SELECCIÓN JUSTA EN EL CONTEXTO DEL MODELO DE INTERACCIÓN ENTRE MÚLTIPLES PARTICIPANTES

DAVID RUIZ CORTÉS
UNIVERSIDAD DE SEVILLA

TESIS DOCTORAL

OCTUBRE, 2003

Financia: Trabajo financiado parcialmente por el Miniterio de Ciencia y Tecnología (proyectos TIC97-0593-C05-05, TIC-2000-1106-C02-01, FIT-150100-2001-78 y TIC-2003-02737-C02-01), y por la Junta de Castilla la Mancha (proyecto PCB-02-001).

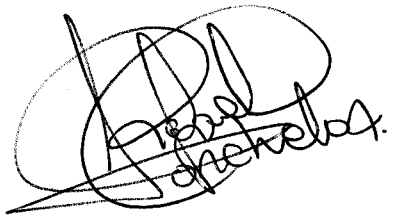
Don Rafael Corchuelo Gil, Profesor Titular de Universidad del Área de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla,

HACE CONSTAR

que don David Ruiz Cortés, Ingeniero en Informática por la Universidad de Sevilla, ha realizado bajo mi supervisión el trabajo de investigación titulado

Selección Justa
en el contexto del
Modelo de Interacción entre Múltiples
Participantes

Una vez revisado, autorizo el comienzo de los trámites para su presentación como Tesis Doctoral al tribunal que ha de juzgarlo.



Fdo. Rafael Corchuelo Gil
Profesor Titular de Universidad
Área de Lenguajes y Sistemas Informáticos
Universidad de Sevilla

Sevilla, Octubre de 2003

UNIVERSIDAD de SEVILLA

RECTORADO

Sevilla, 27 de enero de 2004
Ref: Negociado de Tesis EL/CH
Asunto: Enviando Tesis Doctorales Leídas

Ilmo Sr. Director de la
Biblioteca de la E.T.S. de
Ingeniería Informática
Universidad de Sevilla

Adjunto le remito ejemplares de Tesis Doctorales leídas en Departamentos vinculados a esa Facultad a fin de que pasen a formar parte de fondos bibliográficos de consulta de ese Centro.

AUTORES DE LAS TESÍS LEIDAS

1. García Vázquez, Pedro .
2. Ruíz Cortés, David
3. Valenzuela Muñoz, Jesús

UNIVERSIDAD DE SEVILLA

La Jefa de Sección de Doctorado

Fdo.: Yolanda Díaz Rolando

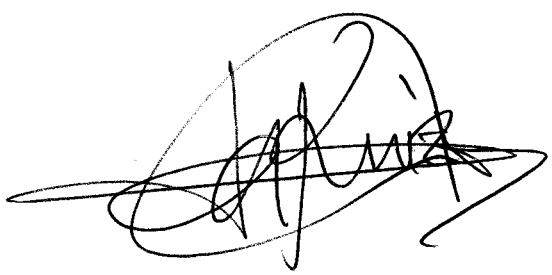
David Ruiz Cortés, con DNI número 28.934.631-X,

DECLARA

Ser el autor de la tesis que se presenta en esta memoria, y cuyo nombre es:

Selección Justa
en el contexto del
Modelo de Interacción entre Múltiples
Participantes

Lo cual firmo en Sevilla, Octubre de 2003.



Fdo. David Ruiz Cortés

A la gorda.

Índice

Índice de Figuras

Índice de Tablas

Índice de programas

Agradecimientos

No cabe duda que uno de los mejores momentos en el desarrollo de una tesis doctoral es aquél en el que terminas de escribir la memoria. No sólo porque terminas un largo camino, sino también porque puedes tener unas palabras de agradecimiento para las personas que han estado contigo.

La primera de ellas es mi director de tesis, Rafael Corchuelo, ya que hubiera sido imposible terminar este trabajo sin su ayuda. Siempre ha confiado en mi trabajo y, desde que me dirigió el proyecto final de carrera, siempre ha tenido las palabras adecuadas en los momentos oportunos.

La ayuda de un grupo de investigación es fundamental para poder hacer realidad una tesis doctoral. En este sentido, *The Distributed Group* me ha dado el soporte y el ánimo necesario desde el comienzo de mi trabajo. En especial, Miguel Toro, por su buena predisposición desde el principio y por las revisiones realizadas. Sin su ayuda jamás hubiera empezado este trabajo. Además, las discusiones mantenidas con José A. Pérez y Antonio, o los desayunos con José L. Arjona, Octavio Martín y Joaquín Peña, han hecho que este trabajo sea mucho más llevadero. Por último, me gustaría hacer mención a David Benavides, que se ha unido recientemente a nuestro grupo de investigación

Por último, pero no menos importante, le doy las gracias a mi familia y a mi novia por haber aguantado mi mal humor, sobre todo los últimos meses de este trabajo en los que he estado al cien por cien dedicado a escribir esta memoria. Seguro que ellos tenían más ganas que yo mismo de que este momento llegara.

Sevilla
30 *de Septiembre de* 2003
D. R. C.

Resumen

Cuando se quiere resolver un problema en el que es preciso que tres o más artefactos software colaboren, hay situaciones en que las llamadas a procedimientos remotos, el *rendez-vous* o el paso de mensajes no resultan las primitivas más adecuadas desde un punto de vista conceptual. El motivo principal es que obligan a describir las interacciones en forma de protocolos de sincronización y comunicación bipartitos en los que, a menudo, se pierde el nivel de abstracción necesario para entender y razonar sobre las propiedades del sistema.

En el grupo de investigación denominado The Distributed Group, llevamos trabajando desde 1997 en modelos de interacción entre múltiples participantes. Estos modelos proporcionan los mecanismos necesarios para describir interacciones complejas desde un punto de vista conceptual y, además, hemos desarrollado implementaciones eficientes de los mismos basadas en las primitivas de sincronización y comunicación tradicionales. Inicialmente, en la tesis doctoral de Corchuelo [34], se presentó un modelo basado en restricciones simbólicas que se aplicó con éxito a la construcción de prototipos tempranos durante la fase de elicitación y análisis de requisitos; posteriormente, en la tesis doctoral de Pérez [105], este modelo se refinó hasta un punto en el que ya era posible una implementación eficiente preparada para la etapa de producción; finalmente, en la tesis de Ruiz [116], se estudiaron aspectos relacionados con la calidad de servicio que ofrecen los artefactos que participan en una interacción. Actualmente, en la tesis de Peña [101] se está estudiando cómo aplicar estos modelos en sistemas multiagentes y en la de Martín [91] se están extrapolando algunos conceptos al campo de la negociación automática.

Entre los muchos problemas que se pueden estudiar en estos modelos de interacción, en esta tesis nos hemos centrado en el estudio de nociones para garantizar la selección justa de interacciones. En el contexto de los sistemas concurrentes, estas nociones se utilizan para garantizar que toda interacción que puede ser ejecutada suficientemente a menudo termina siendo ejecutada

suficientemente a menudo con el tiempo. El problema radica en la interpretación e implementación de los conceptos "suficientemente a menudo" y "con el tiempo". Si se hace una interpretación estricta, entonces aparece el problema de la finitud justa ya que estos conceptos sólo pueden aplicarse a ejecuciones infinitas; si no se tiene cuidado en la implementación, entonces aparece el problema de las conspiraciones ya que se pueden dar situaciones en que un entrelazado desafortunado provoca que una interacción nunca esté preparada para ejecutarse y por lo tanto no puede ser ejecutada en ningún momento, aunque con otro entrelazado pudiera haberlo sido.

En esta tesis doctoral presentamos una nueva noción de selección justa a la que hemos denominado k-*conspiracy-free fairness*. Su ventaja fundamental respecto a otras es que restringe el conjunto de posibles ejecuciones de un programa a aquéllas que no presentan ninguna de las dos anomalías descritas anteriormente y, además, se puede adaptar en la medida de posible a las características propias de los programas ajustando el valor de k. También es importante destacar que nuestra noción es una propiedad *safety*, por lo que es posible implementarla y proporcionamos un marco de trabajo eficiente para conseguirlo. Nuestros resultados experimentales corroboran estas afirmaciones y el estudio detallado del trabajo relacionado demuestra que se trata de una aportación original y novedosa.

Nota sobre la traducción: Según el diccionario Cambridge Advanced Learner's, el vocablo inglés "*fairness*" se define como sigue: "*the quality of treating people equally or in a way that is right or reasonable*". El diccionario Merriam Webster también recoge este significado y abunda en él en su sexta acepción: "*conforming with the established rules, consonant with merit or importance*". Ambas definiciones encajan perfectamente en la primera acepción que el Diccionario de la Real Academia de la lengua española recoge para el vocablo justicia: "una de las cuatro virtudes cardinales que inclina a dar a cada uno lo que le corresponde o pertenece".

Evidentemente, al referirnos a "*fairness*" en este campo no debemos incurrir en la falsa idea de que toda interacción debe tener las mismas oportunidades que el resto de ser ejecutada. Al contrario, debe tener las *justas* en función a las caracteristicas inherentes del programa objeto de estudio. Evidentemente, si la lógica de un programa es tal que obliga a que una interacción sólo se pueda ejecutar tras un millón de ejecuciones de otra, es imposible que la primera pueda ejecutarse en más ocasiones pues iría en contra de las reglas establecidas en el programa y eso no sería ni correcto ni razonable. Lo que resultaría injusto es que pudiendo ejecutar esa interacción lo suficientemen-

te a menudo, no resultase seleccionada nunca o en un número de ocasiones insignificante en proporción al tamaño de la ejecución.

Parte I

Prólogo

Capítulo 1

Introducción

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction
of a new order of things.

Niccolo Machiavelli, 1469–1527
Italian political theorist

El comportamiento de un programa concurrente no depende sólo de su lógica inherente, sino que también influye la forma en que se implementan las interacciones entre los elementos que lo constituyen y la manera en que el sistema operativo las lleva a cabo. Son mucho los factores que afectan, por ejemplo, la velocidad relativa de los procesadores, la planificación de procesos o de hilos, la latencia de la red y de algunos recursos, etcétera. La selección justa es un concepto que nos permite abstraernos de todos estos detalles, ya que normalmente son demasiado complicados para tenerlos todos en cuenta o incluso, a veces, desconocidos a priori.

En esta tesis doctoral se presenta una nueva noción de justicia que puede ser aplicada a cualquier programa concurrente en el que la sincronización y comunicación entre sus elementos se llevan a cabo a través de interacciones multipartitas. En este capítulo primero hacemos una introducción a los conceptos más importantes en nuestro contexto de investigación en la Sección §1.1; después hacemos un resumen de la principales aportaciones en la Sección §1.2; por último, en la Sección §1.3, presentamos la estructura de esta memoria.

1.1 Contexto de investigación

En esta sección presentamos una visión general de los conceptos más importantes que usamos a lo largo de la memoria. Primero definiremos y clasificaremos los programas concurrentes en la Sección §1.1.1; nuestro enfoque de los modelos de interacción se presenta en la Sección §1.1.2; el modelo de interacción entre múltiples participantes se introduce en la Sección §1.1.3; por último, una pequeña introducción al problema de la selección justa se muestra en la Sección §1.1.4.

1.1.1 Programas concurrentes

Un programa informático estará compuesto de conjunto de elementos que podrán ser activos (por ejemplo: procesos, tareas, hilos) o pasivos, en cuyo caso no tendrán actividad propia en sólo serán usados por elementos activos (por ejemplo: instancias de tipos abstractos de datos, recursos del sistema, archivos).

Se dice que un programa informático es concurrente cuando está compuesto de uno o más elementos activos, además, distinguiremos entre dos tipos de concurrencia:

Concurrencia competitiva, que se da cuando dos o más de los elementos que componen el programa han sido diseñados por separado pero tiene que utilizar algún elemento pasivo común. Normalmente, los elementos activos compiten por algún recurso que desconoce la identidad de quién lo utiliza y sólo se limita a servir a sus clientes siempre y cuando no se encuentren éstos ocupados. Por ejemplo, concurrencia competitiva existe en los programas cliente-servidor, en las bases de datos, en los sistemas operativos en los distintas tareas utilizan el mismo recurso (por ejemplo: pilas, colas, impresoras) o en programas concurrentes orientados a objetos en los que sus hilos necesitan tener acceso al código de un mismo objeto.

Concurrencia cooperativa, que se da cuando varios elementos cooperan, es decir, cuando realizan un trabajo conjunto para lograr unos objetivos comunes. La comunicación entre ellos puede ser haciendo uso de recursos compartidos o explícita, lo importante es que han sido diseñados de manera conjunta y que tienen que sincronizar sus ejecuciones y esperarse los unos a los otros. Ejemplos de este tipo de concurrencia se encuentre en los sistemas en tiempo real, en las aplicaciones de control o en los algoritmos sistólicos o de elección de un líder.

Esta clasificación de concurrencia no está explícitamente relacionada con las primitivas que se utilicen para llevar a cabo la sincronización y el intercambio de información. Éstas son generalmente de muy bajo nivel y, en general, se pueden construir programas cooperativos o competitivos utilizando las mismas primitivas, aunque algunas de ellas son más adecuadas para un tipo de concurrencia en concreto. Por ejemplo, las llamadas a procedimientos remotos, la invocación a métodos de objetos, la memoria compartida y los monitores son utilizados normalmente por los elementos activos en un contexto competitivo para tener acceso a los mismo elementos pasivos; no obstante, también pueden ser utilizados en contextos cooperativos por dos clientes que cooperan a través de un servidor. El *Rendez-vous*, el paso de mensajes con emisores y receptores conocidos o las señales son más adecuados en relaciones cooperativas.

En esta tesis doctoral no centramos en ambas concurrencias, la competitiva y la cooperativa, siempre y cuando la sincronización y comunicación le lleve a cabo a través de interacciones multipartitas.

1.1.2 Modelos de interacción

La coordinación es la piedra angular de la mayoría de las actividades actuales. Malone la caracteriza como un campo de investigación emergente con un foco multidisciplinar que juega un papel importante en varias disciplinas tales como la economía o la biología [89], así que no es de sorprender que también afecte a la informática actual. Francez y Forman, por ejemplo, afirman que la coordinación es la razón de ser de los sistemas distribuidos [53].

Por desgracia, no existe una existe una definición de coordinación general que prevalezca sobre las demás. Éstas van desde las más simples como "coordination is managing dependencies between activities" [89], hasta las más complejas como "coordination is the additional information processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goals would not perform" [88].

En el contexto de los lenguajes de programación la definición más ampliamente aceptada parece ser la de Carriero [27]:"coordination is the process of building programs by gluing together active pieces", la forma en la que los elementos activos interactúan entre sí depende completamente del modelo de coordinación utilizado. En otras palabras, es el modelo de coordinación adoptado el que marca cómo los elementos que componen un programa concurrente se sincronizan e intercambian información para alcanzar un objetivo [53]. En la bibliografía a estos modelos se les hace referencia como modelos de interacción, y este es el término que utilizaremos en el resto de la memoria.

Papadopoulos y Arbab presentan un estado del arte en modelos de interacción en la Referencia [99] que los define como tuplas de la forma (E, L, M), donde E representa a los elementos que son coordinados, L el medio que utilizan para llevar a cabo la coordinación y M la semántica del marco de trabajo en el cual el modelo se incorpora. Así, un lenguaje de interacción pode ser visto como la personificación lingüística de un modelo de interacción, ofreciendo facilidades sintácticas para expresar la sincronización, la comunicación y la creación y terminación de actividades.

1.1.3 Interacciones multipartitas

La mayoría de las interacciones que se producen en el mundo real son multipartitas, ya que requieren que varios participantes se coordinen para alcanzar un objetivo. Sin embargo, no son muchos los modelos de interacción que permiten que múltiples elementos interactúen entre sí ya que sólo proporcionan primitivas bipartitas, por ejemplo, el *rendez-vous*, las llamadas a procedimientos remotos, el paso de mensajes, los buzones o los espacios de tuplas. Hasta los más modernos sistemas *peer-to-peer* son bipartitos y hacen hincapié en que sólo dos elementos intercambien información.

Aunque estas primitivas sean cada vez más usadas y existan muchas implementaciones eficientes, no existen aún buenas herramientas que puedan utilizarse para modelar sistemas complejos y reales en los que las interacciones son multipartitas. La razón es que, normalmente, éstas tienen que ser divididas en un conjunto de interacciones bipartitas más un protocolo de coordinación. A menudo, reutilizar o razonar estos protocolos es una tarea difícil, lo que justifica la necesidad de disponer de soluciones que las que las interacciones complejas entre elementos pueda ser expresada con mayor naturalidad.

La Figura §1.1 esquematiza cómo el problema clásico de la elección de líder puede ser representado utilizando interacciones multipartitas y utilizando un conjunto de interacciones bipartitas con un protocolo. Obviamente, la solución de la izquierda es más simple desde el punto de vista conceptual, en cambio, la de la derecha es más eficiente ya que existen multitud de métodos automáticos bien conocidos para transformar interacciones multipartitas en protocolos bipartitos [36, 38, 106]. Así, estos métodos alivian al programados de la tarea tediosa de codificarlas manualmente, y le permite reutilizarlas con seguridad y más fácilmente. Además, está probado que razonar sobre programas que están expresados en términos de interacciones multipartitas es más simple que cuando éstos se expresan con interacciones bipartitas [53].

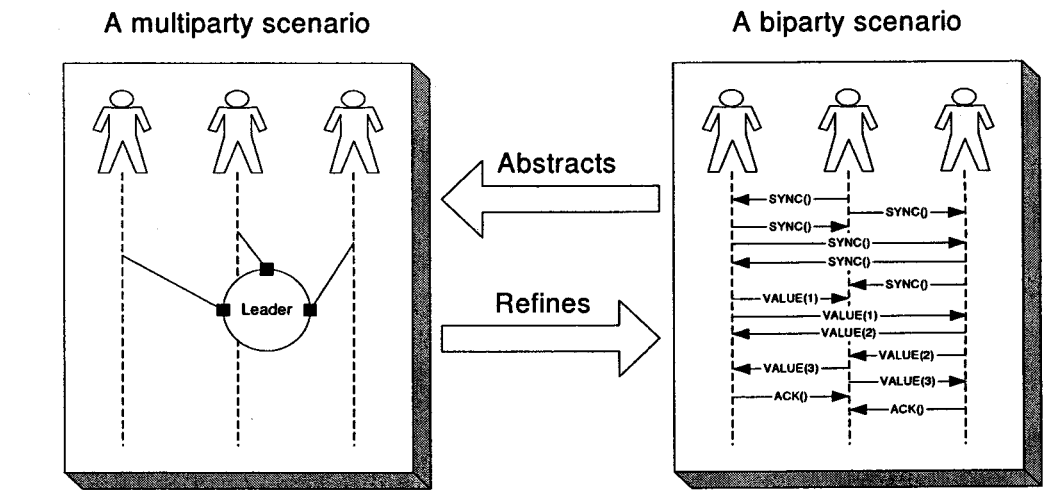


Figura 1.1: *Interacciones multipartitas y bipartitas.*

En la actualidad, existen muchos modeles y lenguajes de carácter académico que soportan interacciones multipartitas, por ejemplo, *e.g.*, POLYPHONIC C♯ [16], IP [53], CAL [36], LOTOS [20], EXTENDED LOTOS [23] o RADDLE [51]; además, el número de lenguajes comerciales que están empezando a soportarlas está en constante crecimiento, prueba de ello es el XLANG de Microsoft [126], el WSFL de IBM [84], o la propuesta conjunta que recientemente ambos han realizado llamada BPEL4WS [121].

