Sincronización Multiproceso en Programas Concurrentes.

Selección Completamente k–justa de Interacciones *

David Ruiz, Rafael Corchuelo, José A. Pérez y Miguel Toro

Universidad de Sevilla
Departamento de Lenguajes y Sistemas Informáticos
Avenida de la Reina Mercedes s/n, Sevilla, 41.012
e-mail: druiz@lsi.us.es

Resumen La selección completamente justa surge en el contexto de los programas no deterministas y sirve para garantizar que todos los elementos que se habilitan infinitamente a menudo se seleccionan infinitamente a menudo. Esta noción de selección presenta dos anomalías: la *finitud justa* y las *conspiraciones*. Este artículo se centra en la selección justa de interacciones y presenta una nueva noción llamada *selección completamente k–justa* cuya principal ventaja sobre otras propuestas es que da solución a las dos anomalías de forma simultánea y que el valor de k se puede establecer a priori para caracterizar su bondad. Para ello, hemos descrito un modelo abstracto de interacción que hace independiente el criterio de selección del lenguaje de programación y que se puede acomodar a gran variedad de modelos de interacción. También presentamos un algoritmo general para implementar la selección completamente k–justa de interacciones que no requiere acceder al estado local de los procesos del sistema.

Palabras clave: programación concurrente y distribuida, modelo abstracto de interacción, acciones conjuntas, eventos compartidos, exclusión mutua, selección justa, conspiraciones.

1 Introducción

La selección justa surge en el contexto de los programas cuya ejecución no es determinista para poder garantizar propiedades de integridad y de viveza [7] en sus ejecuciones. Este factor no determinista puede ser introducido por (i) lenguajes de programación no deterministas, por (ii) el entrelazado de código en los programas concurrentes o por (iii) la configuración de la red en el caso de programas distribuidos.

En la bibliografía no hemos encontrado ninguna definición de justicia que prevalezca sobre las otras, pero en el contexto de los lenguajes de programación

* Este trabajo está subvencionado por la Comisión Interministerial de Ciencia y Tecnología: proyecto GEOZOCO (TIC2000-1106-C02-01)

parece que la *selección completamente justa* es la más adecuada [7] ya que permite garantizar propiedades fundamentales de viveza como son la terminación o la respuesta a eventos. Por definición, la ejecución de un programa es completamente justa cuando los elementos[1] que se habilitan infinitamente a menudo resultan seleccionados infinitamente a menudo.

La figura 1 muestra una solución al problema de los filósofos comensales en IP, en la que, además de garantizar la exclusión mutua de los tenedores compartidos, también podemos garantizar (asumiendo selección completamente justa de interacciones) que todos los filósofos cogen sus dos tenedores para comer infinitamente a menudo.

$S::[||_{i=1}^{N} F_i \ || \ T_i]$

$F_i::*[\text{Coger}_i[] \rightarrow \text{Soltar}_i[]]$

$T_i::*[\ \text{Coger}_i[] \rightarrow \text{Soltar}_i[]$
$\quad [] \ \text{Coger}_{i+1 \bmod N}[] \rightarrow \text{Soltar}_{i+1 \bmod N}[]$
$\quad]$

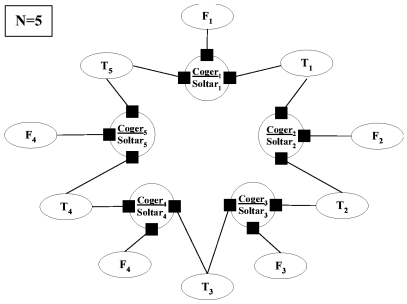

Figura 1. Problema de los filósofos comensales en IP.

El problema que presenta este criterio de selección es que por definición se deduce:

Finitud justa: Que toda ejecución finita de un programa es justa por definición. La figura 2.a muestra una posible ejecución del programa de la figura 1 para $N = 5$. La notación $p.\chi$ indica que el proceso p ha ofrecido el conjunto de interacciones χ, $x[]$ indica que la interacción x se ha ejecutado. El hecho de que para cualquier valor de n finito la ejecución es completamente justa implica que interacciones como $Coger_2$ estén en n ocasiones habilitadas y nunca resulten seleccionadas.

Conspiraciones: Aunque una interacción sea ofrecida por todos sus participantes infinitamente a menudo un entrelazado desafortunado puede provocar que nunca llegue a habilitarse. Esta situación se conoce con el nombre de conspiración y la definición de selección completa no la tiene en cuenta. La figura 2.b muestra un ejemplo de ejecución completamente justa en la que la interacción $Coger_2$ es ofrecida por todos sus participantes infinitamente a menudo pero nunca resulta habilitada.

[1] En nuestro caso los elementos objeto de selección justa son las interacciones entre procesos.

$F_1.\{Coger_1\}$, $F_2.\{Coger_2\}$,
($T_5.\{Coger_5, Coger_1\}$, $T_2.\{Coger_2, Coger_3\}$, $T_1.\{Coger_1, Coger_2\}$, $Coger_1[]$,
$F_1.\{Soltar_1\}$, $T_1.\{Soltar_1, Soltar_2\}$, $T_5.\{Soltar_5, Soltar_1\}$, $Soltar_1[]$, $F_1.\{Coger_1\}$)^n

(a)

$F_1.\{Coger_1\}$, $F_2.\{Coger_2\}F_3.\{Coger_3\}$,
($T_5.\{Coger_5, Coger_1\}$, $T_3.\{Coger_3, Coger_4\}$, $T_1.\{Coger_1, Coger_2\}$, $Coger_1[]$,
$T_2.\{Coger_2, Coger_3\}$, $Coger_3[]$, $F_1.\{Soltar_1\}$, $T_1.\{Soltar_1, Soltar_2\}$,
$T_5.\{Soltar_5, Soltar_1\}$, $Soltar_1[]$, $F_3.\{Soltar_3\}$, $T_2.\{Soltar_2, Soltar_3\}$,
$T_3.\{Soltar_3, Soltar_4\}$, $Soltar_3[]$, $F_1.\{Coger_1\}$,$F_3.\{Coger_3\}$)$^\omega$

(b)

Figura 2. Anomalías de la selección completamente justa de interacciones.

