REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 3, número 1
Págs. 73-92

Vivacidad y Justicia en Entornos no Deterministas*

David Ruiz† Rafael Corchuelo‡ Antonio Ruiz–Cortés§

Resumen

El problema de la vivacidad y la selección justa surge en el contexto de los sistemas con ejecuciones no deterministas. El concepto de *selección completamente justa* sirve para garantizar que todos los elementos que se habilitan infinitamente a menudo se ejecutan 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 en sistemas basados en *interacciones entre múltiples participantes* 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. Para ello, hemos descrito un marco de trabajo teórico para caracterizar los sistemas basados en interacciones entre múltiples participantes que hace independiente el criterio de selección del lenguaje de programación. 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: *sistemas distribuidos, interacciones entre múltiples participantes, vivacidad, selección justa.*

Abstract

Liveliness and fairness issues emerge in the context of systems in which the executions of a programme are non–deterministic. The concept of *strong fairness* ensures that all of the elements of a programme that become enabled infinitely often are selected for execution infinitely often. This notion suffers from two anomalies, namely: *fair finiteness* and *conspiracies*. This article focuses on fair selection of interactions in distributed systems based on *multiparty interactions*, and introduces a new notion called *strong k–fairness* that improves current notions in that it addresses both anomalies simultaneously. We describe a theoretical framework to characterise systems based on multiparty interactions that abstracts the selection criterion from the programming language used. We also present a general algorithm to implement our notion that does not require to have access to the local state of the processes of which a system is composed.

Keywords: *Distributed systems, multiparty interactions, liveliness, fairness*.

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

†Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: druiz@lsi.us.es

‡Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: corchu@lsi.us.es

§Departamento de Lenguajes y Sistemas Informáticos. Escuela Técnica Superior de Ingenieros Informáticos. Universidad de Sevilla, España. e–mail: aruiz@lsi.us.es

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 [11]. Este factor no determinista puede ser introducido por: (i) lenguajes de programación no deterministas, (ii) el entrelazado de código en los programas concurrentes o (iii) la configuración de la red en el caso de programas distribuidos. En este artículo nos centramos en el no determinismo introducido por lenguajes como IP (*Interacting Processes*) [12], Raddle [10], Scripts [11], Unity [7], CAL (*Coordination Aspect Language*) [8] que hacen uso del modelo de interacción entre múltiples participantes.

1.1 El modelo de interacción entre múltiples participantes

Cuando la descripción del comportamiento de un sistema obliga a que más de dos procesos colaboren simultáneamente de forma síncrona, las primitivas clásicas de comunicación como el paso de mensajes, el *rendez–vous* o la llamada a procedimientos remotos no son el mecanismo más adecuado para hacerlo puesto que en ocasiones la solución resulta en exceso sofisticada y en absoluto intuitiva. Por esta razón, muchos autores han considerado adecuado introducir mecanismos que permitan la interacción simultánea entre múltiples participantes [7, 8, 10, 11, 12, 18].

Un ejemplo representativo de interacción entre múltiples procesos es el conocido problema de los filósofos comensales, para el que Tanenbaum [21], por ejemplo, propone varias soluciones haciendo uso de primitivas clásicas con las que es preciso diseñar mecanismos complejos para evitar comportamientos indeseables como son los abrazos mortales o la inanición. Este problema se puede resolver en IP de manera más simple haciendo uso de interacciones tripartitas, como se muestra en la figura 1, en las que cada filósofo toma *simultáneamente* los dos tenedores que tiene asignados, en vez de tomar primero uno y después otro.

{Se considera que la aritmética de subíndices es módulo N}

S::[||$_{i=1}^{N}$ F$_i$ || T$_i$]

F$_i$::*[Coger$_i$[] → Soltar$_i$[]]

T$_i$::*[Coger$_i$[] → Soltar$_i$[]
[] Coger$_{i+1}$[] → Soltar$_{i+1}$[]
]

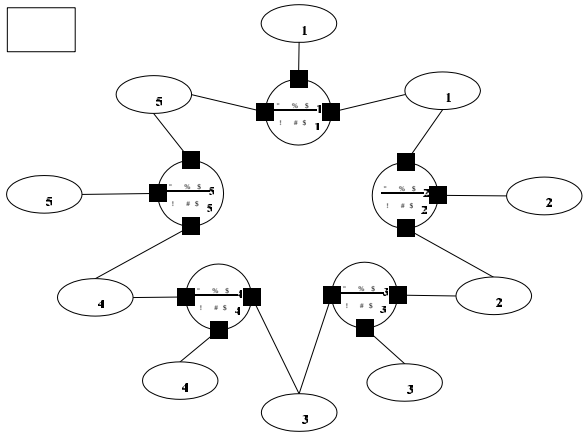


Figura 1: Problema de los filósofos comensales en IP.

Asumimos que el lector está familiarizado con el lenguaje IP, en cualquier caso haremos una pequeña introducción a los conceptos fundamentales. Para saber más sobre este lenguaje diríjase a [12].

Un sistema distribuido en IP se entiende como un conjunto de procesos secuenciales cuyo único mecanismo de sincronización y comunicación son las interacciones entre múltiples participantes.

Las instrucciones de interacción tienen la forma $a[\overline{x{:=}e}]$, donde a es el nombre de la interacción y $\overline{x{:=}e}$ es una asignación opcional que constituye su cuerpo. IP proporciona instrucciones alternativas múltiples de la forma $[[]_{i=1}^{n} G_i \rightarrow S_i]$ e instrucciones iterativas de la forma $*[[]_{i=1}^{n} G_i \rightarrow S_i]$, siendo ambas no deterministas. Los guardas G_i están formados por expresiones lógicas sobre el estado de los procesos y, opcionalmente, por instrucciones de interacción. Así, un guarda se encuentra habilitado para ejecutar su bloque de instrucciones S_i cuando la expresión lógica se evalúa a cierto y todos los procesos que participan en a han llegado a un punto en su ejecución en el que pueden ejecutar la instrucción de interacción a.

1.2 El problema de la selección justa y la vivacidad

De forma intuitiva, una ejecución de un programa no determinista es justa cuando los elementos que se encuentran suficientemente a menudo habilitados se ejecutan suficientemente a menudo. Dependiendo de la definición que hagamos del término suficientemente a menudo tendremos distintos niveles de justicia [5].

En la bibliografía no hemos encontrado ninguna definición de justicia que prevalezca sobre las otras, pero en el contexto de los lenguajes de programación parece que la *selección completamente justa* es la más adecuada [12] 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.

El criterio de selección completamente justo presenta dos anomalías que provocan que ejecuciones completamente justas (aplicando la definición) no sean intuitivamente vivaces:

Finitud justa: 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 sea completamente justa implica que interacciones como $Tomar_2$ 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 completamente justa no la tiene en cuenta. La figura 2.(b) muestra un ejemplo de ejecución completamente justa en la que la interacción $Tomar_2$ es ofrecida por todos sus participantes infinitamente a menudo pero nunca resulta habilitada.