1.1.4 Selección justa

Las nociones de selección justa permiten a los programados abstraerse de los detalles de bajo nivel que conciernen a cómo los elementos que componen el programa son gestionados o cómo el sistema operativo subyacente implementa una primitiva en concreto en un entorno concurrente. No existe una definición simple que prevalezca sobre las demás, pero todas ellas giran en torno a la misma idea: no se pueden rechazar todas las oportunidades que un elemento tiene de ejecutarse [97].

Por ejemplo, consideremos que un elemento del programa ejecuta la siguiente instrucción alternativa múltiple de Dijkstra [44]:

[true $\rightarrow$ **skip** [] true $\rightarrow$ x := x + 1; print(x)]

Ésta establece que no tiene importancia si x es incrementada o no, y una implementación de dicha instrucción que no tuviera la segunda alternativa en

cuenta, en principio, sería tan correcta como otra que seleccionara alternativas de acuerdo a un generador de números aleatorios. Sin embargo, si incorporamos a este pequeño fragmento de código el siguiente bucle, la única forma de garantizar que el programa termina es asumiendo que ambas alternativas son ejecutadas con el tiempo.

```
x := 0;
*[ x < 10 →
   [ true → skip
   [] true → x := x + 1; print(x)
   ]
]
```

Desde un punto de vista intuitivo, parece lógico pensar que este programa imprimirá la secuencia 1, 2, . . ., 10 y terminará. No obstante, suponiendo que el tiempo que tarda en ejecutarse una instrucción ***skip*** es de unos milisegundos, el tiempo que tardaría en imprimir dicha secuencia variaría de ejecución en ejecución. Si suponemos que el no determinismo está embebido en la implementación de este programa, nos gustaría estar seguros de que ambas alternativas tienen oportunidad de ejecutarse independientemente de cómo estas instrucciones son implementadas. En otras palabras, nos gustaría ser capaces de asumir que la implementación es justa.

La selección justa es el concepto clave en la mayoría de los modelos multipartitos ya que éstos normalmente permiten que un elemento esté a la espera de participar en varias interacciones que son mutuamente excluyentes. Así, la única forma de garantizar que toda interacción tiene oportunidad de ser ejecutada cuando se encuentra habilitada es asumir que la implementación es justa.

1.2 Resumen de aportaciones

Nuestra tesis doctoral se centra en el estudio de la selección justa en el contexto del modelo de interacción soportado por el lenguaje IP_{CORE} [53]. Hemos analizados las nociones clásicas de selección justa cuidadosamente y hemos llegado a la conclusión de que éstas normalmente no son adecuadas para ser aplicadas en programas reales ya que adolecen de dos problemas prácticos: están basadas en el concepto de eventualidad y la infinitud, y, como sabemos, estos conceptos no son implementables, además, no tienen en cuenta el problema de las conspiraciones, es decir, aquellas situaciones en las que un entrelazado desafortunado de acciones independientes impide que una interacción se habilite y en consecuencia no pueda ser ejecutada.

Nuestra principal aportación es la definición de una nueva noción a la que hemos llamado k-*conspiracy-free fairness*. Sus principales ventajas son:

i. Soluciona los problema antes mencionados de forma simultanea cuando el resto de nociones lo hacen de forma separada.

ii. El valor de k puede ser asignado de antemano para controlar el bondad con la que se resuelven ambos problemas.

iii. Se puede aplicar a cualquier programa cuando algunas otras nociones sólo pueden ser aplicada a un subconjunto de programas en formal normal.

iv. Proporcionamos un algoritmo genérico no transformacional que la implementa cuando otras nociones son implementadas haciendo uso sólo de gestores *ad hoc*.

También proporcionamos un marco de trabajo teórico para definir el modelo de interacción entre múltiples participantes que nos permite independizarnos del lenguaje de programación que lo soporta. Este marco de trabajo es lo suficientemente genérico como para poder definir otras nociones de selección justa así como propiedades *safety* y *liveness*. Además, también proporcionamos una implementación $\text{J}^\sharp$ eficiente de este marco de trabajo que nos sirve para validarlo.

1.3 Estructura

Esta memoria está organizada de la siguiente forma:

Parte I: Prólogo. Contiene sólo esta introducción.

Parte II: Estado del arte. Nuestro objetivo en esta parte es proporcionar al lector de una profundo entendimiento del contexto de investigación en el que nuestros resultados han sido desarrollados. En el Capítulo §2, documentamos el modelo de interacción en el hemos utilizados y lo ilustramos con algunos ejemplos; ya que no es el único modelo de interacción existente, hemos actualizado la taxonomía de Joung con las propuestas más recientes. En el Capítulo §3 nos centramos en la selección justa y presentamos una idea general de las nociones y marcos de trabajos actuales; también tratamos algunos temas controvertidos sobre la selección

justa. Por último, en el Capítulo §4, algunos algoritmos para implementar selección justa son descritos así como algunos resultados de imposibilidad de implementar *strong fairness* haceindo uso de algoritmos sin esperas; el capítulo concluye con algunos resultados experimentales obtenidos cuando ningún criterio de selección justa es implementado.

Parte III: Nuestra aportación. Es el núcleo de la tesis, y está organizada en 5 capítulos. En el Capítulo §5 motivamos nuestra investigación y mostramos cómo las nociones clásicas de selección justa no lo suficientemente prácticas. En el Capítulo §6 presentamos nuestro marco de trabajo para definir el modelo de interacción que usamos, dando ejemplo de su expresividad. En el Capítulo §7 definimos nuestra noción de selección justa y probamos que es correcta. En el Capítulo §8 presentamos los tipos abstractos de datos y algoritmos necesarios para implementar tanto el marco de trabajo como la noción; además demostramos que es correcto también. Por último, en el Capítulo §9 se muestran los resultamos obtenidos en el estudio experimental que nos ayuda a validar nuestra propuesta.

Parte IV: Notas finales. Es un capítulo en el que resumimos las principales conclusiones y las líneas de investigación futuras.

Parte V: Apéndices. La notación utilizada se encuentra resumida en el Apéndice §C. En el Apéndice §B presentamos avista de pájaro la semántica de IP_{CORE}. Por último, nuestro primer intento de resolver el problema de la finitud y las conspiraciones se muestra en el Apéndice §A.

Parte II

Estado del arte

Capítulo 2

Interacciones multipartitas

(9) Two are better than one, because they have a good return for their work;
(12) Though one may be overpowered, two can defend themselves.
A cord of three strands is not quickly broken

Ecclesiastes, Chapter 4

Nuestro objetivo en este capítulo es proporcionar al lector las bases de las interacciones multipartitas. En la Sección §2.1 motivamos la necesidad de las mismas y presentamos un estudio detallado que identifica los distintos campos en los que éstas juegan un papel importante; en la Sección §2.2 presentamos una taxonomía de los lenguajes de programación que las soportan; ya que existen varios modelos de interacciones multipartitas que, además, son muy similares entre sí, necesariamente tenemos que centrar nuestro estudio sólo en uno, cuya semántica es descrita en la Sección §2.3; ilustramos este modelo a través de ejemplos en la Sección §2.4.

2.1 Introducción

Las primitivas de interacción como las llamadas a procedimientos remoto o el paso de mensajes son el estándar industrial *de facto* que se utiliza para comunicar elementos software entre sí, además, cada el número de propuestas como SOAP [22, 47] o .NET REMOTING [110] está creciendo rápidamente. Utilizar estas primitivas para modelar sistemas con interacciones complejas entre múltiples elementos a menudo requiere dividirlas en interacciones bipartitas y protocolos de comunicación. En general, la construcción de estos protocolos no es trivial y el resultado es a menudo poco reutilizable y dificulta el razonamiento sobre los programas [53]. Por consiguiente, la preocupación por abstraernos y conseguir esta reutilización sustenta las primitivas en las que las interacciones multipartitas puedan ser expresadas directamente, aunque después sean transformadas automáticamente en eficientes protocolos bipartitos. Así evitamos la programador la tarea tediosa de tener que codificar, depurar y validar. [36, 38, 106]. Además, también está demostrado que este tipo de interacciones optiman el nivel de concurrencia en algunos escenarios [125].

Las situaciones en las que varios elementos necesitan interactuar de manera conjunto son habituales en la práctica [53]: la transferencia de dinero de un banco a otro a través de un terminal (tres elementos) [50, 112], el pago de impuestos en España a través de Internet (tres elementos: el pagador, el cobrador y una autoridad de certificación), las aplicaciones de comercio electrónico [49] (un cliente, un proveedor de servidor y una tercera entidad que presenta esos servicios a los clientes) o alcanzar un acuerdo en una subasta (múltiples elementos).

La necesidad de disponer de primitivas de interacción multipartitas ha sido reconocida en varios contextos:

Metodologías de análisis y diseño: La mayoría de ellas proporcionan a los programadores herramientas para modelar colaboraciones entre múltiples objetos. La forma de referirse a éstas difiere según la metodología: diagramas de componentes [21], modelo de procesos [28], conexiones de mensajes [33], digramas de flujo de datos [117], grafos de colaboración [128], diagramas de situación [111] o colaboraciones [46, 118]. Estás propuestas están acompañadas de un conjunto rico de ejemplos que muestran la adecuación de las mismas en diversos campos como son las finanzas, las telecomunicaciones, las aseguradoras, empresas de manufactura, sistemas empotrados, procesos de control, simulación de vuelo, viajes y transportes o sistemas de gestión.

Aplicaciones Web: Recientemente, el aumento de la complejidad de las aplicaciones Web y el más concretamente las tecnologías elaboradas, han

motivado que las mayoría de las compañías de software desarrollen herramientas para soportar interacciones multipartitas en el contexto de Internet. Microsoft, por ejemplo, ha propuesto un marco de trabajo llamado .NET ORCHESTRATION [85] para ayudar a los programadores a coordinar un número arbitrario de elementos de red, como son, servicios Web, colas de mensajes o componentes DCOM y .NET. La calve de este marco de trabajo es el lenguaje XLANG [126], que no es más que una extensión del WSDL. IBM también propuso un lenguaje llamado WSFL [84] como base para describir interacciones complejas entre múltiples procesos de negocio. Fue diseñado para ser el núcleo de su WEB SERVICES TECHNOLOGY FRAMEWORK y complementar otras especificaciones existentes como son XMLP, SOAP, WSDL o UDDI. Recientemente, ambas propuestas se han unificado en macro de trabajo BPEL4WS [121], que consiste en un conjunto de componentes que proporcionan la infraestructura necesaria para soportar interacciones multipartitas entre servicios Web más un lenguaje formal de especificación para definir el proceso de negocio y las interacciones entre los mismos.

Marco de trabajo JAVA para la concurrencia: Las interacciones multipartitas han llamado también la atención de muchos investigadores que han desarrollado varios marcos de trabajo JAVA para soportarlas. En la Referencia [81], los autores proponen una marco que soporta sincronización por barrera, la cual puede ser vista como una forma simple de interacción multipartita; este marco ha sido la base para la especificación #166 del JAVA COMMUNITY PROCESS [82]. En la Referencia [50], lo autores presenta un marco para soportar interacciones 1-a-n en las que un objeto puede solicitar que varios objetos interactúen con él atómicamente. La propuesta de la Referencia [72] parte de SR [7, 59] y extiende JAVA para proporcionar un modelo de concurrencia rico que incluye interacciones bipartitas al estilo de ADA. Estas interacciones fueron generalizadas al caso multipartito en la Referencia [30]. En la Referencia [60], otro marco JAVA que soporta interacciones bipartitas al estilo CSP fue introducida. Desafortunadamente, excepto para el marco de la Referencia [36], ninguna de estas propuestas puede ser vista como una implementación completa de las interacciones multipartitas ya que éstas ni son asimétricas [50, 72], ni pueden ser guardadas por una alternativa múltiple [81, 50].

Sociedades de agentes: Recientemente, la necesidad de las interacciones multipartita también ha sido reconocida en el campo de las sociedades multiagentes [6, 14, 15, 26, 95]. Estas interacciones son normalmente utilizadas en las etapas tempranas durante el diseño de la sociedad, aunque existen varias propuestas que las utilizan para implementar las relaciones de comportamiento entre los agentes [101, 102, 104].

Lenguajes de investigación: Más allá de JAVA, existen muchos lenguajes de investigación que soportan interacciones multipartitas. Una taxonomía puede encontrarse en la Referencia [70], y una actualización de la misma se presenta en la siguiente sección.

2.2 Una taxonomía de modelos de interacción multipartitos

Aunque es usual hacer referencia a *el* modelo de interacción entre múltiples participantes como si éste fuese único, no existe una abstracción simple, sino toda una colección de modelos con una constante común: permitir que varios elementos interactúen de forma coordinada usando primitivas de alto nivel. Desafortunadamente, pocas referencias hacen distinción entre el modelo de interacción o la propia interacción y el lenguaje de programación que las soporta, lo que causa confusión en algunos casos. Esta es la razón por la que nosotros nos hacemos referencia modelos de interacción concretos usando el nombre de uno de los lenguajes que lo implementa.

En la Referencia [65], Joung hizo el esfuerzo de clasificar los lenguajes de programación que soportan interacciones multipartitas. Primero, dividió las primitivas que éstos soportan en cuatro tipos: *channels*, *ports*, *gates* y *teams* (véaseTabla §2.1). Después los clasificó dependiendo de si eran síncronos o asíncronos y del nivel de paralelismo que se permite entre los elementos:

Null: Aunque no es usual, lenguajes comoSCRIPT o COMPACT no permiten a ningún elemento que ejecute más de una interacción al mismo tiempo.

***Conjunctive*:** En este caso, el modelo permite que sus elementos ejecuten grupos de interacciones que son expresadas a través de las llamadas *instrucciones compuestas*. Así, o todas las interacción que forman una interacción compuesta se ejecutan o ninguna de ellas lo hace, es decir, algo parecido a una transacción.

***Disjunctive*:** Este nivel permite que las interacciones sean utilizadas como guardas en instrucciones alternativas múltiples. Así, sus elementos puede expresar que sean interactuar a través de una interacción entre un conjunto de éstas, aunque varias a la vez no pueden ser ejecutadas.

***Disjunctive normal form*:** Este nivel permite que los guardas de una alternativa múltiple incluyan instrucciones compuestas. En este caso, cada alternativa permite que los elementos expresen que quieren participar

Primitiva	Descripción	Lenguajes
Channel	Definen una interacción entre 2 elementos conocidos.	CSP [62], MULTIPARTY CSP [68], OCCAM [61], Pεω [9]
Port	Definen una interacción que entre dos papeles que pueden ser jugados por diferentes elementos.	ADA [96], CCS [93], GMS [58], MEIJE [40]
Gate	Definen una interacción entre un conjunto fijo de elementos conocidos.	CSPS [57], CIRCAL [92], RADDLE [51], LOTOS [20], EXTENDED LOTOS [23], ACTION SYSTEMS [11], IP_{CORE} [53], CAL [37], CLIP [34]
Team	Definen una interacciones entre un conjunto fijo de papeles que pueden ser jugados por diferentes elementos.	SCRIPT [54], RADDLE, IP [53], JPS [101], BPEL4WS [121]

Tabla 2.1: *Taxonomía de lenguajes que soportan interacciones multipartitas.*

en un subconjunto de interacciones, y que todo ese subconjunto sea ejecutado a la vez.

***Disjunctive-conjunctive*:** Este es un nivel restrictivo en el cual se permite *disjunctive* pero la alternativas múltiples no pueden utilizar instrucciones compuestas.

La Figura §2.1 esquematiza y actualiza la taxonomía de Joung. Fíjese que los lenguajes que soportan sólo interacciones bipartitas como ADA, CCS, CSP o OCCAM han sido omitidos. Además, nuevos lenguajes que no existían cuando Joung hizo esta clasificación han sido añadido con un tipo de letra *cursiva*.

2.3 Semántica

En esta tesis nos centramos en el lenguajes IP_{CORE}, que destaca por ha sido concebido para tener un papel dual- Por un lado, viene equipado con una semántica completa que lo convierte en un lenguaje apto para el razonamiento formal; así, resulta bastante adecuado para especificar formalmente sistemas concurrentes. Por otro lado, también fue concebido para como lenguaje ensamblador a través del cual otros lenguajes de más alto nivel puedan ser implementados [34].

En el Apéndice §B, se presenta una introducción detallada a IP_{CORE}. En esta sección nos centraremos estrictamente en la semática del modelo de interacción que proporciona ya que es la clave para el entendimiento de por qué la

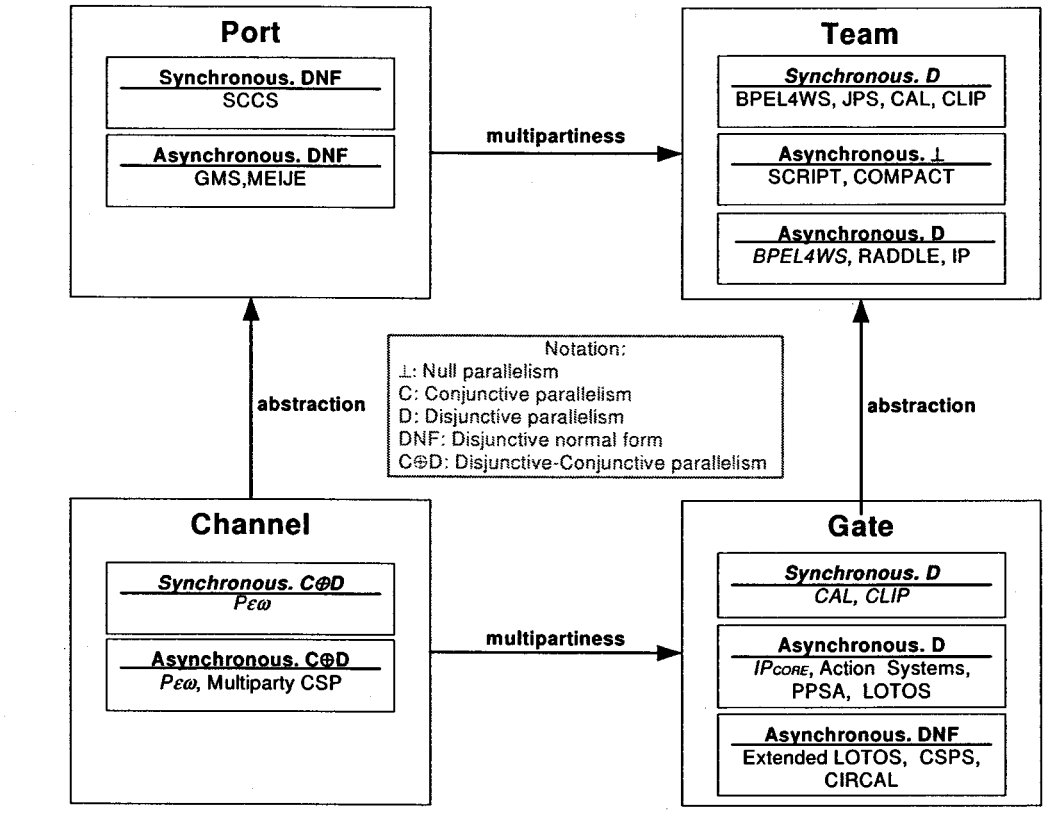


Figura 2.1: *Taxonomía de lenguajes que soportan interacciones multipartitas.*

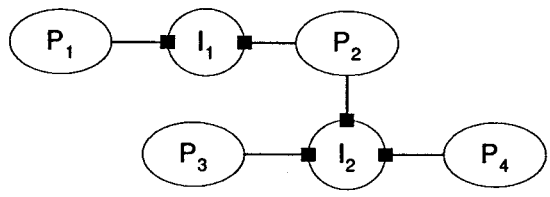


Figura 2.2: *Sistema con 4 participantes y 2 interacciones.*

selección justa es tan importante en este contexto. Como quiera que sea, centrarnos en este lenguaje no implica perdida de generalidad ya que los resultados que presentamos en esta tesis doctoral pueden ser facilmente aplicados a cualquier modelo de interacción o lenguaje de programación en el que el conjunto de interacciones sea finito u conocido de antemano, lo cual supone una restricción bastante real.