Estas anomalías han sido estudiadas por varios autores dando lugar a nuevos criterios de selección más restrictivos. Entre ellos destacaremos: la selección justa finita [1] y la selección hiperjusta [2].

La selección justa finita intenta paliar el primer problema. Para ello lo que hace es sustituir el termino "infinitamente a menudo" de la definición por "al menos una vez cada k veces" siendo k un natural desconocido a priori. Las principales desventajas de esta propuesta es que (i) el valor de k se conoce a posteriori, (ii) que no resuelve el problema de la *finitud* y las conspiraciones, (iii) no hemos encontrado ningún algoritmo para implementarla[2].

La selección hiperjusta surge como respuesta al problema de las *conspiraciones*. Se dice que una ejecución hiperjusta cuando es finita o toda interacción que puede habilitarse infinitamente a menudo lo hace infinitamente a menudo. Con esta noción se pretende garantizar la habilitación de las interacciones y no la selección de las mismas, por lo que se hace necesario combinarla con otra noción de selección de interacciones. Sus principales desventajas son que (i) no resuelve el problema de la *finitud* y (ii) no se conoce ningún algoritmo general para implementarla[3].

Nosotros proponemos una nueva noción para resolver simultáneamente las dos anomalías presentadas: la selección *completamente k–justa*. Una ejecución será completamente k–justa cuando ninguna interacción se ejecuta más de k veces sin conocer el estado definitivo de las interacciones con las que tiene que obtener la exclusión mutua y además cuando se selecciona es la interacción más antigua del grupo (este concepto lo definiremos más adelante).

Nuestra noción está definida sobre un modelo de interacción abstracto basado en el concepto de acción conjunta que abarca a modelos de interacción de alto nivel (como el de múltiples participantes) hasta modelos de interacción más

[2] Los autores proponen un esquema de transformación que puede ser aplicado a programas expresados con autómatas de Büchi [10,5].

[3] Los autores proponen un esquema de transformación de programas IP que implica modificación del código de los procesos y la existencia de gestores a medida para cada programa.

primitivos como el paso de mensajes o el *rendezvous* [8]. Con este modelo de interacción conseguimos hacer independiente el lenguaje de programación del criterio de selección que se adopte. Por ultimo, presentamos un algoritmo general para implementar nuestra noción que no necesita acceder al estado de los procesos.

2 Modelo abstracto de interacción

Entenderemos que nuestros programas concurrentes y/o distribuidos estarán formado por un conjunto fijo no vacío de procesos y un conjunto fijo no vacío de acciones conjuntas (en adelante *interacciones*) entre dichos procesos. Todo proceso puede participar durante su ejecución en un conjunto fijo no vacío de interacciones.

Los procesos se ejecutarán de forma concurrente (real o simulada) y en cada instante de tiempo sólo podrán ejecutar una interacción, es decir, suponemos que los procesos sólo tienen un único hilo de ejecución. El único mecanismo que se proporciona para sincronizar procesos distintos son las interacciones, es decir, los procesos no comparten variables ni se envían mensajes entre ellos.

Los procesos se pueden encontrar en tres estados distintos:

Realizando cálculos locales: Cuando un proceso se encuentra en dicho estado sólo podrá ejecutar acciones sobre su estado local, es decir, acciones que no requieren interactuar con otros procesos para llevar a cabo ninguna tarea.

Esperando interactuar con otros procesos: Cuando un proceso se encuentra en este estado diremos que está ofreciendo un conjunto de interacciones y que se encuentra bloqueado a la espera de que alguna ellas se habilite y resulte seleccionada para ser ejecutada.

Finalizado: Un proceso se encuentra en este estado cuando termina su ejecución. En realidad lo que a nosotros nos interesa es que un proceso que se encuentra en este estado no puede ofrecer (ni ejecutar) ninguna interacción en el futuro ni tampoco efectuar cálculos locales.

La vida de un proceso puede describirse con la siguiente expresión regular:

$$p :: (p.\iota | p.\chi, x)^*, p.\emptyset \tag{1}$$

Donde $p.\iota$ representa la ejecución de cálculos locales, $p.\chi$ el ofrecimiento de un conjunto $\chi \neq \emptyset$ de interacciones, x la sincronización a través de una interacción $x \in \chi$ y $p.\emptyset$ el evento de terminación de p.

Las interacciones (acciones conjuntas síncronas entre un número arbitrario y fijo de procesos) sólo se ejecutan cuando todos los procesos que pueden ofrecerla lo han hecho. Cuando dos o más interacciones pueden ser ofrecidas por los mismos procesos, es decir, tienen algún participante en común diremos que son potencialmente conflictivas. Cuando dos o más interacciones que son potencialmente conflictivas se encuentran habilitadas el mismo tiempo diremos que se ha producido un conflicto y se tendrá que decidir qué interacción ejecutar y

rechazar el resto, ya que como hemos dicho los procesos sólo tienen un único hilo de ejecución.

De esta forma, entenderemos que la ejecución de un programa es la secuencia de *ofrecimiento de interacción/ejecución de interacción* que provoca el cambio de estado en el mismo. Es decir, asumimos que los cálculos locales que realizan los procesos no son visibles durante la ejecución.