1.3 Nuestra solución

Estas anomalías han sido estudiadas por separado por varios autores dando lugar a nuevos criterios de selección más restrictivos. Entre ellos destacaremos la selección justa finita (*finitary fairness*) [1] y la selección hiperjusta (*hyperfairness*) [2], que serán estudiados en detalle en la sección 6. Nosotros proponemos una nueva noción para resolver simultáneamente las dos anomalías: la selección *completamente k–justa*. Una ejecución será completamente k–justa cuando ninguna interacción se

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

$F_1.\{Tomar_1\}$, $F_2.\{Tomar_2\}$,
($T_5.\{Tomar_5, Tomar_1\}$, $T_2.\{Tomar_2, Tomar_3\}$, $T_1.\{Tomar_1, Tomar_2\}$, $Tomar_1$,
$F_1.\{Soltar_1\}$, $T_1.\{Soltar_1, Soltar_2\}$, $T_5.\{Soltar_5, Soltar_1\}$, $Soltar_1$, $F_1.\{Tomar_1\}$)^n

(a)

$F_1.\{Tomar_1\}$, $F_2.\{Tomar_2\}$, $F_3.\{Tomar_3\}$,
($T_5.\{Tomar_5, Tomar_1\}$, $T_3.\{Tomar_3, Tomar_4\}$, $T_1.\{Tomar_1, Tomar_2\}$, $Tomar_1$,
$T_2.\{Tomar_2, Tomar_3\}$, $Tomar_3$, $F_1.\{Soltar_1\}$, $T_1.\{Soltar_1, Soltar_2\}$,
$T_5.\{Soltar_5, Soltar_1\}$, $Soltar_1$, $F_3.\{Soltar_3\}$, $T_2.\{Soltar_2, Soltar_3\}$,
$T_3.\{Soltar_3, Soltar_4\}$, $Soltar_3$, $F_1.\{Tomar_1\}$,$F_3.\{Tomar_3\}$)$^\omega$

(b)

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

ejecute 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 que hace más tiempo que no se ejecuta del grupo (este concepto lo definiremos más adelante).

1.4 Estructura

El artículo está organizado de la siguiente forma: en la sección 2 hacemos una caracterización de los sistemas basados en interacciones entre múltiples participantes. Esta caracterización nos sirve de base para, en la sección 3, poder definir formalmente la selección completamente k–justa de interacciones. En la sección 4 describimos detalladamente un algoritmo para implementar nuestra noción y demostramos su coherencia. En la sección 5 discutimos sobre unos resultados de imposibilidad en la implementación de algoritmos de selección de interacciones. Por último, en las secciones 6 y 7 mostramos el trabajo relacionado y nuestras conclusiones más importantes.

2 Modelo abstracto de interacción

En esta sección presentamos un modelo abstracto para caracterizar los sistemas basados en interacciones entre múltiples participantes. Primero haremos una descripción intuitiva del modelo y después haremos una descripción más rigurosa ilustrando los conceptos más importantes con el ejemplo de los filósofos comensales.

2.1 Descripción intuitiva

Entendemos que nuestros programas concurrentes y/o distribuidos estarán formados por un conjunto fijo no vacío de procesos y un conjunto fijo no vacío de 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 hilo de ejecución. El único mecanismo que se proporciona para sincronizar procesos distintos son las interacciones, es decir, 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 requieran 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.

Las interacciones entre múltiples participantes pueden verse como acciones conjuntas síncronas entre un número arbitrario y fijo de procesos que 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 al 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 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.

2.2 Descripción rigurosa

A continuación presentaremos algunas definiciones para introducir todos estos conceptos de forma rigurosa. Tomaremos como ejemplo para poder ilustrar estas definiciones los filósofos comensales de la figura 1.

Definición 1 (Caracterización estática del sistema) *Un sistema Σ es una tupla (P_Σ, I_Σ), donde $P_\Sigma \neq \emptyset$ es un conjunto finito de procesos 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. Toda interacción $x \in I_\Sigma$ 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. Todo proceso $p \in P_\Sigma$ define un conjunto 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.*

El estado en el que se encuentra la ejecución de un programa en un instante determinado es capturado por su configuración. Generalmente las configuraciones incluyen el estado de los procesos y alguna información de control adicional. Las denotaremos con C, C', C_1, C_2, etcétera.

Las transiciones entre configuraciones de un sistema vienen dadas por eventos. Nuestro modelo define tres tipos: *cálculo local* ($p.\iota$) cuando el proceso p ejecuta una instrucción local, *ofrecimiento* ($p.\chi$) cuando el proceso p ofrece participación en cualquier interacción $x \in \chi$ y *sincronización* (x) cuando el sistema ejecuta la interacción x.

Para el ejemplo de los filósofos comensales con $N = 5$ tendríamos que el sistema[2] estaría formado por 5 procesos "filósofo" F_i y 5 procesos "tenedor" T_i que se sincronizan a través de

[2]Consideraremos que la aritmética de subíndices es módulo N.

5 interacciones $Tomar_i$ para tomar sus tenedores y 5 interacciones $Soltar_i$ para soltarlos. Así, los procesos participantes de $Tomar_i$ son F_i, T_i y T_{i-1} y las interacciones de T_i son $Tomar_i$, $Tomar_{i+1}$, $Soltar_i$ y $Soltar_{i+1}$. Cuando el filósofo F_i ofrece la interacción $Tomar_i$ se produce un evento de ofrecimiento $F_i.\{Tomar_i\}$ y cuando el filósofo F_i toma sus dos tenedores se produce un evento de sincronización $Tomar_i$.

Definición 2 (Caracterización dinámica del sistema) *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 ($i \geq 1$). 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)*

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:

$$C_0 \xrightarrow{e_1}_{\mathrm{L}} C_1 \xrightarrow{e_2}_{\mathrm{L}} C_2 \xrightarrow{e_3}_{\mathrm{L}} \cdots \tag{1}$$

Definición 3 (Caracterización estática de procesos) *Un proceso p estará ofreciendo la interacción x en la configuración i–ésima de su ejecución si y sólo si llega a un punto en el que la ejecución en el que ofrece participar en la interacción x. Un proceso p estará esperando interactuar en la configuración i–ésima de una ejecución λ si y sólo si está ofreciendo alguna interacción. 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.*

$$\begin{aligned}
\mathrm{Readies}(\lambda, p, x, i) &\Leftrightarrow \exists 1 \leq k \leq i \cdot (\lambda(k) = p.\chi \wedge x \in \chi \wedge \nexists k < j \leq i \cdot \lambda_\beta(j) = z \wedge z \in \chi) \\
\mathrm{Waiting}(\lambda, p, i) &\Leftrightarrow \exists x \in I_\Sigma \cdot \mathrm{Readies}(\lambda, p, x, i) \\
\mathrm{Finished}(\lambda, p, i) &\Leftrightarrow \exists 1 \leq k \leq i \cdot \lambda_\beta(k) = p.\emptyset
\end{aligned} \tag{2}$$

En nuestro ejemplo, un filósofo F_i estará ofreciendo la interacción $Tomar_i$ en una determinada configuración C_j si se ha producido una transición $F_i.\{Tomar_i\}$ anterior a C_j y desde entonces ninguna interacción $Tomar_i$ ha sido ejecutada. Por otro lado, un tenedor T_i estará bloqueado a la espera de interactuar si está ofreciendo alguna interacción.

