, la regla es muy similar a la regla del condicional (COND) estudiada antes. De hecho, no hace referencia a la posibilidad de bloqueo, porque la ausencia de *deadlock* se trata en otra prueba.

En lo que respecta a la regla correspondiente a la composición paralela, siguiendo el principio *composicional* sostenido en el marco secuencial su forma natural sería:

$$\frac{\{p_i\}\ S_i\ \{q_i\},\ i = 1, \ldots, n}{\{\bigwedge_{i=1,n} p_i\}\ [\ ||_{i=1,n} S_i\ ]\ \{\bigwedge_{i=1,n} q_i\}}$$

tratando a los procesos S_i como cajas negras, en este caso considerando las conjunciones de sus pre y postcondiciones en lugar de secuencializarlas. Pero desafortunadamente esta regla no es *sensata*, la conclusión no tiene por qué ser verdadera, como se demuestra con el siguiente contraejemplo. Dadas las fórmulas:

$$\{x = 0\}\ x := x + 2\ \{x = 2\}\ y\ \{x = 0\}\ y := x\ \{y = 0\}$$

no se cumple:

$$\{x = 0 \wedge x = 0\}\ [x := x + 2\ ||\ y := x]\ \{x = 2 \wedge y = 0\}$$

sino:

$$\{x = 0 \wedge x = 0\}\ [x := x + 2\ ||\ y := x]\ \{x = 2 \wedge (y = 0 \vee y = 2)\}$$

porque la variable *y* también puede terminar con el valor 2 si la asignación $y := x$ se ejecuta después de la asignación $x := x + 2$. La pérdida de la composicionalidad también se comprueba por el hecho de que ahora puede no ser inocuo reemplazar en un programa un proceso por otro semánticamente equivalente. Por ejemplo, reemplazando el proceso anterior $x := x + 2$ por el proceso $x := x + 1\ ;\ x := x + 1$, de las fórmulas:

$$\{x = 0\}\ x := x + 1\ ;\ x := x + 1\ \{x = 2\}\ y\ \{x = 0\}\ y := x\ \{y = 0\}$$

se obtiene la fórmula:

$$\{x = 0 \wedge x = 0\}\ [x := x + 1\ ;\ x := x + 1\ ||\ y := x]\ \{x = 2 \wedge (y = 0 \vee y = 1 \vee y = 2)\}$$

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

que como se observa, tiene una postcondición distinta a la de la fórmula obtenida con el proceso $x := x + 2$ (la variable y ahora también puede terminar con el valor 1 si la asignación $y := x$ se realiza entre las dos asignaciones $x := x + 1$).

En definitiva, el enfoque composicional que presentamos para los programas secuenciales no sirve para los programas paralelos, consecuencia de las interferencias entre los procesos. Una regla de prueba para la composición paralela debe contemplarlas. La regla que describimos a continuación pertenece a S. Owicki y D. Gries. Si bien no es composicional, es una buena aproximación, proponiendo un esquema sistemático y estructurado para las pruebas como las reglas anteriores. Existen variantes composicionales, por ejemplo la de C. Jones [Jon81], más orientado al desarrollo sistemático de programas, y que en un sentido reformula las ideas de S. Owicki y D. Gries, modificando la manera de especificar los programas. Optamos por la regla no composicional para mantener el estilo de las descripciones anteriores, y también porque es más accesible. La regla se define en términos de *proof outlines* (de correctitud parcial) de los procesos, imprescindibles en este contexto para tratar globalmente el comportamiento de los programas. Plantea, para probar la fórmula $\{p\}\ [\ ||_{i=1,n} S_i\ ]\ \{q\}$, encontrar para todos los procesos de la composición paralela *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ que satisfagan las implicaciones $p \longrightarrow \bigwedge_{i=1,n} p_i$ y $\bigwedge_{i=1,n} q_i \longrightarrow q$ y cumplan lo siguiente:

- para todo par de *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ y $\{p_j\}\ S_j^*\ \{q_j\}$,
- para todo predicado r de $\{p_i\}\ S_i^*\ \{q_i\}$ no interno de un *await* (conocido como *predicado normal*),
- y para toda instrucción T de $\{p_j\}\ S_j^*\ \{q_j\}$ que sea una asignación no interna de un *await* (conocida como *asignación normal*) o un *await*:

$$\{r \wedge pre(T)\}\ T\ \{r\}$$

siendo pre(T) la precondición de T en $\{p_j\}\ S_j^*\ \{q_j\}$. Las *proof outlines* que cumplen dicha condición se conocen como *libres de interferencia*. Efectivamente, notar que en ellas todo predicado normal es invariante en la posición que ocupa, cualquiera sea la asignación normal o el *await* que se ejecute (los predicados no normales no cuentan porque son invisibles globalmente dada la atomicidad de los *await*, y sólo se consideran las instrucciones que pueden modificar variables). En estas condiciones, claramente a partir de un estado que satisface la conjunción de las precondiciones de los procesos, si la composición paralela termina lo hace en un estado que satisface la conjunción de las postcondiciones de los mismos. Con estas definiciones formulamos de la siguiente manera la regla que adoptamos para verificar la correctitud parcial de programas del lenguaje de variables compartidas:

2. Regla de la composición paralela (PAR)

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i = 1, \ldots, n, \text{ son } \textit{proof outlines} \text{ libres de interferencia},\quad p \longrightarrow \bigwedge_{i=1,n} p_i,\ \bigwedge_{i=1,n} q_i \longrightarrow q}{\{p\}\ [\ ||_{i=1,n} S_i\ ]\ \{q\}}$$

PAR es una regla especial, una metarregla, porque además de fórmulas de correctitud tiene *proof outlines* como premisas. Su uso implica un esquema de pruebas en dos etapas, una etapa de *pruebas locales*, para verificar mediante *proof outlines* la correctitud parcial de los procesos considerados aisladamente, y una etapa de *prueba global*, para validar la libertad de interferencia

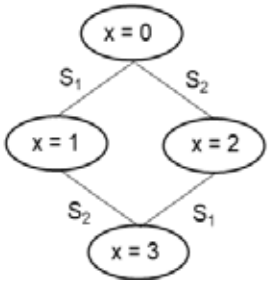
de las *proof outlines*. La prueba global suele ser muy trabajosa, consecuencia de la manera no estructurada con la que se pueden hacer referencias y actualizaciones a las variables compartidas: la cantidad de validaciones con n procesos de tamaño k es del orden de k^n (de todos modos, en la práctica muchas validaciones son triviales). Mostramos en lo que sigue algunos ejemplos de pruebas con el esquema planteado.

Ejemplo 1. Prueba de correctitud parcial de un programa de suma

Vamos a probar $\{x = 0\}\ [S_1 :: x := x + 1 \ ||\ S_2 :: x := x + 2]\ \{x = 3\}$.

Las *proof outlines* naturales de los procesos S_1 y S_2 considerados aisladamente, respectivamente $\{x = 0\}\ S_1 :: x := x + 1\ \{x = 1\}$ y $\{x = 0\}\ S_2 :: x := x + 2\ \{x = 2\}$, no son libres de interferencia (por ejemplo, al final de S_1 también puede cumplirse $x = 3$). Proponemos las siguientes *proof outlines*, justificadas por el diagrama que las acompaña, que representa los estados inicial, intermedios y final del programa y cómo se obtienen:

$\{x = 0 \vee x = 2\} \quad \{x = 0 \vee x = 1\}$
$[S_1 :: x := x + 1 \ ||\ S_2 :: x := x + 2]$
$\{x = 1 \vee x = 3\} \quad \{x = 2 \vee x = 3\}$



Claramente las *proof outlines* son correctas. También libres de interferencia, las siguientes fórmulas, definidas de acuerdo a la regla PAR, son verdaderas:

$$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 0 \vee x = 2\}$$
$$\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 1 \vee x = 3\}$$
$$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$$
$$\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 2 \vee x = 3\}$$

Además se cumple que la precondición del programa implica la conjunción de las precondiciones de los procesos, y que la conjunción de las postcondiciones de los procesos implica la postcondición del programa:

$$x = 0 \longrightarrow ((x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1))$$
$$((x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)) \longrightarrow x = 3$$

lo que completa la prueba. Se observa que para obtener *proof outlines* libres de interferencia hay que debilitar los predicados utilizados en las pruebas locales. Esta técnica no siempre deriva en una prueba exitosa, como mostramos en el siguiente ejemplo.