2.1 Descripción detallada

A continuación haremos algunas definiciones para introducir todos estos conceptos de forma precisa. Tomaremos como ejemplo para poder ilustrar estas definiciones el ejemplo de los filósofos comensales visto en la figura 1.

Modelo de los sistemas

Sistema: Un sistema Σ se define como la la tupla (P_Σ, I_Σ). $P_\Sigma \neq \emptyset$ es un conjunto finito de procesos (con un único hilo de ejecución y un estado local independiente) e $I_\Sigma \neq \emptyset$ es un conjunto finito de interacciones que permiten sincronizar y comunicar a un número arbitrario y fijo de procesos de P_Σ.

Participantes de una interacción: Dada interacción $x \in I_\Sigma$ se define un conjunto estático de procesos $\mathbb{P}(x) \subseteq P_\Sigma$ que estará formado por todos los procesos a los que x puede sincronizar.

Interacciones de un proceso: Todo proceso $p \in P_\Sigma$ define un conjunto también estático de interacciones $\mathbb{I}(p) \subseteq I_\Sigma$ de forma que sólo se podrá sincronizar y comunicar con otros procesos a través de estas interacciones.

Configuración: Una configuración es una estructura matemática que denota el estado en el que se encuentra la ejecución un programa en un instante determinado. Generalmente las configuraciones incluyen el estado de los procesos y alguna información de control adicional. Las denotaremos con C, C', C_1, C_2, etc.

Eventos: Las transiciones entre configuraciones de un sistema vienen dadas por eventos. Nuestro modelo define tres eventos: $p.\iota$ cuando el proceso p ejecuta una instrucción local, $p.\chi$ cuando el proceso p ofrece interacción en alguna interacción $x \in \chi$ y x cuando el sistema ejecuta la interacción x.

Ejecución: Se define una ejecución λ de un sistema Σ como la tupla (C_0, α, β), donde C_0 es la configuración inicial, $\alpha = [C_1, C_2, \ldots]$ es una secuencia maximal (finita o infinita) de configuraciones y $\beta = [e_1, e_2, \ldots]$ también es una secuencia maximal (finita o infinita) de eventos, donde e_i caracteriza la transición entre las configuraciones C_{i-1} y C_i $(i \geq 1)$.

Traza: Sea $\lambda = (C_0, \alpha, \beta)$ una ejecución de un programa Σ. Denotaremos α como λ_α (traza de configuraciones) y β como λ_β (traza de eventos).

Semántica: Asumiremos que nuestros programas han sido escritos en un lenguaje L cuya semántica operativa está definida por la regla de transición $\xrightarrow{e}_{\mathrm{L}}$, donde e es un evento del sistema. Dada una ejecución $\lambda = (C_0, \alpha, \beta)$ también la escribiremos de la siguiente forma:

$$C_0 \xrightarrow{e_1}_{\mathrm{L}} C_1 \xrightarrow{e_2}_{\mathrm{L}} C_2 \xrightarrow{e_3}_{\mathrm{L}} \cdots \tag{2}$$

Modelo de los procesos

Ofrecimiento de interacción: Un proceso p estará ofreciendo la interacción x en la configuración i–ésima de su ejecución si o sólo si llega a un punto en el que la ejecución de la interacción x se una posible continuación.

$$\begin{aligned}&\text{Readies}(\lambda, p, x, i) \text{ sii}\\ &\quad \exists 1 \leq k \leq i \cdot (\lambda(k) = p.\chi \wedge x \in \chi \wedge \nexists k < j \leq i \cdot \lambda_\beta(j) = z \wedge z \in \chi)\end{aligned} \tag{3}$$

Esperando interactuar: Un proceso p estará esterando interactuar en la configuración i–ésima de una ejecución λ si y sólo si está ofreciendo alguna interacción.

$$\text{Waiting}(\lambda, p, i) \text{ sii } \exists x \in I_\Sigma \cdot \text{Readies}(\lambda, p, x, i) \tag{4}$$

Proceso finalizado: Un proceso p habrá finalizado su ejecución en la configuración i–ésima de su ejecución λ si y sólo si ya no puede volver a ejecutar ni cálculos locales ni interacción.

$$\text{Finished}(\lambda, p, i) \text{ sii } \exists 1 \leq k \leq i \cdot \lambda_\beta(k) = p.\emptyset \tag{5}$$

Modelo de las interacciones

Interacciones enlazadas: El conjunto de interacción enlazadas a la interacción x en la configuración i–ésima de la ejecución λ estará formado por aquellas interacciones que han sido ofrecidas por procesos comunes.

$$\begin{aligned}&\text{Linked}(\lambda, x, i) =\\ &\quad \{y \in I_\Sigma \mid y \neq x \wedge \exists p \in P_\Sigma \cdot (\text{Readies}(\lambda, p, x, i) \wedge \text{Readies}(\lambda, p, y, i))\}\end{aligned} \tag{6}$$

Interacciones habilitadas: Una interacción x estará habilitada en la configuración i–ésima de la ejecución λ si y sólo si todos los procesos de $\mathbb{P}(x)$ la están ofreciendo.

$$\text{Enabled}(\lambda, x, i) \text{ sii } \forall p \in \mathbb{P}(x) \cdot \text{Readies}(\lambda, p, x, i) \tag{7}$$

Interacciones deshabilitadas: Una interacción x estará deshabilitada en la configuración i–ésima de la ejecución λ si y sólo si no está habilitada y todos los procesos de $\mathbb{P}(x)$ están esperando interactuar o finalizados.

$$\begin{aligned}&\text{Disabled}(\lambda, x, i) \text{ sii}\\ &\quad \neg\, \text{Enabled}(\lambda, x, i) \wedge \forall p \in \mathbb{P}(x) \cdot (\text{Waiting}(\lambda, p, i) \vee \text{Finished}(\lambda, p, i))\end{aligned} \tag{8}$$