Definición 4 (Caracterización estática de interacciones) *Una interacción x en la configuración i–ésima de una ejecución λ se encuentra en uno de los siguiente estados: (i)* habilitada *cuando todos los procesos de $\mathbb{P}(x)$ la están ofreciendo, (ii)* deshabilitada *cuando no está habilitada y todos los procesos de $\mathbb{P}(x)$ están esperando interactuar o finalizados, (iii)* semihabilitada *cuando no está habilitada y ha sido ofrecida por al menos un proceso o (iv)* estable *cuando está habilitada o deshabilitada.*

$$\begin{aligned}
\mathrm{Enabled}(\lambda, x, i) &\Leftrightarrow \forall p \in \mathbb{P}(x) \cdot \mathrm{Readies}(\lambda, p, x, i) \\
\mathrm{Disabled}(\lambda, x, i) &\Leftrightarrow \neg\, \mathrm{Enabled}(\lambda, x, i) \wedge \forall p \in \mathbb{P}(x) \cdot (\mathrm{Waiting}(\lambda, p, i) \vee \mathrm{Finished}(\lambda, p, i)) \\
\mathrm{SemiEnabled}(\lambda, x, i) &\Leftrightarrow \neg\, \mathrm{Enabled}(\lambda, x, i) \wedge \exists p \in \mathbb{P}(x) \cdot \mathrm{Readies}(\lambda, x, i) \\
\mathrm{Stable}(\lambda, x, i) &\Leftrightarrow \mathrm{Enabled}(\lambda, x, i) \vee \mathrm{Disabled}(\lambda, x, i)
\end{aligned} \tag{3}$$

En nuestro ejemplo la interacción $Tomar_i$ estará habilitada en la configuración C_j sólo cuando todos sus procesos participantes (F_i, T_i y T_{i-1}) la están ofreciendo. En caso de no estar habilitada y que todos sus participantes estén bloqueados, $Tomar_i$ estará deshabilitada. Por último, si quedan procesos por ofrecerla para que se habilite la interacción se encuentra deshabilitada.

Definición 5 (Caracterización dinámica de interacciones) *En la configuración i–ésima de una ejecución λ, una interacción x se encontrará enlazada con un conjunto de interacciones cuando éstas compartan algún proceso que las ofrece. Por otro lado, definimos el conjunto de ejecuciones de una interacción x como las posiciones de β en las que la interacción x ha sido ejecutada. Por último, definimos la edad de una interacción x como el número de configuraciones que hay desde su última ejecución o ∞ si la interacción nunca se ha ejecutado.*

$$\begin{aligned}
&\mathrm{Linked}(\lambda, x, i) = \{y \in I_\Sigma \mid y \neq x \wedge \exists p \in P_\Sigma \cdot (\mathrm{Readies}(\lambda, p, x, i) \wedge \mathrm{Readies}(\lambda, p, y, i))\} \\
&\mathrm{ExeSet}(\lambda, x, i) = \{1 \leq k \leq i \mid \lambda_\beta(k) = x\} \\
&\mathrm{Age}(\lambda, x, i) = \begin{cases} i - \max \mathrm{ExeSet}(\lambda, x, i) & \text{si} \quad \mathrm{ExeSet}(\lambda, x, i) \neq \emptyset \\ \infty & \text{en otro caso} \end{cases}
\end{aligned} \tag{4}$$

Para mostrar estos conceptos haremos uso de la ejecución de los filósofos comensales que se muestra en la figura 2.(a). En dicha traza, la interacción $Tomar_1$ se encuentra enlaza con $Tomar_5$ y $Tomar_2$ en la configuración C_5, ya que dichas interacciones están compartiendo ofrecimientos. Por tratarse de una traza de ejecución con un patrón fijo de comportamiento, el conjunto de ejecuciones de $Tomar_1$ estará formado por los múltiplos de 6, ya que es en estas configuraciones cuando dicha interacción se ejecuta. Por último, la edad de $Tomar_1$ en la configuración C_3 será ∞ porque aún no se ha ejecutado y será 3 en C_9 porque la última vez que se ejecutó lo hizo en C_6.

3 Selección completamente k–justa de interacciones

En esta sección presentamos un nuevo criterio de selección de interacciones con un doble objetivo: por un lado queremos detectar y resolver las anomalías de la selección completamente justa de interacciones vistas en el apartado 1, por otro lado, queremos definir un criterio de selección mejor condicionado para su tratamiento algorítmico en entornos distribuidos evitando introducir conceptos difícilmente implementables tales como “infinitamente a menudo” o “eventualmente”. Como en el apartado anterior, primero haremos una descripción intuitiva y después una descripción más rigurosa.

3.1 Descripción intuitiva

De forma intuitiva entenderemos por ejecución completamente k–justa aquélla que garantiza que ninguna interacción se ejecuta más de k veces sin que las interacciones con las que está enlazada en ese momento estén estables, además, en caso de conflicto se debe seleccionar la interacción de mayor edad.

Para ver cómo con la selección completamente k–justa de interacciones detectamos y resolvemos las anomalías de la selección completamente justa analicemos las ejecuciones del ejemplo de los filósofos comensales visto en la figura 2:

Finitud justa: En la ejecución de la figura 2.(a), en la que se aprecia la anomalía de la finitud justa, se observa como ya no sería completamente k–justa para cualquier k menor que n ya que $Tomar_1$ se ejecuta n veces cuando $Tomar_2$ se encuentra semihabilitada. De esta forma, según nuestro criterio, ejecuciones finitas en las que una interacción es marginada en más de k ocasiones no son justas.

Conspiraciones: La ejecución de la figura 2.(b) no sería 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 $Tomar_1$ no cumpliría el criterio de selección permitiendo de esta forma que $Tomar_2$ se habilite y sea seleccionada. De esta forma resolvemos el problema de las conspiraciones con una grado de bondad tanto mejor cuanto menor sea el valor de k.

El valor de k puede verse como un umbral de las semihabilitaciones durante las ejecuciones de nuestros programas. Es decir, podemos entender que k es el número máximo de veces que una interacción se puede seleccionar en presencia de interacciones enlazadas que se encuentran semihabilitadas. En este sentido, podemos sintonizar el grado de bondad con el que queremos atajar las anomalías de la selección completa: si le damos a k su valor máximo (infinito) eso implicaría que no queremos resolver los problemas de finitud justa y conspiraciones, en cambio, si le damos a k su valor mínimo (véase apéndice 3.2) implicaría que no queremos que ninguna interacción se ejecute sin conocer primero el estado de todas las interacciones con las que podría estar en conflicto.

Por último, las ejecuciones completamente k–justas de un programa pueden verse como un subconjunto de las ejecuciones completamente justas 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 que 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 sólo se exige que las interacciones se encuentren habilitadas para ser seleccionadas), por lo que el conjunto de posibles ejecuciones es máximo (igual al de las ejecuciones completamente justas).

