Verificación de Programas Distribuidos

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

Resumen

Este artículo completa nuestra serie de cuatro artículos sobre la verificación axiomática de programas, que planteamos en el marco del proyecto del CAETI para construir un ambiente de soporte al desarrollo de software. En particular, concluimos el análisis de los programas concurrentes iniciado en la publicación anterior, considerando ahora la familia de los programas distribuidos, caracterizados por contar con procesos con variables disjuntas y que se comunican mediante mensajes. Como siempre, destacamos el principio de utilizar las axiomáticas como guías para la obtención de programas correctos por construcción, y la observación de que las nociones fundamentales de predicado invariante y función variante constituyen la base metodológica en todos los paradigmas de programación.

PALABRAS CLAVE: PROGRAMA DISTRIBUIDO, VERIFICACIÓN, AXIOMÁTICA.

Verification of Distributed Programs

Abstract

This article completes our series of four articles on the axiomatic verification of programas, which we propose within the framework of the CAETI project to build an environment to support software development. In particular, we conclude the analysis of concurrent programs begun in the previous publication, now considering the family of distributed programs, characterized by having processes with disjoint variables and that communicate through messages. As always, we highlight the principle of using axiomatics as guides for obtaining correct programs by construction, and the observation that the fundamental notions of invariant predicate and variant function constitute the methodological basis in all programming paradigms.

KEYWORDS: DISTRIBUTED PROGRAM, VERIFICATION, AXIOMATICS.

Recibido: septiembre 2023./ Aceptado: noviembre 2023 / Publicado: diciembre 2023

1. Introducción

Con este artículo completamos nuestra serie de cuatro artículos sobre la *verificación axiomática de programas*, desarrollada en el marco del proyecto del CAETI para la construcción de un ambiente que asiste en el desarrollo de software. En particular, completamos el análisis de los *programas concurrentes*. En la publicación anterior consideramos la familia de los *programas paralelos*, así que en este último trabajo nos enfocamos en la verificación axiomática de los *programas distribuidos*, con procesos con *variables disjuntas* y que se comunican mediante *mensajes* Las axiomáticas que presentamos se basan en los trabajos de K. Apt, N. Francez y W. de Roever de 1980 [AFdR80], adaptación, también *no composicional*, de las axiomáticas para programas paralelos de S. Owicki y D. Gries [OG76a, OG76b].

Primero introducimos los programas con los que trabajamos y las propiedades a verificar, luego presentamos las axiomáticas correspondientes junto a ejemplos de aplicación, y finalmente concluimos con un par de notas adicionales (una de ellas sobre alternativas axiomáticas composicionales para la concurrencia, en contraste con las que venimos describiendo) y observaciones finales. Recomendamos al lector leer o releer los artículos anteriores, [Ros22a, Ros22b, Ros23], especialmente el último.

2. Lenguaje de pasajes de mensajes

El lenguaje de programación que utilizamos es una extensión concurrente del lenguaje de programación secuencial no determinístico considerado en el segundo artículo de la serie [Ros22b]. Se basa en el lenguaje CSP o *Communicating Sequential Processes* (Comunicación de Procesos Secuenciales o Procesos Secuenciales Comunicantes), creado por C. Hoare [Hoa78]. Los procesos no comparten variables, y se comunican y sincronizan por medio de *pasajes de mensajes*, generados por instrucciones de *entrada/salida*, también conocidas como instrucciones de *comunicación*.

Consideramos comunicaciones *sincrónicas*, esquema según el cual el envío de un mensaje por parte de un proceso y la recepción del mensaje por parte de otro se ejecutan atómicamente (un proceso puede quedar bloqueado a la espera de la comunicación con el otro, lo que puede producir una situación de *deadlock*). En una de las notas adicionales hacemos una breve referencia al modelo *asincrónico*.

Los programas tienen la siguiente forma:

$$S :: [P_1 :: S_1 \,||\, \ldots \,||\, P_n :: S_n]$$

que se abrevia con $[\,||_{i=1,n} P_i\,]$. Es decir, los programas son directamente composiciones distribuidas de procesos. Los procesos siempre se etiquetan, para identificarlos en los pasajes de mensajes. No hay anidamiento de concurrencia ni procesos dinámicos. Además del *skip*, la asignación, la secuencia y la composición concurrente, en este caso distribuida, que conocemos de los lenguajes utilizados previamente, el repertorio de instrucciones del lenguaje incluye otras cuatro instrucciones, que pasamos a describir. Comenzamos con las instrucciones de entrada/salida:

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

- Instrucción de *entrada*. Su sintaxis es:

$$P_i ? x$$

En un proceso P_j, con $j \neq i$, que tiene una variable local x, la instrucción efectúa un pedido al proceso P_i para que asigne un valor a x.

- Instrucción de *salida*. Su sintaxis es:

$$P_j ! e$$

En un proceso P_i, con $i \neq j$, la instrucción efectúa un pedido al proceso P_j para que reciba el valor de la expresión *e*, definida con variables locales de P_i.

$P_i ? x$ (en P_j) y $P_j ! e$ (en P_i) progresan simultáneamente. Un proceso debe esperar al otro para que se produzca la comunicación entre ellos, cuyo efecto es la asignación del valor de *e* a x. También se pueden comunicar tuplas de valores.

Se define que $P_i ? x$ (en P_j) y $P_j ! e$ (en P_i) *coinciden sintácticamente*, si además de la coincidencia de los índices coinciden los tipos de datos de x y *e*. Se utiliza el símbolo α para denotar una instrucción de entrada/salida, y (α, α´) para denotar instrucciones de entrada/salida que coinciden sintácticamente. La comunicación (α, α´) está *habilitada* cuando α y α´ son las siguientes instrucciones a ser ejecutadas por los procesos respectivos y los mismos están *preparados* o *listos* para ejecutarlas.

Las instrucciones de entrada/salida pueden aparecer aisladamente en un proceso o bien ser parte de *selecciones condicionales y repeticiones con comunicaciones*, las dos instrucciones del lenguaje que nos quedan por describir:

- *Selección condicional con comunicaciones*. Su sintaxis más general es:

$$\text{if } B_1 ; \alpha_1 \rightarrow S_1 \text{ or } \ldots \text{ or } B_n ; \alpha_n \rightarrow S_n \text{ fi}$$

siendo $B_1, \ldots, B_n$ expresiones booleanas y $\alpha_1, \ldots, \alpha_n$ instrucciones de entrada/salida, estas últimas opcionales. Es decir, la selección tiene la misma forma que la del lenguaje secuencial de comandos guardados que vimos antes, pero ahora una guardia puede incluir una instrucción de entrada/salida, en cuyo caso un comando guardado $B_i ; \alpha_i \rightarrow S_i$ puede ejecutarse (o lo que es lo mismo, la *dirección* i está *habilitada*) si B_i es verdadera y α_i es parte de una comunicación habilitada. La evaluación de las guardias es atómica. Por lo tanto, una selección condicional con comunicaciones se comporta de la siguiente manera: si tiene direcciones habilitadas, puede progresar no determinísticamente por alguna de ellas y pasar a la siguiente instrucción si el componente S_i respectivo termina; y si no tiene direcciones habilitadas, si es porque todas las B_i son falsas la selección falla, y si no, queda bloqueada, temporaria o definitivamente, de acuerdo a lo que suceda en el programa más adelante.