Interacciones semihabilitadas: Una interacción x estará semihabilitada en la configuración i–ésima de la ejecución λ si y sólo si no está habilitada y ha sido ofrecida por al menos un proceso.

$$\text{SemiEnabled}(\lambda, x, i) \text{ sii } \neg\, \text{Enabled}(\lambda, x, i) \wedge \exists p \in \mathbb{P}(x) \cdot \text{Readies}(\lambda, x, i) \tag{9}$$

Interacciones consolidadas: Una interacción x estará consolidada en la configuración i–ésima de la ejecución λ si y sólo si está habilitada o deshabilitada.

$$\text{Stable}(\lambda, x, i) \text{ sii } \text{Enabled}(\lambda, x, i) \vee \text{Disabled}(\lambda, x, i) \tag{10}$$

Ejecuciones de interacción: Se define el conjunto de ejecuciones de una interacción x en la configuración i–ésima de la ejecución λ como las posiciones de β en las que la interacción x ha sido ejecutada.

$$\mathrm{ExeSet}(\lambda, x, i) = \{1 \leq k \leq i \mid \lambda_\beta(k) = x\} \tag{11}$$

Edad de una interacción: Se define la edad de una interacción x en la configuración i–ésima de su ejecución λ como el número de configuraciones que hay desde la última ejecución de x o ∞ si la interacción nunca se ha ejecutado.

$$\mathrm{Age}(\lambda, x, i) = \begin{cases} i - \max \mathrm{ExeSet}(\lambda, x, i) & \text{si } \mathrm{ExeSet}(\lambda, x, i) \neq \emptyset \\ \infty & \text{en otro caso} \end{cases} \tag{12}$$

Nuestro modelo abstracto está compuesto por dos módulos cuya relación se muestra en la figura 3. El programa informa de los ofrecimientos al módulo de selección de interacciones, que se encargará de detectar cuándo una interacción se habilita y debe ser ejecutada por el programa.

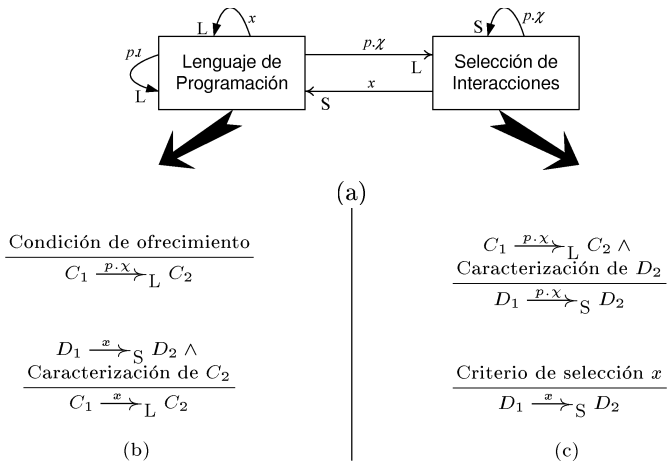

Figura 3. Modelo abstracto: (a) Relación entre el programa y el algoritmo de selección de interacciones. (b) Esquema abstracto de programa (c) Esquema abstracto de algoritmo de selección.

Está relación puede modelarse haciendo uso de reglas de inferencia. La figura 3.b muestra el aspecto que tendrían las reglas que describen el comportamiento de los programas ($\longrightarrow_{\mathrm{I}}$) y la figura 3.c el algoritmo de selección de interacciones ($\longrightarrow_{\mathrm{S}}$). Como se observa, $\xrightarrow{x}_{\mathrm{S}}$ aparece en el antecedente de la regla de inferencia del programa. Esto implica que la transición de ejecución de interacción viene dada por el algoritmo de selección de interacciones al programa. Asimismo, $\xrightarrow{p.\chi}_{\mathrm{I}}$ aparece en el antecedente de la regla de inferencia del algoritmo de selección por lo que la transición de ofrecimiento de interacción viene dada por el programa.

Hay que destacar que este modelo abstracto de interacción sólo muestra el aspecto que deben tener las reglas de inferencia que lo describen, y que para valores concretos de $\longrightarrow_{\mathrm{L}}$ y $\longrightarrow_{\mathrm{S}}$ tendremos distintos casos de modelos de interacción.

La figura 4 muestra dos reglas de inferencia que caracterizan (desde un punto de vista abstracto) a nuestros programas. La regla 13 describe la semántica de un ofrecimiento por parte de un proceso p. El consecuente solo exige que la transición venga dada por un evento de ofrecimiento. Por otro lado, el antecedente obliga a que para que un proceso p pueda ofrecer interacción primero llegue a un punto de su ejecución en la que ya no puede ejecutar cálculos locales, y además, para poder volver a ofrecer interacción o ejecutar cálculos locales ($\omega \in \{p.\iota, p.\chi\}$) es necesario que el programa ejecute una interacción en la que p sea participante. La regla 14 captura la semántica de la terminación de un proceso p. La transición $\longrightarrow_{\mathrm{L}}$ del consecuente esta caracterizada por el evento de terminación. Con el antecedente expresamos que cuando un proceso p termina su ejecución ya no volverá a ejecutar ni cálculos locales ni a ofrecer interacciones ($\omega \in \{p.\iota, p.\chi\}$).