3.2 Definición rigurosa

Formalmente, definimos la selección completamente k–justa de interacciones de la siguiente forma:

Definición 6 (Selección Completamente k–Justa) *Sea una ejecución $\lambda = (C_0, \alpha, \beta)$ y k un número natural no nulo. Diremos que λ es una ejecución completamente k–justa si el predicado $\mathcal{SKF}(\lambda, k)$, definido como sigue, se satisface:*

$$\begin{aligned}&\mathcal{SKF}(\lambda, k) \Leftrightarrow \\ &\quad \forall x \in I_\Sigma, i \in \mathrm{ExeSet}(\lambda, x, \infty) \cdot \mathrm{Enabled}(\lambda, x, i) \wedge \\ &\quad (\mathrm{LStable}(\lambda, x, i) \wedge \mathrm{LOldest}(\lambda, x, i) \vee \neg\, \mathrm{LStable}(\lambda, x, i) \wedge \Delta(\lambda, x, i) \leq k)\end{aligned} \tag{5}$$

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 estables.

$$\mathrm{LStable}(\lambda, x, i) \Leftrightarrow \forall y \in \mathrm{Linked}(\lambda, x, i) \cdot \mathrm{Stable}(\lambda, y, i) \tag{6}$$

$\mathrm{LOldest}(\lambda, x, i)$ 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.

$$\mathrm{LOldest}(\lambda, x, i) \Leftrightarrow \forall y \in \mathrm{Linked}(\lambda, x, i) \cdot \mathrm{Age}(\lambda, x, i) \geq \mathrm{Age}(\lambda, y, i) \tag{7}$$

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

$$\Delta(\lambda, x, i) = \sum_{\phi \leq k < i} (\lambda_\beta(k) = x \wedge \mathrm{Linked}(\lambda, x, k) \neq \emptyset \wedge \neg\, \mathrm{LStable}(\lambda, x, k)) \tag{8}$$

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 lo denotamos como $\perp$):

$$\phi \triangleq \begin{cases} j & \text{if} \quad j = \max\{k \in \mathrm{ExeSet}(\lambda, x, i) \mid \mathrm{LStable}(\lambda, x, k)\} \wedge j \neq \perp \\ 1 & \text{en otro caso} \end{cases} \tag{9}$$

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[3] y debe verificar que:

$$k \geq \max\{|\mathbb{I}(p) \cap \mathbb{I}(q)| \cdot p, q \in P_\Sigma \wedge p \neq q\} \tag{10}$$

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 estabiliza 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 número máximo de interacciones comunes que tienen los procesos del sistema dos a dos.

4 Algoritmo $\mathcal{SKF}$

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.

Nuestro algoritmo está basado en la propuesta presentada en [9]. En la implementación distribuida que realizamos de dicho algoritmo [20] se detectaron las anomalías que sufre la selección completamente justa de interacciones. En [19] presentamos un algoritmo de detección de habilitaciones y en [17] un algoritmo para la selección de interacciones que no tenía en cuenta el problema de la selección justa. El resultado obtenido es que las trazas de nuestros programas no son justas y que el comportamiento de los programas distribuidos depende de factores ajenos su lógica como la topología de la red, la carga de la red, la carga de los nodos, la velocidad de los nodos, etcétera.

[3] 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.

Primero haremos una descripción informal a grandes rasgos del algoritmo y después haremos una descripción detallada de las estructuras de datos y de las funciones de actualización para, haciendo uso de reglas de inferencia, definir nuestro algoritmo. Finalmente mostraremos una prueba de coherencia.

4.1 Descripción intuitiva

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 [9]). 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 estable.

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 no comparta procesos con un conjunto de interacciones estables o, en caso contrario, que el valor de δ de ninguna de ellas haya superado el valor de k.

4.2 Descripción rigurosa

Vamos a describir la semántica operativa de nuestro algoritmo de selección completamente k–justo haciendo uso de la regla de transición $\longrightarrow\!\!\!\!\!\!\rightarrow_{\mathcal{SKF}}$ sobre configuraciones que denotaremos como D, D', D_1, etcétera. 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.

Definición 7 (Estructuras de datos) *Definimos un mapa de ofrecimientos φ de forma que a cada interacción $x \in I_\Sigma$ se le asocia un conjunto de procesos $\varphi(x)$[4]. Definimos un conjunto de procesos finalizados $\vartheta \subseteq P_\Sigma$ que contendrá a todos los procesos que han finalizado su ejecución. Definimos un mapa de semihabilitaciones δ de forma que a cada interacción x se le asocia un número natural $\delta(x)$. Por último, definimos cola 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.*

$$\begin{aligned}
&\varphi \in \{f : I_\Sigma \longrightarrow 2^{P_\Sigma} \wedge \operatorname{dom} f = I_\Sigma\} \\
&\vartheta \subseteq P_\Sigma \\
&\delta \in \{f : I_\Sigma \longrightarrow \mathbb{N} \wedge \operatorname{dom} f = I_\Sigma\} \\
&\tau = [x_1, \ldots, x_n] \wedge n = |I_\Sigma| \wedge \operatorname{img} \tau = I_\Sigma
\end{aligned} \tag{11}$$

[4]En esta definición, 2^{P_Σ} denota el conjunto de las partes de P_Σ. Se define de la siguiente forma: $2^{P_\Sigma} = \{x \subseteq P_\Sigma\}$

φ 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.

En δ 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 de ellas puede estar siendo marginada.

En τ mantendremos ordenadas las interacciones del sistema de forma que al final de la misma se encontrarán aquellas interacciones que han sido seleccionadas más recientemente y en la cabeza las que hacen más tiempo que no son seleccionadas.

De esta forma, la configuración extendida del algoritmo en un instante i viene dada por la tupla $D_i = (C_i, \tau_i, \varphi_i, \delta_i, \vartheta_i)$, 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 un conjunto de procesos finalizados $\vartheta_0 = \emptyset$.

Definición 8 (Funciones de actualización) *Cuando se producen eventos de ofrecimiento, ejecución y finalización, utilizamos las funciones* $\mathrm{AddOffer}(\varphi, p, \chi)$ *y* $\mathrm{RemoveOffer}(\varphi, x)$ *para actualizar el mapa de ofrecimientos* φ *y la función* $\mathrm{AddFinished}(\vartheta, p, \chi)$ *para actualizar el conjunto de procesos finalizados* ϑ*. Cuando una interacción* x *resulta seleccionada para ser ejecutada usaremos las funciones* $\mathrm{Order}(\tau, \delta)$ *y* $\mathrm{Update}(\varphi, \delta, x)$ *para retrasar la posición de* x *en* τ *y para crear un nuevo mapa de semihabilitaciones respectivamente*[5]*. La función* $\mathrm{Offered}(\Upsilon, \varphi, \vartheta)$ *la utilizamos para saber cuándo un conjunto de interacciones se encuentra ofrecido. Por último, definimos una función* $\mathrm{Ready}(\tau, \varphi)$ *para saber en tiempo de ejecución cuáles son las interacciones habilitadas disyuntas que se encuentran primero en la cola .*