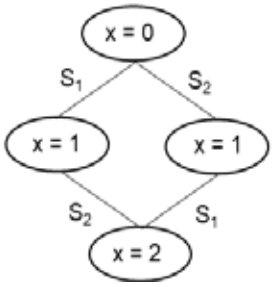
Ejemplo 2. Prueba de correctitud parcial de otro programa de suma

Vamos a probar $\{x = 0\}\ [S_1 :: x := x + 1 \ ||\ S_2 :: x := x + 1]\ \{x = 2\}$.

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

Las *proof outlines* $\{x = 0\}\ S_1 :: x := x + 1\ \{x = 1\}$ y $\{x = 0\}\ S_2 :: x := x + 1\ \{x = 1\}$ son las naturales pero no son libres de interferencia (al final de un proceso también se puede cumplir x = 2). Siguiendo el mecanismo del ejercicio anterior, proponemos las siguientes *proof outlines*:

$\{x = 0 \vee x = 1\} \quad \{x = 0 \vee x = 1\}$
$[S_1 :: x := x + 1 \ \|\ S_2 :: x := x + 1]$
$\{x = 1 \vee x = 2\} \quad \{x = 1 \vee x = 2\}$



pero tampoco son libres de interferencia. Notar por ejemplo que la fórmula:

$$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 1)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$$

no es verdadera. Se puede demostrar formalmente que no existen *proof outlines* libres de interferencia en este caso. Intuitivamente, lo que sucede es que expresando el estado intermedio solamente con x = 1 no alcanza para registrar la *historia* del *interleaving* llevado a cabo, es decir, si el proceso que ejecutó la primera asignación $x := x + 1$ fue S_1 o S_2.

Para resolver estos casos inevitables de incompletitud, el esquema de pruebas contempla, antes de la validación de libertad de interferencia, la ampliación del programa considerado con *variables auxiliares* (técnica propuesta en [Cli73]). La idea es agregarlas a los procesos, sin alterar el cómputo básico del programa, con el objetivo de reforzar los predicados de sus *proof outlines*. Primero completamos la prueba de este ejemplo con esta idea, y luego formalizamos la extensión necesaria en la axiomática. Proponemos la siguiente ampliación del programa (resaltamos las partes que se refieren a las variables auxiliares para distinguirlas del programa original):

$\mathbf{y := 0}\ ;\ \mathbf{z := 0}\ ;$
$\{(x = 0 \wedge \mathbf{y = 0} \wedge \mathbf{z = 0}) \vee (x = 1 \wedge \mathbf{y = 0} \wedge \mathbf{z = 1})\}$
$[S_1 ::$ await true $\rightarrow x := x + 1\ ;\ \mathbf{y := 1}$ end $\|$
$\{(x = 1 \wedge \mathbf{y = 1} \wedge \mathbf{z = 0}) \vee (x = 2 \wedge \mathbf{y = 1} \wedge \mathbf{z = 1})\}$

$\{(x = 0 \wedge \mathbf{y = 0} \wedge \mathbf{z = 0}) \vee (x = 1 \wedge \mathbf{y = 1} \wedge \mathbf{z = 0})\}$
$S_2 ::$ await true $\rightarrow x := x + 1\ ;\ \mathbf{z := 1}$ end$]$
$\{(x = 1 \wedge \mathbf{y = 0} \wedge \mathbf{z = 1}) \vee (x = 2 \wedge \mathbf{y = 1} \wedge \mathbf{z = 1})\}$

La idea es la siguiente. Se agregan la variable *y* al proceso S_1 y la variable z al proceso S_2, inicializadas en 0, y se las cambia a 1 después de que el proceso respectivo ejecuta su asignación $x := x + 1$ (los *await* son necesarios porque las dos asignaciones de cada proceso deben ejecutarse atómicamente). Esto refuerza los predicados y permite obtener *proof outlines* libres de interferencia (se comprueba fácilmente, lo mismo que la postcondición x = 2 a partir de la precondición x = 0). Y dado que el agregado de las variables auxiliares y sus asignaciones no alteran el cómputo básico, las *proof outlines* también sirven para probar el programa original.

La regla de prueba que formaliza el uso de variables auxiliares se formula de la siguiente manera:

3. Regla de las variables auxiliares (AUX)

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_{|A}\ \{q\}}$$

tal que:

- S es un programa ampliado con variables auxiliares de un conjunto A, y $S_{|A}$ es el programa original.
- Las variables auxiliares sólo aparecen en asignaciones de S, pero nunca en la parte derecha de una asignación a una variable original (para no alterar el cómputo básico).
- La postcondición q no incluye variables auxiliares.

Con estas definiciones, queda claro que de {p} S {q} se deriva {p} $S_{|A}$ {q}.
Completamos la serie de ejemplos mostrando la prueba de correctitud parcial del programa que calcula el factorial presentado previamente.

Ejemplo 3. Prueba de correctitud parcial del programa que calcula el factorial

Vamos a probar:

$\{c_1 \wedge c_2 \wedge i = 1 \wedge j = N \wedge n = N \wedge N \geq 1\}$

```
[S1 :: while c1 do                          S2 :: while c2 do
       await true → if i + 1 < j                   await true → if j – 1 > i
       then i := i + 1 ; n := n . i   ||           then j := j – 1 ; n := n . j
       else c1 := false fi end                     else c2 := false fi end
       od                                          od]
```

$\{n = N!\}$

Proponemos como invariantes de los *while* los siguientes predicados:

$p_1 = (i \leq j \wedge (\neg c_1 \rightarrow i + 1 = j) \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!)$, para el *while* del proceso S_1
$p_2 = (i \leq j \wedge (\neg c_2 \rightarrow j - 1 = i) \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!)$, para el *while* del proceso S_2

y como *proof outlines* de los procesos:

```
{i ≤ j ∧ (¬c1 → i + 1 = j) ∧                {i ≤ j ∧ (¬c2 → j – 1 = i) ∧
n . (i + 1) . … . (j – 1) = N!}             n . (i + 1) . … . (j – 1) = N!}
[S1 :: while c1 do                          S2 :: while c2 do
{c1 ∧ i ≤ j ∧ n . (i + 1) . … . (j – 1) = N!}  ||  {c2 ∧ i ≤ j ∧ n . (i + 1) . … . (j – 1) = N!}
       await true → if i + 1 < j                       await true → if j – 1 > i
       then i := i + 1 ; n := n . i                    then j := j – 1 ; n := n . j
       else c1 := false fi end                         else c2 := false fi end
{i ≤ j ∧ (¬c1 → i + 1 = j) ∧                {i ≤ j ∧ (¬c2 → j – 1 = i) ∧
n . (i + 1) . … . (j – 1) = N!}             n . (i + 1) . … . (j – 1) = N!}
       od                                          od]
{i ≤ j ∧ i + 1 = j ∧                        {i ≤ j ∧ j – 1 = i ∧ n . (i + 1) . … . (j – 1) =
n . (i + 1) ….. (j – 1) = N!}                       n . (i + 1) . … . (j – 1) = N!}
```


Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

Se prueba fácilmente que las *proof outlines* son correctas (no incluimos los predicados no normales) y libres de interferencia (como en los dos ejemplos previos, la validación se simplifica porque sólo hay que considerar una instrucción por proceso, el *await*). También se prueba fácilmente que se cumplen las implicaciones requeridas entre la precondición y la postcondición del programa y las conjunciones de las precondiciones y las postcondiciones de los procesos.

2.2. Axiomática para las pruebas de no divergencia

La descripción del esquema de verificación en dos etapas que acabamos de introducir sirve como base para el resto del artículo. Repasándolo: en una primera etapa de pruebas locales se prueban los procesos aisladamente por medio de *proof outlines*, y en una segunda etapa, la prueba global, se validan las *proof outlines* obtenidas en la primera etapa, con un criterio que depende de la propiedad considerada y el lenguaje utilizado.

Para el caso de la verificación de no divergencia de programas del lenguaje de variables compartidas, la axiomática cuenta para las *proof outlines* (de no divergencia) de las pruebas locales con la regla REP* introducida en la primera publicación, asociada a los *while* (los *while* constituyen la única fuente posible de computaciones infinitas). En cuanto a la prueba global, utiliza una regla de composición paralela que establece una validación de las *proof outlines* con más requisitos que la de la prueba de correctitud parcial, consecuencia del uso no sólo de invariantes sino también de variantes para los *while*. Concretamente, chequea también que el valor de un variante de un proceso no sea impactado por la interferencia de otro, de modo tal que provoque la divergencia del programa. Este segundo requisito se explica porque, tal como las definimos, las *proof outlines* no registran las pruebas de no divergencia sino sólo las definiciones de los variantes.