$$\frac{\begin{array}{c}\chi \subseteq \mathbb{I}(p) \wedge \chi \neq \emptyset \wedge (\nexists C' \cdot C_1 \xrightarrow{p.\iota}_{\mathrm{L}} C') \wedge \\ \exists C', C'', C''', C_3 \cdot (C_2 \longrightarrow_{\mathrm{L}}^{*} C' \xrightarrow{\omega}_{\mathrm{L}} C_3 \Rightarrow C_2 \longrightarrow_{\mathrm{L}}^{*} C'' \xrightarrow{x}_{\mathrm{L}} C''' \longrightarrow_{\mathrm{L}}^{*} C_3 \wedge x \in \chi)\end{array}}{C_1 \xrightarrow{p.\chi}_{\mathrm{L}} C_2} \quad (13)$$

$$\frac{\nexists C', C'' \cdot C_2 \longrightarrow_{\mathrm{L}}^{*} C' \xrightarrow{\omega}_{\mathrm{L}} C''}{C_1 \xrightarrow{p.\emptyset}_{\mathrm{L}} C_2} \quad (14)$$

Figura 4. Modelo abstracto de interacción entre procesos de un programa

En el caso del criterio de selección podemos encontrar en la bibliografía varios propuestas de selección de interacciones. De forma abstracta, todos ellos se comportan según el modelo de la figura 3, lo único que cambia entre ellos es la condición de selección que cada uno adopta. Por ejemplo, en [3] se reciben los ofrecimientos y tan pronto como una interacción habilitada consigue la exclusión mutua se selecciona (no tiene en cuenta la selección justa). En [4] se trata el problema de la selección justa de alternativas en instrucciones de selección múltiple no deterministas seleccionando aquella alternativa que hace más tiempo que no se ejecuta haciendo uso de contadores de selección. En [9] se reciben ofrecimientos aleatorios y se selecciona aquella interacción que se habilita primero (se apoya en la *Teoría de los Grandes Números* para garantizar la selección justa de interacciones). En [6] se espera a recibir todos los ofrecimientos y selecciona aquella interacción que puede ser rechazada en menor número de ocasiones haciendo uso de contadores de rechazo.

3 Selección completamente k–justa de interacciones

De forma intuitiva entenderemos por ejecución completamente k–justa aquella que garantiza que ninguna interacción se ejecuta más de k veces sin que las interacciones con las que está enlazada en ese momento estén consolidadas, además, en caso de conflicto debe seleccionarse la interacción de mayor edad. Formalmente la definimos de la siguiente forma:

Definición 1 (Selección Completamente k–Justa) *Sea una ejecución $\lambda = (C_0, \alpha, \beta)$ y k un natural no nulo. Diremos que λ es una ejecución completamente k–justa si el predicado $\mathcal{SKF}(\lambda, k)$ se satisface.*

$$\begin{array}{l}\mathcal{SKF}(\lambda, k) \textit{ sii} \\ \quad \forall x \in I_\Sigma, i \in \mathrm{ExeSet}(\lambda, x, \infty) \cdot \mathrm{Enabled}(\lambda, x, i) \wedge \\ \quad\quad (\mathrm{LStable}(\lambda, x, i) \wedge \mathrm{LOldest}(\lambda, x, i) \vee \neg\, \mathrm{LStable}(\lambda, x, i) \wedge \Delta(\lambda, x, i) \leq k)\end{array} \tag{15}$$

Esta definición hace uso de varias funciones y predicados auxiliares que detallaremos a continuación. $\mathrm{LStable}(\lambda, x, i)$ es un predicado que utilizamos para saber si las interacciones enlazadas con x en la configuración i–ésima están en un estado consolidado.

$$\mathrm{LStable}(\lambda, x, i) \text{ sii } \forall y \in \mathrm{Linked}(\lambda, x, i) \cdot \mathrm{Stable}(\lambda, y, i) \tag{16}$$

$\mathrm{LOldest}(\lambda, x, i)$ es un predicado que se satisface cuando la interacción x es la de mayor edad en la configuración i–ésima de la ejecución λ de entre todas las interacciones que están enlazadas con ella.

$$\mathrm{LOldest}(\lambda, x, i) \text{ sii } \forall y \in \mathrm{Linked}(\lambda, x, i) \cdot \mathrm{Age}(\lambda, x, i) \geq \mathrm{Age}(\lambda, y, i) \tag{17}$$

$\Delta(\lambda, x, i)$ es una función que devuelve el número de veces que la interacción x se ha ejecutado en presencia de interacciones enlazadas no consolidadas desde la última configuración que se ejecutó hasta la configuración i.

$$\Delta(\lambda, x, i) = \sum_{\phi \leq k < i} (\lambda_\beta(k) = x \wedge \mathrm{Linked}(\lambda, x, k) \neq \emptyset \wedge \neg\, \mathrm{LStable}(\lambda, x, k)) \tag{18}$$

donde $\sum_{a \in A} P(a) \triangleq |\{a \in A \mid P(a)\}|)$, y ϕ se define de la siguiente forma (observe que el máximo de un conjunto vacío es $\perp$):

$$\phi \triangleq \begin{cases} j \text{ if } j = \max\{k \in \mathrm{ExeSet}(\lambda, x, i) \mid \mathrm{LStable}(\lambda, x, k)\} \wedge j \neq \perp \\ 1 \text{ en otro caso} \end{cases} \tag{19}$$

De la definición se deduce que toda ejecución que contenga menos de k ejecuciones de interacción es completamente k–justa (ya que ninguna interacción se ha podido ejecutar más de k veces) y que además k debe ser conocido a priori[4] y debe verificar que:

$$k \geq \max\{|\,\mathbb{I}(p) \cap \mathbb{I}(q)\,| \cdot p, q \in P_\Sigma \wedge p \neq q\} \tag{20}$$

[4] Observe que k caracteriza a la noción de selección completamente k–justa, es decir, es un dato de partida. En la noción de selección justa finita el valor de k se calcula a posteriori sobre la propia ejecución.