$$
\begin{aligned}
\mathrm{AddOffer}(\varphi, p, \chi) =& \{x \mapsto \varphi(x) \cdot x \in \operatorname{dom}\varphi \wedge x \notin \chi\} \cup \{x \mapsto \varphi(x) \cup \{p\} \cdot x \in \operatorname{dom}\varphi \wedge x \in \chi\} \\
\mathrm{RemoveOffer}(\varphi, x) =& \{x \mapsto \varphi(x) \setminus \{p\} \cdot x \in \operatorname{dom}\varphi\} \\
\mathrm{AddFinished}(\vartheta, p, \chi) =& \begin{cases} \vartheta \cup \{p\} & \text{si } \chi \neq \emptyset \\ \vartheta & \text{si } \chi = \emptyset \end{cases} \\
\mathrm{Order}(\tau, \delta) =& \tau' \Leftrightarrow \operatorname{dom}\tau = \operatorname{dom}\tau' \wedge \operatorname{ran}\tau = \operatorname{ran}\tau' \wedge \\
& \quad \forall x_1, x_2 \in \operatorname{ran}\tau \cdot \tau'^{-1}(x_1) \leq \tau'^{-1}(x_2) \Rightarrow \delta(x_1) \geq \delta(x_2) \\
\mathrm{Offered}(\Upsilon, \varphi, \vartheta) \Leftrightarrow& \Upsilon \subseteq I_\Sigma \wedge \forall x \in \Upsilon \cdot \forall p \in \{p\}(x) p \in \varphi(x) \vee x \in \vartheta \\
\mathrm{Update}(\varphi, \delta, x) =& \delta \otimes \{x \mapsto 0\} \otimes \{y \mapsto \delta(y) + 1 \cdot y \in \mathcal{S} \setminus \{x\} \wedge \neg\, \mathrm{Offered}(\mathcal{S}, \varphi, \vartheta)\} \\
\mathrm{Ready}(\tau, \varphi) =& \{x \in \operatorname{dom}\varphi \cdot \{p\}(x) = \varphi(x) \wedge \nexists y \in \mathcal{S} \cdot \{p\}(y) = \varphi(y) \wedge \tau^{-1}(y) < \tau^{-1}(x)\} \\
\text{Siendo } \mathcal{S} =& \{z \in \operatorname{dom}\varphi \cdot \varphi(z) \cap \varphi(x) \neq \emptyset\}
\end{aligned}
\tag{12}
$$

El objetivo de cada una de estas funciones es el siguiente:

[5] En esta definición, el operador binario $\otimes$ (*override*) denota la composición de mapas.
Se define de la siguiente forma: $\delta_1 \otimes \delta_2 = \{x \mapsto \delta_1(x) \cdot x \in \operatorname{dom}\delta_1 \wedge x \notin \operatorname{dom}\delta_2\} \cup \{x \mapsto \delta_2(x) \cdot x \in \operatorname{dom}\delta_2\}$

- La función $\mathrm{AddOffer}(\varphi, p, \chi)$ recibe un mapa de ofrecimientos φ, un proceso p y el conjunto de interacciones χ que ese proceso está ofreciendo y devuelve un nuevo mapa en el que se ha insertado dicho proceso en el conjunto de ofrecimientos de cada interacción de χ.
- La función $\mathrm{RemoveOffer}(\varphi, x)$ recibe un mapa de ofrecimientos φ y la interacción seleccionada x y devuelve un mapa en el que los participantes de la interacción x han sido eliminados de todos los ofrecimientos.
- La función $\mathrm{AddFinished}(\vartheta, p, \chi)$ recibe un conjunto de procesos finalizados ϑ, un proceso p y el conjunto de interacciones que dicho proceso está ofreciendo y devuelve el mismo conjunto ϑ si $\chi = \emptyset$ o $\vartheta \cup \{p\}$ en caso contrario.
- La función $\mathrm{Order}(\tau, \delta)$ recibe una cola de interacciones τ y un mapa de semihabilitaciones δ y devuelve cualquier cola ordenada de mayor a menor según δ. Fíjese que tal y como está definida esta función, su valor de retorno es no determinista, ya que para un mapa de semihabilitaciones con empates pueden existir distintas colas que cumplan el criterio de ordenación.
- El predicado $\mathrm{Offered}(\Upsilon, \varphi, \vartheta)$ recibe un conjunto de interacciones Υ, un mapa de ofrecimientos φ y un conjunto de procesos finalizados ϑ y se evalúa a verdadero cuando todos los procesos participantes de todas las interacciones de Υ están ofreciendo interacción o han finalizado su ejecución.
- La función $\mathrm{Update}(\varphi, \delta, x)$ recibe un mapa de ofrecimientos φ, un mapa de semihabilitaciones δ y la interacción seleccionada x y devuelve otro mapa de semihabilitaciones igual que el de entrada en el que el contador de x se pone a cero y se incrementa en una unidad el contador de todas las interacciones que comparten algún ofrecimiento con x.
- Por último, la función $\mathrm{Ready}(\tau, \varphi)$ recibe una cola de interacciones τ y un mapa de ofrecimientos φ y devuelve todas las interacciones que se encuentran habilitadas, teniendo en cuenta que, en el caso de que varias interacciones habilitadas compartan algún ofrecimiento, se devuelve la que se encuentre primera en la cola.

Una vez definidas las estructuras de datos y las correspondientes funciones para actualizar y consultar dichas estructuras podemos definir el algoritmo de selección de interacciones completamente k–justo haciendo uso de reglas de inferencia para describir bajo qué condiciones una interacción habilitada resulta seleccionada.

Definición 9 (Algoritmo de selección completamente k–justo) *Hemos descrito el algoritmo de selección completamente k–justo de interacción haciendo uso de las siguientes reglas de inferencia:*

$$\frac{\begin{array}{c} C \xrightarrow{p.\chi}_{\mathrm{L}} C' \wedge \\ \varphi' = \mathrm{AddOffer}(\varphi, p, \chi) \wedge \vartheta' = \mathrm{AddFinished}(\vartheta, p, \chi) \end{array}}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{p.\chi}_{\mathcal{SKF}} (C', \tau, \varphi'\delta, \vartheta')} \tag{13}$$

$$\frac{\begin{array}{c} x \in \mathrm{Ready}(\tau, \varphi) \wedge \mathcal{S} = \{z \in \mathrm{dom}\,\varphi \cdot (\varphi(z) \cap \varphi(x) \neq \emptyset)\} \wedge \\ \tau' = \mathrm{Order}(\tau, \delta') \wedge \varphi' = \mathrm{RemoveOffer}(\varphi, x) \wedge \delta' = \mathrm{Update}(\varphi, \delta, x) \wedge \\ (\mathcal{S} = \{x\} \vee \mathrm{Offered}(\mathcal{S}, \varphi, \vartheta) \vee (\mathcal{S} \neq \{x\} \wedge \max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k)) \end{array}}{(C, \tau, \varphi, \delta, \vartheta) \xrightarrow{x}_{\mathcal{SKF}} (C', \tau', \varphi', \delta', \vartheta) \wedge C \xrightarrow{x}_{\mathrm{L}} C'} \tag{14}$$

En la regla 13 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 14 se describe qué interacción se debe seleccionar para que la ejecución que se genere sea completamente k–justa (obsérvese que $C \xrightarrow{x}_{\mathrm{L}} C'$ 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 estables.
3. El contador de semihabilitaciones asociado de todas las interacciones enlazadas con x es menor que k.