- *Repetición con comunicaciones*. Su sintaxis más general es:

$$\text{do } B_1 ; \alpha_1 \rightarrow S_1 \text{ or } \ldots \text{ or } B_n ; \alpha_n \rightarrow S_n \text{ od}$$

  Valen las mismas definiciones del item anterior, y mantenemos la restricción del lenguaje secuencial no determinístico de comandos guardados, para simplificar la presentación, de no permitir repeticiones anidadas. Por lo tanto, una repetición con comunicaciones se comporta de la siguiente manera: si tiene direcciones habilitadas, puede progresar no determinísticamente por alguna de ellas y repetir el mismo ciclo mientras existan; si en algún momento ninguna dirección está habilitada, si es porque todas las B_i son falsas la repetición termina, y si no, queda bloqueada, temporaria o definitivamente, según lo que ocurra después en el programa; y si siempre existe una dirección habilitada, la repetición diverge.

Al igual que los programas paralelos, los programas distribuidos terminan, fallan (por causa de una selección condicional sin ninguna B_i verdadera o por *deadlock*) o divergen. La ejecución de una composición distribuida consiste en el *interleaving* de las instrucciones atómicas de los distintos procesos que la constituyen (instrucciones *skip*, asignaciones y comunicaciones), en base a la *propiedad de progreso fundamental*, que sólo asegura que algún proceso avance si puede (luego nos referimos al *fairness*). La novedad semántica en los programas distribuidos es la ejecución conjunta de una instrucción de entrada y una instrucción de salida, y así el avance en dos procesos simultáneamente. Otro aspecto distintivo relacionado con las instrucciones de comunicación es la manifestación de dos tipos de no determinismo, uno *local*, proveniente del lenguaje secuencial no determinístico de comandos guardados, y el otro *global*, propio de la semántica de *interleaving*, que ejemplificamos a continuación. Los procesos:

$$P_1 :: \text{if } P_3 \text{ ? x} \rightarrow \text{skip or } P_3 \text{ ! 0} \rightarrow \text{skip fi}$$
$$P_2 :: \text{if true} \rightarrow P_3 \text{ ? x or true} \rightarrow P_3 \text{ ! 0 fi}$$

implementan de dos maneras distintas las alternativas de recibir un valor del proceso P_3 o enviarle el valor 0. Que P_1 ejecute P_3 ? x o P_3 ! 0 depende de la situación global del programa. En cambio, en P_2 la elección entre una u otra instrucción de entrada/salida es local. Esta diferencia es irrelevante en términos de la correctitud parcial, pero crucial con respecto a la ausencia de *deadlock*. Por ejemplo, si el proceso P_3 envía el valor 1, mientras que el programa:

$$[P_1 :: \text{if } P_3 \text{ ? x} \rightarrow \text{skip or } P_3 \text{ ! 0} \rightarrow \text{skip fi} \,||\, P_3 :: P_1 \text{ ! 1}]$$

termina con x = 1, en el programa:

$$[P_2 :: \text{if true} \rightarrow P_3 \text{ ? x or true} \rightarrow P_3 \text{ ! 0 fi} \,||\, P_3 :: P_2 \text{ ! 1}]$$

se puede producir una situación de *deadlock* si el proceso P_2 elige la segunda dirección.

Completamos la descripción del lenguaje de pasajes de mensajes con otros dos ejemplos de programas. El primero consiste en la propagación de un valor desde un proceso P_1 hasta un proceso P_n:

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

S_{prop} :: [P_1 || … || P_i || … || P_n], con:
P_1 :: P_2 ! y_1
…..
P_i :: P_{i-1} ? y_i ; P_{i+1} ! y_i
…..
P_n :: P_{n-1} ? y_n

que no requiere mayores explicaciones. El último ejemplo es un programa que obtiene el mínimo de un conjunto de valores:

S_{min} :: [P_1 || … || P_i || … || P_n || Q], con:
P_i :: (mi_min_i, $mi\_tamaño_i$) := (a_i, 1) ;
do $0 < mi\_tamaño_i < n$; P_1 ! (mi_min_i, $mi\_tamaño_i$) → $mi\_tamaño_i := 0$
……
or $0 < mi\_tamaño_i < n$; P_{i-1} ! (mi_min_i, $mi\_tamaño_i$) → $mi\_tamaño_i := 0$
or $0 < mi\_tamaño_i < n$; P_{i+1} ! (mi_min_i, $mi\_tamaño_i$) → $mi\_tamaño_i := 0$
……
or $0 < mi\_tamaño_i < n$; P_n ! (mi_min_i, $mi\_tamaño_i$) → $mi\_tamaño_i := 0$
or $0 < mi\_tamaño_i < n$; P_1 ? (su_min_i, $su\_tamaño_i$) →
(mi_min_i, $mi\_tamaño_i$) := (min(mi_min_i, su_min_i), $mi\_tamaño_i$ + $su\_tamaño_i$)
……
or $0 < mi\_tamaño_i < n$; P_{i-1} ? (su_min_i, $su\_tamaño_i$) →
(mi_min_i, $mi\_tamaño_i$) := (min(mi_min_i, su_min_i), $mi\_tamaño_i$ + $su\_tamaño_i$)
or $0 < mi\_tamaño_i < n$; P_{i+1} ? (su_min_i, $su\_tamaño_i$) →
(mi_min_i, $mi\_tamaño_i$) := (min(mi_min_i, su_min_i), $mi\_tamaño_i$ + $su\_tamaño_i$)
……
or $0 < mi\_tamaño_i < n$; P_n ? (su_min_i, $su\_tamaño_i$) →
(mi_min_i, $mi\_tamaño_i$) := (min(mi_min_i, su_min_i), $mi\_tamaño_i$ + $su\_tamaño_i$)
od ;
if $mi\_tamaño_i = 0$ → skip or $mi\_tamaño_i = n$ → Q ! mi_min_i fi

Q :: if P_1 ? min → skip or … or P_n ? min → skip fi

Para facilitar la escritura usamos asignaciones simultáneas de la forma (x_1, x_2) := (e_1, e_2). Cada proceso P_i del programa S_{min} se inicializa con un valor, su mínimo inicial, y un tamaño, su tamaño inicial, de valor 1. Luego ejecuta una repetición, en la que en cada iteración hace lo siguiente. O bien le envía a algún proceso P_j, con $i \neq j$, su mínimo actual y su tamaño actual y termina su participación en el programa asignándose un 0 a su tamaño. O bien recibe de algún proceso P_j, con $i \neq j$, su mínimo actual y su tamaño actual, por medio de la función min se queda con el mínimo entre el mínimo actual propio y el que recibe, incrementa su tamaño actual con el tamaño que recibe, y continúa participando en el programa a menos que su tamaño actual sea n, lo que implica que obtuvo el mínimo buscado, en cuyo caso se lo envía al proceso Q.

En las secciones que siguen describimos la variante axiomática para verificar programas distribuidos. Por las características del lenguaje de pasajes de mensajes, la variante se basa en el método de verificación que presentamos para los programas secuenciales no determinísticos.

2.1. Axiomática para las pruebas de correctitud parcial

Comenzamos con la descripción de la axiomática para la prueba de correctitud parcial. Como con los programas paralelos, la meta es contar con una regla basada en la siguiente estructura:

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

siendo las $\{p_i\}\ P_i^*\ \{q_i\}$ *proof outlines* de correctitud parcial de los procesos que integran el programa distribuido $[\ ||_{i=1,n} P_i\ ]$, obtenidas en una primera etapa de pruebas locales y chequeadas en una segunda etapa de prueba global. En este caso no hace falta chequear que las *proof outlines* sean libres de interferencia, porque los procesos no comparten variables. Lo que sí se necesita es ratificar los predicados que necesariamente deben asumirse en las pruebas locales, producto de los pasajes de mensajes definidos entre los procesos.