En IP_{CORE}, los elementos son modelados como procesos (también llamados participantes) que poseen su propio estado local y que se comunican unos con los otros sólo a través de interacciones multipartitas. Los procesos poseen un único hilo y soportan paralelismo *disjunctive*. Así, aunque, esto puede ser visto como una desventaja, en realidad no los es, ya que tres hilos puedes ser vistos como tres procesos IP_{CORE} independientes.

Las interacciones son identificadas por un nombre único y tienen un conjunto fijo de participantes que necesita ser conocido de antemano. Para que una interacción se habilite, es necesario que todos sus participantes la ofrezcan ejecutando una instrucción de la forma $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, donde A es el nombre de la interacción y $x_1 := e_1; \ldots; x_n := e_n$ es la instrucción de comunicación. Una vez que una interacción se habilita, podrá ser ejecuta tan pronto sus participantes confirme su interés, es decir, retiren los ofrecimientos realizados a otras interacciones. (A menudo, se dice que la interacción seleccionada consigue la exclusión mutua sobre sus participantes). Cuando una interacción es seleccionada para su ejecución, un estado que combina una copia local de los estados de cada uno de los participantes de la interacción se crea. Este estado combinado es compartido por todos ellos y puede ser visto como una pizarra que permite que un proceso tenga acceso a la información de otros procesos a través de instrucciones de asignación. En el ejemplo anterior, cuando un proceso ejecuta con éxito $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, se tiene que se ha puesto de acuerdo con los otros participantes para ejecutar la interacción, y que luego ha accedido a las expresiones $e_1, e_2, \ldots, e_n$ del estado de los otros participantes transfiriendo los resultados a las variables $x_1, x_2, \ldots, x_n$ de su propio estado local. Este simple mecanismo de comunicación fue mejorado, por ejemplo, en [39] pero los detalles se salen del alcance de esta tesis.

Es importante destacar que una interacción habilitada puede no ser ejecutada. La Figura §2.2 esquematiza un sistema simple que está formado por cuatro procesos y dos interacciones bipartitas y 1 tripartita. Fíjese que el proceso P_1 puede ofrecer participar en I_1 y que P_3, P_4 pueden ofrecer participar en I_2 y P_2 puede ofrecer participar en I_1 o I_2. Esto significa que estas interacciones no pueden ser ejecutadas simultáneamente ya que IP_{CORE} sólo soporta paralelismo *disjunctive*. De esta forma, una selección entre ellas debe ser llevada a cabo y es importante selecciones justamente para evitar rechazar una de ellas por siempre.

```
        { We assume that subindex arithmetic is module N }
DINNER :: [||_{i=1}^{N} P_i || F_i], where
    P_i :: *[ Get_i⟨⟩ → eat; Rel_i⟨⟩; think ];
    F_i :: *[ Get_i⟨⟩ → Rel_i⟨⟩
            [] Get_{i+1}⟨⟩ → Rel_{i+1}⟨⟩
            ].
```

Programa 2.1: *Los filósofos comensales en* IP_{CORE}.

2.4 Algunos ejemplos

Nuestro objetivo en esta sección es ilustrar IP_{CORE} a través de algunos ejemplos no triviales. Consulte el Apéndice §B para una descripciçon detallada de las instrucciones que utilizamos.

2.4.1 Los filósofos comensales

Ilustraremos la sincronización multipartita a través del conocido problema de los filósofos comensales [43]. Consiste en sentar en una mesa a cinco filósofos que se dedican es estar un tiempo pensando y luego comer. Existe un único tenedor entre cada par de filósofos vecinos y éstos necesitan coger los tenedores de ambos lados antes de comer. Además, cada filósofo debe comer tantas veces como el resto, es decir, las ejecuciones de este programa han de ser justas. Este problema es núcleo de una amplia clase de problemas en los que se requiere que un elemento adquiera un conjunto de recursos en exclusión mutua.

La solución obvia a este problema usando interacciones bipartitas consiste en coger los tenedores en orden. Sin embargo, el problema surge cuando todos los filósofos cogen los tenedores que están a su izquierda y luego esperan a que el tenedor que tiene a su derecha quede libre. En este caso, un interbloqueo ocurre todos los filósofos sufrirían inanición. En la Referencia [87], los autores demuestran que cualquier solución en la que los filósofos sean simétricos y no exista comunicación entre ellos puede conducir a interbloqueo. Así, una solución correcta debe contar con que los exista alguna diferencia entre los filósofos. Estas soluciones normalmente no son escalables y reutilizables, ya que las diferencias que los filósofos deben implementar depende de la topología del problema en consideración.

Si utilizamos interacciones multipartitas, cada filósofo cogería sus dos tenedores al mismo tiempo de forma que es imposible que se den interbloqueos.

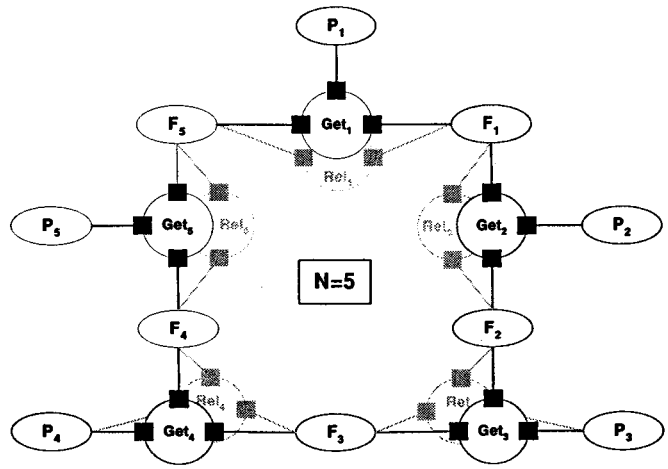


Figura 2.3: *Vista estática de los filósofos comensales.*

El Programa §2.1 muestra una solución a este problema y la Figura §2.3 una vista estática. Los filósofos son representados por P_i y los tenedores por F_i ($i \in [1, N]$). Cada P_i está siempre intentando coger sus tenedores para participar en la interacción tripartita llamada Get_i junto con F_i y F_{i-1}. Así, adquirir un recurso es especificado como la sincronización entre los procesos participantes en la interacción. Después de que P_i ha cogido sus tenedores, come, los suelta y después vuelve a pasar un tiempo pensando. Fíjese que las interacciones Get_i y Get_{i+1} están en conflicto cuando ambas se encuentran habilitadas al mismo tiempo. La única forma de garantizar que se habilita lo suficiente a menudo termina siendo ejecutada consiste en asumir que el resolutor de conflictos subyacente es justo.

2.4.2 La elección del líder

Ilustramos como funciona la comunicación multipartita a través del problema de la elección del líder [80]. Éste tiene una gran importancia ya que también es el núcleo de una gran gama de problemas en los que varios elementos tienen que ejecutar un algoritmo, pero no existe un candidato para hacerlo de antemano. Por lo tanto, una elección entre los mismos debe ser llevada a cabo.

Esta situación puede ser el caso de un proceso de inicialización que debe ser ejecutado al comienzo de una actividad o de un procedimiento de rescate que deba ser ejecutado después de una caída de un sistema distribuido. Es imposible asignar a un elemento el papel de líder porque quizás sea el que esté

LEADER :: $[\|_{i=1}^{N} M_i]$, **where**
M_i :: { w_i: *natural; leader*$_i$: *boolean* }
w_i := *a weight;*
Elect⟨leader$_i$:= (w_i = $\max_{1 \le j \le n} \{w_j\}$)⟩;
[leader$_i$ → *execute algorithm*].

Programa 2.2: *La elección del líder en* IP_{CORE}.

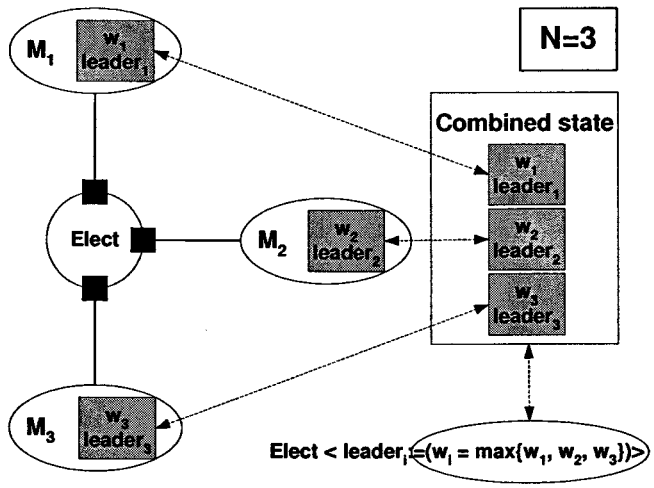


Figura 2.4: *Vista estática de la elección del líder.*

fallando y entonces el sistema tendría que ser reiniciado. El criterio utilizado para seleccionar a un líder es muy simple: cada uno de los elementos tiene asociado un peso natural diferente, por ejemplo, su dirección de red, y el líder es aquel que tenga el mayor peso. Fíjese que se trata de un claro ejemplo en el que se necesita que varios elementos colaboren simultáneamente en una interacción multipartita ya que en caso contrario no hay forma de seleccionar un líder si un elemento no conoce los pesos del resto de elementos.

Una solución IP_{CORE} a este problema se muestra en el Programa §2.2, cuya vista estática es la mostrada en la Figura §2.4. La interacción multipartita *Elect* sincroniza a todos los procesos y les permite que intercambien información para decidir cuál de ellos va a jugar el papel de líder. Cuando varios procesos se sincronizan e interactúan, un estado global combinado temporal es creado de forma que todos los procesos pueden leer la información del estado del resto de procesos y transferirla a su propio estado local. De esta forma, la instrucción de asignación *leader*$_i$:= (w_i = $\max_{1 \le j \le N} \{w_j\}$) nos permite que cada participante

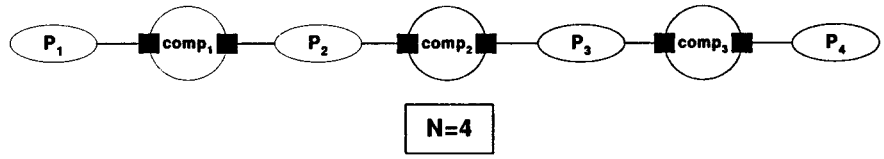


Figura 2.5: *Vista estática del algoritmo* OE SORT.

calcule el si su peso es el máximo y guardar el resultado en la variable local $leader_i$. Una vez que la interacción es ejecutada, aquél que detecte que tiene el mayor peso ejecutará el algoritmo adecuado.

Recapitulando, la comunicación multipartita es una comunicación simétrica que atañe a un número arbitrario de procesos. Esto es una forma distinta de ver la comunicación, que tradicionalmente a sido dividida en enviar y recibir datos.

2.4.3 El algoritmo OE SORT

Supongamos un programa compuesto de N procesos a los que denotamos como $P_1, P_2, \ldots, P_n$, y que almacenan un valor al que llamamos $a_1, a_2, \ldots, a_n$, respectivamente. El objetivo el coordinarlos de forma que después un número finito de interacciones para interacambiar sus valores el estado final satiface que $a_1 \leq a_2 \leq \ldots \leq a_n$. El algoritmo OE SORT de Batcher [13] resulta el más adecuado en este escenario, y Tang y Muraoka lo utilizan para mostrar que las interacciones multipartitas pueden ser usadas para conseguir un paralelismo óptimo [124].

El algoritmo OE SORT puede resolverse en $\frac{N}{2}$ fases duales: en las fases impares, los participantes comparan e intercambian (si procede) los datos con los vecinos pares que le encuentra en la derecha; en las fases pares serán los procesos pares son los que comparan e intercambian sus valores con los vecinos impares de la derecha.

Por ejemplo, durante una fase impar, P_1 y P_2, P_3 y P_4, P_5 y P_6, etcétera son los que intercambian sus valores si estos no se encuentran ya ordenados; durante las fases pares, P_2 y P_3, P_4 y P_5, P_6 y P_7, etcétera harán lo mismo. Está probado que después de que k fases para-impar son ejecutadas, ningún procesos está a más de $N - 2k$ pasos del estado final ordenado. Así, cuando k is igual a $\lceil \frac{N}{2} \rceil$, todos los procesos habrán llegado a su estado final.

```
OESORT :: [||_{i=1}^{N} P_i], where
    P_1 :: { a_1: natural; result_1: natural; j: natural }
        a_1 := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ & comp_1⟨a_1 := min(a_1, a_2)⟩ → j := j + 1 ];
        result_1 := a_1;
    P_N :: { a_N: natural; result_N: natural; j: natural }
        a_N := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ & comp_{N-1}⟨a_N := max(a_{N-1}, a_N)⟩ → j := j + 1 ];
        result_N := a_N;
P_{i ∈ (1, N)} :: { a_i: natural; result_i: natural; j: natural }
        a_i := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ →
            [ odd(i) & comp_i⟨a_i := min(a_i, a_{i+1})⟩ → skip
            []   even(i) & comp_{i-1}⟨a_i := max(a_{i-1}, a_i)⟩ → skip
            ];
            [ even(i) & comp_i⟨a_i := min(a_i, a_{i+1})⟩ → skip
            [] odd(i) & comp_{i-1}⟨a_i := max(a_{i-1}, a_i)⟩ → skip
            ];
            j := j + 1;
        ];
        result_i := a_i.
```

Programa 2.3: *El algoritmo* OE SORT *en* IP_{CORE}.

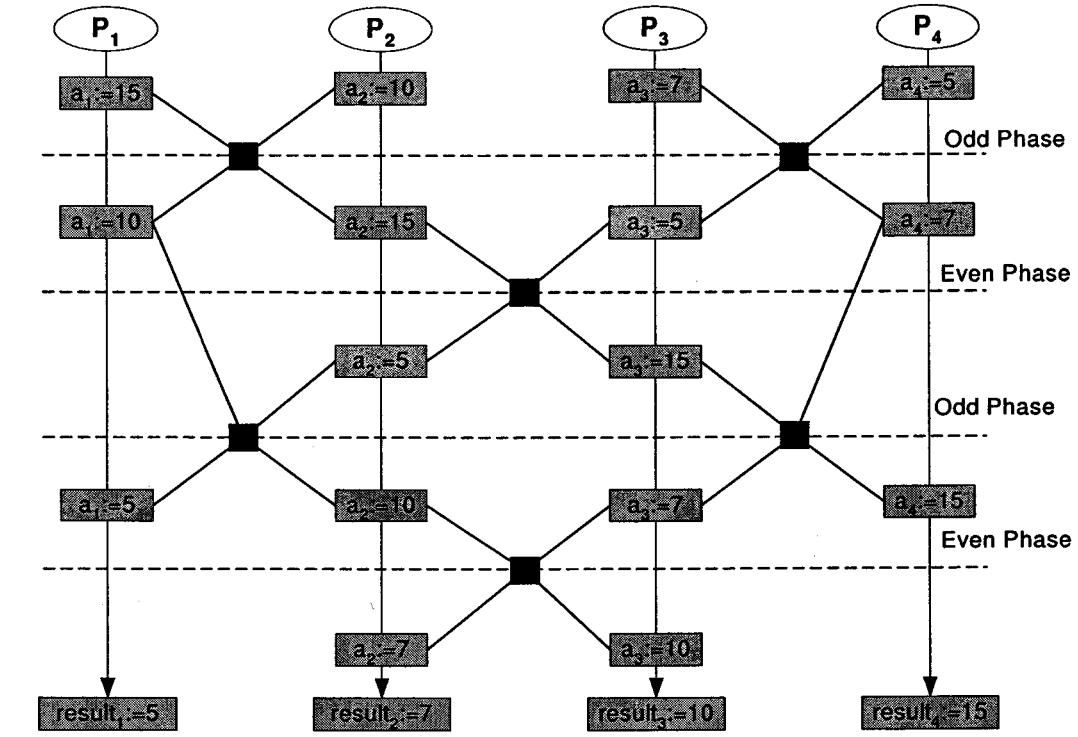


Figura 2.6: *Ejecución de ejemplo del algoritmo* OE SORT.

El Programa §2.3 muestra una implementación IP_{CORE} inspirada en la solución de Quinn [109], y la Figura §2.5 muestra una vista estática cuando N = 4. La Figura §2.6 ilustra una ejecución en la que sólo 2 fases par-impar son necesarios para ordenar lso valores.

2.4.4 El protocolo ALOHA

Existen muchos protocolos para comunicar estaciones de trabajo a través de un medio de transmisión compartido que no es capaz de transmitir más de un mensaje al mismo tiempo. El protocolo ALOHA es una solución clásica de gran interés ya que es la base de la red más universal, la Ethernet[1, 123].

En el protocolo ALOHA original, la estaciones de trabajo se comunican las unas con las otras a través de un satélite. Cuando una estación de trabajo necesita transmitir un mensaje, es el satélite el encargado de difundirlo a todas las estaciones de trabajos (incluida la emisora); en cambio, cuando varias estaciones quieren transmitir un mensaje a la vez, el satélite detecta esta colisión y propaga un mensaje espacial para indicarlo, entonces las estaciones tienen que

```
SALOHA :: [Satellite || ||_{i=1}^{N} Station_i], where
  Satellite :: { broadcast: message; vector: array (1..N) of message }
               *[ true → ascend⟨vector(1) := sent_1; ...; vector(N) := sent_N⟩;
                  [ collision(vector) → broadcast := COLLISION
                  [] ¬ collision(vector) → broadcast := extract(vector)
                  ];
                  descend⟨⟩
               ];
Workstation_i :: { sent_i, recv_i, save_i: message; waitfor_i: natural }
               sent_i := EMPTY; recv_i := EMPTY;
               *[ true →
                  [ sent_i = EMPTY → sent_i := prepare message ];
                  ascend⟨⟩; descend⟨recv_i := broadcast⟩;
                  [ recv_i = sent_i → sent_i := EMPTY
                  [] recipient(recv_i) = i → process message
                  [] recv_i = COLLISION ∧ sent_i ≠ EMPTY →
                     waitfor_i := a random number;
                     save_i := sent_i; sent_i := EMPTY;
                     *[ waitfor_i > 0 →
                          ascend⟨⟩; desced⟨recv_i := broadcast⟩;
                          [ recipient(recv_i) = i → process message ];
                          waitfor_i := waitfor_i - 1
                     ];
                     sent_i = save_i
                  ]
               ].
```

Programa 2.4: *El protocolo* ALOHA *en* IP_{CORE}.

espera por un tiempo aleatorio antes de intentar enviar el mensaje de nuevo. Afortunadamente, este retraso aleatorio provoca que las estaciones implicadas en la colisión no vuelvan a colisionar.