Esto es así porque en un grupo de N interacciones enlazadas tenemos que garantizar que, en el peor de los casos (cuando el grupo se consolida y todas se habilitan), al menos toda interacción se ejecuta 1 vez cada N ejecuciones de interacciones enlazadas, es decir $k = N$. Para poder garantizar esto en todos los grupos de interacciones enlazadas lo que tenemos que hacer es dar a k un valor mayor o igual que el máximo de interacciones enlazadas, lo que se obtiene calculando el numero máximo de interacciones comunes que tienen los procesos del sistema dos a dos.

El valor de k puede verse como un umbral de semihabilitaciones en nuestras ejecuciones. Es decir, podemos entender que k es el numero máximo de veces que una interacción se puede seleccionar en presencia de interacciones enlazadas que se encuentran semihabilitadas.

Por otro lado, las ejecuciones completamente k–justas de un programa pueden verse como un subconjunto de las ejecuciones completamente justa a las que en el criterio de selección se les ha añadido una condición en función de k. Así si k es mínimo tenemos que el criterio de selección de interacciones es muy exigente de forma el conjunto de interacciones que lo cumplen es mínimo, por lo que el subconjunto de las posibles ejecuciones completamente k–justa también es mínimo. Por el contrario, si k tiende a infinito el criterio de selección es muy relajado (de hecho solo se exige que las interacciones se encuentren habilitadas), por lo que el conjunto de posibles ejecuciones es máximo (igual al de las ejecuciones completamente justa).

Para ver cómo con la selección completamente k–justa de interacciones detectamos y resolvemos las anomalías de la selección completamente justa analicemos las ejecuciones de las figuras 2.a. En la primera se observa como dicha ejecución no seria completamente k–justa para cualquier k menor que n ya que Get_1 se ejecuta n veces cuando Get_2 se encuentra semihabilitada. En la segunda ejecución (figura 2.b) también tenemos que no seria completamente k–justa para cualquier k menor que n por el mismo motivo con la diferencia de que en la configuración $k+1$ la interacción Get_1 no cumpliría el criterio de selección permitiendo de esta forma que Get_2 se habilite y sea seleccionada.

4 Algoritmo de selección de interacciones completamente k–justo

A continuación vamos a describir en detalle un algoritmo que a partir de cualquier programa cuyo modelo abstracto de interacción sea el descrito en la sección 2 obtenga ejecuciones que sean completamente k–justas para un valor de k dado.

Primero haremos una descripción informal a grandes rasgos del algoritmo. Después haremos una descripción más detallada haciendo uso de reglas de inferencia para describir el cambio de estado (configuración) que provocan los distintos eventos.

4.1 Descripción informal

La idea del algoritmo consiste en ordenar todas las interacciones en una cola (τ) de forma que las interacciones que están al final son las que se han ejecutado más recientemente (seleccionar la primera interacción habilitada de una cola ordenada garantiza que nuestras ejecuciones sean completamente justas [6]). Además a cada interacción x se le asocia un contador ($\delta(x)$) que sirve para llevar la cuenta del número de veces que otras interacciones enlazadas con x se han ejecutado cuando el grupo no estaba consolidado.

Para conocer el estado de los procesos y las interacciones utilizaremos un mapa de ofrecimientos (φ), de forma que cada interacción tiene asociado en tiempo de ejecución cuáles son los procesos que la han ofrecido. Para ello cada vez que se produce una transición de ofrecimiento/interacción se actualiza dicho mapa de la forma adecuada.

Así, nuestro algoritmo siempre seleccionará para su ejecución aquella interacción habilitada que se encuentre primero en la cola y que comparta procesos con un conjunto de interacciones consolidado o que el valor de δ de ninguna de ellas haya superado el valor de k.

4.2 Descripción detallada

Vamos a describir la semántica operativa de nuestro algoritmo de selección completamente k–justo haciendo uso de la regla de transición $\longrightarrow_{SK\mathcal{F}}$ sobre configuraciones que denotaremos como $D, D', D_1, \ldots$ Dichas configuraciones estarán compuestas por la configuración C del programa más las estructuras de datos necesarias para llevar a cabo la selección.

A continuación definiremos las estructuras de datos necesarias así como las funciones que las actualizan.

Estructuras de datos

Mapa de ofrecimientos: Definimos el mapa φ de forma que a cada interacción $x \in I_\Sigma$ se le asocia un conjunto de procesos $\varphi(x)$.

$$\varphi \in \{f : I_\Sigma \longrightarrow 2^{P_\Sigma} \wedge \operatorname{dom} f = I_\Sigma\} \tag{21}$$

Este mapa sirve para disponer en tiempo de ejecución de cuáles son los procesos que han ofrecido cada interacción. De esta forma, podremos saber cuándo una interacción se habilita o cuándo comparte procesos con otras interacciones.

Procesos que finalizados: Se define un conjunto de procesos $\vartheta \subseteq P_\Sigma$ que contendrá a todos los procesos que han finalizado su ejecución. En este conjunto mantendremos todos los procesos que han terminado su ejecución de forma que se podrá detectar cuándo un proceso ha terminado su ejecución o cuando una interacción no volverá jamás a habilitarse.

Mapa de semihabilitaciones: Definimos un mapa δ de forma que a cada interacción se le asocia un número natural.

$$\delta \in \{f : I_\Sigma \longrightarrow \mathbb{N} \wedge \operatorname{dom} f = I_\Sigma\} \tag{22}$$

En este mapa mantendremos información acerca del número de veces que una interacción ha sido rechazada en presencia de otras interacciones con las que compartía algún proceso. De esta forma podremos saber cuándo dentro de un grupo de interacciones que comparten procesos alguna interacción puede estar siendo marginada.