Precisando, la nueva regla plantea, para probar $\langle p \rangle [\|_{i=1,n} S_i] \langle q \rangle$, encontrar para todos los procesos *proof outlines* $\langle p_i \rangle S_i^{**} \langle q_i \rangle$ que cumplan las implicaciones $p \longrightarrow \bigwedge_{i=1,n} p_i$ y $\bigwedge_{i=1,n} q_i \longrightarrow q$, que sean libres de interferencia, y que satisfagan las siguientes dos condiciones adicionales:

- para todo par de *proof outlines* $\langle p_i \rangle S_i^{**} \langle q_i \rangle$ y $\langle p_j \rangle S_j^{**} \langle q_j \rangle$,
- para todo variante t de $\langle p_i \rangle S_i^{**} \langle q_i \rangle$,
- y para toda asignación normal o await T de $\langle p_j \rangle S_j^{**} \langle q_j \rangle$:

1. $\langle t = Z \wedge pre(T) \rangle$ T $\langle t \leq Z \rangle$, siendo pre(T) la precondición de T en $\langle p_j \rangle S_j^{**} \langle q_j \rangle$.
2. Toda *secuencia sintácticamente posible* de asignaciones normales y *await* dentro de un *while* decrementa el valor de su variante asociado.

Las secuencias sintácticamente posibles de instrucciones de un fragmento de programa U se obtienen teniendo en cuenta solamente la sintaxis del fragmento. Por ejemplo, las de un fragmento U_1 ; U_2 se obtienen secuencializando las de U_1 con las de U_2, y las de un fragmento *if B then* U_1 *else* U_2 *fi* contienen las de U_1 y las de U_2. Notar que la condición (1) permite que un proceso acelere la terminación de otro. La necesidad de la condición (2) se aclara más adelante por medio de un ejemplo. Las *proof outlines* libres de interferencia que cumplen las dos condiciones adicionales se conocen como *fuertemente libres de interferencia*. Con estas definiciones enunciamos la regla que adoptamos para verificar la no divergencia de programas del lenguaje de variables compartidas:

4. Regla de la no divergencia paralela (PAR*)

$\langle p_i \rangle\ S_i^{**}\ \langle q_i \rangle$, i = 1, …, n, son *proof outlines* fuertemente libres de interferencia,
$p \longrightarrow \bigwedge_{i=1,n} p_i$, $\bigwedge_{i=1,n} q_i \longrightarrow q$

$$\langle p \rangle\ [\ ||_{i=1,n} S_i\ ]\ \langle q \rangle$$

La regla PAR* tiene la misma característica que la regla PAR definida para la prueba de correctitud parcial, es una metarregla, porque incluye *proof outlines* como premisas. Al igual que las reglas que describimos para los programas secuenciales, permite probar conjuntamente la correctitud parcial y la no divergencia (la *correctitud total débil* tal como la definen algunos autores). No cubre la prueba de ausencia de *deadlock*, en su aplicación se asume que dicha propiedad se verifica aparte. Y como en el caso de los programas secuenciales no determinísticos, se puede relajar si se asume alguna hipótesis de *fairness*. Por ejemplo, en el caso del programa:

$$S_{loop} :: [\text{while } x = 1 \text{ do skip od} \ ||\ x := 0]$$

no sería necesario probar la no divergencia del primer proceso localmente, es decir considerándolo aisladamente, contando con una hipótesis de *fairness* que asegure que la asignación *x := 0* se va a ejecutar alguna vez.

Finalizamos la sección mostrando el uso de la regla con un par de ejemplos.

Ejemplo 4. Prueba de no divergencia de un programa de resta

Vamos a probar:

$$\langle x > 0 \wedge par(x) \rangle\ [S_1 :: \text{while } x > 2 \text{ do } x := x - 2 \text{ od} \ ||\ S_2 :: x := x - 1]\ \langle x = 1 \rangle$$

siendo el predicado par(x) verdadero sii x es par. Por lo simple que es el programa, verificaremos directamente su correctitud total (no hay posibilidad de *deadlock* por la inexistencia de instrucciones *await*).

Proponemos para la prueba las siguientes *proof outlines* (el predicado impar(x) es verdadero sii x es impar):

⟨inv: x > 0, var: x⟩		⟨par(x)⟩
[S_1 :: while x > 2 do	||	S_2 :: x := x − 1]
⟨x > 2⟩		⟨impar(x)⟩
x := x − 2		
⟨x > 0⟩		
od		
⟨x = 1 ∨ x = 2⟩		

Se comprueba fácilmente que las *proof outlines* son correctas y libres de interferencia, y que se cumplen las implicaciones requeridas entre la especificación del programa y las especificaciones de los procesos. Las *proof outlines* también cumplen las dos condiciones adicionales requeridas para ser fuertemente libres de interferencia:

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

1. La asignación *x := x – 1* del proceso S_2 no aumenta el valor del variante t = x.
2. La única secuencia sintácticamente posible de asignaciones normales e instrucciones *await* del *while* del proceso S_1 es la asignación *x := x – 2*, la cual decrementa el valor del variante.

Ejemplo 5. Prueba de no divergencia del programa del productor-consumidor

Vamos a considerar solamente la composición concurrente del programa referido, con la precondición in = 0 ∧ out = 0 ∧ i = 1 ∧ j = 1 ∧ b > 0 ∧ n > 0, obtenida de las asignaciones iniciales *in := 0*, *out := 0*, *i := 1* y *j := 1* (b es el tamaño del arreglo *buffer* y n es el tamaño de los arreglos *a* y b). Proponemos las siguientes *proof outlines*:

```
⟨inv: 1 ≤ i ≤ n + 1, var: n + 1 – i⟩
[prod :: while i ≤ n do  ⟨1 ≤ i ≤ n⟩
      x :=  a[i] ;  ⟨1 ≤ i ≤ n⟩
      await in – out < b →
      skip end;  ⟨1 ≤ i ≤ n⟩
      buffer[in mod b] := x ;  ⟨1 ≤ i ≤ n⟩
      in := in + 1 ;  ⟨1 ≤ i ≤ n⟩
      i := i + 1  ⟨1 ≤ i ≤ n + 1⟩
      od
⟨true⟩
```

```
⟨inv: 1 ≤ j ≤ n + 1, var: n + 1 – j⟩
cons :: while j ≤ n do  ⟨1 ≤ j ≤ n⟩
      await in – out > 0 →
      skip end ;  ⟨1 ≤ j ≤ n⟩
      y := buffer[out mod b] ;  ⟨1 ≤ j ≤ n⟩
      out := out + 1 ;  ⟨1 ≤ j ≤ n⟩
      b[j] := y ;  ⟨1 ≤ j ≤ n⟩
      j := j + 1  ⟨1 ≤ j ≤ n + 1⟩
      od]
⟨true⟩
```

También en este caso es fácil comprobar la correctitud de la prueba desarrollada. La precondición implica la conjunción de los invariantes, las *proof outlines* son correctas y libres de interferencia, ninguna instrucción de un proceso aumenta el variante del *while* del otro proceso, y cada *while* tiene una única secuencia sintácticamente posible de instrucciones, la cual decrementa el variante respectivo.

En los dos ejemplos anteriores no puede apreciarse bien el requerimiento de la regla de prueba PAR* de que todas las secuencias sintácticamente posibles de asignaciones normales y *await* de los *while* decrementen sus variantes. Para aclarar este punto presentamos el siguiente programa:

```
[S1 :: while x > 0 do
      y := 0;
      if y = 0 then x := 0
      else y := 0 fi
      od
||
S2 :: while x > 0 do
      y := 1;
      if y = 1 then x := 0
      else y := 1 fi
      od]
```

Considerados individualmente, los procesos S_1 y S_2 terminan, lo que se puede probar fácilmente, por ejemplo, con *proof outlines* de no divergencia con todos predicados *true*, incluyendo los invariantes de los *while*, y con los dos variantes definidos con el valor max(x, 0), es decir el máximo entre x y 0. De esta forma, las *proof outlines* resultan libres de interferencia, y además cumplen con la primera condición adicional de la regla PAR*: las asignaciones de un proceso no aumentan el valor del variante del *while* del otro proceso.