Esta idea básica tiene muchas variantes. Una de las más interesantes es la llamada SLOTTED ALOHA que difiere en que no se permiten que las estaciones envíen los mensajes cuando ellas quieren, sino que lo tienen que hacer al comienzo de unos intervalos de tamaño fijo llamados *slots*. Está demostrado que esta variación mejora la eficiencia del protocolo ALOHA en un 35%.

El Programa §2.4 muestra una implementación IP_{CORE} de este protocolo, y la Figure §2.7 su vista estática cuando $N = 3$. Fíjese en que utilizamos dos interacciones $(N + 1)$-partitas: una para el enlace ascendente, que permite que las estaciones envíen mensajes al satélite, y otra para el enlace descendente, que permite que el satélite propague los mensajes a todas las estaciones. En aras de la simplicidad, suponemos que existe un tipo abstracto de datos llamado *message* con el propósito obvio. También suponemos que existen dos

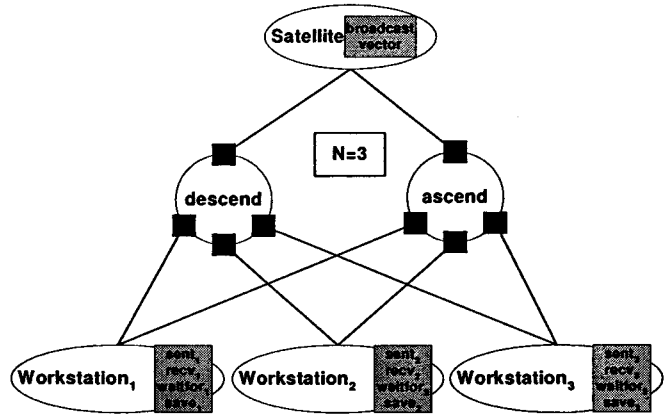


Figura 2.7: *Vista estática del protocolo* ALOHA.

mensajes especiales llamados *EMPTY* y *COLLISION*. El primero denota que una estación no necesita transmitir mensajes, el segundo se usa para denotar que una colisión ha sido detectada.

El satélite tiene una variable llamada *vector* en la que se almacenan los mensajes recibidos de las estaciones de trabajo. Su ciclo de vida es muy simple ya que primero participa en una interacción *ascend* para coordinarse con todas las estaciones de trabajo t conocer qué mensajes quieren transmitir. Cada estación almacena sus mensajes en una variable llamada $sent_i$ de forma que cuando el satélite ejecuta Intascendvector(1) := $sent_1$; ...; vector(N) := $sent_N$, se almacena en *vector* los mensajes que las estaciones quieren transmitir. Así, una colisión existirá cuando una o más celdas de este vector esté no vacía, y el satélite utiliza la función *collision* para detectarlo.

En cuanto a las estaciones, su ciclo de vida es algo más complicado. Primero participan en una interacción *ascend* para hacerle saber al satélite qué mensaje quieren transmitir. Después participan en una interacción *descend* y reciben el mensaje que el satélite ha propagado. Si el mensaje coincide con el mensaje enviado por ellas, esto significa que no ha habido colisión y que el mensaje ha llegado correctamente a todas las estaciones. En caso contrario, si el mensaje recibido es *COLLISION* y el mensaje que la estación envió previeamente no estaba vacío entonces se tiene que otras estaciones han colisionado. En este caso, las estaciones implicadas esperaran un número aleatorio de *slots* antes de volver a enviar. Fíjese que tiene que borrar $sent_i$ o de lo contrario el satélite pensaría que están intentando transmitir de nuevo. Para que los mensajes no se pierdan se utiliza la variable auxiliar$save_i$.

Capítulo 3

Selección justa

Fairness does not consist so much of everybody's doing the same thing, but of everybody's being willing to do something that others don't want to do.

Judith Martin, 1938–
Washington Post columnist

En este capítulo, nos centramos en la selección justa y argumentamos el por qué de su importancia en el contexto de los sistemas basados en interacciones multipartitas. Está organizado de la siguiente forma: En la Sección §3.1, introducimos el concepto; en la Sección §3.2, definimos las propiedades safety y liveness y las ilustramos con algunos ejemplos; en la Sección §3.3, se muestran algunas taxonomías de selección justa; en la Sección §3.4, presentamos algunos marcos de trabajo para definir nociones de selección justa; algunos aspectos controvertidos sobre el tema son presentados en la Sección §3.5.

```
x := 0; goon := true;
*[ goon → x := x + 1
[] goon → goon := false
].
```
(a)

```
SYSTEM :: [Provider || Client_A || Client_B], where
Provider :: *[ A⟨⟩ → skip [] B⟨⟩ → skip ];
Client_A :: *[ A⟨⟩ → skip ];
Client_B :: B⟨⟩.
```
(b)

Programa 3.1: *Selección justa para garantizar la terminación y la respuesta a una solicitud de servicio con el tiempo.*

3.1 Introducción

Las nociones de selección justa que normalmente se utilizan fueron introducidas en el contexto del lenguaje de Dijkstra [44] y de las redes de Petri [31]. Desde entonces, es usual hacer referencia a la selección justa como se fuese un concepto único. Sin embargo, existe una rica gama de nociones que no son equivalente entre ellas aunque comparten una constante: todas intentan restringir el conjunto de ejecuciones potenciales de un sistema no permitiendo aquellas en las que un proceso es rechazado siempre que tiene una oportunidad de ejecutarse [97]. Esto le permite a los programadores abstraerse de cualquier detalle de bajo nivel ya que pueden suponer que sus procesos progresarán de forma independiente a la velocidad relativa de los procesadores, la latencia de la red o la política de planificación.

Consideremos, por ejemplo, el Programa §3.1.(a). Para determinar si éste termina o no necesitamos saber cómo el compilador implementa las instrucciones alternativas múltiples. En teoría, cuando varias alternativas pueden ser ejecutadas el criterio de selección debe ser no determinista, pero el cómo se implementa este no determinismo depende del compilador y del soporte que proporcione el sistema en tiempo de ejecución. Si sabemos que la implementación asegura que la probabilidad de seleccionar cada alternativa es 0.5 el programa debería terminar en un tiempo finito aunque desconocido; de los contrario, si el criterio es seleccionar siempre la primera alternativa, entonces el programa nunca terminará. El Programa §3.1.(b) es parecido. En este caso, existen un proceso proveedor que puede servir a $Client_A$ o $Client_B$ participando en las interacciones A o B, respectivamente. Por desgracia, a menos que sepamos cçomo el programa está implementado, no podremos demostrar que $Client_B$ tenga garantías de ser servido.

Estos hechos cambian drásticamente cuando sabemos cómo el compilador implementa la noción de selección justa. En este caso, el Programa §3.1.(a) debe terminar y $Client_B$ debe ser servido en un tiempo finito aunque desconocido ya que ninguna alternativa es rechazada por siempre. En este caso, la selecci-

Propiedad	"Algo malo"
Exclusión mutua	Dos procesos no entran en su sección crítica al mismo tiempo.
Ausencia de interbloqueos	En un conjunto de procesos que los que unos se tienen que esperar a los otros para acceder a un recurso.
Corrección parcial	El estado inicial satisface una precondición, pero el estado final no satisface la post-condición correspondiente.
Primero en llegar primero en ser servido	Dos solicitudes de servicio no son atendidas en el orden en el que se producen.

Tabla 3.1: *Ejemplos de propiedades safety.*

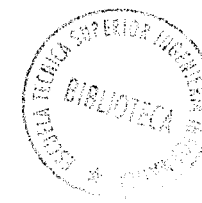
Propiedad	"Algo bueno"
Progreso	Todo proceso progresa y ejecuta cálculo local con el tiempo.
Terminación	La última instrucción de un proceso siempre es ejecutada con el tiempo, es decir, llega a un punto en lo que puede ni ejecutar cálculo local ni interactuar.
Ausencia de inanición	Toda solicitud de servicio es atendida con el tiempo.

Tabla 3.2: *Ejemplos de propiedades liveness.*

ón justa nos permite demostrar dos propiedades *liveness* que no hubiéramos podido demostrar de otra forma. Además, la selección justa también nos puede ayudar a demostrar algunas propiedades *safety* ya que es un resultado ampliamente conocido que la selección justa es esencial en sistemas en los que sus participantes necesitan obtener la exclusión mutua sobre un recurso, lo cual es una propiedad *safety* [74].

3.2 Propiedades *safety* y *liveness*

En la Referencia [77], Lamport algunas de las propiedades que merecen una especial atención en el contexto de los programas concurrentes. Las clasificó en dos grupos: propiedades *safety*, con las que se garantiza que "algo malo" nunca occurre, y propiedades *liveness*, con las que se garantiza que "algo bueno" termina ocurriendo con el tiempo. Las Tablas §3.1, §3.2 y §3.3 muestran algunas propiedades *safety* y *liveness*.



Safety	*Liveness*
Toda instrucción se ejecuta en menos de 10 milisegundos.	Toda instrucción se ejecuta en un tiempo finito.
El programa terminará en 1.7E24 siglos.	El programa terminará con el tiempo.
Ninguna interacción es rechazada más de 5 veces consecutivas.	Ninguna interacción es rechazada por siempre.

Tabla 3.3: *Ejemplo de propiedades safety con sus homólogas liveness.*

La distinción entre ambas es muy importante, ya que demostrar que un programa satisface una propiedad *safety* concreta equivale a demostrar el cumplimiento de una invariante; por lo contrario, demostrar que un programa cumple una propiedad *liveness* es equivalente a encontrar un orden bien formado en el que una función de *ranking* se decrementa o permanece constate en cada paso computacional. En resumen, las propiedades *safety* se puede ver como características individuales de los estados, mientras que las propiedades *liveness* sólo pueden ser vistas como propiedades de las ejecuciones.

Después de que fuesen introducidas informalmente por Lamport, han sido muchos los autores que se han esforzado en caracterizarlas formalmente: Alford *et al.* caracterizan la ejecución de un sistema concurrente como secuencia infinita de estados, y una propiedad como un subconjunto de estas secuencias [2]; Alpern y Schneider las caracterizan a través de propiedades estructurales de los autómatas de Büchi [3]; Sistla se centro en la estructura de las formulas temporales, lo que después se encontró equivalente con la propuesta anterior ya que existen procesos automáticos para traducir fórmulas de lógica temporal en los equivalentes autómatas de Büchi [32]; Lichtenstein *et al.* presentaron otra propuesta de caracterización basada en formulas temporales que no tenían en cuenta operadores para el pasado.

Nosotros pensamos que la propuesta de Alpern y Schneider es la más adecuada en el contexto de los sistema concurrentes, ya que está probado que la Referencia[129] que existen propiedades que pueden ser expresadas a través de autómatas de Büchi pero que no pueden ser expresadas con lógica temporal. La definición literal que dan es la siguiente [3]:

> *Let S be the set of program states, S^∞ the set of infinite sequences of programs states, and S^* the set of finite sequences of programs states. The elements of S^∞ are called executions, and the elements of S^* are called partial executions. A property is a subset of S^∞. By $\lambda \models P$, we denote that execution λ satisfies P, i.e., it belongs to the subset of*

executions defined by this property. Property P *holds for a program if the whole set of executions defined by the program is contained within the property. Finally, let* λ_i *denote the partial execution consisting of the first* i *states in* λ. P *is a safety property iff*

$$\forall \lambda \in S^{\infty} \cdot (\lambda \not\models P \Rightarrow \exists i > 0 \cdot \forall \beta \in S^{\infty} \cdot \lambda_i \beta \not\models P).$$

On the contrary, it is a liveness property iff

$$\forall \alpha \in S^{*} \cdot \exists \beta \in S^{\infty} \cdot \alpha\beta \models P.$$

Fíjese que la definición de propiedad *safety* incondicionalmente prohíbe que "algo malo" ocurra, ya que, de otra forma, existiría un instante en el que ese "algo" podría ser detectado. Por lo contrario, la definición de propiedad *liveness* indica que ninguna ejecución parcial es irremediable ya que siempre es posible a partir de ella extenderla en una ejecución infinita que preserva la propiedad *liveness* en consideración. Es importante destacar que, de acuerdo a esta definición, una propiedad *liveness* estipula que "algo bueno" debe ocurrir con el tiempo, pero no el instante de tiempo en que esto ocurre. Este es un fenómeno conocido al cual se suele hacer referencia como indeterminismo no acotado [97].

Aunque esta caracterización de *liveness* de Alpern y Schneider es ampliamente aceptada, existen otras definiciones complementarias que son más restrictivas, por ejemplo, *uniform liveness* y *absolut liveness* [4]. Una propiedad es *uniform liveness* si y sólo si existe un prefijo que puede ser usado para extenderse a cualquier ejecución parcial de forma que el resultado preserva la propiedad considerada; por contra, una propiedad es *absolute liveness* si y sólo si no está vacía y cualquier ejecución que la satisface puede ser concatenada con cualquier ejecución parcial de forma que el resultado sigue satisfaciendo la propiedad. Por desgracia, *absolute fairness* no incluye propiedades como la respuesta a una solicitud de servicio con el tiempo, y *uniform liveness* no captura la idea intuitiva de *liveness*.

3.3 Taxonomías de nociones de selección justa

Como mencionamos en la introducción, la constante que existe detrás de toda noción de selección justa es que ningún elemento es rechazado siempre que tiene la oportunidad de ejecutarse [97]. En otras palabras, todo elemento del sistema que pueda ejecutarse suficientemente a menudo realmente se

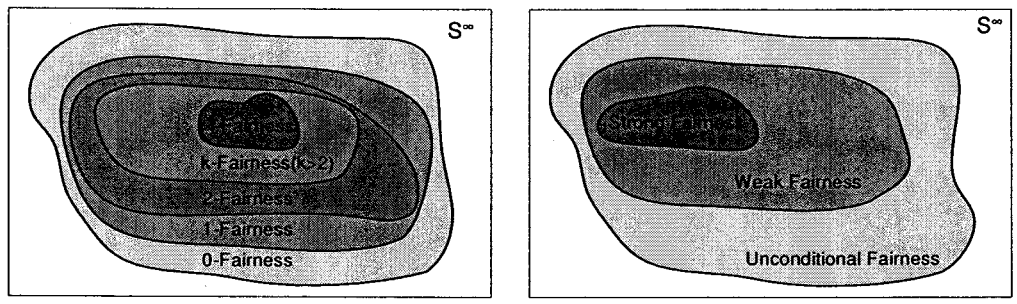


Figura 3.1: *Relación entre distintos niveles de selección justa.*

ejecute suficientemente a menudo. Diferentes nociones de justicia son posibles dependiendo del significado que le demos a los términos "elemento" y "suficientemente a menudo".

En el contexto de las interacciones multipartita, los elementos más comunes son las propias interacciones, así, es habitual que se estudie la selección justa de interacciones. Existen algunos autores que también consideran la selección justa de procesos. En nuestra tesis nos centramos en la selección justa de interacciones ya que una ejecución que sea justa con respecto a las interacciones también lo es con respecto a los procesos. La razón de esto es que podemos suponer que los procesos sólo se sincronizan e intercambian información a través de interacciones, lo que implica que cada vez que una interacción es seleccionada de forma justa, los procesos que la están ofreciendo lo están haciendo también de forma justa. Obviamente, esta definición de selección justa de interacciones supone que la propiedad de progreso mínimo está implementada de forma que todo proceso que puede ejecutar una instrucción sobre su estado local lo termina haciendo en un tiempo finito.

Además, cada una de estas nociones tiene un nivel de severidad dependiendo de la definición del término "suficientemente a menudo". En las siguientes subsecciones, presentaremos varias taxonomías para dar al lector una idea de la gama tan amplia que niveles de selección justa existentes. Merece la pena destacar que ninguno de estos niveles tiene en cuenta las ejecuciones finitas, ya que las consideran justas por definición [19]. Las relaciones conocidas entre los distintos niveles de es esquematizada en la Figura §3.1. Por desgracia, existen algunos niveles cuya relación con los restantes no es conocida, pero los detalles de los motivos de ésto se salen del alcance de esta tesis.

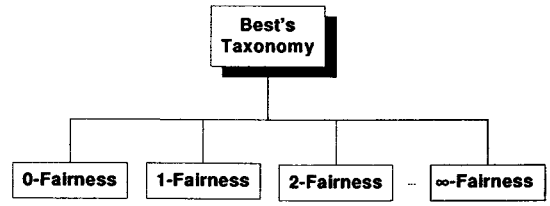


Figura 3.2: *Taxonomía de Best.*

3.3.1 Taxonomía de Best

Best consideró la eventualidad como la propiedad básica para distinguir entre los distintos niveles [18, 17, 19], y diseñó una taxonomía construida sobre el trabajo previo de Chrząstowski-Wachtel en el contexto de las redes de Petri.

Para entender esta taxonomía, ilustrada en la Figura §3.2, es necesario introducir el concepto de k-activación: se dice que una ejecución parcial k-activa una interacción si y sólo si ésta puede ser extendida de forma que sea posible que dicha interacción se habilite después de $k+1$ ejecuciones de otras interacciones; en general, se dice que una ejecución parcial ∞-activa una interacción si y sólo si existe algún $k \in \mathbb{N}$ que k-activa esta interacción. Este concepto le permite definir el nivel k-*fairness* para cualquier $k \in \mathbb{N} \cup \{\infty\}$ de la siguiente forma: una ejecución es k-*fair* si y sçolo so todo interacción que se k-activa infinitamente a menudo se ejecuta infinitamente a menudo.

Fíjese que la 0-activación de una interacción implica que esta debe estar habilitada inmediatamente (en el momento presente), mientras que k-activación para $k > 0$ implica que es posible que la interacción debe habilitarse después de que $k+1$ interacciones sean habilitadas. Esto implica que k-*fairness* no depende las ejecuciones por sí mismas, sino de las posibles extensiones de éstas. En el caso de ∞-*fairness*, este matiz ha motivado a algunos autores a afirmar la noción como sigue [10, 73, 78]: una ejecución es ∞-*fair* si y sólo si toda interacción que puede habilitarse infinitamente a menudo, se habilita infinitamente a menudo, y, es ejecutada infinitamente a menudo. Fíjese en complejidad del matiz en esta frase, ya que la noción no tiene en cuenta la ejecución objeto de estudio, sino la posibilidad para una interacción de habilitarse si dicha ejecución tomase otro camino.

Como conclusión, no existe un planificador genérico razonable para implementar k-*fairness* si $k > 0$, ya que éste debería de ser capaz de predecir si una interacción puede habilitarse en el futuro. Fíjese, sin embargo, que algunas implementaciones son posibles en algunos casos muy restrictivos que son explicados en la Sección §4.3 y en las Referencias [10, 73, 78, 114].

También merece la pena destacar que k-*fairness* se conoce con el nombre de propiedad de *compassion* [83] o *strong fairness* [54], y es uno de los niveles más prácticos. ∞-*fairness* ha llamado de la atención de muchos investigadores recientemente, y muy a menudo se le referencia como *hyperfairness* [10, 78]. En lo que queda de memoria, nosotros usaremos el término de *strong fairness* y *hyperfairness* ya que son más usuales que los que utiliza Best en su trabajo original.

3.3.2 Taxonomía de Francez

La taxonomía de Francez [54] se organiza en dos categorías estructurales principales en función del subconjunto de interacción que son objeto de selección justa: si ésta es aplicada a todas las interacciones en su conjunto, entonces se conoce como *all-levels fair*; por el contrario, si sólo consideramos los programas en forma normal y el criterio de selección sólo se aplica a las interacciones del primer nivel, entonces se conoce como *top-level fair* (véase Section §B.4).