4.3 Demostración de coherencia

Demostrar la coherencia de nuestro algoritmo equivale a demostrar que todas las ejecuciones que genera son completamente k–justas. Haremos la demostración por reducción al absurdo.

Por definición, cualquier ejecución que contenga menos de k ejecuciones de interacción el completamente k–justa, siendo la demostración trivial.

Supongamos que partimos de una configuración extendida D_n completamente k–justa y que nuestro algoritmo lleva a cabo una transición $\longrightarrow_{\mathcal{SKF}}$ que nos lleva a un configuración D_{n+1} que deja de ser completamente k–justa. Es decir, que se selecciona una interacción x que desde la última vez que se estabilizó el conjunto de interacciones con las que compartía ofrecimientos se ha ejecutado k veces (ver definición 6), es decir, cuando

$$\Delta(\lambda, x, n-1) \leq k < \Delta(\lambda, x, n) \tag{15}$$

Analicemos las reglas una a una:

- La regla de transición $\xrightarrow{p.\chi}_{\mathcal{SKF}}$ (13) nos sirve para actualizar el mapa de ofrecimientos φ y el conjunto ϑ de forma que para toda interacción x, en $\varphi(x)$ se encuentran los procesos que la han ofrecido desde su última ejecución y en ϑ se encuentran todos los procesos p que han finalizado su ejecución (han ejecutado $p.\emptyset$). Esta transición viene dada por el programa y además no desencadena la selección de ninguna interacción por lo que es imposible que D_{n+1} deje de ser completamente k–justa.
- Para que se satisfaga el antecedente de la regla de transición $\xrightarrow{x}_{\mathcal{SKF}}$ (14) es necesario que se cumplan alguna de las siguientes tres condiciones :
    1. $\mathcal{S} = \{x\}$. En $\mathcal{S}$ están todas las interacciones que comparten ofrecimientos con x desde la última vez que se ejecutó, ya que cuando se selecciona una interacción x se actualiza $\varphi(x)$ a $\emptyset$ ($\mathrm{RemoveOffer}(\varphi, x)$) y cuando se realiza un ofrecimiento $p.\chi$ se añade p a cada interacción de χ.
    Si $\mathcal{S}$ sólo contiene la interacción x es porque ésta no comparte ningún ofrecimiento con ninguna interacción z en la configuración D_n, luego $z \notin \mathrm{Linked}(\lambda, x, n)$, lo que implica que $\Delta(\lambda, x, n) = \Delta(\lambda, x, n-1)$, llegando a una contradicción.

2. $\mathrm{Offered}(\mathcal{S}, \varphi_n, \vartheta_n)$. Este predicado se verifica cuando todos los procesos de todas las interacciones del conjunto $\mathcal{S}$ han ofrecido interacción o han terminado su ejecución, es decir, el conjunto $\mathcal{S}$ se encuentra estable en la configuración D_n. Así el predicado $\mathrm{Offered}(\mathcal{S}, \varphi_n, \vartheta_n)$ es un refinamiento del predicado $\mathrm{Stable}(\lambda, \Upsilon, n)$ (definido sobre el modelo abstracto de ejecución) sobre la configuración extendida D_n de nuestro algoritmo. Al igual que en caso anterior es fácil comprobar que cuando esto ocurre se tiene que $\Delta(\lambda, x, n) = \Delta(\lambda, x, n-1)$, volviendo a llegar a una contradicción.
3. $\max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k$. En la configuración D_n tenemos que $\delta(y)$ contiene el número de veces que otras interacciones han sido seleccionadas cuando y aún no estaba estable. Esto es así porque la función $\mathrm{Update}(\varphi, \delta, x)$ se encarga de poner $\delta(x)$ a cero e incrementar en una unidad el valor de δ para las interacciones y tal que $y \in \mathcal{S} \setminus \{x\} \wedge \neg\, \mathrm{Offered}(\mathcal{S}, \varphi, \vartheta)$ cada vez que se selecciona una interacción y.

   De esta forma se tiene que $\max_{y \in \mathcal{S} \setminus \{x\}} \delta(y)$ es un refinamiento de la función $\Delta(\lambda, x, n)$ sobre la configuración D_n, por lo que si $\max_{y \in \mathcal{S} \setminus \{x\}} \delta(y) < k$ en D_n entonces $\Delta(\lambda, x, n) < k$, volviendo a llegar a una contradicción.

Como el antecedente no se verifica en ninguno de los casos anteriores podemos asegurar que hemos llegado a una contradicción concluyendo así la demostración.

5 Sobre los resultados de imposibilidad

Fíjese que nuestra noción de selección es más restrictiva que la selección completamente justa, lo que implica que bajo las mismas condiciones, una interacción habilitada puede ser que cumpla la condición de selección completa pero no cumpla la condición de selección $\mathcal{SKF}$. Desde el punto de vista de la ejecución de nuestros programas esto puede verse como un retraso a la hora de tomar una decisión. Esto es cierto, ya que Tsay y Bagrodia en [22] demuestran que, en general, la selección completamente justa de interacciones es imposible en interacciones binarias y, por consiguiente también lo es para las interacciones entre múltiples participantes. También demuestran que la selección completamente justa de procesos es imposible en el caso de interacciones entre múltiples participantes, no siendo así para interacciones binarias. Al usar el término "imposible" queremos decir que no existe un algoritmo determinista libre de esperas que implemente la selección justa. Para llegar a estos resultados de imposibilidad hace las siguientes suposiciones:

1. Los procesos no tienen por qué ofrecer interacción en un tiempo finito. Es decir, un proceso puede estar indefinidamente realizando cálculos locales.
2. El hecho de que un proceso ofrezca interacción no depende del estado local de estos procesos. Es decir, cuando un proceso ofrece interacción lo hace de una manera autónoma.
3. Los cambios de estado en los procesos no son observables de manera inmediata por los otros procesos o por sus gestores. Es decir, el tiempo necesario para comunicar a dos procesos del sistema no es despreciable.

Para demostrar la imposibilidad de diseñar un algoritmo determinista de selección completamente justa de interacciones asumiendo las suposiciones anteriores se puede hacer uso del ejemplo de la figura 3. Supongamos que tenemos un sistema con tres procesos P_1, P_2 y P_3 y tres interacciones

binarias A, B y C. Teniendo en cuenta las suposiciones iniciales una posible traza de interacción sería aquélla en la que todos los procesos inicialmente están realizando cálculos locales hasta que P_1 y P_2 ofrecen todas sus interacciones. Esto haría que A se habilitase y empezara su ejecución (para cumplir la suposición 1). Antes de que la ejecución de la interacción A comience P_3 decide ofrecer todas sus interacciones lo que implicaría que B y C se habilitasen al mismo tiempo que la interacción A. En esta situación tendríamos que todas las interacciones están habilitadas pero es A la que termina siendo seleccionada. Esta situación puede repetirse en infinitas ocasiones lo que nos llevaría a una ejecución en la que las interacciones B y C están infinitamente a menudo habilitadas pero nunca llegan a ser seleccionadas.