Sin embargo, el programa puede divergir, si ejecuta repetidamente la secuencia formada por la primera asignación *y := 0* de S_1, la primera asignación *y := 1* de S_2, la segunda asignación y := 0 de S_1 y la segunda asignación y := 1 de S_2.

Sucede en este caso que la segunda condición adicional de la regla PAR* no se cumple: notar que la secuencia *y := 0 ; y := 0* de S_1 y la secuencia *y := 1 ; y := 1* de S_2 son sintácticamente posibles y no decrementan el variante definido.

2.3. Axiomática para las pruebas de ausencia de deadlock

La regla para verificar la ausencia de *deadlock*, única falla que contemplamos en los programas paralelos descriptos, es otro ejemplo de metarregla. En verdad, esta característica se repite en todas las reglas siguientes que se aplican sobre la composición paralela de un programa.

Otra característica común de esta clase de reglas es que plantean arrancar de un conjunto de *proof outlines* de correctitud parcial libres de interferencia, el más adecuado según la propiedad que se pretenda verificar.

La particularidad en el caso de la propiedad de ausencia de *deadlock* es que no puede analizarse localmente, sino que su tratamiento requiere considerar directamente el programa completo. Adicionalmente, por limitaciones de la lógica de predicados, para su prueba se debe recurrir a configuraciones expresadas en términos de los puntos de control de los procesos. Específicamente, la prueba consiste en definir todas las configuraciones que representan potenciales casos de *deadlock*, y comprobar que ninguno es posible. Considerando un programa $[\|\|_{i = 1,n} S_i]$ y una precondición p, la regla de prueba de ausencia de *deadlock* se formula de la siguiente manera:

5. Regla de la ausencia de deadlock en programas paralelos (DEADLOCK)

- Paso 1. Obtener *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ de los procesos S_i del programa, libres de interferencia, y que cumplan la implicación $p \longrightarrow \bigwedge_{i = 1,n} p_i$.
- Paso 2. Definir sintácticamente en las *proof outlines* todas las configuraciones de la forma $C = (l_1, \ldots, l_n)$ que representan casos posibles de *deadlock*: cada l_i de una configuración es una etiqueta asociada a una instrucción *await* del proceso S_i o al final del mismo, con la salvedad de que en toda configuración debe haber al menos una etiqueta asociada a un *await*, porque si no no representaría una situación de *deadlock*.
- Paso 3. Caracterizar semánticamente las configuraciones definidas en el paso (2), mediante predicados, denominados *imágenes semánticas*. La imagen semántica M de una configuración C es una conjunción de predicados $r_1 \wedge \ldots \wedge r_n$ que tiene la siguiente forma: Si la etiqueta l_i de C está asociada a una instrucción *T :: await B → T´ end* del proceso S_i, entonces r_i es $pre(T) \wedge \neg B$, siendo pre(T) la precondición de T en la *proof outline* de S_i. Y si la etiqueta l_i de C está asociada al final de S_i, entonces r_i es q_i.
- Paso 4. Probar que todas las imágenes semánticas son falsas, lo que significa que no existe ningún caso de *deadlock* en el programa.

El siguiente ejemplo sobre un esquema de programa con tres procesos aclara los pasos anteriores. Supongamos que luego del paso 2 llegamos a:

$$\{p_1\}\ [S_1 :: \ldots \{pre(T)\}\ l_{11} \Rightarrow T :: \text{await } B_T \rightarrow T' \text{ end} \ldots l_{12} \Rightarrow \{q_1\}$$
$$\|$$
$$\{p_2\}\ S_2 :: \ldots l_2 \Rightarrow \{q_2\}$$
$$\|$$
$$\{p_3\}\ S_3 :: \ldots \{pre(U)\}\ l_{31} \Rightarrow U :: \text{await } B_U \rightarrow U' \text{ end} \ldots]\ l_{32} \Rightarrow \{q_3\}$$

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

es decir que las configuraciones que representan los casos posibles de *deadlock* son:

$$C_1 = (l_{11}, l_2, l_{31}),\ C_2 = (l_{12}, l_2, l_{31}) \text{ y } C_3 = (l_{11}, l_2, l_{32})$$

En el paso 3 definimos las imágenes semánticas de las configuraciones:

$$M_1 = (pre(T) \land \neg B_T) \land q_2 \land (pre(U) \land \neg B_U)$$
$$M_2 = q_1 \land q_2 \land (pre(U) \land \neg B_U)$$
$$M_3 = (pre(T) \land \neg B_T) \land q_2 \land q_3$$

Finalmente, en el paso 4 verificamos que todas las imágenes semánticas sean falsas. En lo que sigue ejemplificamos el uso de la regla DEADLOCK con un programa concreto. Volvemos una vez más al programa del productor-consumidor.

Ejemplo 6. Prueba de ausencia de deadlock en el programa del productor-consumidor

Partimos como antes de la precondición $in = 0 \land out = 0 \land i = 1 \land j = 1 \land b > 0 \land n > 0$. Proponemos la siguiente *proof outline* para el proceso del productor, en el que ya insertamos las etiquetas apropiadas:

$\{inv: 0 \le in - out \le b \land 1 \le i \le n + 1 \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
while $i \le n$ do
$\{0 \le in - out \le b \land 1 \le i \le n \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
$x := A[i]$;
$\{0 \le in - out \le b \land 1 \le i \le n \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
$l_{p1} \Rightarrow$ await $in - out < b \rightarrow$ skip end ;
$\{0 \le in - out < b \land 1 \le i \le n \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
buffer[in mod b] := x ;
$\{0 \le in - out < b \land 1 \le i \le n \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
in := in + 1 ;
$\{0 \le in - out \le b \land 1 \le i \le n \land 1 \le j \le n + 1 \land b > 0 \land i = in\}$
i := i + 1
$\{0 \le in - out \le b \land 1 \le i \le n + 1 \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$
od
$l_{p2} \Rightarrow \{0 \le in - out \le b \land i = n + 1 \land 1 \le j \le n + 1 \land b > 0 \land i = in + 1\}$

De una manera similar se puede plantear la *proof outline* etiquetada para el proceso del consumidor, libre de interferencia con respecto a la del proceso del productor. No la desarrollamos, sólo formulamos los predicados que interesan para la prueba, y asociamos la etiqueta l_{c1} a la instrucción *await* y la etiqueta l_{c2} al final del proceso. El invariante del *while* es:

$$0 \le in - out \le b \land 1 \le i \le n + 1 \land 1 \le j \le n + 1 \land b > 0 \land j = out + 1$$

la precondición de la instrucción *await in – out > 0 → skip end* es:

$$0 \le in - out \le b \land 1 \le i \le n + 1 \land 1 \le j \le n \land b > 0 \land j = out + 1$$

y la postcondición del proceso es:

$$0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge j = n + 1 \wedge b > 0 \wedge j = out + 1$$

Claramente, la precondición del programa implica la conjunción de los invariantes de los *while*. Hay tres casos posibles de *deadlock*, representados por $C_1 = (l_{p1}, l_{c1})$, $C_2 = (l_{p2}, l_{c1})$ y $C_3 = (l_{p1}, l_{c2})$. Veamos a qué situaciones se refieren y si sus imágenes semánticas son falsas:

- C_1 representa el caso en el que el productor y el consumidor están bloqueados en sus *await*. De la conjunción de las precondiciones de los *await* y las negaciones de sus expresiones booleanas se obtiene $b > 0 \wedge in - out \geq b \wedge in - out \leq 0$, que implica la conjunción falsa $b > 0 \wedge b \leq 0$.
- C_2 representa el caso en el que el productor termina y el consumidor está bloqueado en su *await*. De la postcondición del productor se obtiene $in = n$. De la precondición del *await* del consumidor se obtiene $0 \leq in - out \leq b \wedge 1 \leq j \leq n \wedge j = out + 1$. Y de la negación de la expresión booleana de dicho *await* se obtiene $in - out \leq 0$. A partir de dichos predicados llegamos primero a $0 \leq in - out \leq 0$, luego a $in = out = n$, y finalmente a $j = out + 1 = n + 1 \leq n$ (falso).
- El último caso posible de *deadlock*, representado por la configuración C_3, corresponde a cuando el productor está bloqueado en su *await* y el consumidor termina. También la imagen semántica resulta falsa, lo que se prueba razonando como en el caso anterior.