Para las pruebas locales, la axiomática incluye, además de los axiomas SKIP y ASI y la regla SEC, reglas para las instrucciones de entrada/salida y adaptaciones de las reglas de la selección condicional y la repetición no determinísticas que utilizamos en la verificación de programas secuenciales no determinísticos con comandos guardados (NCOND y NREP, respectivamente):

1. Axioma de la instrucción de entrada (IN)
$\{p\}\ P_i\ ?\ x\ \{q\}$

p y q son dos predicados arbitrarios con variables locales de un proceso P_j que incluye la instrucción P_i ? x. La arbitrariedad de la relación entre p y q se explica por las características de P_i ? x: en la prueba local de P_j, el valor asignado a la variable x proveniente del proceso P_i sólo puede asumirse, lo mismo que si P_i está preparado para comunicarse con P_j. Las asunciones deben validarse en la segunda etapa de la prueba, en la que se tiene en cuenta el programa completo.

2. Axioma de la instrucción de salida (OUT)
$\{p\}\ P_j\ !\ e\ \{p\}$

p es un predicado con variables locales de un proceso P_i que incluye la instrucción P_j ! e, la cual no tiene efecto sobre las variables locales de P_i. También puede utilizarse el axioma más general $\{p\}\ P_j\ !\ e\ \{q\}$, para incluir, como en el axioma IN, una asunción acerca del estado de P_j con respecto a su preparación para comunicarse con P_i. Cualquiera sea el caso, lo que se asume debe chequearse en la segunda etapa de la prueba.

3. Regla del condicional con comunicaciones (DCOND)

$$\frac{\{p \wedge B_i\}\ \alpha_i ; S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if } B_1 ; \alpha_1 \rightarrow S_1 \text{ or } \ldots \text{ or } B_n ; \alpha_n \rightarrow S_n \text{ fi } \{q\}}$$

La regla es la misma que utilizamos para los programas secuenciales no determinísticos con comandos guardados, pero adaptada considerando la posibilidad de que existan guardias con instrucciones de entrada/salida. La forma planteada es la más general (en los casos en los que los comandos guardados son $B_i \rightarrow S_i$, las premisas correspondientes son $\{p \wedge B_i\}\ S_i\ \{q\}$).

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

4. Regla de la repetición con comunicaciones (DREP)

$$\frac{\{p \wedge B_i\}\ \alpha_i ; S_i\ \{p\},\ i = 1, \ldots, n}{\{p\}\ \text{do}\ B_1 ; \alpha_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n ; \alpha_n \rightarrow S_n\ \text{do}\ \{p \wedge \bigwedge_i \neg B_i\}}$$

También esta regla, presentada en su forma más general, adapta contemplando guardias con instrucciones de entrada/salida una regla de la axiomática para programas secuenciales con comandos guardados, en este caso la que corresponde a la prueba de no divergencia.

Complementariamente, para la prueba global de la segunda etapa, en la que deben validarse las asunciones de las *proof outlines*, la axiomática incluye los siguientes componentes:

5. Axioma de comunicación (COM)

$$\{p[x|e]\}\ P_i\ ?\ x \ ||\ P_j\ !\ e\ \{p\}$$

El axioma captura la semántica de una comunicación entre dos instrucciones de entrada/salida que coinciden sintácticamente, que es la asignación del valor de la expresión *e* a la variable x. Su utilidad se aprecia en el chequeo de consistencia de las *proof outline*s, que en el marco de los programas distribuidos se conoce como *test de cooperación*, el cual pasamos a describir.

El test de cooperación requiere validar, por cada par de *proof outlines* $p_i\}\ P_i^*\ \{q_i\}$ del proceso P_i y $\{p_j\}\ P_j^*\ \{q_j\}$ del proceso P_j, que para todo par de instrucciones de entrada/salida que coinciden sintácticamente, P_i ? x en P_j por un lado y P_j ! e en P_i por otro lado, la fórmula $\{p_1\}\ P_i$? x $\{q_1\}$ correspondiente a P_i ? x en la *proof outline* de P_j y la fórmula $\{p_2\}\ P_j$! e $\{q_2\}$ correspondiente a P_j ! e en la *proof outline* de P_i satisfagan:

$$\{p_1 \wedge p_2\}\ P_i\ ?\ x \ ||\ P_j\ !\ e\ \{q_1 \wedge\ q_2\}$$

De esta manera, el axioma COM permite validar todas las asunciones de las pruebas locales. Si se cumplen, se define que las *proof outlines cooperan*. En base a lo definido, la regla de prueba de correctitud parcial de una composición distribuida debería ser:

$$\frac{\{p_i\}\ P_i^*\ \{q_i\},\ i = 1, \ldots, n, \text{ son } \textit{proof outlines} \text{ que cooperan}}{\{\bigwedge_{i=1,n} p_i\}\ [\ ||_{i=1,n} P_i\ ]\ \{\bigwedge_{i=1,n} q_i\}}$$

pero así planteada, la regla está incompleta. Su formulación definitiva la mostramos después de los dos ejemplos de aplicación que presentamos a continuación, en el segundo de los cuales se explicita cómo completarla.

Ejemplo. Prueba de correctitud parcial del programa que propaga un valor

Vamos a verificar la correctitud parcial del programa presentado en la sección previa, que propaga un valor desde un proceso P_1 hasta un proceso P_n. Para simplificar la prueba, acotamos el programa a tres procesos:

S_{prop} :: $[P_1 \,||\, P_2 \,||\, P_n]$, con:
P_1 :: P_2 ! y_1
P_2 :: P_1 ? y_2 ; P_3 ! y_2
P_3 :: P_2 ? y_3

Probaremos la fórmula $\{y_1 = Y\}\ S_{prop}\ \{y_3 = Y\}$.

Proponemos las siguientes *proof outlines*:

P_1 :: $\{y_1 = Y\}\ P_2$! y_1 $\{y_1 = Y\}$
P_2 :: $\{true\}\ P_1$? y_2 $\{y_2 = Y\}$; P_3 ! y_2 $\{y_2 = Y\}$
P_3 :: $\{true\}\ P_2$? y_3 $\{y_3 = Y\}$

La correctitud de las *proof outlines* se verifica fácilmente, aplicando los axiomas IN y OUT y la regla SEC. Resta probar que las *proof outlines* cooperan. Hay dos pares de instrucciones de entrada/salida que coinciden sintácticamente: P_2 ! y_1 (en P_1) y P_1 ? y_2 (en P_2), y P_3 ! y_2 (en P_2) y P_2 ? y_3 (en P_3). Por lo tanto, de acuerdo al test de cooperación, hay que verificar:

$\{y_1 = Y \wedge true\}\ P_2$! y_1 $||$ P_1 ? y_2 $\{y_1 = Y \wedge y_2 = Y\}$
$\{y_2 = Y \wedge true\}\ P_3$! y_2 $||$ P_2 ? y_3 $\{y_2 = Y \wedge y_3 = Y\}$

Las fórmulas se prueban aplicando el axioma COM y la regla CONS. Así obtenemos la fórmula $\{y_1 = Y \wedge true \wedge true\}\ S_{prop}\ \{y_1 = Y \wedge y_2 = Y \wedge y_3 = Y\}$, y por la regla CONS, $\{y_1 = Y\}\ S_{prop}\ \{y_3 = Y\}$.