Independientemente de la estructura, Francez también distingue entre los niveles cuantitativos y los basados en eventualidades, que son esquematizados en la Figura §3.3. La idea detrás de los primeros es relajar la eventualidad y considerar sólo el orden relativo en que las interacciones son ejecutadas; por el contrario, en el segundo, no se permiten las ejecuciones en las que una interacción que está habilitada en infinitas ocasiones sólo se ejecute un número finito de veces.

Aunque los niveles basados en el orden parecen atractivos, éstos no son habituales en la práctica. Estos niveles incluyen: la noción *order-based fairness*, que garantiza que una interacción se ejecuta tan pronto se habilita y obtiene la exclusión mutua sobre el conjunto de interacciones con las que comparte algún participante [108]; la noción *absolute fairness*, que el orden en el que las interacciones son ejecutadas permutan infinitamente a menudo; y la noción *bounded-delay fairness*, que consiste en fijar una cota (a ser posible pequeña) en los intervalos entre las ejecuciones consecutivas de la misma interacción.

Por contra, las nociones basadas en la eventualidad son más comunes en la práctica, los cuales, han atraído la atención de muchos investigadores. Se incluyen: la noción *unconditional fairness*, que garantiza que una que está permanentemente habilitada es ejecuta infinitamente a menudo; la noción *weak fairness*, que es más relajada y requiere que las que se ejecuten infinitamente a menudo estén habilitadas desde un momento de su ejecución en adelante; la noción *strong fairness*, que es la más restrictiva ya que requiere que las

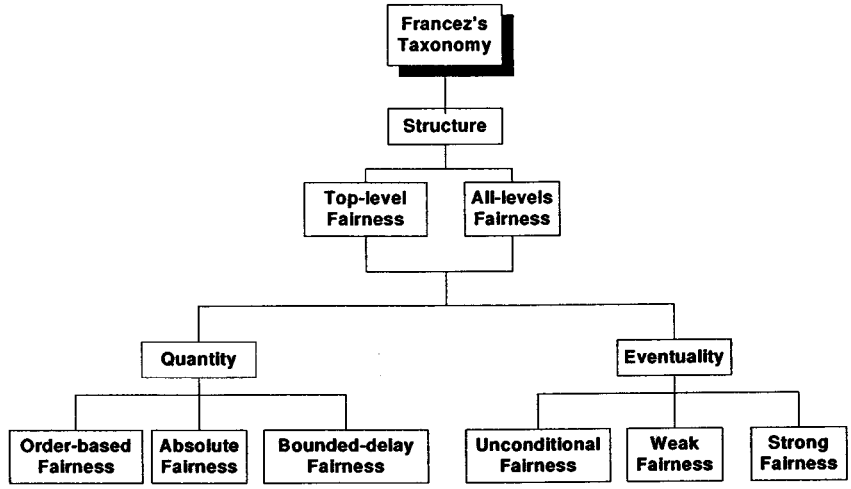


Figura 3.3: *Taxonomía de Francez.*

interacciones que se ejecutan infinitamente a menudo estén habilitadas infinitamente a menudo, no necesariamente de forma continuada. A pesar de que las nociones basadas en la eventualidad son las más utilizadas en la práctica, sus nombres no son universales. Por ejemplo, Lehmann *et al.* hacen referencia a *unconditional* y *weak* como propiedades de *justice* y *minimal progress* respectivamente, y Best se refiere a *strong fairness* como *0-fairness*.

Fíjese que la aplicación de estos niveles a las interacciones multipartitas depende completamente de la semántica que consideremos. Como se puede tener varias semánticas equivalentes para el mismo modelo, tenemos que existe una estrecha relación éstas y las aplicabilidad de las nociones. Por ejemplo, en nuestro marco de trabajo (véase Capítulor §6), los niveles *unconditional* y *weak* no son aplicables ya que ninguna interacción puede estar permanentemente habilitada. La razón es que una vez que una interacción se ejecuta, ésta se deshabilita y sus participantes necesitan ofrecer participar en ella de nuevo; esto implica que algún tiempo pase desde el momento en el que la interacción se ejecutada hasta que se habilita de nuevo, y éste no es ignorado por nuestro marco de trabajo. Sin embargo, si consideramos la semántica original de IP [53], una interacción puede estar habilitada permanentemente ya que es considerado un evento distinto el hecho de que un proceso ofrezca una interacción. Así aunque algún tiempo tenga que pasar entre dos habilitaciones consecutivas de la misma interacción, éste no es tenido en cuenta.

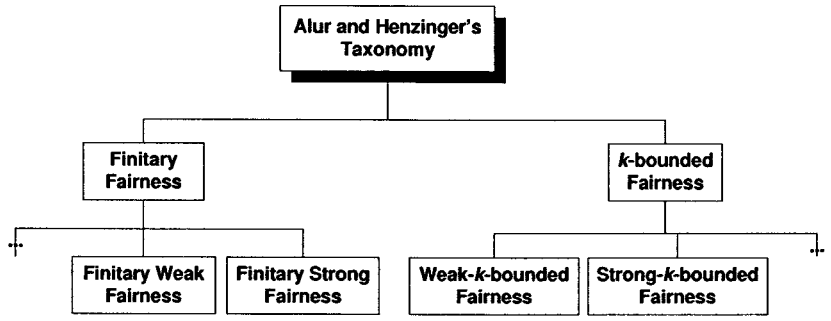


Figura 3.4: *Taxonomía de Alur y Henzinger.*

3.3.3 Taxonomía de Alur y Henzinger

El principal problema de la nociones basadas en eventualidades de Francez es que no ponen una cota del número de veces que una interacción puede ser rechazada. Las relaciones entre estas nociones y el indeterminismo no acotado es bien conocido en la bibliografía [97]. Esto motivó a Alur y Henzinger (véase Figura §3.4) para introducir la noción *finitary fairness* en la que el no determinismo inherente a programa está acotado por un número natural [5].

Por ejemplo, consideremos el siguiente programa:

```
*[ true & a1⟨⟩ → x := x + 1
[] true & a2⟨⟩ → x := x - 1
].
```

Una ejecución en la que las interacciones son seleccionadas de acuerdo al siguiente patrón $[a_1a_2a_1a_1a_2a_1a_1a_1a_2\ldots]$ no es *finitary* a ningún nivel ya que no existe una cota en el número de veces que la interacción a_1 pueda ser ejecutada después de que a_2 sea ejecuta. Por el contrario, si el patrón tiene la siguiente forma $[a_1^n a_2^m]^\infty$ siendo n y m números naturales, se tiene que sí existe una cota en la ejecución y es *finitary*.

Fíjese que la cota debe existir, pero no es establecida de antemano. Si esto es así, entonces a la noción se le conoce como k-bounded fairness, y Alur y Henzinger demuestran que la versión *finitary* de una noción de justicia puede ser vista como la unión infinita de sus correspondientes nociones k-*bounded* siendo $k \in \mathbb{N}$. Recientemente, otros autores han trabajado en nociones *bounded*, que durante años se han considerado como parte del folclore [64, 41].

EJEMPLO §3.3.1 ::
```
*[ true & a1⟨⟩ → x := x + 1
[] true & a2⟨⟩ → x := x - 1
].
```

EJEMPLO §3.3.2 ::
```
goon := true; A := true; B := true;
*[ goon & a1⟨⟩ → A := ¬A
[] goon & a2⟨⟩ → B := ¬B
[] ¬A ∧ ¬B & a3⟨⟩ → goon := false
].
```

EJEMPLO §3.3.3 ::
```
goon := true; x := 1;
*[ goon & a1⟨⟩ → x := x + 2
[] goon & a2⟨⟩ → x := x - 1
[] x ≤ 0 & a3⟨⟩ → goon := false
].
```

EJEMPLO §3.3.4 ::
```
goon := true; x := 0;
*[ goon & a1⟨⟩ → x := 1
[] x = 0 & a2⟨⟩ → goon := false
].
```

Programa 3.2: *Ejemplo de programas para mostrar ejecuciones justas e injustas.*

3.3.4 Ejemplos de ejecuciones justas e injustas

Nuestro objetivo en esta sección es ilustrar algunos de los niveles anteriores a través de los ejemplos de los programas §3.2. En aras de la simplicidad, hemos usado sólo interacciones unipartitas; además, nos centramos en sólo en las interacciones que son ejecutas, es decir, las trazas de interacción, ya que no es necesario entrar en detalles de bajo nivel concernientes a los estados de los procesos o a los eventos que éstos producen.

Ejemplo 3.3.1 *Una ejecución que genera la traza de interacción* $[a_1]^\infty$ *no es* unconditional *según la semántica original de* IP. *La razón es que la interacción* a_2 *está siempre habilitada pero nunca resulta seleccionada. En consecuencia, esta ejecución tampoco es* weak, strong, k-fair *para ningún* $k \in \mathbb{N}$ *o* ∞-fair. *Concretamente, existen infinitos prefijos que* 0-, k- *o* ∞-*activan* a_2, *pero ésta no resulta ejecutada en infinitas ocasiones. Teniendo en cuenta la semántica de muestro marco de trabajo, los niveles* unconditional *y* weak *no son aplicables.*

Por el contrario, una ejecución que genera la traza de interacción $[a_1a_2]^\infty$ *cumple los requisitos de todos los niveles basados en eventualidades ya que ambas interacciones se* 0-, k- *y* ∞-*activan y son ejecutadas en infinitas ocasiones. Obviamente,*

también es finitary a cualquier nivel ya que el número de veces que las interacciones son rechazadas no excede uno.

Ejemplo 3.3.2 *Cualquier ejecución que genere una traza de la forma* $[a_1a_1a_2a_2]^\infty$ *es* 0*-fair y* 1*-fair pero no* k*-fair para cualquier* $k \geq 2$*, y, en consecuencia, tampoco es* ∞*-fair. Fíjese que la interacción* a_3 *nunca se habilita, y por eso no puede ejecutarse. Sin embargo,* a_3 *puede habilitarse después de la ejecución de las alternativas* a_1 *y* a_2 *ya que existen infinitos prefijos de esta ejecución que puede ser extendidos por la ejecución de dos interacciones que conducen a una secuencia en la que* a_3 *puede ejecutarse.*

Este es un claro ejemplo del interés teórico de estas nociones, ya que es imposible implementar un planificador capaz de predecir qué el guarda de la interacción a_3 *pueda habilitarse si las interacciones* a_1 *y* a_2 *son ejecutadas en este o en otro orden.*

Ejemplo 3.3.3 *Cualquier ejecución cuya traza de interacción sea* $[a_1a_2]^\infty$ *es* k*-fair para cualquier* $k \geq 0$*, pero no* ∞*-fair. Fíjese que la interacción* a_3 *nunca es ejecutada, pero todo prefijo finito de esta ejecución puede ser extendido tal que active* a_3 *ejecutando la interacción* a_2 *las suficientes veces; sin embargo, esta interacción no puede ser ejecutada en infinitas ocasiones.*

Ejemplo 3.3.4 *Cualquier ejecución cuya traza de interacción sea* $[a_1]^\infty$ *es* k*-fair para todo* $k \geq 0$ *así como* ∞*-fair. Ahora, el guarda de la interacción* a_2 *no puede habilitarse aunque se ejecute la interacción* a_1*.*

3.4 Marcos de trabajo para definir nociones

Aunque varios son los marcos de trabajos que aparecen en la literatura para definir nociones de selección justa, nosotros pensamos que las propuestas de Olderog y Apt [98] y Lamport [78] son las más maduras. En las siguientes subsecciones, haremos una breve introducción.

3.4.1 Marco de trabajo de Olderog y Apt

Olderog y Apt fueron los primeros en introducir un marco de trabajo para definir nociones de selección justa en el contexto de los lenguajes de programación paralelos con variables compartidas [98]. En aras de simplificar el estudio, definen la selección justa en términos de elementos (interacciones en

nuestro contexto) habilitados y seleccionados. Así, una ejecución es una secuencia de pares de la forma (E, i), donde E es un conjunto no vacío de índices que denotan las interacciones que están habilitadas y $i \in E$ denota el índice de la interacción seleccionada para ser ejecutada. Con este simple modelo, ellos definen dos predicados que capturan los niveles *weak* y *strong*:

$$WF(\lambda) \iff \forall i \in \{1,\ldots,n\} \cdot (\forall^{\infty} j \in \mathbb{N} \cdot i \in E_j) \Rightarrow (\exists^{\infty} j \in \mathbb{N} \cdot i = i_j)$$
$$SF(\lambda) \iff \forall i \in \{1,\ldots,n\} \cdot (\exists^{\infty} j \in \mathbb{N} \cdot i \in E_j) \Rightarrow (\exists^{\infty} j \in \mathbb{N} \cdot i = i_j)$$

donde λ denota la ejecución genérica $[(E_0, i_0)(E_1, i_1)\ldots(E_j, i_j)\ldots]$.

En cuanto a la implementación, presenta un método transformacional construido en base a la propuesta previa de Dijkstra [42]. Éste nos permite transformar los programas en sus equivalentes versiones *weak* o *strong* que pueden ser problemáticos en la medida en que la mayoría de los sistemas actuales son desarrollados como componentes binarios cuyo código fuente normalmente no está disponible [122]. La transformación asigna un contador natural a cada interacción y los inicializa a un valir aleatorio positivo. El criterio es seleccionar la interacción cuyo contador es mínimo, asignarle un nuevo número aleatorio positivo y decrementar el contador de las restantes interacciones en una unidad.

Fíjese que la implementación de este marco de trabajo requiere que el estado de todas las interacciones en sea conocido antes de tomar una decisión, lo que tiene un impacto muy negativo en la eficiencia.

3.4.2 Marco de trabajo de Lamport

En la Referencia [78], Lamport se inspiró en Francez y Gabbay *et al.* [55, 56] para usar lógica temporal de acciones para definir nociones de selección justa. Esta propuesta es más adecuada y potente que la que de Olderog Apt's, aunque se aleja bastante de una posible implementación [52].

Existen dos versiones dependiendo de si hablamos de lógica lineal o de salto. La primera tiene conectivas especiales que ocultan la cuantificación explícita del tiempo y la hacen más intuitiva y compacta muchas propiedades de los programas; por el contrario, la segunda, se construye sobre árboles en lugar de secuencias, y sus ventajas no están lo suficientemente claras en la comunidad investigadora.

Por lo tanto, la versión lineal es la más común. Sus conectivas de tiempo son denotadas como $\Box$ y $\Diamond$. Si F es una fórmula lógica, $\Box F$ significa que

Noción	Ejemplo de ejecuciones permitidas
Strong	$(a_1 \| a_2)^\infty$
Finitary strong	$\bigcup_{i \in \mathbb{N}} (a_1^{\{0,i\}} a_2 \| a_1 a_2^{\{0,i\}})^\infty$
k-Bounded strong	$(a_1^{\{0,k\}} a_2 \| a_1 a_2^{\{0,k\}})^\infty$

Tabla 3.4: *Diferencias entre nociones clásicas, finitary y bounded.*

F se cumple siempre, y $\Diamond$F significa que debe cumplirse con el tiempo. Así, se dice que una ejecución es de nivel *weak* si y sólo si $\Diamond\Box$enabled(x) $\Rightarrow$ $\Box\Diamond$selected(x) se cumple para toda interacción x. Análogamente, se dice que una ejecución se de nivel *strong* si y sólo si $\Box\Diamond$enabled(x) $\Rightarrow$ $\Box\Diamond$selected(x) se cumple para toda interacción x.

No existen implementaciones genéricas de este marco de trabajo ya que las conectivas temporales sobre las que se construye no son implementables. Éstas se basan en el infinito y en eventualidades, conceptos que son muy adecuados para expresar propiedades *liveness* y *safety* complejas, pero imposibles de implementar en máquinas finitas.

3.5 Aspectos controvertidos

Aunque la selección justa parece ser muy deseable desde un punto de vista práctico, existen algunas complicaciones que han llevado a algunos investigadores a criticarla. En algunos casos argumentan desechar todas las nociones presentadas. En las siguientes subsecciones abordaremos la mayoría de las críticas.

3.5.1 ¿Es la selección justa una propiedad *safety* o *liveness*?

El objetivo de las nociones *unconditional*, *weak* y k-*fairness* es evitar que los elementos que están preparados para progresar suficientemente a menudo se bloqueen para siempre. Fíjese, sin embargo, que no implica que exista una cota superior en el tiempo que un elementos habilitado puede ser retrasado antes de ser ejecutado, y esta es la razón del por qué las nociones *finitary* y k-*bounded* fueron introducidas.

Existen, sin embargo, importantes diferencias entre las nociones k-*bounded* y las restantes ya que las primeras son una propiedad *safety* y las restantes una

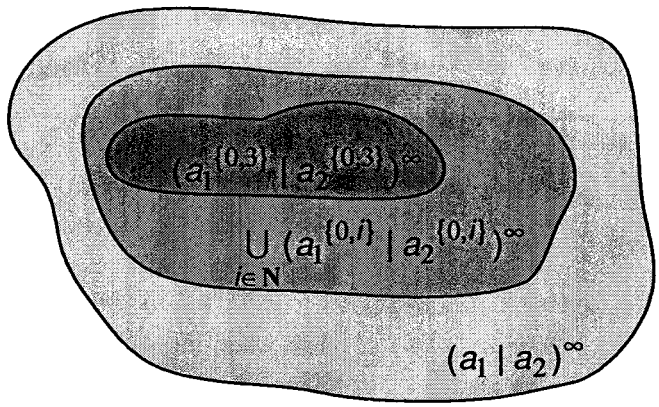


Figura 3.5: *Relaciones entre las ejecuciones strong, finitary y bounded.*

propiedad *liveness*. La razón es que la noción k-*bounded* no es una característica de todas las ejecuciones, sino de algunas en concreto, ya que es posible contar el número de veces que una interacción ha sido seleccionada en detrimento de las restantes. Por lo tanto, el cumplimiento de esta propiedad puede ser comprobado en cada paso de la ejecución. Si existe una cota del tiempo que los elementos de un sistema están ejecutando cálculo local, la noción k-*bounded* puede ser vista como una materialización del nivel *bounded-delay* en la taxonomía de Francez.

La Tabla §3.4 ilustra las diferencias entre las nociones clásicas, las *finitary* y las *bounded*, y la Figura §3.5 muestra sus relaciones gráficamente. Fíjese que el primero y segundo ejemplo son propiedades *liveness* ya que garantizan que algo "bueno" debe terminar ocurriendo ; por el contrario, el tercer ejemplo es una propiedad *safety* ya que garantiza que cuando una interacción habilitada es rechazada en más de $k + 1$ ocasiones algo "malo" ocurre.

En resumen, ya que la selección justa no es un concepto universal, sino un conjunto de abstracciones, puede ser visto como una propiedad *safety* o *liveness* dependiendo del nivel considerado.

3.5.2 ¿Podemos ignorar la selección justa?

Ya que la violación de una propiedad *safety* puede ser detectada en un tiempo finito [3], esta cuestión cobra sentido en el contexto de las nociones de selección justa ya que éstas sólo son propiedades *liveness*. En general, como se basan en el futuro y describen características de todas las ejecuciones de un

b := true; *[b → print(0) [] b → print(1); b := false].	(a) Programa original.
print(1).	(b) Implementación 1.
*[true → print(0)].	(c) Implementación 2.

Programa 3.3: *Ejemplo de Dijkstra para mostrar que la selección justa es una propiedad void.*

programa y de ejecuciones concretas, éstas no puede ser testadas en un tiempo finito. La única excepción son las propiedades llamadas *finitary liveness* [76], aunque ninguna noción de selección justa encaja en esta categoría.