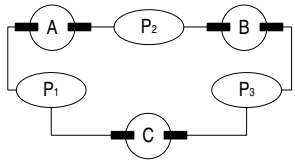


Figura 3: Grafo de interacción para el ejemplo de Tsay.

Como puede observarse, nuestro modelo de interacción no cumple el primer requisito impuesto para que la selección completamente justa de interacciones pueda ser tratada de forma algorítmica en sistemas distribuidos, por lo que se puede definir un algoritmo determinista libre de esperas.

6 Trabajos relacionados

El problema de la selección justa en el contexto de los sistemas basados en interacciones entre múltiples participantes ha sido estudiado por varios autores dando lugar a distintas nociones y algoritmos de selección justa. De todos ellos destacamos los tres trabajos que se muestran en la tabla 1 junto con el nuestro[6]. Las columnas *Fin.* y *Con.* indican si dichas propuestas dan solución al problema de la *finitud justa* y las *conspiraciones* respectivamente.

6.1 Un algoritmo de selección completamente justa probabilístico

En [14] se presentan unos algoritmos para la selección completamente justa de interacciones entre múltiples participantes que son totalmente simétricos y descentralizados, es decir, que todos los procesos del sistema ejecutan el mismo código. Presentan dos soluciones, una haciendo uso de paso de mensajes y otra haciendo uso de variables compartidas *single–write*, es decir, variables que sólo pueden ser escritas por un proceso y que pueden ser leídas por varios procesos a la vez. Su propuesta resuelve los siguientes problemas: (i) *sincronización*, detectar la habilitación de interacción sin necesidad de hacer uso de gestores de interacción que vayan anotando los ofrecimientos de los procesos participantes; (ii) *exclusión mutua*, garantizar que de un grupo de interacciones conflictivas sólo se podrá ejecutar una interacción en cada momento; (iii) *selección completamente justa de interacciones*, garantizar que toda interacción que está infinitamente a menudo habilitada terminará siendo seleccionada para ser ejecuta en infinitas ocasiones.

[6] De todos los algoritmos para implementar selección completamente justa de interacciones hemos escogido los descritos en [14] por ser uno de los últimos y más interesantes. En cualquier caso, ninguno de los estudiados resuelve las anomalías estudiadas de forma simultánea.

Propuesta	Algoritmo	Fin.	Con.
Strong fairness [14]	Algoritmo descentralizado que utiliza la *La Ley de los Grandes Números* para garantizar la selección completamente justa de interacciones.	No	No
Hyperfairness [2]	Modifica el código fuente de los programas para garantizar la selección completamente justa de interacciones y la ausencia de conspiraciones.	No	Si
Finitary fairness [1]	Propone esquemas de transformación de autómatas para resolver el problema de la finitud en cualquier noción de selección.	Si	No
$\mathcal{SKF}$	Algoritmo basado en colas de prioridad para resolver la finitud justa y contadores de semihabilitaciones para resolver el problema de las conspiraciones.	Si	Si

Tabla 1: Trabajos relacionados con el presentado.

Los algoritmos que presentan Joung et al. [14] tienen sus antecedentes en otros algoritmos realizados por Smolka [15] y Francez [13]. Con respecto al primero presenta la diferencia de que utiliza temporizadores (*timeouts*) dinámicos en lugar de estáticos, lo que supone una mejora importante en el rendimiento. Con respecto al segundo la diferencia radica en que utiliza variables compartidas *single–write* frente a las variables *multi–write* que utiliza Francez, que son muy costosas de implementar.

El esquema que siguen sus algoritmos son de *"tentativa, espera y comprobación"* de manera que un proceso escoge una única interacción y espera a que todos los procesos que participan en dicha interacción la ofrezcan, transcurrido un cierto tiempo comprueban si esto ocurre. En caso negativo se vuelve a intentar con otra interacción.

Como la selección de la interacción que un proceso intentará ejecutar es aleatoria, para poder garantizar la selección completamente justa de las mismas se apoya en la *Ley de los Grandes Números* de la teoría de la probabilidad, que nos dice que si un conjunto de procesos ofrece un conjunto aleatorio de interacciones infinitamente a menudo se tendrá que con probabilidad 1 dichos procesos ofrecerán la misma interacción al mismo tiempo infinitamente a menudo.

6.2 *Hyperfairness*

En [2] se estudia el problema de las conspiraciones en el contexto de las interacciones entre múltiples participantes y se presenta la noción de selección hiperjusta (*hyperfairness*) como respuesta a dicho problema. Se dice que una ejecución es 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: (i) no resuelve el problema de la *finitud justa* y (ii) no se conoce ningún algoritmo general para implementarla ya que los autores sólo 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, algo que se antoja bastante complicado en entornos distribuidos en los que no se tiene acceso al código de los procesos.

Para dar un definición de *hyperfairness* lo primero que hacen es identificar las interacciones que son resistentes a conspiraciones. Una interacción x será resistente a conspiraciones en un sistema Σ si y sólo si bajo la hipótesis que todas las interacciones conflictivas se encuentran deshabilitadas, x termina habilitándose en un tiempo finito. Partiendo de esta premisa se entiende que una ejecución es hiperjusta (*hyperfair*) si es finita o si es infinita y toda interacción resistente a conspiraciones termina habilitándose infinitamente a menudo.

La diferencia fundamental entre esta noción y las clásicas es que las segundas garantizan la selección de interacciones habilitadas y *hyperfairness* sólo garantiza la condición de habilitación de interacciones resistentes a conspiraciones.

El algoritmo presentado en este trabajo consiste en tener un planificador como un proceso más que se ejecuta en paralelo al resto de procesos del sistema y que tiene acceso a las variables locales y al código de los mismos. A cada interacción x se le asocia un entero z_x que indica la prioridad de la interacción, siendo inicialmente aleatorias. Para conseguir que toda interacción resistente a conspiraciones termine habilitándose modifica los guardas de los ofrecimientos de todas las interacciones de forma que sólo se evalúe a verdadero aquella interacción x cuyo z_x es mínimo. Una vez que dicha interacción se habilita existe otro planificador (completamente justo) que decide si ejecutar o rechazar la interacción. En el caso de ejecutarla decrementa en una unidad el valor z_y de todas las interacciones y que son conflictivas con x y asigna un valor positivo aleatorio a z_x.

6.3 *Finitary fairness*

La selección justa finita [1] (*finitary fairness*) intenta resolver el problema de la finitud justa. 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 son que: (i) el valor de k se conoce a posteriori, (ii) no resuelve el problema de las conspiraciones y (iii) no se conoce ningún algoritmo que la implemente, sólo un esquema de transformación que puede ser aplicado a programas expresados con autómatas de Büchi [6, 16].

La noción de selección justa finita se define como un operador unario (fin), que dado un lenguaje (Π) devuelve todas las palabras finitamente justas, definiendo que una palabra de longitud infinita $\overline{\omega}$ es finitamente justa si el prefijo de dicha palabra es finito y pertenece al lenguaje.