Ejemplo. Prueba de correctitud parcial de un programa que devuelve el número uno

Dado el programa:

S_{0a1} :: $[P_1 \,||\, P_2]$, con:
P_1 :: x := 0 ; P_2 ! x ; x := x + 1 ; P_2 ! x
P_2 :: P_1 ? y ; P_1 ? y

vamos a verificar $\{true\}\ S_{0a1}\ \{y = 1\}$. Las *proof outlines* naturales de P_1 y P_2 son:

P_1 :: $\{true\}$ x := 0 $\{x = 0\}$; P_2 ! x $\{x = 0\}$; x := x + 1 $\{x = 1\}$; P_2 ! x $\{x = 1\}$
P_2 :: $\{true\}\ P_1$? y $\{y = 0\}$; P_1 ? y $\{y = 1\}$

que de acuerdo al test de cooperación deben satisfacer:

a) $\{x = 0 \wedge true\}\ P_2$! x $||$ P_1 ? y $\{x = 0 \wedge y = 0\}$
b) $\{x = 1 \wedge true\}\ P_2$! x $||$ P_1 ? y $\{x = 1 \wedge y = 0\}$
c) $\{x = 0 \wedge y = 0\}\ P_2$! x $||$ P_1 ? y $\{x = 0 \wedge y = 1\}$
d) $\{x = 1 \wedge y = 0\}\ P_2$! x $||$ P_1 ? y $\{x = 1 \wedge y = 1\}$

Las fórmulas de (a) y (d) se prueban fácilmente por medio del axioma COM y la regla CONS. Sin embargo, las fórmulas de (b) y (c) no se cumplen. Por el axioma COM debe valer:

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

$(x = 1 \wedge true) \longrightarrow (x = 1 \wedge x = 0)$
$(x = 0 \wedge y = 0) \longrightarrow (x = 0 \wedge x = 1)$

y ambas implicaciones son falsas. Esto va a ocurrir indefectiblemente con cualquier par de *proof outlines* que se propongan, porque las fórmulas de (b) y (c) corresponden a comunicaciones imposibles de ejecutarse en el programa, situación que el test de cooperación no contempla tal como está definido. Así, el test debe ajustarse, para que no requiera tratar pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente.

Lo que plantea la axiomática en este caso es, como antes, recurrir a variables auxiliares (y la regla AUX), para reforzar predicados que precisen la historia del *interleaving* ejecutado. Pero además introduce una novedad, los *invariantes globales*, que en alguna medida recuerdan los invariantes de recursos usados con los programas paralelos. Antes de reformular el test de cooperación, completamos la prueba del ejemplo con la idea comentada:

- Primero agregamos una variable local auxiliar c_1 en el proceso P_1 y una variable local auxiliar c_2 en el proceso P_2, ámbas inicializadas en 0 y que se incrementan en 1 toda vez que el proceso respectivo ejecuta una instrucción de entrada/salida. El incremento y la instrucción de entrada/salida asociada se ejecutan atómicamente. De esta manera, los valores de las variables c_1 y c_2 siempre son iguales antes y después de cada comunicación del programa, lo que resulta útil para la prueba, como mostramos enseguida. Las *proof outlines* del programa ampliado quedan así (las secciones atómicas se delimitan con los símbolos < >):

$$
\begin{array}{lll}
 & \{c_1 = 0\} & \{c_2 = 0\} \\
[P'_1 :: & x := 0 \; ; \; \{x = 0 \wedge c_1 = 0\} & P'_2 :: <P_1 \; ? \; y \; ; \; c_2 := c_2 + 1>; \{y = 0 \wedge c_2 = 1\} \\
 & <P_2 \; ! \; x; c_1 := c_1 + 1>; \{x = 0 \wedge c_1 = 1\} \; || & < P_1 \; ? \; y \; ; \; c_2 := c_2 + 1 >] \\
 & x := x + 1 \; ; \; \{x = 1 \wedge c_1 = 1\} & \{y = 1 \wedge c_2 = 2\} \\
 & < P_2 \; ! \; x \; ; \; c_1 := c_1 + 1 > & \\
 & \{x = 1 \wedge c_1 = 2\} &
\end{array}
$$

- Complementariamente, definimos un invariante global $IG = (c_1 = c_2)$, que en sintonía con lo anterior, al agregarlo en los chequeos del test de cooperación permite descartar los pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente, es decir las comunicaciones imposibles de ejecutarse en el programa. Las nuevas fórmulas a verificar con este agregado son las siguientes:

a) $\{x = 0 \wedge c_1 = 0 \wedge c_2 = 0 \wedge (c_1 = c_2)\}$
$< P_2 \; ! \; x \; ; \; c_1 := c_1 + 1 > \; || \; < P_1 \; ? \; y \; ; \; c_2 := c_2 + 1 >$
$\{x = 0 \wedge c_1 = 1 \wedge y = 0 \wedge \; c_2 = 1 \wedge (c_1 = c_2)\}$

b) $\{x = 1 \wedge c_1 = 1 \wedge c_2 = 0 \wedge (c_1 = c_2)\}$
$< P_2 \; ! \; x \; ; \; c_1 := c_1 + 1 > \; || \; < P_1 \; ? \; y \; ; \; c_2 := c_2 + 1 >$
$\{x = 1 \wedge c_1 = 2 \wedge y = 0 \wedge \; c_2 = 1 \wedge (c_1 = c_2)\}$

c) $\{x = 0 \wedge c_1 = 0 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$
$< P_2 \; ! \; x \; ; \; c_1 := c_1 + 1 > \; || \; < P_1 \; ? \; y \; ; \; c_2 := c_2 + 1 >$
$\{x = 0 \wedge c_1 = 1 \wedge y = 1 \wedge \; c_2 = 2 \wedge (c_1 = c_2)\}$

d) $\{x = 1 \wedge c_1 = 1 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$
$< P_2 \,!\, x \,;\, c_1 := c_1 + 1 > \,||\, < P_1 \,?\, y \,;\, c_2 := c_2 + 1 >$
$\{x = 1 \wedge c_1 = 2 \wedge y = 1 \wedge\ c_2 = 2 \wedge (c_1 = c_2)\}$

Se comprueba fácilmente que ahora se cumplen todas las fórmulas (asumiendo que las reglas utilizan secciones atómicas, lo que comentamos más adelante). Las fórmulas de (b) y (c) se cumplen trivialmente porque las precondiciones son falsas, producto de la información global aportada por el invariante IG, que determina que las comunicaciones correspondientes no pueden ocurrir. Así llegamos a $\{true \wedge true\}\ S_{0a1}\ \{x = 1 \wedge y = 1\}$, por aplicación de la regla AUX, y por la regla CONS, a $\{true\}\ S_{0a1}\ \{y = 1\}$.

Completado el ejemplo, formalizamos en lo que sigue las modificaciones necesarias para obtener la forma definitiva de la regla de la composición distribuida:

- Según cómo aparezca en el programa, toda instrucción de entrada/salida α de toda *proof outline* se encapsula en una sección atómica $< S_1 \,;\, \alpha \,;\, S_2 >$ o $< \alpha \rightarrow S_1 >$, en que ni S_1 ni S_2 contienen instrucciones de entrada/salida (se define que dos secciones atómicas coinciden sintácticamente si lo hacen las instrucciones de entrada/salida que encapsulan).
- Se especifica un invariante global IG en términos de las variables de los procesos y de variables auxiliares locales que se agregan al programa, que sólo pueden modificarse dentro de las secciones atómicas. IG debe valer al inicio y ser preservado por toda comunicación. Su rol es proveer información para evitar en las pruebas el tratamiento de pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente, y por eso se agrega en los chequeos del test de cooperación.
- Se reformula el test de cooperación de la siguiente manera. Por cada par de *proof outlines* $\{p_i\}\ P_i^*\ \{q_i\}$ y $\{p_j\}\ P_j^*\ \{q_i\}$, para todo par de secciones atómicas $< S >$ (en P_j) y $< S' >$ (en P_i) que coinciden sintácticamente, la fórmula $\{p_1\} < S > \{q_1\}$ asociada a $< S >$ en la *proof outline* de P_j y la fórmula $\{p_2\} < S' > \{q_2\}$ asociada a $< S' >$ en la *proof outline* de P_i deben satisfacer:

$$\{p_1 \wedge p_2 \wedge IG\} < S > \,||\, < S' > \{q_1 \wedge q_2 \wedge IG\}$$

  Se define en este caso que las *proof outlines* cooperan con respecto al invariante global IG.