Teniendo en cuenta el problema anterior, Dijkstra sostiene que la selección justa puede ser ignorada con "impunidad", ya que implementarla es una obligación *void* [45]. Por ejemplo, si suponemos que la instrucción alternativa múltiple del Programa §3.3.(a) is implementada con cualquier noción basada en eventualidades, entonces podemos concluir que el programa escribe un número aleatorio de ceros y para tras escribir un único cero. Sin embargo, los Programas §3.3.(b) y §3.3.(c) son implementaciones correctas ninguno de ellos tiene el comportamiento esperado, ya que el primero nunca escribe un cero, y el segundo nunca para. Por *implementación correcta*, Dijkstra entiende que se puede implementar un experimento finito para demostrar su corrección (véase Programa §3.3.(b)) o, en otro caso, no se puede implementar una experimento finito para demostrar su incorrección (véase Programa §3.3.(c)).

Traducimos el programa anterior a varios lenguajes que soportan instrucciones alternativas múltiples y los resultados fueron siempre los mismos: parecía iterar para siempre y no se escribía ningún cero. Obviamente nosotros no pensamos que éste sea el comportamiento adecuado, aunque podría tratarse de un error, que causara que el programador que implementó el compilador pasara una noche en blanco ya que el indeterminismo es una obligación *void*, es decir, una obligación que nadie puede comprobar que no ha sido realizada a través de experimentos finito.

Por lo tanto, de acuerdo con Dijkstra, no hay necesidad de implementar la selección justa ya que nadie es capaz de comprobar que ésta implementada, y la distinción entre ejecuciones justas e injustas es inútil.

Este argumento fue refutado por Schneider y Lamport con el ejemplo del Programa §3.4 [120]. Según el razonamiento de Dijkstra, una instrucción de asignación puede ser implementada correctamente con una instrucción de espera indeterminada, ya que no existen experimento finitos que demuestren que dicha asignación no va a ser ejecutada. En general, demostrar la terminación de un programa indeterminista es insoluble [97], aunque algunos argu-

x := 1.	(a) Programa original.
*[true → **skip**].	(b) Implementación 1.

Programa 3.4: *Contraejemplo de Schneider y Lamport.*

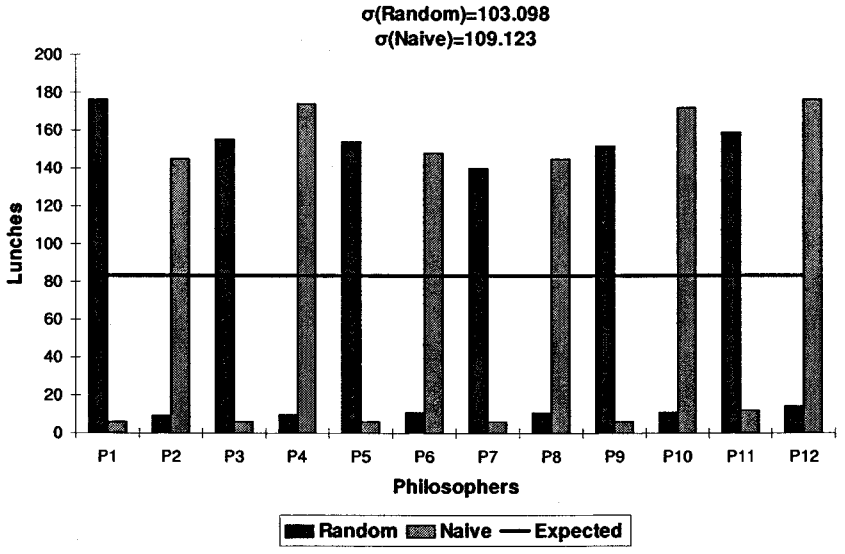
mentan que la distinción entre programas que terminan y programas que no terminan es inútil.

Broy y Nelson [24] profundizan en el problema de la selección justa y el indeterminismo no acotado, y concluyen que éste es inútil sólo si pensamos que los programas justos no pueden ser definidos por un algoritmo, aunque la forma en que definamos el algoritmo para cualquier planificador sea correcta con respecto a la noción de selección justa considerada.

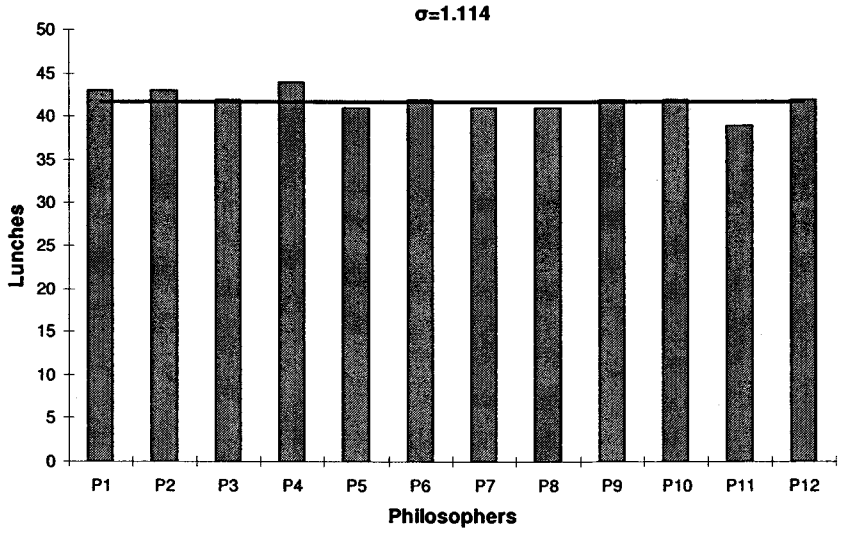
Merece la pena destacar que aunque la selección justa sea una obligación *void*, existe una clara diferencia entre los programas que son controlados por un planificador justo y lo que no. Para ilustrar este aspecto, usamos un banco de pruebas que consiste en ejecutar el Programa §2.1 de los filósofos comensales y estudiamos cómo las comidas son distribuidas entre éstos. Fíjese que si el tiempo que los filósofos están pensando y comiendo es el mismo, entonces todos ellos tendrían las mismas oportunidades de comer, lo que implica que la desviación estándar en el número de comidas debe ser pequeña o nula. En las Figuras §3.6 y §3.7, presentamos los resultados obtenidos de ejecutar este experimentos en el que doce filósofos se reparten 500 comidas. Los detalles de los planificadores utilizados se encuentran en el Capítulo §4 y los detalles del estudio experimental en el Capítulo §9.

En la Figura §3.6.(a) presentamos los resultados obtenidos con un par de planificadores llamados *Random* y *Naive*. Ambos seleccionan las interacciones tan pronto como ellas se habilitan, aunque el primero resuelve los conflictos sorteando un número aleatorio y el segundo selecciona la primera interacción habilitada sin intentar resolver los conflictos de ninguna forma. Como muestra la figura, la desviación estándar obtenida es grande y existen algunos filósofos que no comen ninguna vez en el caso del planificador *Naive*. En las Figuras §3.6.(b), §3.7.(a) y §3.7.(b) presentamos los resultados de ejecutar el mismo experimento con los planificadores *strong* de Francez y Forman [53], Corchuelo [34] y Joung [67]. La diferencia es obvia, en todos la desviación es más pequeña, lo que es deseable en este escenario.

Fíjese, sin embargo, que todas las ejecuciones son *strong*, ya que son finitas. También es importante destacar que este tipo de comportamiento depende completamente del generador de números aleatorios utilizado por los

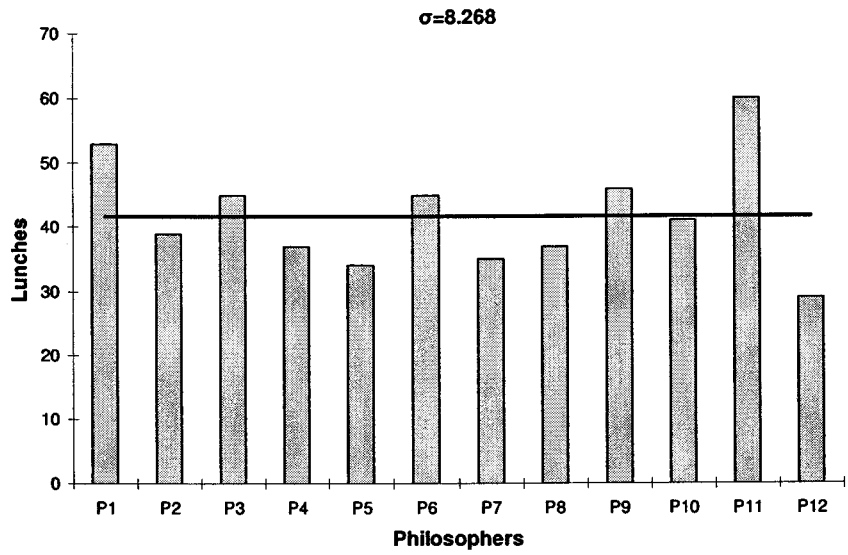


(a) Planificadores injustos.

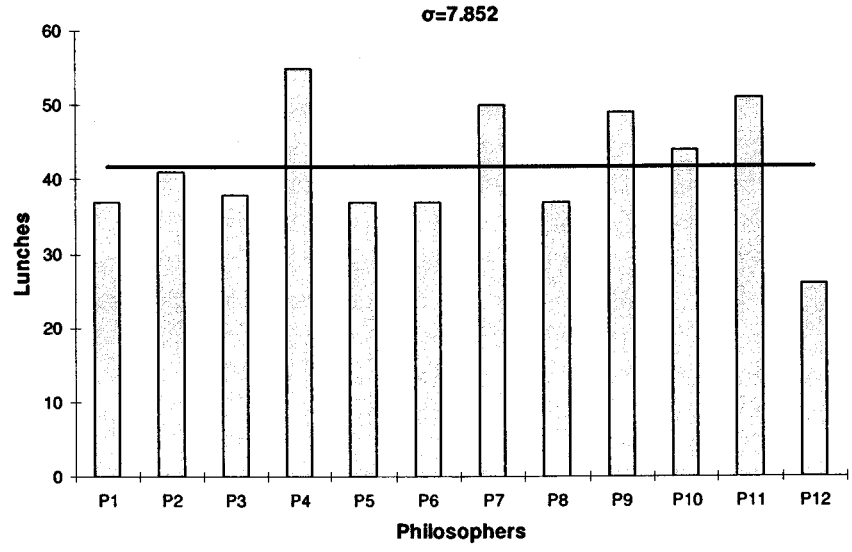


(b) Planificador *strong* de Francez y Forman.

Figura 3.6: *Distribución de comidas con planificadores justos e injustos (I).*



(b) Planificador *strong* de Corchuelo.



(b) Planificador *strong* de Joung.

Figura 3.7: *Distribución de comidas con planificadores justos e injustos (II).*

planificadores. Por ejemplo, si usamos un generador que tienda a producir números en el intervalo [10 000, 50 000] en lugar de [1, 10], tenemos que los planificadores de Francez y Forman o de Corchuelo podrían generar ejecuciones muy parecidas a las obtenidas con *Random* o *Naive*. La diferencia es que los planificadores justos pueden garantizar que ninguna interacción es rechazada por siempre. Ya que en nuestra batería de pruebas es deseable que todos los filósofos tengan las mismas oportunidades de comer, resulta obvio que una cuidadosa sintonización del generador de números de aleatorios puede ayudar a conseguir este objetivo. Esto no es posible en el caso de los planificadores no justos.

3.5.3 ¿Son las implementaciones justas completas?

De acuerdo a la semántica del la instrucción alternativa múltiple presentada en la Referencia [53], ésta es no determinista. Fíjese que estrictamente hablando, el no determinismo no es ninguna propiedad *safety* o *liveness*. Existe sólo la promesa de seleccionar entre las alternativas habilitadas arbitrariamente, así que podemos esperar cualquier salida. En otras palabras, la posible salida del Programa §3.1, por ejemplo, debe ser que a x se la asigne un valor natural aleatorio o el programa itere, y ambas son salidas correctas. Fíjese que implementar el programa justamente implica que la implementación prohíba la segunda salida, lo que deja de ser completo.

Algunos autores han argumentado que la selección justa no debe ser parte de la implementación de los lenguaje, ya que responsabilidad del programador demostrar que sus programas son correctos sin depender de ninguna propiedad de la implementación, pero sólo de la semántica del lenguaje [97]. Sin embargo, esto no es práctico, ya que demostrar que un programa termina, por ejemplo, implica demostrar que éste satisface una propiedad *liveness*, al menos, lo que requiere un conocimiento profundo de cómo el programa está implementado a menos que supongamos que el planificador es justo, de forma que nos podamos abstraer de estos detalles.

Broy y Nelson [24] zanjan la discusión y concluyen que la mejor idea es considerar ambos, el no determinismo y las instrucciones alternativas múltiples justas, por separado. La implementación de una instrucción no determinista debe garantizar que todas las posibles salidas pueden ser generadas, en cambio, la implementación de instrucciones justas debe garantizar que ninguna alternativa es rechazada siempre si ésta está habilitada los suficientemente a menudo.

Ellos diseñan un operador justo para las alternativas múltiples llamado *dovetail*, que se denota como ∇ y puede ser incorporado a IP_{CORE}. Utilizando

este operador, la semántica del siguiente programa es "a x se le asigna un numero positivo aleatorio" y ninguna implementación que itere por siempre sería correcta:

```
x := 0; goon := true;
*[ goon → x := x + 1
∇ goon → goon := false
].
```

El principal problema de considerar operadores como el ∇ es que la selección justa pasa a formar parte del lenguaje. Ya que no existe una definición que prevalezca sobre el resto y que no son equivalentes unas con las otras, esto implica que usar estos operadores es equivalente a congelar el programa para siempre, y que ninguna noción alternativa de selección justa pueda ser aplicada. Esta es la razón del por qué nosotros pensamos que la idea de incorporar la selección justa en la implementación es más atractiva que considerarla parte del lenguaje, ya que es más fácil cambiar el planificador y probar con varias alternativas sin cambiar el programa. Toda ejecución justa debe ser correcta, aunque no toda ejecución correcta sea justa. Esto parece que es un defecto, pero en realidad, es una propiedad deseable.

Capítulo 4

Implementando selección justa de interacciones

It is important to build a kingdom through ruling in fairness.

King Solomon, 970–928 BC
Ancient israelian sovereign

El objetivo de este capítulo es presentar una idea de los algoritmos más modernos para planificar la selección justa de interacciones multipartitas. (La mayoría de ellos son comparados desde el punto de vista empírico en el Capítulo §9.) Empezaremos con una descripción de tres propuestas para implementar el nivel strong *y un discusión sobre algunos resultados de imposibilidad en la Sección §4.2; describimos la única implementación del nivel* hyperfair *que conocemos que en la Sección §4.3; en la Sección §4.4, presentaremos un algoritmo que fue desarrollado en el contexto de la planificación de* carpool *pero que puede ser facilmente adaptable a las interacciones multipartitas para implementar el nivel* bounded-delay*; por último, hablaremos de algunos aspectos de la implementación de los niveles* finitary *y* k-bounded*, los cuales son un tema de candente actualidad.*

4.1 Introducción

En la bibliografía, existen varias técnicas centralizadas y distribuidas para implementar las interacciones multipartitas que nos proporciona IP_{CORE}. El problema normalmente se descompone en tres tareas: implementar sincronización, la obtención de la exclusión mutua y la comunicación entre los participantes.

Las primeras propuestas fueron introducidas en el contexto de CSP [62], pero estaban restringidas sólo a interacciones bipartitas. Después, el problema fue cobrando interés y Chandy y Misra [29] desarrollaron dos algoritmos que son la base del algoritmo MEM de Bagrodia[12], que es uno de los más populares ya que permite ser configurado para obtener un rendimiento óptimo dependiendo del sistema. Desde que la primera solución fue desarrollada, han habido muchas propuestas: para implementar interacciones multipartitas, es habitual utilizar encuestas [34], mensajes contadores [12] o recursos auxiliares como pueden ser los testigos [29]; el problema de la exclusión mutua se puede resolver utilizando prioridades [75], estampas de tiempo[69], recursos auxiliares [12, 75], técnicas probabilisticas [71, 67] o colas [86]; sin embargo, muy poco trabajo hay seido desarrollado con respecto a cómo implementar la comunicación multipartita [130].

La mayoría de las técnicas entes mencionadas tienen en común el inconveniente de que el único nivel de selección justa que tienen en cuenta es el *weak*, lo cual suele ser resultado de las primitivas de paso de mensajes o de memoria compartida utilizadas en la implementación. Este es el motivo del por qué las investigaciones actuales en este campo se centran en implementar nociones de nivel *strong*. Sin embargo, son pocas las nociones que han sido implementadas con éxito: el nivel *strong*, el nivel *hyperfairness* en algunos casos restrictivos y el nivel *bounded-delay*. Existen algunos resultados sobre la implementación de los niveles *finitary* y k-*bounded*, pero todavía no son concluyentes. En las siguientes sección, describiremos las mejores propuestas de las que tenemos constancia.

4.2 Implementando el nivel *strong*

El nivel *strong*, también conocido como nivel 0-*fairness* o propiedad de *compassion*, es probablemente la noción más habitual de selección justa, de ahí el motivo de que sea el nivel que más interés despierta en el campo de la investigación. En esta sección, primero exponemos algunos ressultados que

demustran que el nivel *strong* no puede ser implementado utilizando planificadores sin esperas, luego, describimos las tres propuestas de Francez y Forman, Corchuelo *et al.* y Joung.

4.2.1 Resultados de imposibilidad

Tsay y Bagrodia fueron los primeros en demostrar que la selección de interacciones a nivel *strong* no puede ser implementada utilizando planificadores deterministas libres de esperas [127]. Este resultado de imposibilidad es demostrado para interacciones bipartitas y después es generalizado para interacciones multipartitas. Las suposiciones que ellos consideran son las siguientes:

i. Ningún participantes de una interacción puede estar ejecutando cálculo local indefinidamente, es decir, los participantes deben ofrecer alguna interacción de vez en cuando. Esto implica que éstos no nunca terminan.

ii. Los participantes de las interacciones son autónomos e independientes los unos de los otros, es decir, el subconjunto de interacciones en las que un proceso puede participar en un instante dado nunca depende de otros procesos.

iii. Los cambios de estado de los participantes no son detectados por el resto de participantes inmediatamente, es decir, el tiempo necesario para notificar los cambios no es despreciable.

La demostración de estos resultados la hacen a través del contraejemplo del Programa §4.1, en el que la siguiente ejecución el posible: inicialmente todos los procesos están ejecutando cálculo local hasta que los procesos P_1 y P_2 ofrecen participar en las interacciones A o B, lo que habilita A su ejecución. Si la planificación es determinista y sin esperas, la interacción A debe ser ejecutada tan pronto se habilita. Sin embargo, algún tiempo no despreciable pasa entre estos eventos. Como los procesos son autónomos, es posible que P_3 ofrezca participar en B o C justo en entre estos dos eventos. Esto implica que todas las interacciones están habilitadas inmediatamente después de haber decidido ejecutar la interacción A, pero justo antes de que haya sido ejecutada. Obviamente, sieste escenario se repite en infinitas ocasiones, ni la interacción B ni la C resultan seleccionadas para ser ejecutadas aunque están habilitadas en infinitas ocasiones, es decir, esta ejecución no es de nivel *strong* en cuanto a la selección de interacciones.