Cola de interacciones: Se define la secuencia de interacciones τ de forma que las interacciones que se encuentran en las primeras posiciones son aquéllas que hace más tiempo que no se ejecutan.

$$\tau = [x_1, \ldots, x_n] \wedge n = |I_\Sigma| \wedge \operatorname{img} \tau = I_\Sigma \tag{23}$$

Configuración extendida: La configuración extendida del algoritmo en un instante i viene dada por la tupla $(C_i, \tau_i, \varphi_i, \delta_i, \vartheta_i)$, donde C_i es la configuración del programa en el instante i, τ_i es la cola de interacciones, φ_i es el mapa de ofrecimientos, δ_i es el mapa de semihabilitaciones y ϑ_i es el conjunto de procesos finalizados.

La configuración inicial del algoritmo viene determinada por la configuración inicial del programa, cualquier cola de interacciones (no importa el orden), un mapa de ofrecimientos φ_0 de forma que $\forall x \in \operatorname{dom} I_\Sigma \cdot \varphi_0(x) = \emptyset$, un mapa de semihabilitaciones δ_0 de forma que $\forall x \in I_\Sigma \cdot \delta_0(x) = 0$ y una secuencia de procesos finalizados $\vartheta_0 = \emptyset$.

Funciones de consulta y actualización de las estructuras de datos

Actualización de φ y ϑ: Cuando un proceso p decide interactuar a través de conjunto de interacciones $\chi \subseteq I_\Sigma$ utilizaremos la función $\operatorname{AddOffer}(\varphi, p, \chi)$ para crear un nuevo mapa de ofrecimientos actualizado de la siguiente forma:

$$\begin{aligned} \operatorname{AddOffer}(\varphi, p, \chi) = &\{x \mapsto \varphi(x) \cdot x \in \operatorname{dom}\varphi \wedge x \notin \chi\} \cup \\ &\{x \mapsto \varphi(x) \cup \{p\} \cdot x \in \operatorname{dom}\varphi \wedge x \in \chi\} \end{aligned} \tag{24}$$

Cuando una interacción x se selecciona para ser ejecutada será la función $\operatorname{RemoveOffer}(\varphi, x)$ la que se encargue de crear un nuevo mapa de ofrecimientos de la siguiente manera:

$$\operatorname{RemoveOffer}(\varphi, x) = \{x \mapsto \varphi(x) \setminus \mathbb{P}(x) \cdot x \in \operatorname{dom}\varphi\} \tag{25}$$

Cuando un proceso p finaliza su ejecución y ofrece el conjunto de interacciones χ vacío $(p.\emptyset)$ la función $\operatorname{AddFinished}(\vartheta, p)$ crea un nuevo conjunto de procesos finalizados en el que incluye a p.

$$\operatorname{AddFinished}(\vartheta, p) = \begin{cases} \vartheta \cup \{p\} & \text{si } \chi \neq \emptyset \\ \vartheta & \text{si } \chi = \emptyset \end{cases} \tag{26}$$

Actualización de τ: Cuando una interacción x resulta seleccionada para ser ejecutada usaremos la función Order para retrasar la posición de x en τ.

$$\begin{aligned}\mathrm{Order}(\tau,\delta) = \tau' \text{ si } &\mathrm{dom}\,\tau = \mathrm{dom}\,\tau' \wedge \mathrm{ran}\,\tau = \mathrm{ran}\,\tau' \wedge \\ &\forall x_1, x_2 \in \mathrm{ran}\,\tau \cdot \tau'^{-1}(x_1) \leq \tau'^{-1}(x_2) \Rightarrow \delta(x_1) \geq \delta(x_2)\end{aligned} \tag{27}$$

Interacciones habilitadas disyuntas: Dada una cola de interacciones τ y un mapa de ofrecimientos φ se define el conjunto de todas las interacciones habilitadas disyuntas $\mathrm{Ready}(\tau,\varphi)$ de la siguiente forma:

$$\mathrm{Ready}(\tau,\varphi) = \{x \in \mathrm{dom}\,\varphi \cdot \mathbb{P}(x) = \varphi(x) \wedge \nexists y \in \mathcal{S} \cdot \mathbb{P}(y) = \varphi(y) \wedge \tau^{-1}(y) < \tau^{-1}(x)\} \tag{28}$$

Siendo $\mathcal{S} = \{z \in \mathrm{dom}\,\varphi \cdot \varphi(z) \cap \varphi(x) \neq \emptyset\}$.

Interacciones consolidadas: Dado un conjunto de interacciones Υ, un mapa de ofrecimientos φ y un conjunto de procesos finalizados ϑ definimos el predicado $\mathrm{Consolidated}(\Upsilon,\varphi,\vartheta)$ que se verifica cuando todos los procesos de todas las interacciones de Υ han ofrecido interacción[5].

$$\mathrm{Consolidated}(\Upsilon,\varphi,\vartheta) \text{ sii } \forall x \in \Upsilon \cdot \forall p \in \mathbb{P}(x) \cdot p \in \mathrm{ran}\,\varphi \vee p \in \vartheta \tag{29}$$

Actualización de δ: Cuando una interacción x es seleccionada para su ejecución utilizaremos la función $\mathrm{Update}(\varphi,\delta.x)$ para crear un nuevo mapa de semihabilitaciones de la siguiente forma:

$$\mathrm{Update}(\varphi,\delta.x) = \delta \otimes \{x \mapsto 0\} \otimes \{y \mapsto \delta(y)+1 \cdot y \in \mathcal{S} \setminus \{x\} \wedge \neg\,\mathrm{Offered}(\mathcal{S},\varphi,\vartheta)\} \tag{30}$$

Definiéndose $\mathcal{S}$ igual que en la función $\mathrm{Ready}(\tau,\varphi)$.

Hemos descrito el algoritmo de selección completamente k–justo de interacción haciendo uso de las dos reglas de inferencia que se muestran en la figura 5.