- Por el uso de secciones atómicas, deben agregarse a la axiomática algunas reglas auxiliares relacionadas con las dos formas definidas, $< S_1 \,;\, \alpha \,;\, S_2 >$ y $< \alpha\ \ S_1 >$. Por ejemplo:

$$\frac{\{p\}\ S_1 \,;\, S_3\ \{r_1\},\ \{r_1\}\ \alpha \,||\, \alpha'\ \{r_2\},\ \{r_2\}\ S_2 \,;\, S_4\ \{q\}}{\{p\} < S_1 \,;\, \alpha \,;\, S_2 > \,||\, < S_3 \,;\, \alpha' \,;\, S_4 > \{q\}}$$

$$\frac{\{p\}\ \alpha \,||\, \alpha'\ \{r\},\ \{r\}\ S_1 \,;\, S_2\ \{q\}}{\{p\} < \alpha \longrightarrow S_1 > \,||\, < \alpha' \longrightarrow S_2 > \{q\}}$$

Con la nueva definición de cooperación, necesaria en realidad en la prueba de toda propiedad de un programa distribuido para descartar los pares de instrucciones de entrada/salida que

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

coinciden sintácticamente pero no semánticamente, llegamos a la versión definitiva de la regla de la composición distribuida (cuya aplicación luego es seguida por la de la regla AUX):

6. Regla de la composición distribuida (DIST)
$\{p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\bigwedge_{i=1,n} p_i \wedge IG),\ (\bigwedge_{i=1,n} q_i \wedge IG) \longrightarrow q$

$$\{p\}\ [\ ||_{i=1,n} P_i\ ]\ \{q\}$$

tal que ninguna variable de IG puede ser modificada fuera de las secciones atómicas.

2.2. Axiomática para las pruebas de no divergencia

En los programas distribuidos, a diferencia de lo que observamos en los programas paralelos, al no haber variables compartidas las pruebas no tienen que reparar en el impacto de un proceso por esta vía sobre el valor del variante definido para una repetición de otro proceso. Pero aparece otra dificultad, que es que la no divergencia de un proceso no siempre puede probarse localmente, dadas sus comunicaciones con otros procesos. Por ejemplo, que el proceso:

$$P_1 :: \text{do } x \neq 0 \ ; P_2 \ ? \ x \rightarrow \text{skip od}$$

no diverja depende de si recibe alguna vez del proceso P_2 el valor 0. Así que en este caso también hay que recurrir a asunciones en las pruebas locales y verificarlas en el test de cooperación de la prueba global (incluyendo variables auxiliares locales y un invariante global). En particular, puede ser necesario asumir el valor inicial de algunos variantes, si la cantidad de iteraciones de las repeticiones correspondientes depende de información global.

En las pruebas locales, para verificar la no divergencia de una repetición se utiliza la misma regla que definimos para los programas secuenciales con comandos guardados, pero ahora contemplando instrucciones de entrada/salida en las guardias. Su forma más general es:

7. Regla de la no divergencia con comunicaciones (DREP*)
$\langle p \wedge B_i \rangle\ \alpha_i\ ; S_i\ \langle p \rangle$, i = 1, …, n, $\langle p \wedge B_i \wedge t = Z \rangle\ \alpha_i\ ; S_i\ \langle t < Z \rangle$, i = 1, …, n, $p \rightarrow t \geq 0$

$$\langle p \rangle\ \text{do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od}\ \ \langle p \wedge \bigwedge_{i=1,n} \neg B_i \rangle$$

tal que p y t son el invariante y el variante de la repetición, respectivamente. Para la prueba global se recurre al test de cooperación de la misma manera que en la prueba de correctitud parcial:

8. Regla de la no divergencia distribuida (DIST*)
$\langle p_i \rangle\ S_i^{**}\ \langle q_i \rangle$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\bigwedge_{i=1,n} p_i \wedge IG),\ (\bigwedge_{i=1,n} q_i \wedge IG) \longrightarrow q$

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

tal que ninguna variable de IG puede ser modificada fuera de las secciones atómicas. Mostramos a continuación un ejemplo de aplicación de estas reglas sobre un programa muy sencillo.

Ejemplo. Prueba de no divergencia de un programa con diez iteraciones

Vamos a probar:

$\langle 0 \leq x \leq 10 \wedge y > 0 \rangle\ S_{diez} :: [P_1 \,||\, P_2\,]\ \langle true \rangle$, con:
$P_1 :: do\ x \geq 0\ ;\ P_2\ !\ x \rightarrow x := x - 1\ od$
$P_2 :: do\ y \neq 0\ ;\ P_1\ ?\ y \rightarrow skip\ od$

Primero definimos las secciones atómicas del programa, y agregamos en el proceso P_2 una variable auxiliar w para diferenciar las instancias anterior y posterior de la primera iteración de su instrucción de repetición, cuya utilidad explicamos enseguida:

$P'_1 :: do\ x \geq 0\ ;\ <P_2\ !\ x \rightarrow x := x - 1 > od$
$P'_2 :: w := 0\ ;\ do\ y \neq 0\ ;\ <P_1\ ?\ y \rightarrow skip\ ;\ w := 1 > od$

Para la prueba de no divergencia del proceso P_1 definimos como invariante de su repetición el predicado $p_1 = (x \geq -1)$, y como variante la función $t_1 = x + 1$. De acuerdo a la regla DREP*, deben cumplirse las siguientes fórmulas:

1. $\langle x \geq -1 \wedge x \geq 0 \rangle\ P_2\ !\ x\ ;\ x := x - 1\ \langle x \geq -1 \rangle$
2. $\langle x \geq -1 \wedge x \geq 0 \wedge x + 1 = Z \rangle\ P_2\ !\ x\ ;\ x := x - 1\ \langle x + 1 < Z \rangle$
3. $x \geq -1 \rightarrow x + 1 \geq 0$

que se verifican fácilmente por medio de los axiomas ASI y OUT y las reglas SEC y CONS.

La prueba de no divergencia del proceso P_2 es más complicada, porque el valor de la variable *y* depende de lo que recibe de P_1. Definimos para la repetición de P_2 el invariante $p_2 = (y \geq 0)$ y el variante t_2 = (if w = 0 then 11 else y fi). La variable auxiliar w sirve para asignar un valor inicial al variante. En este caso, las fórmulas que deben cumplirse por DREP* son:

1. $\langle y \geq 0 \wedge y \neq 0 \rangle\ P_1\ ?\ y\ ;\ skip\ ;\ w := 1\ \langle y \geq 0 \rangle$
2. $\langle y \geq 0 \wedge y \neq 0 \wedge$ (if w = 0 then 11 else y fi) $= Z \rangle$
   $P_1\ ?\ y\ ;\ skip\ ;\ w := 1$
   $\langle$(if w = 0 then 11 else y fi) $< Z \rangle$
3. $y \geq 0 \rightarrow$ (if w = 0 then 11 else y fi) ≥ 0

que también se prueban fácilmente, aplicando los axiomas SKIP, ASI e IN y las reglas SEC y CONS. Las *proof outlines* de los procesos ampliados quedan así:

$\langle$inv: $x \geq -1$, var: $x + 1\rangle$
$P'_1 :: do\ x \geq 0\ ;\ \langle x \geq -1 \wedge x \geq 0 \rangle < P_2\ !\ x \rightarrow x := x - 1 > od$
$\langle x \geq -1 \rangle$
$\langle y > 0 \rangle$
$P'_2 :: w := 0\ ;$
$\langle$inv: $y \geq 0$, var: if w = 0 then 11 else y fi$\rangle$
$do\ y \neq 0\ ;\ \langle y \geq 0 \wedge y \neq 0 \rangle < P_1\ ?\ y \rightarrow skip\ ;\ w := 1 > od$
$\langle y \geq 0 \rangle$

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

Se cumple que la precondición del programa implica la conjunción de las precondiciones de los procesos:

$(0 \leq x \leq 10 \wedge y > 0) \rightarrow (x \geq -1 \wedge y > 0)$.