Por lo tanto, se cumple la ausencia de *deadlock*. Notar que las *proof outlines* utilizadas son distintas de las que consideramos para la prueba de no divergencia del programa.

3. Lenguaje de recursos de variables

Por medio del lenguaje de variables compartidas, utilizado previamente, con una instrucción de sincronización muy elemental, el *await*, con la que S. Owicki y D. Gries publican originalmente su axiomática, aprovechamos a presentar aspectos salientes de la verificación de programas concurrentes en general. Entre otros: *proof outlines* de correctitud parcial para probar todas las propiedades; pruebas en dos etapas como acercamiento a un esquema composicional, con chequeo de consistencia de las *proof outlines*; variables auxiliares; y configuraciones expresadas en términos de puntos de control de los procesos.

Con esta base introducimos en esta sección una segunda clase de programas paralelos, con una instrucción de sincronización más estructurada, sin la permisividad irrestricta sobre referencias y actualizaciones a las variables compartidas observada antes, lo que facilita la construcción y verificación de programas. La nueva instrucción fuerza la exclusión mutua entre las secciones críticas especificadas. Además, las variables compartidas se disponen en conjuntos denominados *recursos*, sobre los que los procesos obtienen acceso exclusivo. Un programa de este segundo lenguaje (*lenguaje de recursos de variables*) tiene la siguiente forma:

$$S :: \text{resource } r_1 \text{ (lista-var}_1\text{)} ; \ldots ; \text{resource } r_m \text{ (lista-var}_m\text{)}; S_0 ; [S_1 \parallel \ldots \parallel S_n]$$

Los componentes $r_1, \ldots, r_m$ son recursos de variables, conjuntos disjuntos de variables definidos mediante las listas *lista-var*$_1$, …, *lista-var*$_m$. El subprograma S_0 tiene las mismas características que

en el lenguaje anterior. Los procesos S_1, …, S_n también, salvo que cuentan con una instrucción de sincronización distinta, el *with*, que provee acceso exclusivo a los recursos. La sintaxis del *with*, también conocido como *sección crítica condicional*, es la siguiente:

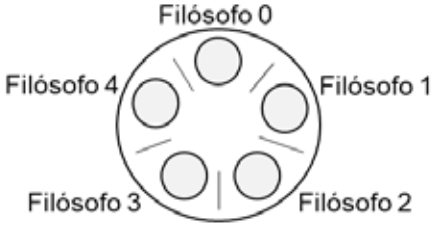
with r_j when B do S endwith

r_j es un recurso, B una expresión booleana, y S un subprograma sin *with* y que siempre termina (tiene las mismas restricciones del *await* para simplificar la presentación). Toda variable de un recurso referida en un proceso debe estar dentro de un *with* del proceso, y toda variable compartida modificable debe estar definida en un recurso. Cuando un proceso S_i ejecuta una instrucción *with* r_j *when B do S endwith*, si r_j está libre (es decir, si no está ocupado por otro proceso), y B es verdadera, entonces S_i puede ocupar el recurso, utilizar sus variables y progresar en su ejecución. Ningún proceso tiene acceso a un recurso ocupado por otro. La liberación de un recurso se produce cuando el proceso que lo ocupa termina de ejecutar el *with* correspondiente. Si dos o más procesos compiten por un recurso, su ocupación se decide no determinísticamente, no hay ninguna hipótesis de *fairness* al respecto.

Otra instrucción de sincronización estructurada es el *monitor*, que permite liberaciones temporarias de los recursos [Bri73, Hoa74]. En [NB85] se describe una axiomática que lo contempla.

Como en la clase de programas anterior, el uso indebido del *with* puede causar *deadlock*, y el *while* sigue siendo la única fuente posible de divergencia. Notar que ahora las secciones críticas se pueden identificar sintácticamente, lo que hace que las pruebas de correctitud resulten más estructuradas.

Como ejemplo de programa, presentamos a continuación un esquema muy simple que modeliza el clásico problema de la cena de los filósofos, propuesto por E. Dijkstra para plantear una típica situación de sincronización en los sistemas operativos. El problema considera cinco filósofos sentados alrededor de una mesa, en la que hay cinco platos con espagueti y cinco tenedores, dispuestos como lo muestra la siguiente figura:



Los filósofos alternativamente comen o piensan. Para comer, un filósofo debe utilizar dos tenedores, pero sólo el de su derecha y el de su izquierda, por lo que dos filósofos vecinos no pueden comer al mismo tiempo. El esquema de implementación propuesto es el siguiente:

S_{fcp} :: resource forks(f[0:4]) ; f[0]:= 2 ; f[1] := 2 ; f[2] := 2 ; f[3] := 2 ; f[4] := 2 ;
[$||_{i = 0,4}$ F_i]

con:

F_i :: while true do
with forks when f[i] = 2 do f[i – 1] := f[i – 1] – 1 ; f[i + 1] := f[i + 1] – 1 endwith ;
… comer …
with forks when true do f[i – 1] := f[i – 1] + 1 ; f[i + 1] := f[i + 1] + 1 endwith ;
… pensar …
od

El proceso F_i representa al filósofo i, y el elemento f[i] del arreglo compartido f (por eso está en un recurso, el recurso *forks*) representa la cantidad de tenedores a los que el filosofo tiene acceso en todo momento (la aritmética sobre los índices de f es módulo 5). Las secciones correspondientes a cuando el filósofo come o piensa no modifican f. El primer *with* representa la espera del filósofo para conseguir los dos tenedores que le permitan comer, y si lo logra, la inhabilitación para que los tomen sus vecinos. Y el segundo *with* implementa la situación en la que el filósofo, habiendo terminado de comer y disponiéndose a pensar, libera los tenedores para permitir que los utilicen sus vecinos.

3.1. Axiomática para las pruebas de correctitud parcial

Las únicas dos reglas de prueba de correctitud parcial que cambian, con respecto a la axiomática para los programas del lenguaje de variables compartidas, son las relacionadas con la instrucción de sincronización y la composición paralela. Las nuevas reglas, en conjunto, determinan pruebas más simples y estructuradas que las anteriores, explotando la característica del lenguaje de programación, que fuerza a agrupar las variables compartidas modificables, incluidas en recursos, dentro de los *with*. La idea central de esta variante axiomática es definir por cada recurso r un invariante I_r, expresado en términos de las variables del recurso, que debe valer toda vez que el recurso está libre, es decir, en el momento en que un proceso lo ocupa y en el momento en que es liberado. Entre ambas instancias el invariante no tiene por qué valer (los estados intermedios correspondientes a la ejecución del proceso que utiliza el recurso son invisibles para el resto de los procesos).

La regla correspondiente a la instrucción de sincronización *with* (WITH) se formula de la siguiente manera (de ahora en más asumimos un solo recurso, para simplificar la presentación):

6. Regla del with (WITH)

$$\frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with r when B do S endwith}\ \{q\}}$$

tal que p y q no tienen variables compartidas modificables e I_r se define con las variables de r. Como con el *await*, la regla WITH tiene por la forma de la instrucción la misma estructura que la regla del condicional (COND). En este caso, la regla establece que pasar de la precondición p a la postcondición q mediante la ejecución del *with* es posible siempre que el subprograma S, a partir de $p \wedge B$, preserve el invariante I_r. Por lo dicho antes, I_r no se traslada a la conclusión de la regla. El uso de WITH se aclara formulando la segunda regla novedosa de la axiomática, correspondiente a la composición paralela:

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

7. Regla de la composición paralela con recursos (RPAR)

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i = 1, \ldots, n,\ \text{son } \textit{proof outlines} \text{ que utilizan } I_r \qquad p \longrightarrow (I_r \wedge \bigwedge_{i=1,n} p_i),\ (I_r \wedge \bigwedge_{i=1,n} q_i) \longrightarrow q}{\{p\}\ [\ ||_{i=1,n}\ S_i\ ]\ \{q\}}$$

tal que los predicados de las *proof outlines* no incluyen variables compartidas modificables, las cuales figuran exclusivamente en el invariante I_r. Las *proof outlines* obtenidas en la primera etapa de pruebas locales utilizan el invariante del recurso a través de la aplicación de la regla WITH. Por cómo se conforman las *proof outlines*, la regla RPAR no requiere que se efectúe el chequeo de libertad de interferencia de las mismas en la etapa de prueba global. La conclusión de la regla se justifica por lo siguiente:

- La información de las variables del recurso r (variables compartidas modificables) se propaga desde la precondición hasta la postcondición de la composición paralela a través del invariante I_r. Al comienzo y al final I_r vale porque en ambas instancias el recurso está libre.
- La información del resto de las variables (variables no compartidas o compartidas de solo lectura) se propaga desde la precondición hasta la postcondición de la composición paralela a través de los predicados de las *proof outlines*.