En la regla 31 describimos cómo cambian las estructuras de datos cuando se produce una transición de ofrecimiento, es decir, cada vez que un proceso p ofrece el conjunto de interacciones χ nuestro algoritmo añade dicho ofrecimiento en el mapa de ofrecimientos. En el caso de que un proceso p finalice lo que hacemos es añadir dicho proceso al conjunto de procesos finalizados.

En la regla 32 se describe qué interacción debe seleccionar para que la ejecución que se genere sea completamente k–justa (obsérvese que $C_1 \xrightarrow{x}_{\mathrm{L}} C_2$ aparece en el consecuente). El antecedente de esta regla asume que x es la primera interacción habilitada de la cola τ, de forma que el criterio de selección se satisface si se da alguna de las siguiente condiciones:

1. Si x no es conflictiva con ninguna interacción, es decir, $\mathcal{S} = \{x\}$.
2. Las interacciones con las que x está enlazada están consolidadas.
3. El contador de semihabilitaciones asociado de todas las interacciones enlazadas con x es menor que k.

[5] Fíjese en que consolidación no implica habilitación pero en cambio habilitación si implica consolidación.

$$\frac{\begin{array}{c} C \xrightarrow{p.\chi}_{\mathrm{L}} C' \wedge \\ \varphi' = \mathrm{AddOffer}(\varphi, p, \chi) \wedge \vartheta' = \mathrm{AddFinished}(\vartheta, p) \end{array}}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{p.\chi}_{\mathcal{SKF}} (C', \tau, \varphi'\delta, \vartheta')} \tag{31}$$

$$\frac{\begin{array}{c} x \in \mathrm{Ready}(\tau, \varphi) \wedge \mathcal{S} = \{z \in \mathrm{dom}\,\varphi \cdot (\varphi(z) \cap \varphi(x) \neq \emptyset)\} \wedge \\ \tau' = \mathrm{Order}(\tau, \delta') \wedge \varphi' = \mathrm{RemoveOffer}(\varphi, x) \wedge \delta' = \mathrm{Update}(\varphi, \delta.x) \wedge \\ (\mathcal{S} = \{x\} \vee \mathrm{Offered}(\mathcal{S}, \varphi, \vartheta) \vee (\mathcal{S} \neq \{x\} \wedge \max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k)) \end{array}}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{x}_{\mathcal{SKF}} (C', \tau', \varphi', \delta', \vartheta) \wedge C \xrightarrow{x}_{\mathrm{L}} C'} \tag{32}$$

Figura 5. Algoritmo de selección $\mathcal{SKF}$.

5 Conclusiones

En este artículo hemos presentamos un nuevo criterio de selección de interacciones para poder garantizar propiedades fundamentales en programas cuyo comportamiento es no determinista que con otros criterios resultan imposibles. Este criterio de selección de interacciones resuelve las anomalías de la selección completamente justa mostradas en la figura 2.a y 2.b y sus principales ventajas son:

- No asume que cualquier ejecución finita sea justa por definición, lo que nos puede llevar a ejecuciones como las mostradas en 2.a.
- El valor de k se ajusta empíricamente a priori para cada programa. Dicho valor caracteriza las ejecuciones que cumplen el criterio de selección.
- Garantiza la ejecución de todas las interacciones que pueden habilitarse en un tiempo finito y acotado superiormente (el valor de dicha cota varía en función de k).
- Resuelve el problema de las conspiraciones con una bondad que aumenta conforme disminuye el valor de k.
- Proporcionamos un algoritmo para implementarla que no requiere acceder al estado local de los procesos.

Una de las aportaciones más importantes es el concepto de umbral de semihabilitaciones k, que utilizamos para detectar las posibles situaciones de conspiración y resolverlas a tiempo. Este umbral caracteriza la ejecución de nuestros programas de forma que nos sirve para regular la velocidad a la que se ejecutarán las interacciones que comparten procesos en tiempo de ejecución. Si k es mínimo entonces las interacciones enlazadas se ejecutarán a la velocidad del participante más lento, ya que en este caso ninguna interacción se selecciona hasta conocer el estado final de todas las interacciones con las que comparte ofrecimientos. En este contexto nuestro algoritmo es similar a [4] resolviendo el problema de las conspiraciones a costa de ralentizar los procesos que componen un programa concurrente. Si k es máximo (tiende a infinito) entonces las interacciones enlazadas se ejecutan a la velocidad del participante más rápido, ya que en este caso

toda interacción se selecciona tan pronto como obtiene la exclusión mutua con las que es conflictiva. En este contexto nuestro algoritmo se parece a [3] y [11], en los que no se tiene en cuenta el problema de la selección justa de interacciones pero se consigue ejecutar un gran número de interacciones por segundo.

Referencias

1. R. Alur and T. A. Henzinger. Finitary fairness. *ACM Transactions on Programming Languages and Systems*, 20(6):1171–1194, November 1998.
2. P.C. Attie, N. Francez, and O. Grumberg. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.
3. R.L. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.
4. E. Best. Fairness and conspiracies. *Information Processing Letters*, 18(4):215–220, 1984.
5. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.
6. R. Corchuelo, D. Ruiz, M. Toro, and A. Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the XXV^{th} Euromicro Conference (Workshop on Network Computing)*, Milan (Italy), September 1999. IEEE Press.
7. N. Francez. *Fairness*. Springer–Verlag, 1986.
8. N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.
9. Y.J. Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science*, 243(1–2):307–338, 2000.
10. E. Olderog and K.R. Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10(3):420–455, July 1988.
11. J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. A framework for aspect–oriented multiparty coordination. In Kluwer, editor, *Proceedings of the Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Krakow, Poland, September 2001. To appear soon.