Para la segunda etapa de la prueba planteamos el siguiente invariante global:

$IG = (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1))$

alineado con el comportamiento del programa modificado, por lo que el chequeo requerido por el test de cooperación utilizado por la regla DIST* queda planteado de esta forma:

$\langle x \geq -1 \wedge x \geq 0 \wedge y \geq 0 \wedge y \neq 0 \wedge (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1)\rangle$
$< P_2 ! x \longrightarrow x := x - 1 > \,||\, < P_1 ? y \longrightarrow skip ; w := 1 >$
$\langle x \geq -1 \wedge y \geq 0 \wedge (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1)\rangle$

La fórmula se verifica por medio de la segunda regla auxiliar presentada en la sección anterior, los axiomas SKIP, ASI y COM y las reglas SEC y CONS. La prueba se completa aplicando la regla AUX.

Con hipótesis de *fairness* en los programas distribuidos, en las pruebas de no divergencia (y en general en las pruebas de cualquier propiedad de tipo *liveness*) se plantean más escenarios que el de nivel *proceso* que mencionamos en el marco de los programas paralelos. Por ejemplo, con hipótesis de *fairness* fuerte se pueden definir los siguientes dos niveles adicionales:

1. Nivel *canal*: todo par de procesos infinitas veces preparados para comunicarse se comunican infinitas veces.
2. Nivel *comunicación*: toda comunicación infinitas veces habilitada se ejecuta infinitas veces.

El siguiente programa sirve para aclarar las definiciones anteriores. En este caso, la no divergencia sólo se logra con *fairness* fuerte de nivel comunicación:

$[P_1 :: b := true ;$ $\quad P_2 :: c := true ;$
$\quad do\ b ; P_2 ? b \rightarrow skip\ od \;\; || \;\; do\ c ; P_1 ! true \rightarrow skip$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad or\ c ; P_1 ! false \rightarrow c := false$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad od]$

El programa termina si P_2 toma alguna vez la segunda dirección de su instrucción de repetición, lo que no se puede asegurar con *fairness* fuerte de niveles proceso o canal.

La no divergencia con *fairness* se puede probar en base a *direcciones útiles*, como hicimos con los programas secuenciales no determinísticos, pero ahora considerando, en lugar de direcciones habilitadas, pares de direcciones habilitadas.

2.3. Axiomática para las pruebas de ausencia de falla

Completamos la presentación de la axiomática con una breve referencia a la verificación de la propiedad de ausencia de falla. Un programa distribuido falla si durante su ejecución, en una

selección condicional todas las expresiones booleanas de sus guardias son falsas, o si se produce una situación de *deadlock*. Para verificar la ausencia de falla del primer caso, la axiomática incluye una regla similar a la regla DCOND utilizada para la prueba de correctitud parcial, con una premisa adicional. Su forma más general es:

9. Regla del condicional con comunicaciones sin falla (DCOND*)

$$\frac{p \rightarrow \bigvee_{i=1,n} B_i,\ \{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if}\ B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or}\ \ldots\ \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{fi}\ \{q\}}$$

La premisa $p \rightarrow \bigvee_{i=1,n} B_i$ asegura que al menos una guardia de la selección condicional tiene una expresión booleana verdadera. Con respecto a la prueba de ausencia de *deadlock*, la regla correspondiente se basa en el mismo esquema que planteamos para los programas paralelos:

10. Regla de la ausencia de deadlock en programas distribuidos (DDEADLOCK)

Dadas *proof outlines* (con secciones atómicas y que cooperan con respecto a un invariante global IG), se identifican sintácticamente todas las posibles situaciones de *deadlock*, se obtienen las imágenes semánticas asociadas, y se prueba que todas resultan falsas. Por las características de los programas distribuidos, los predicados que integran las imágenes semánticas pueden ser:

- post(P_i), la postcondición de la *proof outline* del proceso P_i.
- pre(< S >), la precondición de una sección atómica S de la forma S_1 ; α ; S_2.
- $pre(S) \wedge \bigwedge_{i \in H} B_i \wedge \bigwedge_{j \notin H} \neg B_j$, la precondición de una selección condicional o una repetición S, en conjunción con un predicado que establece que S tiene un subconjunto H no vacío de direcciones con expresiones booleanas verdaderas.

No puede haber sólo predicados del primer tipo, porque si no la imagen semántica no representaría una situación de *deadlock*. Como ejemplo de aplicación de esta regla, consideramos nuevamente el programa de propagación de un valor.

Ejemplo. Prueba de ausencia de *deadlock* en el programa que propaga un valor

Para verificar la correctitud parcial de $P_{prop} :: [P_1 :: P_2 ! y_1 \,||\, P_1 ? y_2 ; P_3 ! y_2 \,||\, P_2 ? y_3]$ hemos utilizado en una sección anterior las *proof outlines*:

$P_1 :: \{y_1 = Y\}\ P_2 ! y_1\ \{y_1 = Y\}$
$P_2 :: \{true\}\ P_1 ? y_2\ \{y_2 = Y\}; P_3 ! y_2\ \{y_2 = Y\}$
$P_3 :: \{true\}\ P_2 ? y_3\ \{y_3 = Y\}$

que no sirven en este caso para probar la ausencia de *deadlock*. Por ejemplo, la imagen semántica correspondiente a la situación de *deadlock* en que P_1 terminó y P_2 y P_3 no empezaron:

$$y_1 = Y \wedge true \wedge true$$

no es falsa. Se necesita recurrir a variables auxiliares y un invariante global. Agregamos una variable w_i en cada proceso P_i, tal que $w_i = 0$ significa que P_i está preparado para recibir un

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

valor, $w_i = 1$ que P_i está preparado para enviar un valor, y $w_i = 2$ que P_i terminó. La variable w_1 se inicializa en 1, y las variables w_2 y w_3 en 0. Correspondientemente con esta idea, definimos el siguiente invariante global:

$$IG = (w_1 = 1 \wedge w_2 = 0 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 1 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 2 \wedge w_3 = 2)$$

Las *proof outlines* de los procesos ampliados quedan de la siguiente manera:

$\{w_1 = 1\}$	$\{w_2 = 0\}$	$\{w_3 = 0\}$
$[< P_2 \,!\, y_1 \,;\, w_1 := 2>$ $\|\|$	$<P_1 \,?\, y_2 \,;\, w_2 := 1>$ $\|\|$	$<P_2 \,?\, y_3 \,;\, w_3 := 2>]$
$\{w_1 = 2\}$	$\{w_2 = 1\}$	$\{w_3 = 2\}$
	$<P_3 \,!\, y_2 \,;\, w_2 := 2>$	
	$\{w_2 = 2\}$	

La cooperación entre las *proof outlines* con respecto a IG se prueba fácilmente, empleando la primera regla auxiliar presentada previamente. Notar que ahora la imagen semántica que representa la situación en la que P_1 terminó y P_2 y P_3 no empezaron, incluyendo el invariante IG, resulta falsa:

$$(w_1 = 2 \wedge w_2 = 0 \wedge w_3 = 0) \wedge$$
$$[(w_1 = 1 \wedge w_2 = 0 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 1 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 2 \wedge w_3 = 2)]$$

Se comprueba fácilmente lo mismo con el resto de las situaciones posibles de *deadlock*.