Una de las aportaciones más interesantes de este trabajo es que propone un esquema de transformación para obtener a partir de un sistema Σ (descrito con un autómata) otro programa equivalente $\mathrm{fin}(\Sigma)$ sobre el que se pueden demostrar propiedades de viveza con mayor facilidad.

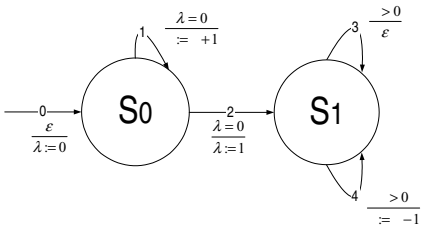
La transformación que propone pasa por asociar a cada transición τ del autómata inicial un umbral μ_τ y un contador δ_τ. A la condición de habilitación de toda transición τ se le añade la condición $\delta_\tau < \mu_\tau$ y cada vez que se habilita τ se incrementa en una unidad el valor de δ_τ y se ponen a cero los contadores de las transiciones cuyo estado inicial es el mismo que el de τ. Por ejemplo, la figura 4 muestra la versión finita de un programa (autómata) que requiere la selección moderadamente justa de transiciones habilitadas para poder garantizar la terminación del mismo.

7 Conclusiones

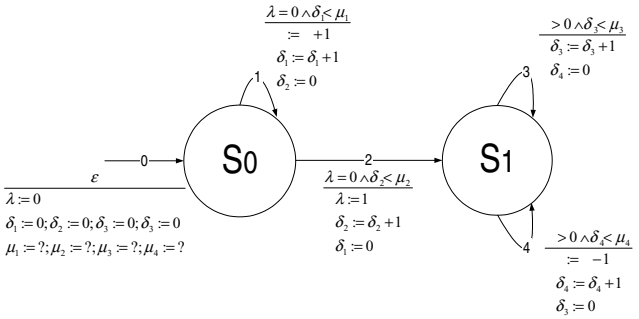
En este artículo presentamos un nuevo criterio de selección de interacciones que permite garantizar propiedades fundamentales en programas cuya ejecución es no determinista que con otros criterios resultan imposibles.

```
P::
  λ := 0
  *[ λ = 0 → x := x + 1          {opción 1}
  [] λ = 0 → λ := 1              {opción 2}
  [] λ = 1 ∧ x > 0 → skip        {opción 3}
  [] λ = 1 ∧ x > 0 → x := x − 1  {opción 4}
  ]
```

(a) Programa de ejemplo



(b) Autómata del programa de ejemplo



(c) Autómata transformado del programa de ejemplo

Figura 4: Transformación de programas propuesta por Alur [1] .

Hemos definido formalmente un modelo formal de interacción que: (i) permite sincronizar a un número arbitrario y fijo de procesos, (ii) hace uso de eventos para detectar el ofrecimiento de interacciones de un proceso, la terminación de un proceso y la ejecución de una acción conjunta (interacción) por parte de un conjunto de procesos y (iii) modela una ejecución como una secuencia maximal de estos eventos.

Utilizando este modelo, hemos definido de forma rigurosa una ejecución completamente k–justa, como aquélla en la que ninguna interacción se ejecuta más de k veces sin que lo hagan otras interacciones habilitadas con las que comparte algún participante. Hemos visto que una ejecución completamente k–justa 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 la definición de un umbral k que utilizamos para detectar las posibles situaciones de conspiración y resolverlas a tiempo. Este umbral caracteriza la ejecución de nuestros programas concurrentes, así, el valor que tome 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 conflictivas se ejecutarán a la velocidad del participante más lento. Además, dentro de un grupo de interacciones conflictivas se establecerá un turno de ejecución. En este contexto nuestra solución para k mínimo es similar a la propuesta de Best [4] para resolver el problema de las conspiraciones a costa de ralentizar los procesos que forman un programa concurrente.
- Si k es máximo (tiende a infinito) entonces las interacciones conflictivas se ejecutan a la velocidad del participante más rápido. Como sabemos, cuando las velocidades de los participantes es muy distinta los programas toman un comportamiento no deseado. En este contexto nuestra solución se parece a las propuestas por Bagradia [3] y Pérez et al. [17], en las 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.

Nuestra noción de selección justa presenta varias ventajas con respecto a la propuesta por Attie [2], ya que (i) no requiere que los programas objeto de selección sean infinitos y reactivos, (ii) el algoritmo que implementa la noción no necesita acceder al estado local de los procesos que se ejecutan en el programa y (iii) la responsabilidad final de la selección de interacciones no recae en un gestor externo.

Nuestra noción de selección justa no asume que cualquier ejecución finita sea justa por definición. Es decir no se requiere que las ejecuciones sean infinitas, sino que éstas tengan una longitud mínima (que será función del umbral k). En este sentido, la principal ventaja de nuestra propuesta con respecto a otras que intentan resolver este problema [1] es que (i) garantiza la ejecución de una interacción en un tiempo finito dando una cota superior del mismo, (ii) garantiza la ausencia de conspiraciones y (iii) dispone de un algoritmo que la implementa.

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] E. Best. *Semantics of Sequential and Parallel Programs*. Prentice Hall, New York, 1996.

[6] 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.

[7] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison–Wesley, 1988.

[8] R. Corchuelo, J.A. Pérez, and M. Toro. A multiparty coordination aspect language. *ACM Sigplan*, 35(12):24–32, December 2000.

[9] R. Corchuelo, D. Ruiz, M. Toro, and A. Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the XXVth Euromicro Conference (Workshop on Network Computing)*, Milan, September 1999. IEEE Press.

[10] M. Evangelist, V.Y. Shen, I.R. Forman, and M. Graf. Using Raddle to design distributed systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 102–115. IEEE Computer Society Press, April 1988.

[11] N. Francez. *Fairness*. Springer–Verlag, 1986.

[12] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[13] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science*, pages 373–379, 1980.

[14] 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.

[15] Y.J. Joung and S.A. Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):137–149, February 1998.

[16] 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.

[17] J. A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An order-based, distributed algorithm for implementing multiparty interactions. In *Fifth International Conference on Coordination Models and Languages COORDINATION 2002*, pages 250–257, York, UK, 2002. Springer–Verlag.

[18] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. A framework for aspect–oriented multiparty coordination. In *New Developments in Ditributed Applications and Interoperable Systems*, pages 161–173. Kluwer Academic Publishers, 2001.

[19] J.A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An enablement detection algorithm for open multiparty interactions. In *ACM Symposium on Applied Computing SAC'02*, pages 378–384, Madrid, Spain, 2002. Springer–Verlag.

[20] D. Ruiz, R. Corchuelo, J.A. Pérez, and M. Toro. Un algoritmo descentralizado de selección justa de interacciones entre múltiples participantes. In *Simposio Español de Informática Distribuida 2000*, pages 419–427, Ourense, Spain, 2000.

[21] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

[22] Y.K. Tsay and R.L. Bagrodia. Some impossibility results in interprocess synchronization. *Distributed Computing*, 6(4):221–231, 1993.