Al igual que en los programas del lenguaje de variables compartidas, se sigue necesitando la regla AUX.

Como ejemplo de aplicación, probamos en lo que sigue la correctitud parcial de un esquema de programa de recursos de variables que implementa un *semáforo binario*, otro mecanismo de sincronización. El semáforo es un tipo de datos, creado por E. Dijkstra, con variables enteras y tres operaciones, que definidas en términos de la instrucción *await* tienen la siguiente forma:

- Instrucción I(s): s := k, con k ≥ 0.
- Instrucción P(s): await s > 0 → s := s – 1 end.
- Instrucción V(s): s := s + 1.

Su efecto es similar al del *await* (el *await* es más flexible y estructurado, pero como contrapartida menos eficiente, porque cuando un proceso lo ejecuta los otros procesos quedan suspendidos). Un semáforo es binario si sus variables pueden valer sólo 0 o 1.

Ejemplo 7. Prueba de correctitud parcial de un esquema de programa que implementa un semáforo binario

Vamos a probar:

```
resource sem(s) ; s = 1 ;
{s = 1}
[with sem when s = 1 do s := 0 endwith ;          with sem when s = 1 do s := 0 endwith ;
… sección 1 …                              ||     … sección 2 …
with sem when true do s := 1 endwith              with sem when true do s := 1 endwith]
{s = 1}
```

La sección i entre los dos *with* del proceso i no modifica el valor de la variable s. Para la prueba tenemos que incluir variables auxiliares, porque de lo contrario estamos forzados a utilizar solamente predicados *true* en las *proof outlines* y el predicado $I_{sem} = 0 \leq s \leq 1$ como invariante del semáforo, que no conducen a la postcondición s = 1. Agregamos una variable *a* inicializada en 0 en el primer proceso, para indicar con el valor 1 o 0, respectivamente, si el proceso está ejecutando o no su sección entre los *with*, y lo mismo hacemos en el segundo proceso con una variable b. Proponemos las siguientes *proof outlines*:

```
{a = 0}                                    {b = 0}
[with semáforo when s = 1 do               with semáforo when s = 1 do
 s := 0 ; a := 1 endwith ;   {a = 1}        s := 0 ; b := 1 endwith ;   {b = 1}
 … sección 1 …               {a = 1}   ||   … sección 2 …               {b = 1}
 with semáforo when true do                 with semáforo when true do
  s := 1 ; a := 0 endwith                    s := 1 ; b := 0 endwith]
 {a = 0}                                    {b = 0}
```

utilizando como invariante del semáforo:

$$I_{sem} = 0 \leq s \leq 1 \wedge ((a = 0 \wedge b = 0) \vee (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \wedge$$
$$((a = 0 \wedge b = 0) \rightarrow s = 1) \wedge (((a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \rightarrow s = 0)$$

Al comienzo vale el invariante I_{sem}, implicado por la precondición $s = 1 \wedge a = 0 \wedge b = 0$.

La *proof outline* del primer proceso se obtiene de la siguiente manera. Aplicando la regla WITH, como se cumple:

$$\{I_{sem} \wedge a = 0 \wedge s = 1\}\ s := 0 ; a := 1\ \{I_{sem} \wedge a = 1\}$$

se deriva:

$$\{a = 0\}\ \text{with semaforo when } s = 1 \text{ do } s := 0 ; a := 1 \text{ endwith } \{a = 1\}$$

en lo que hace al primer *with*, y como se cumple:

$$\{I_{sem} \wedge a = 1\}\ s := 1 ; a := 0\ \{I_{sem} \wedge a = 0\}$$

se deriva:

$$\{a = 1\}\ \text{with semaforo when true do } s := 1 ; a := 0 \text{ endwith } \{a = 0\}$$

en lo que hace al segundo *with*. La *proof outline* del segundo proceso se obtiene de manera similar, utilizando la variable b.

Finalmente, a la terminación del programa vale $I_{sem} \wedge a = 0 \wedge b = 0$, que implica la postcondición s = 1, válida también para el programa original, sin las variables *a* y b, por la aplicación de la regla AUX.

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

3.2. Axiomática para las pruebas de otras propiedades

Las reglas para probar no divergencia y ausencia de *deadlock* en los programas paralelos del lenguaje de recursos de variables tienen las mismas características que las descriptas para los programas paralelos del lenguaje de variables compartidas:

8. Regla de la no divergencia paralela con recursos (RPAR*)
La única diferencia de esta regla con la regla PAR* utilizada para los programas anteriores es que no requiere chequear la libertad de interferencia de las *proof outlines* de correctitud parcial, elaboradas como base para la prueba de no divergencia.

9. Regla de la ausencia de deadlock con recursos (RDEADLOCK)
En este caso la regla tiene dos diferencias con respecto a la regla DEADLOCK utilizada para los programas del lenguaje de variables compartidas. La primera diferencia es la misma que mencionamos antes, no requiere chequear la libertad de interferencia de las *proof outlines* de correctitud parcial elaboradas en las pruebas locales. La segunda diferencia consiste en que ahora las imágenes semánticas de las configuraciones correspondientes a los posibles casos de *deadlock*, definidas en términos de puntos de control asociados a instrucciones *with*, incluyen el invariante definido para el recurso utilizado, porque en una situación de *deadlock* el recurso no está ocupado.

Otras dos propiedades comúnmente consideradas en el paradigma concurrente son la *exclusión mutua* y la *ausencia de inanición*, pertenecientes, respectivamente, a las familias de propiedades *safety* y *liveness*. No las hemos tratado en los programas paralelos anteriores, así que aprovechamos esta sección para analizarlas al menos resumidamente. Sus pruebas tienen sentido sobre todo en programas que no terminan, usualmente denominados *reactivos*, cuya correctitud se evalúa en términos de su comportamiento continuo.

El lenguaje de recursos de variables garantiza la exclusión mutua entre las secciones críticas condicionales, por lo que la verificación de la propiedad tiene sentido únicamente para secciones de programa de mayor nivel de abstracción, sobre las que las interferencias indeseables sólo pueden ser evitadas algorítmicamente. La manera de probar la exclusión mutua es similar a cómo se prueba la ausencia de *deadlock*: a partir de *proof outlines* de correctitud parcial, con postcondiciones cualesquiera, hay que identificar sintácticamente todas las situaciones posibles de interferencia, y comprobar que las imágenes semánticas asociadas son falsas. La prueba es más laboriosa que la de ausencia de *deadlock*, porque ahora no sólo se deben considerar los *with* sino también todos los puntos de control internos de las secciones definidas.

Ejemplificamos a continuación la aplicación de la regla sobre el programa que implementa el problema de la cena de los filósofos.

Ejemplo 8. Prueba de exclusión mutua en el programa que implementa el problema de la cena de los filósofos

Vamos a probar que en la implementación elegida nunca se produce la situación en la que dos filósofos vecinos comen al mismo tiempo. Para ello, agregamos al programa un arreglo auxiliar *e*, de igual tamaño que el arreglo compartido f incluido en el recurso *forks*. El elemento e[i] se utiliza para indicar la residencia o no en la sección *comer* del proceso F_i que representa al filósofo i-ésimo

(valor 1 o 0, respectivamente). El arreglo *e* se inicializa con todos sus elementos en 0 (recordar que todos los f[i] se inicializan con 2). De esta manera, la precondición de las *proof outlines* es:

$$p = (I_{forks} \wedge \bigwedge_{i=0,4} e[i] = 0)$$

Como invariante del recurso *forks* proponemos el siguiente predicado:

$$I_{forks} = (\bigwedge_{i=0,4} ((0 \leq e[i] \leq 1) \wedge (e[i] = 1 \longrightarrow f[i] = 2) \wedge (f[i] = 2 - e[i - 1] - e[i + 1])))$$

y como *proof outline* de cada proceso F_i:

{e[i] = 0}
while true do
{e[i] = 0}
with forks when f[i] = 2 do
{e[i] = 0}
f[i – 1] := f[i – 1] – 1 ; f[i + 1] := f[i + 1] – 1 ; e[i] := 1 endwith ;
{e[i] = 1}
… comer …
{e[i] = 1}
with forks when true do
{e[i] = 1}
f[i – 1] := f[i – 1] + 1 ; f[i + 1] := f[i + 1] + 1 ; e[i] := 0 endwith ;
{e[i] = 0}
… pensar …
{e[i] = 0}
od
{e[i] = 0}

El cumplimiento del invariante I_{forks} al comienzo de la composición paralela se prueba fácilmente. Tampoco reviste dificultad probar la correctitud de las *proof outlines*, con la aplicación de la regla WITH sobre los dos *with* de cada proceso. Veamos cómo con las *proof outlines* planteadas se verifica la exclusión mutua de dos procesos F_i y F_{i+1} con respecto a sus secciones *comer*. Supongamos por el contrario que no se cumple la propiedad, para llegar a una contradicción:

- De acuerdo a las *proof outlines*, vale: $e[i] = 1 \wedge e[i + 1] = 1$.
- También vale el invariante I_{forks}, lo que se justifica de la siguiente manera. Por un lado, cuando el programa reside en las secciones *comer* de F_i y F_{i+1}, ninguno de los dos procesos está ocupando el recurso *forks*. Pero por otro lado, el recurso sí podría estar ocupado por un tercer proceso (representante de un filósofo no vecino a los filósofos i e i + 1). De todos modos, se puede probar que si existe una computación con estas características, entonces también existe otra, semánticamente equivalente, sin que el tercer proceso resida en una sección crítica condicional (intuitivamente, se obtiene modificando la secuencia de intercalación de las acciones atómicas).
- A partir de lo anterior, se obtiene la imagen semántica $e[i] = 1 \wedge e[i + 1] = 1 \wedge I_{forks}$, que implica el predicado falso $f[i] = 2 \wedge f[i] < 2$, lo que completa la prueba.

La prueba de ausencia de inanición de un proceso con respecto a un evento determinado (que en los programas paralelos descriptos sería el acceso a una sección crítica condicional), como la prueba de no divergencia, y en general de toda propiedad *liveness*, se basa en la utilización de una función variante definida en un conjunto bien fundado. Y propio de las propiedades *liveness*, en general la ausencia de inanición no puede garantizarse si no se incorpora alguna asunción de *fairness* en la semántica del lenguaje de programación. Por ejemplo, sin *fairness* en el programa que implementa el problema de la cena de los filósofos puede producirse la situación en la que un filósofo nunca accede a sus dos tenedores, por su apropiación permanente por parte de los filósofos vecinos. Tal caso de inanición es imposible con una hipótesis de *fairness* que asegure que no pueden haber computaciones con infinitas ocurrencias de estados con f[i] = 2 sin que el proceso F_i acceda a su sección *comer*.

4. Ejemplo de desarrollo sistemático de programa

Como único ejemplo de desarrollo sistemático de programa de este artículo, consideramos a continuación un programa paralelo muy sencillo, con dos procesos y sin instrucciones de sincronización. A pesar de su simplicidad, el ejemplo sirve para destacar cómo se mantienen las ideas fundamentales descriptas anteriormente. En efecto, el esquema de desarrollo de los procesos es básicamente el que planteamos para los programas secuenciales determinísticos, con la salvedad, por la no composicionalidad del método axiomático, de que en la construcción de un proceso se deben tener en cuenta las interferencias del otro, y así la prueba de correctitud del programa obtenido debe incluir, además de las *proof outlines* de los procesos, el chequeo de libertad de interferencia de las mismas (la alternativa composicional que mencionamos previamente pero no describimos, es derivar los programas a partir de especificaciones que incluyan la información de las interferencias entre sus procesos).

Ejemplo 9. Desarrollo sistemático de un programa que busca el primer elemento positivo de un arreglo de números enteros

Construiremos un programa que busca el primer elemento positivo de un arreglo a[1:N] de números enteros, de solo lectura, con $N \geq 1$. Su especificación es:

$$\langle \text{true} \rangle \; S_{pep} \; \langle 1 \leq k \leq N + 1 \wedge \forall m: (1 \leq m < k \longrightarrow a[m] \leq 0) \wedge k \leq N \longrightarrow a[k] > 0 \rangle$$

que expresa que el programa debe obtener el menor índice k del arreglo *a* con a[k] > 0, si existe, o el valor k = N + 1 en caso contrario. Como esquema general de solución proponemos un programa con dos procesos, S_1 y S_2, uno con un *while* que recorra los elementos del arreglo con índice impar y el otro con un *while* que recorra los elementos con índice par, hasta que alguno de los procesos encuentre un elemento positivo o hasta que ámbos alcancen el final del arreglo. La forma del programa a construir sería entonces:

S_{pep} :: i := 1 ; j := 2 ; k_1 := N + 1 ; k_2 := N + 1 ;
[S_1 || S_2] ;
k := min(k_1, k_2)

i y j son los índices manejados por los procesos S_1 y S_2 para recorrer el arreglo. k_1 y k_2 son las variables de S_1 y S_2 que se utilizan para guardar el índice requerido si se encuentra. Deben ser

variables compartidas para que un proceso termine ni bien el otro tenga eventualmente éxito en la búsqueda. El resultado del programa es el contenido menor de las dos variables, que denotamos con la función $\min(k_1, k_2)$. De acuerdo a esta idea de solución, especificamos el proceso S_1 del siguiente modo:

$$\langle N \geq 1 \wedge i = 1 \wedge k_1 = N + 1 \rangle \; S_1 \; \langle q_1 \rangle$$

con:

$$q_1 = 1 \leq k_1 \leq N + 1 \wedge (\forall m: \text{impar}(m) \wedge 1 \leq m < k_1 \longrightarrow a[m] \leq 0) \wedge (k_1 \leq N \longrightarrow a[k_1] > 0)$$

donde impar(m) es verdadero sii m es impar. Simétricamente, la especificación del proceso S_2 es:

$$\langle N \geq 1 \wedge j = 2 \wedge k_2 = N + 1 \rangle \; S_2 \; \langle q_2 \rangle$$

con:

$$q_2 = 2 \leq k_2 \leq N + 1 \wedge (\forall m: \text{par}(m) \wedge 1 \leq m < k_2 \longrightarrow a[m] \leq 0) \wedge (k_2 \leq N \longrightarrow a[k_2] > 0)$$

siendo par(m) verdadero sii m es par. Luego de las inicializaciones del programa se cumple la conjunción de las precondiciones de S_1 y S_2, y a partir de la conjunción de las postcondiciones de los mismos, con la asignación $k := \min(k_1, k_2)$ se obtiene la postcondición del programa.

El paso siguiente es definir los invariantes y variantes de los *while* de los procesos. Para el primer *while*, generalizando la postcondición de S_1 con el índice i llegamos al invariante:

$$\begin{aligned} p_1 = {} & 1 \leq k_1 \leq N + 1 \wedge \text{impar}(i) \wedge 1 \leq i \leq k_1 + 1 \wedge \\ & (\forall m: \text{impar}(m) \wedge 1 \leq m < i \longrightarrow a[m] \leq 0) \wedge \\ & k_1 \leq N \longrightarrow a[k_1] > 0 \end{aligned}$$

La cota superior $k_1 + 1$ del índice i determina la forma del *while*: primero se evalúa el elemento corriente a[i] y luego se avanza al siguiente elemento con índice impar. Como variante del *while* proponemos:

$$t_1 = k_1 + 1 - i$$

que a partir de N + 1 se decrementa después de cada iteración, y cumple $p_1 \longrightarrow t_1 \geq 0$. Análogamente, para el segundo *while* definimos el invariante:

$$\begin{aligned} p_2 = {} & 2 \leq k_2 \leq N + 1 \wedge \text{par}(j) \wedge 2 \leq j \leq k_2 + 1 \wedge \\ & (\forall m: \text{par}(m) \wedge 1 \leq m < j \longrightarrow a[m] \leq 0) \wedge \\ & k_2 \leq N \longrightarrow a[k_2] > 0 \end{aligned}$$

y el variante:

$$t_2 = k_2 + 1 - j$$

Revista Abierta de Informática Aplicada/vol 7 Nº1 (2023): 51-77
ISSN 2591-5320

Con las definiciones anteriores derivamos la siguiente composición paralela del programa. Presentamos directamente las *proof outlines* de los procesos:

$\langle \text{inv: } p_1, \text{ var: } t_1 \rangle$		$\langle \text{inv: } p_2, \text{ var: } t_2 \rangle$
$[S_1 :: \text{while } i < \min(k_1, k_2) \text{ do } \langle p_1 \wedge i < k_1 \rangle$		$S_2 :: \text{while } j < \min(k_1, k_2) \text{ do } \langle p_2 \wedge j < k_2 \rangle$
$\text{if } a[i] > 0 \text{ then } \langle p_1 \wedge i < k_1 \wedge a[i] > 0 \rangle$		$\text{if } a[j] > 0 \text{ then } \langle p_2 \wedge j < k_2 \wedge a[j] > 0 \rangle$
$k_1 := i$	$\|\|$	$k_2 := j$
$\text{else } \langle p_1 \wedge i < k_1 \wedge a[i] \leq 0 \rangle$		$\text{else } \langle p_2 \wedge j < k_2 \wedge a[j] \leq 0 \rangle$
$i := i + 2 \text{ fi od}$		$j := j + 2 \text{ fi od}]$
$\langle p_1 \wedge i \geq \min(k_1, k_2) \rangle$		$\langle p_2 \wedge j \geq \min(k_1, k_2) \rangle$

Se verifica fácilmente la correctitud de las *proof outlines*. También que la conjunción de las precondiciones de S_1 y S_2 implica $p_1 \wedge p_2$, y que la conjunción de las postcondiciones de dichos procesos es implicada por $p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j \geq \min(k_1, k_2)$.

Para concluir la prueba de correctitud parcial resta chequear que las *proof outlines* son libres de interferencia. Todas las validaciones correspondientes son triviales, salvo las que consideran el predicado $p_1 \wedge i \geq \min(k_1, k_2)$ con respecto a la asignación $k_2 := j$, y el predicado $p_2 \wedge j \geq \min(k_1, k_2)$ con respecto a la asignación $k_1 := i$. La primera validación requiere el cumplimiento de la fórmula:

$$\{p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j < k_2 \wedge a[j] > 0\}\ k_2 := j\ \{p_1 \wedge i \geq \min(k_1, k_2)\}$$

la cual se verifica porque de la precondición de la asignación derivamos:

$$p_1 \wedge i \geq \min(k_1, j)$$

de la siguiente manera:

$$\begin{aligned} p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j < k_2 \wedge a[i] > 0 &\longrightarrow \\ p_1 \wedge i \geq \min(k_1, k_2) \wedge j < k_2 &\longrightarrow \\ p_1 \wedge i \geq \min(k_1, j) & \end{aligned}$$

y entonces, por el axioma ASI y la regla CONS llegamos a:

$$\{p_1 \wedge i \geq \min(k_1, j)\}\ k_2 := j\ \{p_1 \wedge i \geq \min(k_1, k_2)\}$$

Del mismo modo verificamos el cumplimiento de la fórmula:

$$\{p_2 \wedge j \geq \min(k_1, k_2) \wedge p_1 \wedge i < k_1 \wedge a[i] > 0\}\ k_1 := i\ \{p_2 \wedge j \geq \min(k_1, k_2)\}$$

Por último, la no divergencia del programa se justifica porque las asignaciones de un proceso no alteran el valor del variante del *while* del otro proceso, al tiempo que cada variante decrece a lo largo de las secuencias sintácticamente posibles asociadas. No hay posibilidad de *deadlock* porque el programa no tiene instrucciones de sincronización.

5. Observaciones finales

Las axiomáticas descriptas en este artículo, como así también el enfoque para desarrollar software sistemáticamente en base a las mismas, por un lado sostienen en líneas generales los mismos principios estudiados previamente. Por otro lado, cabe observarles una diferencia relevante con respecto a lo analizado en el paradigma secuencial, que es la no composicionalidad, producto del tipo de especificación que manejan. Así y todo, constituyen una muy buena aproximación composicional, por medio de pruebas guiadas por la sintaxis de los programas, estructuradas en dos etapas, una primera etapa de elaboración de *proof outlines* de procesos, obtenidas composicionalmente, y una segunda etapa de validación de las mismas, con un criterio y un costo que dependen de la propiedad considerada y el lenguaje de programación utilizado.

Existen varias propuestas axiomáticas composicionales. La idea principal es incluir en la especificación la información de las interferencias entre los procesos y el entorno de ejecución. A cada proceso, además de su pre y postcondición, se le asocia una *condición de fiabilidad*, que establece lo que el proceso espera del entorno, y una *condición de garantía*, que establece cómo el proceso influencia en el entorno. Así, la verificación debe contemplar las relaciones entre dichas condiciones (por ejemplo, debe asegurar que lo que garantiza un proceso implica lo que esperan los otros procesos de él).

Otras características diferenciales del método presentado es que debe recurrir a variables auxiliares para registrar las historias de las computaciones, cuando las variables de programa no resultan suficientes, y en el caso particular de los programas con recursos de variables, a invariantes de alcance global, para tratar la información compartida de los procesos.

En la verificación de un programa paralelo (en realidad de uno concurrente en general) se tratan más propiedades que en un programa secuencial, mínimamente la ausencia de *deadlock*. La prueba de esta propiedad, al igual que otras propiedades *safety*, se basa en asignar a determinadas configuraciones, definidas en términos de puntos de control de procesos, predicados que la contradigan. En lo que hace a las pruebas de propiedades *liveness*, como la no divergencia, se depende marcadamente de la hipótesis de progreso de las computaciones establecida por la semántica de los programas.

Referencias

» [Bar85] Barringer, H. A survey of verification techniques for parallel programs. Lecture Notes in Computer Science, 191, Springer, 1985.

» [Bri73] Brinch-Hansen, P. Operating systems principles. Englewood Cliffs, 1973.

» [Cli73] Clint, M. Program proving: coroutines. Acta Informática, 2, 50-63, 1973.

» [Dij65] Dijkstra, E. Solution of a problem in concurrent programming control. Comm. of the ACM, 8, 9, 569, 1965.

» [Dij68] Dijkstra, E. Cooperating sequential processes. Programming Languages, 43-112, Academic Press, 1968.

» [Hoa 69] Hoare, C. An axiomatic basis for computer programming. Comm. of the ACM, 12, 10, 576-583, 1969.

» [Hoa72] Hoare, C. Towards a theory of parallel programming. Operating Systems Techniques, 61-71, Academic Press, 1972.

» [Hoa74] Hoare. C. Monitors: an operating system structuring concept. Comm. of the ACM, 17, 10, 549-557, 1974.

» [Hoa75] Hoare, C. Parallel programming: an axiomatic approach. Computer Languages, 1, 2, 151-160, 1975.

» [Jon81] Jones, C. Development methods for computer programs including a notion of interference. Technical Monograph PRG-25, Oxford University, 1981.

» [Jon92] Jones, C. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, University of Manchester, 1992.

» [Lam15] Lamport, L. The computer science of concurrency: the early years. Comm. of the ACM, 58, 6, 71-76, 2015.

» [Lam77] Lamport, L. Proving the correctness of multiprocess programs. IEEE Trans. Software Engineering, 3, 2, 125-143, 1977.

» [Mal19] Malkhi, D. Concurrency. The works of Leslie Lamport. ACM Books, 2019.

» [NB85] Nielsen, L. & Black, A. Proving monitor proof rules. Technical Report 85-08-01. Department of Computer Science, University of Washington, Seattle WA, 1985.

» [OG76a] Owicki, S. & Gries, D. An axiomatic proof technique for parallel programs. Acta Informatica, 6, 319-340, 1976.

» [OG76b] Owicki, S. & Gries, D. Verifying properties of parallel programs: an axiomatic approach. Comm. of the ACM, 19, 279-285, 1976.

» [Ros22a] Rosenfeld, R. Introducción a la verificación de programas. Revista Abierta de Informática Aplicada, 6, 1, 79-100, 2022.

» [Ros22b] Rosenfeld, R. Verificación de programas no determinísticos. Revista Abierta de Informática Aplicada, 6, 2, 54-79, 2022.

» [SA86] Schneider, F & Andrews, G. Concepts for concurrent programming. Current Trends in Concurrency, Lecture Notes in Computer Science, 224, Springer, 1986.

» [Sch18] Schneider, F. History and context for defining liveness. ACM SIGACT News, 49, 4, 60-63, 2018.