3. Notas adicionales

Incluimos una nota en la que nos referimos resumidamente a la verificación axiomática de programas distribuidos con *comunicaciones asincrónicas*, y otra sobre alternativas *composicionales* al método no composicional de pruebas locales y globales que hemos considerado en este artículo y el anterior.

3.1. Verificación con comunicaciones asincrónicas

En los programas distribuidos con comunicaciones asincrónicas los procesos ejecutan las instrucciones de salida sin tener en cuenta la disponibilidad de sus contrapartes para recibir los mensajes. Sólo las instrucciones de entrada pueden tener que esperar para ejecutarse, y así provocar eventualmente bloqueos en los procesos que las contienen. La semántica informal de ambas instrucciones en esta clase de programas es la siguiente (consideramos un lenguaje distribuido similar al presentado antes, reemplazando los símbolos ? y ! por ?? y !!, respectivamente, y utilizando identificadores de canales en lugar de identificadores de procesos y guardias sin instrucciones de salida):

- Instrucción de *entrada* c ?? x (en un proceso receptor P_r): Si en el canal c existe un valor disponible, éste se asigna a la variable local x de P_r. Si no, P_r queda bloqueado hasta que haya un valor en c.

- Instrucción de *salida* c !! e (en un proceso emisor P_e): P_e envía al canal c el valor de la expresión *e*, definida en términos de sus variables locales, y continúa con la ejecución de la siguiente instrucción.

Otras características del lenguaje son: todo canal conecta a un único proceso emisor con un único proceso receptor, la única hipótesis sobre el tiempo entre el envío y la recepción de un mensaje es que es finito, y los mensajes llegan en el orden en el que se envían. Por ejemplo, el siguiente programa calcula el máximo de un conjunto de n valores x_i, todos distintos, en base a una estructura de anillo de n procesos distribuidos:

S_{max} :: [$||_{i = 1,n} P_i$], con:

P_i :: $z_i := 0$; $y_i := 0$; c_i !! x_i ;
do $y_i \neq N \rightarrow c_{i-1}$?? y_i ;
if $x_i = y_i \rightarrow z_i := 1$; $y_i := N$; c_i !! N
or $x_i > y_i \rightarrow$ skip
or $x_i < y_i \rightarrow c_i$!! y_i
fi
od

El canal c_i comunica al proceso P_i con su vecino de la derecha, P_{i+1} (o P_1 si i = n). P_i contiene el valor x_i, que envía al inicio. Luego, iterativamente, P_i envía todo valor y_i mayor que su valor x_i, que recibe de su vecino de la izquierda, P_{i-1}, en el canal c_{i-1} (o P_n en c_n si i = 1). De esta manera, sólo el valor máximo recorre el anillo completo de procesos, situación que detecta el proceso P_i que recibe su valor x_i, en cuyo caso se identifica ejecutando la asignación $z_i := 1$ y hace finalizar el programa enviando un valor N mayor que todos los x_i al resto de los procesos. En [SS84] se describe una variante axiomática para verificar programas del lenguaje descripto, basada en el esquema de pruebas en dos etapas de S. Owicki y D. Gries. A cada canal c se le asocian dos variables de tipo *secuencia* para utilizar en las pruebas, una variable IN_c, local del proceso receptor, y una variable OUT_c, local del proceso emisor, que representan la secuencia de valores recibidos por uno y la secuencia de valores enviados por el otro, respectivamente.

Los axiomas para las pruebas locales relacionados con las instrucciones de entrada/salida son:

Axioma de la instrucción de entrada asincrónica (AIN)
{p} c ?? x {q}

Al igual que el axioma IN para las comunicaciones sincrónicas, plantea en el caso más general asunciones sobre el valor recibido y sobre la posibilidad de recibir un mensaje, a ser validadas en el test de cooperación de la prueba global.

Axioma de la instrucción de salida asincrónica (AOUT)
{p[OUT_c|OUT_C.e]} c !! e {p}

El axioma captura el efecto de la instrucción de salida: el envío del valor de la expresión *e* al final de la secuencia de valores enviados al canal c (OUT_C.e expresa la concatenación de la secuencia representada por OUT_c con *e*). En este caso, la precondición y la postcondición coinciden siempre,

Revista Abierta de Informática Aplicada/vol 7 Nº2 (2023): 51-70
ISSN 2591-5320

porque en una comunicación asincrónica no se necesita asumir nada sobre la disponibilidad del proceso receptor para recibir un mensaje.

Axioma del orden de llegada (FIFO)
Para todo k, si $|IN_c| = k$ entonces $|OUT_c| \geq k$, y $IN_c[k] = OUT_c[k]$

$|IN_c|$ expresa el tamaño de la secuencia IN_c, e $IN_c[k]$ el valor k-ésimo de la misma. Lo mismo se define para OUT_c. El axioma establece que los mensajes se reciben en el orden en el que se envían (FIFO abrevia *first in first out*, es decir, el primero que entra es el primero que sale).

En lo que hace a los componentes de la axiomática para la prueba global, cabe destacar que por el uso de las variables IN_c y OUT_c, en el test de cooperación no hace falta recurrir a un invariante global. El test consiste en confrontar, para cada par de *proof outlines*, todos los pares de instrucciones de entrada/salida que coinciden sintácticamente, que en este caso están constituidos por una instrucción de entrada y una instrucción de salida que actúan sobre un mismo canal. Específicamente, dadas $\{p_1\}$ c ?? x $\{q_1\}$ y $\{p_2\}$ c !! e $\{q_2\}$ debe cumplirse:

$$\{p_1 \wedge p_2 \wedge IN_c = OUT_c\} \text{ c ?? x } || \text{ c !! e } \{q_1\}$$

No se considera la postcondición de la instrucción de salida, acorde con la semántica de las comunicaciones asincrónicas. La igualdad $IN_c = OUT_c$ permite descartar las instrucciones de entrada/salida que no coinciden semánticamente. Para llevar a cabo el test se emplea el siguiente el axioma:

Axioma de comunicación asincrónica (ACOM)
$\{p[x|e][IN_c|IN_C.e]\}\ P_i\ ?\ x\ ||\ P_j\ !\ e\ \{p\}$

que captura la semántica de la comunicación asincrónica. Según lo definido, se puede formular la siguiente regla para la prueba de correctitud parcial:

Regla de la composición distribuida con comunicaciones asincrónicas (ADIST)
$p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* que cooperan
$p \longrightarrow (\bigwedge_{i=1,n} p_i \wedge \bigwedge_c IN_c = OUT_c = \lambda), (\bigwedge_{i=1,n} q_i) \longrightarrow q$

$$\{p\}\ [\ ||_{i=1,n} P_i\ ]\ \{q\}$$

Se agrega $\bigwedge_c IN_c = OUT_c = \lambda$ en la precondición para indicar que al comienzo todos los canales están vacíos (tienen la secuencia vacía λ). El resto de las propiedades se prueban a partir de *proof outlines* de correctitud parcial que cooperan.