SYSTEM :: [P_1 || P_2 || P_3], **where**
P_1 :: *[A⟨⟩ → **skip** [] C⟨⟩ → **skip**];
P_2 :: *[A⟨⟩ → **skip** [] B⟨⟩ → **skip**];
P_3 :: *[B⟨⟩ → **skip** [] C⟨⟩ → **skip**].

Programa 4.1: *Contraejemplo de Tsay y Bagrodia.*

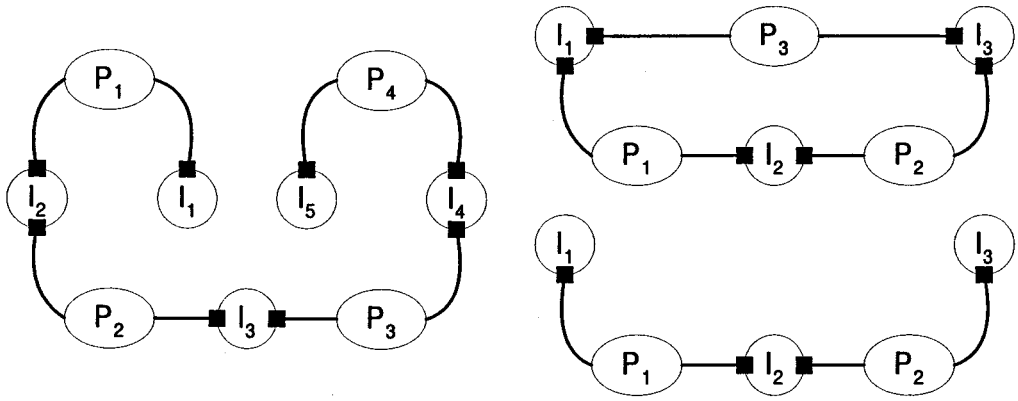


Figura 4.1: *Resultados de imposibilidad de Joung y Smolka.*

De forma independiente, Joung y Smolka [66] obtienen similares conclusiones, pero ellos, además, caracterizan la topología exacta de los subsistemas en los que la selección *strong* de interacción no puede ser implementada con planificadores deterministas sin esperas. Estos subsistemas, que son muy habituales en la práctica, son mostrados en la Figura §4.1.

En resumen, la única forma de implementa selección justa de interacciones a nivel *strong* es, o bien utilizar planificadores deterministas que puedan retrasar la ejecución de algunas interacciones por un tiempo arbitrario, o bien utilizando técnicas probabilísticas. Esta claro que ambas soluciones repercuten en la eficiencia pero no exista otra alternativa posible.

4.2.2 Un planificador simple

Francez y Forman idean un planificador que se construye sobre los resultados de Olderog y Apt en el contexto de programas paralelos con variables compartidas [98]. Esta solución asocia un contador inicializado a cero a cada interacciones. Así, una interacción habilitada es seleccionadas si y sólo si su

contador tiene el valor mínimo de entre todas las habilitadas. En este caso, su contador es reiniciado a un valor natural aleatorio y el contador del resto de interacciones es decrementado en una unidad, de esta forma se incrementa la prioridad de las mismas.

Merece la pena destacar que este algoritmo requiere conocer el estado de todas las interacciones antes de tomar una decisión, es decir, esto retrase la ejecución de las interacciones hasta que todo el sistema se estabiliza. Además, este planificador requiere contadores infinitos para ser implementado, o de lo contrario, pierde la completitud y, lo que es peor, la corrección. Este hecho motivó a Corchuelo a llevar a cabo una versión en la que los contadores pueden ser finitos y no se pierde la corrección, aunque la completitud se pierda como es habitual en estos casos [34].

4.2.3 Un planificador incremental

La principal motivación detrás del planificador de Corchuelo es producir una versión incremental de la propuesta de Francez y Forman en la que el estado de todo el conjunto de interacciones sólo necesita ser conocido en el peor de los casos, lo que mejora notablemente la eficiencia [34]. Este planificador organiza las interacciones en una cola ordenada por sus contadores de forma que la interacción más cercana a la principio es aquella cuyo contador es menor. Así, las interacciones son examinadas en secuencia y se para cuando se encuentra la primera interacción habilitada. Entonces, su contador es reiniciado a un valor aleatorio, y el contador de las restantes interacciones no examinadas es decrementado en una unidad.

Está demostrado que los tiempos medios de este planificador son mejores que la propuesta original de Francez y Forman, y que, por contra, no pierde la corrección al ser implementado con contadores finitos.

4.2.4 Un planificador aleatorio

Recientemente, Joung [67] ha presentado dos planificadores basados en *La Teoría de lo Grandes Números* que permiten la selección de interacciones a nivel *strong* con probabilidad uno. Ambos son equivalentes, la única diferencia entre ellos es que uno utiliza primitivas de paso de mensajes y el otro utiliza primitivas de memoria compartida. Son una mejora de un resultado previo, ya que el tiempo que los elementos están realizando cálculo local no necesita estar acotado por una constante predeterminada, aunque la solución que propone

sigue sin poder trabajar con elementos en los que este tiempo crece de manera monótona [71].

Ambos algoritmos están basados en la idea de "intento, estera y test": cuando un procesos quiere participat en un subconjunto de interacciones, selecciona una aleatoriamente y espera por un tiempo (referido como ventana de monitorización) a que los otros participantes también estén interesados en la misma interacción, y si esto es así la ejecutan; de lo contrario, un nuevo intento es realizado y el procedimiento se repite de nuevo.

El coste de un ciclo intento-espera-test puede ser considerablemente alto. El tiempo que un participantes está intentando establecer una interacción se ha probado que no es más grande que

$$\frac{|\mathcal{P}|\eta}{\prod_{p\in\mathcal{P}}\psi_p} + (|\mathcal{P}| - 1)\eta + \epsilon,$$

donde $\mathcal{P}$ es el conjunto de procesos que pueden participantar en esa interacción, ψ_p is la probabilidad de que el participante p escoja a esa interacción en un sorteo aleatorio y $\eta - \epsilon, \eta > \epsilon > 0$ es la longitud de la ventana de monitorización. El orden de complejidad se incremente ligeramente conforme $|P|$ se incrementa y resulta no práctico para aplicaciones con interacciones 4- o 5-partitas o aplicaciones en las que un participante puede ofrecer más de tres interacciones al mismo timepo.

4.3 Implementando el nivel *hyperfairness*

La primera implementación del nivel *hyperfairness*, también conocido como nivel 0-*fairvess*, fue presentada por Attie *et al.* en la Referencia [10]. Su propuesta es una técnica tranformacional en la que el código fuente de los programas objeto de estudio necesita ser cambiado de forma que integra un planificador *hyperfair ad hoc*.

La transformación consiste en añadir un nuevo proceso al sistema. Al que se le conoce como planificador ya que controla la habilitación y ejecución de las interacciones de forma independiente al planificador proporcionado por el sistema en tiempo de ejecución o por la máquina. Éste tiene un contador asociado a cada interacción y una variable que referencia a la interacción cuyo contador es mínimo. Los contadores son inicializado a un valor natural aleatorio. Cualquier proceso puede leer estas variables, pero el único que puede cambiarlas es el planificador. Además, los procesos tienen que estar en forma

```
S :: [P || Q], where
P :: *[ true & x⟨⟩ → skip
     [] true & y⟨⟩ → skip
     ];
Q :: s := 0;
     *[ s ∈ {0, 1} & y⟨⟩ → s := 1
     [] s = 1 & x⟨⟩ → skip
     ].
```

Programa 4.2: *Transformación hyperfair: programa original.*

normal (véase Appendix §B), y los guardas de sus bucles principales tiene que se cambiado de la siguiente forma:

$$B_x^P \wedge (\forall \text{ interacción } y \cdot P \text{ puede participar en} y \wedge y = MIN \Rightarrow (x = y \vee \neg B_y^P)),$$

siendo B_x^P el guarda original, x la interacción correspondiente y MIN la interacción con el contador mínimo. Cuando una interacción es ejecutada su contador es iniciado a un natural aleatorio y los contadores de las restantes interacción es decrementado en uno, actualizando MIN se procede.

Ilustraremos la transformación a través del Programa §4.3, cuyo equivalente *hyperfair* es el §4.2. Fíjese que esta transformación puede y debe ser simplificada, pero esto no es fundamental para el algoritmo sea correcto.

En resumen, la idea detrás de esta implementación es retrasar la habilitación de las interacciones que no sean las más prioritarias con la esperanza de que está se termina habilitando con el tiempo y, así, pueda ser seleccionada para ser ejecutada. Fíjese que esto podría conducir al planificador a un interbloqueo la interacción más prioritaria no se habilita y las restantes están esperando a que esto ocurra. Esto ocurre, por ejemplo, en el Programa §4.2 si la inicialización aleatoria asignada a los contadores hace que *x* sea la primera interacción más prioritaria. En este caso el planificador retrasa la ejecución de *y* hasta que las otras se habiliten; sin embargo, esto no puede ocurrir ya que *x* no puede ser ejecutada a menos que *y* se ejecute antes.

Attie *et al.* caracterizaron el subconjunto de programas en los que este algoritmo puede ser aplicado, y concluyeron que puede ser utilizado siempre que los procesos están en forma normal y toda interacción es resistente a conspiraciones. A grandes rasgos, una interacción es resistente a conspiraciones si y sólo el hecho de retrasar la ejecución de las restantes interacciones no impide que ésta se habilite en un tiempo finito. Sin embargo, no proporcionan un algoritmo para calcular el conjunto de interacciones en un programa que son resistentes a conspiraciones, ya que esto es un problema de difícil solución

```
S :: [P || Q || H], where
P :: *[ true ∧
        (x = MIN ⇒ (x = x ∨ ¬true)) ∧
        (y = MIN ⇒ (x = y ∨ ¬true)) & x⟨⟩ → skip
     [] true ∧
        x = MIN ⇒ (y = x ∨ ¬true) ∧
        y = MIN ⇒ (y = y ∨ ¬true) & y⟨⟩ → skip
     ];
Q :: s := 1;
     *[ s ∈ {0, 1} ∧
        x = MIN ⇒ (y = x ∨ s ≠ 1)) ∧
        y = MIN ⇒ (y = y ∨ s ∉ {0, 1})) & y⟨⟩ → s := 1
     [] s = 1 ∧
        x = MIN ⇒ (x = x ∨ s ≠ 1)) ∧
        y = MIN ⇒ (x = y ∨ s ∉ {0, 1})) & x⟨⟩ → skip
     ];
H :: { This is the scheduler }
     z(x) := Ξ; z(y) := Ξ; MIN := (z(x) ≤ z(y) ? x : y);
     *[ Enabled(x) → z(x) := Ξ; z(y) := z(y) - 1; MIN := (z(x) ≤ z(y) ? x : y)
     [] Enabled(y) → z(x) := z(x) - 1; z(y) := Ξ; MIN := (z(x) ≤ z(y) ? x : y)
     ].
```

Programa 4.3: *Transformación hyperfair: programa hyperfair.*

4.4 Implementando el nivel *bounded-delay*

Hasta lo que sabemos, no existen en la bibliografía implementaciones de este nivel en el contexto de la selección justa de interacciones. Sin embargo, los resultados de Fagin y Williams en el contexto de la planificación de *carpool* pueden ser fácilmente adaptables al contexto de las interacciones multipartitas si consideramos que existe una cota en el tiempo que un proceso puede estar ejecutando cálculo local [48].

Su algoritmo primero define U como el mínimo común múltiplo de los números $1, 2, \ldots, N$, donde N es el número de interacciones en el sistema. Se necesita una tabla de contadores en la que cada columna representa una interacción y cada fila la selección de una de ellas para su ejecución. Inicialmente, la tabla contiene una única fila inicializada a ceros. Para que una interacción sea seleccionada el sistema debe estar estable, es decir, ningún proceso puede estar ejecutando cálculo local ya que todos ellos estarán esperando a que una interacción sea ejecutada. Cuando esto ocurre, la interacción seleccionada es aquella cuyo contador en la última fila es mínimo. Una nueva fila se añade y los contadores son actualizados como sigue: el que está asociado a la integración seleccionada es incrementado en $U(k-1)/k$ unidades, donde k es el

número de interacciones que estuvieran habilitadas; el resto de contadores son decrementados en U/k unidades. Los autores demuestran que el algoritmo es correcto y que los contadores se hallan en el intervalo $[-(N-1)f(N)U, f(N)U]$, donde $f(N)$ se halla en el intervalo $\left[\frac{(N-1)}{3}, a(N)\right]$ con

$$a(x) = \begin{cases} 0 & \text{Si } x = 1 \\ (x-1)a(x-1)+1 & \text{en caso contrario.} \end{cases}$$

(Los autores atribuyen a Don Coppersmith de *T.J. Watson Research Centre* bajar la cota superior que es exponencial de $f(N)$ a $\frac{(N-1)}{2}$, pero estos resultados no están publicados aún.)

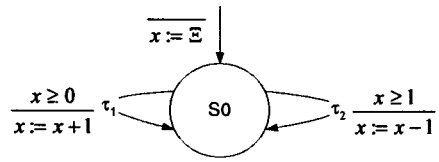
Aunque el algoritmo es un poco complicado, la idea en la que se apoya es garantizar que la diferencia entre el número ideal de veces que una interacción debería se ejecutada u el número real de ejecuciones no excede una cota. El número ideal de ejecuciones es

$$\frac{1}{2}b_2 + \frac{1}{3}b_3 + \ldots + \frac{1}{N}b_N = \sum_{k=2}^{N} \frac{1}{k}b_k,$$

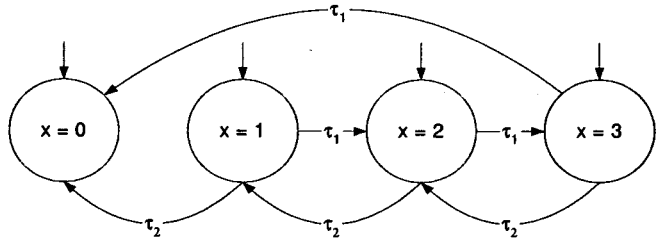
donde b_k es el número de veces que exactamente k interacciones estaban habilitadas. Los autores demuestran que el valor absoluto de esta diferencia se encuentra en el intervalo $\left[\frac{(N-1)^2}{3}, \frac{(N-1)a(N)}{2}\right]$. (Si la cota superior de Coppersmith es cierta, es fácil demostrar que la cota superior exponencial se reduce a $\frac{(N-1)^2}{2}$.)

4.5 Implementando los niveles *finitary* y k-*bounded*

Alur y Henzinger desarrollaron un método transformacional para implementar las nociones k-*bounded* y *finitary* a nivel *weak* en el contexto de sistemas de transiciones estándar [5, 90]. El trabajo de implementar interacciones multipartitas utilizando este modelo de concurrencia aún está en curso [103, 100], pero resulta fácil darse cuenta de que la ejecución de una interacción en un sistema basado en interacciones multipartitas puede ser interpretado como la ejecución de la transición adecuada en su sistema de transiciones equivalente, y que la habilitación de una interacción es equivalente a la habilitación de su correspondiente transición.



(a) Máquina de estados.



(a) Sistema de transiciones estándar.

Figura 4.2: *Transformación finitary: sistema original.*

Un sistema de transiciones estándar puede ser visto como una tupla de la forma (Q, Q_0, T), donde $Q = \{q_0, q_1, \ldots\}$ es un conjunto de estados, $Q_0 \subseteq Q$ es un conjunto de estados iniciales y $T = \{\tau_0, \tau_1, \ldots, \tau_m\}$ es un conjunto finito de transiciones. Todo estado $q \in Q$ es el resultado de asignar valores a todas las variables. Para una transición $\tau \in T$ y un estado $q \in Q$, $\tau(q)$ denota el conjunto $\{q' \mid (q, q') \in \tau\}$. La transformación de Alur y Henzinger consiste en añadir un contador por cada transición y una variable umbral que ayuda a prohibir que una transición que está habilitada k veces no se ejecute. Es decir, el estado de la transformación es $Q_0 \times \{0\}^m \times \mathbb{N}$ y el conjunto que de transiciones contiene $((q, c_1, c_2, \ldots, c_m, b), (q', c_1', c_2', \ldots, c_m', b))$ si y sólo si $q' \in \tau(q)$, $b' = b$ y $\forall i \in [1, m]$, $\tau_i(q) = \emptyset \vee q' \in \tau_i(q)$ implica que $c_i' = 0$, o sino $c_i < b \wedge c_i' = c_i + 1$.

A grandes rasgos, el objetivo de esta transformación es contar el número de veces que una transición ha sido rechazada y prohibir que esto ocurra más de b veces. Por ejemplo, la Figura §4.2.(a) muestra una máquina de estados simple que modela un sistema con una variable cuyo dominio es el conjunto $\{0, 1, 2, 3\}$ a la que inicialmente se le asigna un valor aleatorio; existen dos transiciones que pueden incrementar y decrementar su valor siempre que este sea positivo. La Figura §4.2.(b) representa a su sistema de transiciones estándar, y la Figura §4.3 la correspondiente transformación para un $b \geq 4$.

Fíjese que, en general, el número de estados de un sistema de transiciones estándar es muy grande aunque el sistema sea simple. En nuestro ejemplo, si la variable x fuese una variable natural típica de 32 bits, el sistema de transiciones estándar tendría 2^{32} estados distintos, y la transformación equivalente

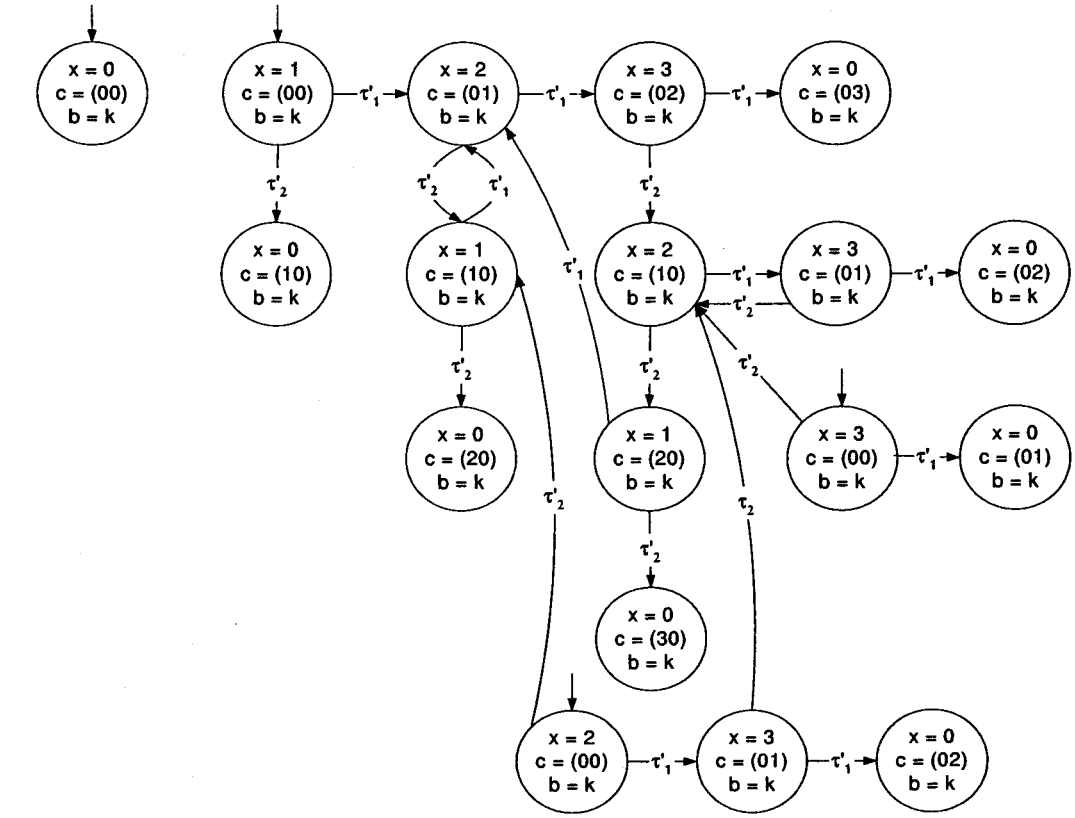


Figura 4.3: *Transformación finitary: sistema resultante.*

excedería a este número en varios ordenes de magnitud. En la práctica, poner un valor de antemano a la constate k es deseable, en cuyo caso la transformación es k-*bounded* y de nivel *weak*, pero el número de estado aún excedería en varios ordenes de magnitud al original. Hasta lo que nosotros sabemos, no existen resultados publicados sobre cómo transformar maquinas estado de forma más práctica que la presentada.