3.2. Métodos de prueba composicionales

A la variante axiomática no composicional de S. Owicki y D. Gries le sucedieron varias propuestas composicionales. La primera, para la correctitud parcial de programas paralelos, es [Jon81]. En [XdRH97] se la describe y amplía para la prueba de correctitud total, incluyendo además consideraciones sobre la sensatez y la completitud de la axiomática. La idea básica 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). Más en detalle, utilizando para simplificar sólo dos procesos S_1 y S_2, se plantea lo siguiente:

- En cuanto a la especificación: dada una precondición p y una postcondición q de una composición concurrente $[S_1 \,||\, S_2]$, definir un conjunto de variables auxiliares y dos tuplas de predicados $(p_1, q_1, rely_1, guar_1)$ y $(p_2, q_2, rely_2, guar_2)$ para S_1 y S_2, respectivamente, siendo p_1 y p_2 las precondiciones, q_1 y q_2 las postcondiciones, y $rely_1$, $guar_1$, $rely_2$ y $guar_2$ relaciones que establecen transformaciones entre las variables compartidas y auxiliares. El significado de la tupla $(p_i, q_i, rely_i, guar_i)$ es que cuando S_i se ejecuta a partir de un estado que satisfice p_i, en un entorno en el que se pueden modificar las variables compartidas y auxiliares de acuerdo a la condición $rely_i$, el proceso lleva a cabo transformaciones que respetan la condición $guar_i$, y si termina, lo hace en un estado que satisface q_i.
- En cuanto a la verificación: elaborar *proof outlines* de S_1 y S_2 que satisfagan las condiciones establecidas (por ejemplo, que los invariantes se preserven teniendo en cuenta las condiciones $rely_i$), para llegar a la fórmula $\{p_1 \wedge p_2\}\ [S_1 \,||\, S_2]\ \{q_1 \wedge q_2\}$, y finalmente, mediante las reglas de las variables auxiliares y de consecuencia, a $\{p\}\ [S_1 \,||\, S_2]\ \{q\}$.

Para el tratamiento de la ausencia de *deadlock* se sigue una línea similar, agregando en la especificación información acerca de los posibles bloqueos de los procesos. Con respecto a la no divergencia, la idea es directamente adaptar la regla de no divergencia de la variante no composicional, considerando las nuevas condiciones. Las mismas nociones generales pueden aplicarse a los programas distribuidos. El enfoque descripto convive con el de S. Owicki y D. Gries, lo que se explica porque este último resulta más accesible, sobre todo para especificar los invariantes. Un método axiomático composicional es más conveniente fundamentalmente para el desarrollo sistemático de programas, permite que los procesos se deriven y prueben directamente desde las especificaciones, y que en general las propiedades de los programas se infieran a partir de las propiedades de los procesos, sin necesidad de chequear la consistencia de las *proof outlines*. Entre las metodologías de desarrollo con los principios mencionados se destaca UNITY o *Unbounded Nondeterministic Iterative Transformations* (Transformaciones Iterativas no Determinísticas no Acotadas), presentada por primera vez en [CM88]. Esta obra se considera fundacional para el desarrollo sistemático de programas concurrentes. La metodología incluye las etapas de especificación y verificación. Aunque se basa en la lógica temporal, comentamos en lo que sigue algunas de sus características para profundizar en lo que aporta un método composicional. Un programa UNITY es esencialmente una repetición infinita de asignaciones condicionales que se ejecutan no determinísticamente en tanto estén habilitadas infinitas veces. Existe una noción de terminación, que se manifiesta si al cabo de una cantidad finita de iteraciones se alcanza un estado que no se modifica más (se lo conoce como *punto fijo*). A partir de una primera versión de programa, la idea es refinarla gradualmente, hasta alcanzar el nivel de detalle esperado, instanciado a una de varias arquitecturas posibles. Existen dos estrategias de construcción de programas, la *unión* y la *superposición*, que representan la composición y el refinamiento, respectivamente. La unión de dos procesos S_1 y S_2, denotada con $[S_1 \,||\, S_2]$, consiste en su concatenación, mientras que la superposición establece una programación por capas, de modo tal que a una nueva capa se le agregan variables y asignaciones que no alteran el cómputo

de las capas inferiores. Las especificaciones se definen en términos de cuatro operadores básicos, *FP*, *unless*, *ensures* y *leads-to*:

- FP.S establece el conjunto de puntos fijos del programa S.
- p *unless* q establece que si se cumple p, q no se cumple nunca y p se cumple siempre, o q se cumple a futuro y p se cumple mientras q no se cumpla. De este operador se derivan *invariant* p (p se cumple siempre) y *stable* p, equivalente a p *unless false*.
- p *ensures* q establece que si se cumple p, q se cumple a futuro y p se cumple mientras q no se cumpla.
- p *leads-to* q establece que si se cumple p, q se cumple a futuro.

La metodología propone hacer gran parte de la tarea de desarrollo de un programa en la etapa de especificación, para facilitar su cálculo *top-down* y la verificación de sus propiedades a partir de las propiedades de los procesos. En este último sentido, se demuestra por ejemplo que:

- Se cumple p *unless* q en $[S_1 \,||\, S_2]$ si se cumple p *unless* q en los dos procesos.
- Se cumple p *ensures* q en $[S_1 \,||\, S_2]$ si se cumple p *ensures* q en un proceso y p *unless* q en el otro.

En general, no se puede inferir que *p leads-to* q se cumple en $[S_1 \,||\, S_2]$ aún cumpliéndose en los dos procesos. Con respecto a la estrategia de superposición, mientras facilita la verificación, como contrapartida exige conocer en detalle las capas previamente desarrolladas, al tiempo que su tratamiento algebraico es limitado.

4. Observaciones finales

El método descripto utiliza los mismos conceptos generales estudiados previamente. Las axiomáticas resultan excelentes guías para la obtención de programas correctos por construcción, y las nociones de *invariante* y *variante* constituyen la parte fundamental de la metodología de pruebas.

Sin embargo, tiene una diferencia relevante con respecto a las variantes axiomáticas para los programas secuenciales, que ya destacamos en el caso de los programas paralelos: *no es composicional*, producto del tipo de especificación que maneja (en la nota adicional sobre alternativas composicionales se aprecian las características principales que debe tener una axiomática composicional).

Otra característica diferencial es el uso de *variables auxiliares* para registrar las historias de las computaciones cuando las variables de programa no resultan suficientes (lo que también comentamos con los programas paralelos), y de *invariantes globales* ante la ausencia de estados que reúnan información común a todos los procesos.

Desde el punto de vista del desarrollo sistemático de programas, en general se acepta que el modelo distribuido tiene ventajas en relación al modelo paralelo, lo que se percibe en la práctica por la existencia de metodologías de construcción de programas del primer modelo más disciplinadas, característica que se refleja también en las pruebas asociadas.

Referencias

» [AFdR80] Apt, K., Francez, N. y de Roever, W. A proof system for communicating sequential processes. ACM Trans. Prog. Lang. Syst., 2, 3, 359-385, 1980.

» [CM88] Chandy, K. y Misra J. Parallel program design: a foundation. Addison-Wesley, 1988.

» [Hoa78] Hoare, C. Communicating secuential processes. Comm. ACM, 21, 8, 666-677, 1978.

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

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

» [OG76b] Owicki, S. y Gries, D. Verifying properties of parallel programs: an axiomatic approach. Comm. ACM, 19, 5, 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.

» [Ros23] Rosenfeld, R. Verificación de programas paralelos. Revista Abierta de Informática Aplicada, 7, 1, 67-93, 2023.

» [SS84]Schlichting, R. y Schneider, F. Message passing for distributed programming. ACM Trans. Prog. Lang. Syst., 6, 3, 402-431, 1984.

» [XdRH97] Xu, Q., de Roever, W y He, J. The rely-guarantee method for verifying shared variable concurrent pograms. Formal Aspects of Computing, 9, 2, 149-174, 1997.