Parte III

Nuestra aportación

Capítulo 5

Motivación

Motivation will almost always beat mere talent.

Norman R. Augustine, 1935–
Chairman of Lockheed Martin Corporation

Recientemente, la selección justa se ha convertido en una de las áreas de investigación más importantes en el campo de las interacciones multipartitas. Sin embargo, las nociones actuales no son lo suficientemente prácticas y no resuelven los problemas de la finitud justa y las conspiraciones simultáneamente. Nuestro objetivo en este capítulo es presentar en detalle estos problemas y motivar la necesidad de nociones más restrictivas a la vez que implementables. En las Secciones §5.1, §5.2 y §5.3, presentamos estos problemas en detalle e identificamos sus causas; en la Sección §5.4, demostramos que ninguna de las nociones actuales de selección justa los resuelven simultáneamente; en la Sección §5.5, discutimos sobre nuestros resultados y argumentamos por qué otros autores no han intentado proporcionar una solución satisfactoria.

5.1 Introducción

La selección justa fue diseñada para ayudar a verificar algunas propiedades complejas de los programas no deterministas desde un punto de vista teórico. El objetivo era proporcionar una abstracción a través de la cual fuese posible alejarse de los detalles de bajo nivel concernientes a la planificación de los conflictos entre las distintas alternativas sin tener que ignórarlas. En otras palabras, el objetivo era identificar una propiedad común capaz de generalizar todos los comportamientos deseables de un programa como punto de partida para el razonamiento y la verificación de otras propiedades [54].

La eventualidad fue el punto de partida, pero, por desgracia, ésta puede ser vista como el mayor problema práctico ya que ningún planificador real en un maquina finita puede implementar la eventualidad de una forma completa. Después se presentó la idea de la eventualidad acotada como forma de poner una cota a la eventualidad que un planificador puede implementar. Recientemente, esto ha atraído la atención de varios investigadores ya que ésta ayuda a demostrar algunas propiedades más fácilmente que cuando se razona sobre eventualidades puras [5, 41].

Algunos autores también han propuesto nociones de justicia sobre la base de la ocurrencia eventual de algunos eventos a través de los cuales se tiene la ejecución de un programa bajo control. Por desgracia, ninguna de las nociones actuales parece ser lo suficientemente apropiada en el contexto de las interacciones multipartitas. El motivo es que ninguna de ellas tiene una implementación razonable o no pueden atacar a los dos problemas antes mencionados. Esto sustenta la necesidad de una nueva noción de selección justa, y esto es la principal motivación de esta tesis.

5.2 La finitud justa

Ya que las nociones basadas en eventualidades se apoyan en la ejecución eventual de toda interacción que se habilita infinitamente a menudo, se tiene que ningún planificador real en una máquina finita puede generar el conjunto de todas las posibles ejecuciones que satisfacen estas nociones [5, 54]. El motivo es que siempre hay una cota física de la eventualidad en que un ordenador puede implementar que depende del número de estados, del rango de los contadores, de la disponibilidad de a memoria o de cualquier otro recurso que sea finito por naturaleza. Por ejemplo, el planificador basado en contadores de Corchuelo presentado en la Sección §4.2.3 no puede generar ninguna

$[P_2.\{Get_2\}\ P_4.\{Get_4\}\ P_5.\{Get_5\}\ F_4.\{Get_4, Get_5\}] \frown$
$[P_1.\{Get_1\}\ P_3.\{Get_3\}\ F_1.\{Get_1, Get_2\}\ F_2.\{Get_2, Get_3\}\ F_3.\{Get_3, Get_4\}\ F_5.\{Get_5, Get_1\}$
$\mathbf{Get_1}\ \mathbf{Get_3}\ P_1.\{Rel_1\}\ F_1.\{Rel_1\}\ F_5.\{Rel_1\}\ P_3.\{Rel_3\}\ F_2.\{Rel_3\}\ F_3.\{Rel_3\}\ \mathbf{Rel_1}\ \mathbf{Rel_3}]^*$

Figura 5.1: *Ejemplo de ejecución para ilustrar la finitud justa.*

ejecución finita en la que una interacción sea ejecutada después de $N + 1$ ejecuciones de otras interacciones si los contadores sólo pueden tomar N valores distintos.

Sin embargo, como se demostró en la Sección §3.5.2, utilizar planificadores aunque sean incompletos parece deseable en la práctica ya que estos pueden producir ejecuciones en las que la planificación de las interacciones no difiere demasiado de la distribución teórica esperada. Como también mostramos, este comportamiento depende del generador de números aleatorio que se utilice, lo que implica que la bondad de los resultados no depende de la propia noción, sino que de la implementación del planificador. Esto no es deseable generalmente, ya que aunque todas las implementaciones garantizan que las interacción son planificadas a un nivel *strong*, la diferencia entre ellas es importante desde el punto de vista práctico.

Además, ningún programa puede estar ejecutándose por siempre ya que necesariamente debe para o ser parado algún día. Esto implica que toda ejecución real es finita por naturaleza, lo que implica que todas las ejecuciones reales son trivialmente justas por definición. El motivo es que ninguna interacción real puede estar habilitada infinitamente a menudo en una ejecución finita; así, ninguna noción de justicia basada en eventualidad puede forzar a una interacción a ser ejecutada de vez en cuando. Alguien puede pensar que no tiene importancia si un planificador es justo o no ya que el comportamiento observable en cualquier programa es de nivel k-*fair* para algún $k \in \mathbb{N}$, ∞ [24, 45, 97, 120].

Por ejemplo, la Figura §5.1 ilustra una ejecución parcial del conocido problema de los filósofos comensales presentado en el Programa §2.4.1. Este programa no termina, pero esta ejecución es posible se pulsamos la tecla de escape, se le envía una señal de terminación al proceso, el sistema es reiniciado por cuestiones de mantenimiento o cualquier otra causa razonable. Hemos obviado los detalles concernientes a los estados de los procesos y nos hemos centrado en los momentos en los que los procesos ofrecen participar en un subconjunto de interacciones y los momentos en los que las interacciones son ejecutadas. Los primeros eventos son denotados como $P.\{I_1, I_2, \ldots, I_n\}$, y los segundos son denotados con el nombre de la interacción. Fíjese que sólo las

$[P_1.\{Get_1\}\ P_2.\ \{Get_2\}\ P_3.\{Get_3\}\ P_4.\{Get_4\}\ P_5.\{Get_5\}] \frown$
$[F_1.\{Get_1, Get_2\}\ F_3.\{Get_3, Get_4\}\ F_5.\{Get_5, Get_1\}\ \mathbf{Get_1}\ P_1.\{Rel_1\}\ F_1.\{Rel_1\}\ F_5.\{Rel_1\}\ \mathbf{Rel_1}$
$F_2.\{Get_2, Get_3\}\ \mathbf{Get_3}\ P_3.\{Rel_3\}\ F_2.\{Rel_3\}\ F_3.\{Rel_3\}\ \mathbf{Rel_3}\ P_1.\{Get_1\}\ P_3.\{Get_3\}$
$F_3.\{Get_3, Get_4\}\ F_4.\{Get_4, Get_4\}\ \mathbf{Get_4}\ P_4.\{Rel_4\}\ F_3.\{Rel_4\}\ F_4.\{Rel_4\}\ \mathbf{Rel_4}\ P_4.\{Get_4\}]^{\infty}$

Figura 5.2: *Ejemplo de ejecución para ilustrar las conspiraciones.*

interacciones Get_1, Get_3 y sus correspondientes Rel_1 y Rel_3 son ejecutadas. Sin embargo, Get_2, Get_4 yGet_5 están habilitadas cuando Get_1 y Get_3 lo están.

A pesar de esta observación, esta ejecución es justa para cualquier noción basada en eventualidades. Aunque las interacciones de Get_1 y Get_3 sean ejecutadas un *googolplex*[†1] de veces en detrimento del resto, la ejecución seguiría satisfaciendo los requisitos de todas las nociones basadas en eventualidades. Nosotros nos referimos a este problema como "finitud justa". Las nociones de nivel *bounded* pueden ser la respuesta, pero existe un problema que estás no resuelven: las conspiraciones.

5.3 Las conspiraciones

Las conspiraciones son uno de los mayores y más complicados obstáculos para permitir que una noción sea adecuada de acuerdo al criterio de Apt *et al.* [8, 10, 17, 18]. A grandes rasgos, una conspiración ocurre si una interacción no se habilita aunque todos sus participantes las ofrezcan en infinitas ocasiones, pero no las mismo tiempo, es decir, que si el entrelazado fuese diferente, existiría la posibilidad de que todos la ofrecieran al mismo tiempo.

Por ejemplo, la Figura §5.2 ilustra una ejecución infinita de los filósofos comensales en a que se satisfacen todas las nociones basadas en eventualidades. Fíjese, sin embargo, que ni la interacción Get_2 ni la Get_5 se habilitan en esta ejecución. En el caso de Get_2, el motivo es que el tenedor F_1 se compromete con Get_1 antes de que el tenedor F_2 pueda ofrecer participar en Get_2. Una simple reorganización de estas acciones independientes en la que el tenedor F_2 ofrezca su participación en Get_2 o Get_3 justo antes de que Get_1 se ejecutase permitiría a Get_2 habilitarse y, en consecuencia, tendría infinitas oportunidades de ejecutarse. El razonamiento es similar en el caso de Get_5.

[†1] El matemático americano Edward Kasner le pidió a su sobrino que se inventara un nombre para un número muy grande: 10 elevado a 100. El niño le llamó *googol*. El pensó que con esto sería bastante, pero pero otro matemático definió el *googolplex* como 10 elevado a un *googol*. Este término se hizo popular a partir de un programa de la televisión americana.

```
SYSTEM :: [P || Q], where
        P :: *[ A⟨⟩ → B⟨⟩ [] C⟨⟩ → skip ];
        Q :: *[ A⟨⟩ → [ B⟨⟩ → skip [] C⟨⟩ → skip ] ].
```

Programa 5.1: *Un programa con conspiraciones inherentes.*

Merece la pena destacar que no todas las conspiraciones pueden ser evitadas con una reorganización. Por ejemplo, en el Programa §5.1, es imposible que la interacción C se habilite ya que este programa es totalmente determinista y primero se ejecuta la interacción A y después la B, no existiendo ninguna posibilidad para P y Q de poder participar en la interacción C al mismo tiempo. En este caso decimos que se trata de una conspiración inherente al programa, mientras que en el anterior decimos que se trata de una conspiración no inherente. Esta distinción es muy importante ya que nosotros tratamos la conspiraciones no inherentes retrasando la ejecución de algunas interacciones, es decir, reordenando las acciones atómicas que se ejecutan en el programa. Sin embargo, si existe reordenación posible para tratar la conspiración inherente del Programa §5.1.

Al hilo de esta observación, está claro que sería deseable para una noción de selección justa práctica en el contexto de las interacciones multipartitas se capaz las conspiraciones, dando a toda interacción que puede habilitarse la oportunidad de se ejecutada.

5.4 Análisis de las soluciones actuales

Nuestro objetivo en esta sección es demostrar que ninguna de las nociones implementables del Capítulo §4 resuelve los problemas ante mencionados simultáneamente, aunque si de forma independiente.

5.4.1 Nivel *economy*

El nivel *economy* de la taxonomía de Francez requiere que la diferencia entre el número de veces que una interacciones se ejecuta y el número ideal esté acotada por una constante predefinida que dependa sólo del número de interacciones.

Esto no permite aquellas ejecuciones como las presentadas en la Figura §5.1. En este caso la diferencia se incrementaría monótonamente para las interacci-

ones Get_2, Get_4, Get_5, Rel_2, Rel_4 y Rel_5 ya que no resultan nunca ejecutadas. El problema con esta noción es que permite ejecuciones con conspiraciones como las presentadas en la Figura §5.2 ya que las interacciones que no se habilitan nunca son tenidas en cuenta.

Resaltaremos que cuando se asume que existe una cota del tiempo que un proceso puede estar ejecutando cálculo local sin interactuar con otros, entonces una implementación del nivel *economy* puede ser visto como una implementación del nivel *bounded-delay*. En este caso, este nivel no permitiría algunas ejecuciones finitas, aunque seguiría sin resolver el problema de las conspiraciones.

5.4.2 Niveles *strong*, *weak* y *unconditional*

Técnicamente, el nivel *strong* previene que las interacciones que están habilitadas en infinitas ocasiones no se ejecuten. Así, éste subsume a los nivel *weak* y *unconditional*. Por desgracia, no resuelve ninguno de los problemas objeto de estudio.

Por ejemplo, la ejecución parcial de la Figura §5.1 es de nivel *strong*, *weak* y *unconditional* con respecto a las interacciones, ya que ninguna de ellas se habilita en infinitas ocasiones, lo que implica que ninguna tiene obligación de ser ejecutada. Análogamente, la ejecución de la Figura §5.2 también satisface los requisitos de estos nivel de justicia ya que la conspiración contra Get_2 y Get_5 prohíben que éstas se habiliten.

5.4.3 Niveles ∞-*fair* y *hyperfair*

El único nivel que puede resolver el problema de las conspiraciones es el ∞-*fair*, ya que no depende de la ejecución objeto de estudio, sino que todas las posibles continuaciones de sus prefijos finitos. Así, si una ejecución puede habilitar en infinitas ocasiones durante una ejecución, este nivel obliga a que se habilite en infinitas ocasiones, lo que implicaría que podría ser ejecutada en infinitas ocasiones.

La única materialización de la tenemos conocimiento en el contexto de las interacciones multipartitas es el nivel *hyperfair* [10]. Por desgracia, este nivel tiene varios inconvenientes prácticos, ya que toda ejecución finita es de nivel *hyperfair* por definición; además, cuando existe una interacción en el programa que no sea resistente a conspiraciones, entonces cualquier ejecución que produzca el programa será también de nivel *hyperfair* por definición; por otra

parte, en la bibliografía no existe ningún algoritmo genérico para determinar cuándo una interacción es resistente a conspiraciones; por último, la noción sólo puede ser aplicada a programas en forma normal, aunque esto no es importante ya que cualquier programa tiene su equivalente en forma normal si introducimos algunas variables de estado.

Estos inconvenientes implican que las dos ejecuciones de las Figuras §5.1 y §5.2 son *hyperfair* ya que la primera es finita y la segunda es producida por un programa que contiene interacciones que no son resistentes a conspiraciones. Resaltamos que una interacción es resistente a conspiraciones cuando retrasar la ejecución del resto de interacciones no prohíbe que la primera se habilite en un tiempo finito. Fíjese que si la ejecución, por ejemplo, de Get_1 es retrasada, entonces Rel_1 no puede habilitarse en un tiempo finito porque Rel_1 necesita que Get_1 se ejecute antes que ella, es decir, un tenedor no puede ser soltado a menos que primero haya sido cogido.

5.4.4 Niveles k-*bounded* y *finitary*

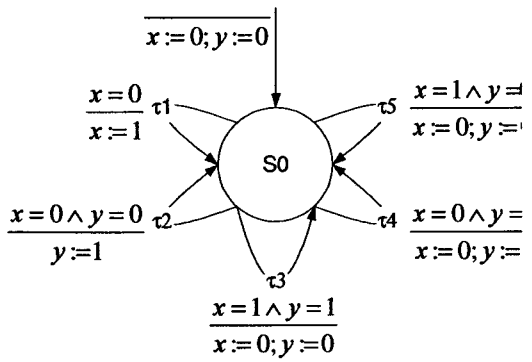
Las nociones k-*bounded* y *finitary* de Alur y Henzinger fueron diseñados en el contexto de los sistemas de transiciones estándar [90]. El trabajo de implementar interacciones multipartitas usando este modelo de concurrencia está aún en desarrollo [103, 100], pero es fácil darse cuenta que la ejecución de una interacción en un sistema basado en interacciones multipartitas puede ser interpretado como la ejecución de la transición adecuada de su sistema de transición equivalente, y una interacción se habilita es equivalente a que sus correspondiente transición se habilite. A pesar de que estos resultados no son concluyentes, podemos demostrar que ni el nivel k-*bounded* ni el *finitary* pueden resolver las conspiraciones.

En cuanto al problema de la finitud justa, ambas nociones no permiten las ejecuciones finitas como la mostrada en la Figura §5.1. Esta no satisface los requisitos de estas nociones ya que la interacción Get_2 se habilita más de k veces para cualquier natural k mayor que la longitud de la ejecución, aunque nunca resulta ejecutada. No obstante, la generalización del nivel k-*bounded* al nivel *finitary* no resuelve el problema de la finitud justa ya que también toda ejecución finita es de nivel *finitary* por definición. El motivo es que existe siempre una cota superior del número de veces que una interacción puede ser ejecutada si la ejecución es finita.

Como quiera que sea, ninguno de estos niveles considera las conspiraciones. Para demostrarlo, utilizaremos el programa de la Figura §5.3 ya que el sistema de transiciones asociado al problema de los filósofos comensales es

```
S :: [P || Q], where
P :: { x: natural }
   *[ x = 0 & τ1⟨⟩ → x := 1
   [] x = 0 & τ2⟨⟩ → skip
   [] x = 1 & τ3⟨⟩ → x := 0
   [] x = 0 & τ4⟨⟩ → x := 0
   [] x = 1 & τ5⟨⟩ → x := 0
   ]
Q :: { y: natural }
   *[ y = 0 & τ2⟨⟩ → y := 1
   [] y = 1 & τ3⟨⟩ → y := 0
   [] y = 1 & τ4⟨⟩ → y := 0
   [] y = 0 & τ5⟨⟩ → y := 0
   ]
```

(a) Programa de ejemplo.



(b) Máquina de estados.

Figura 5.3: *Programa conspiratorio y su correspondiente máquina de estados.*

complicado y dificulta innecesariamente la comprensión del ejemplo. Fíjese que es posible una ejecución en la que las interacciones sean seleccionadas de acuerdo al patrón $[\tau 2\tau 4\tau 1\tau 5]^{\infty}$ y que prohíbe que la interacción $\tau 3$ puede habilitarse y que se ejecute. El motivo es que las transiciones $\tau 1$ y $\tau 2$ habilitan el guarda de la transición $\tau 3$ intermitentemente, aunque la ejecución de la transición $\tau 4$ entre medio prohíbe que $\tau 3$ se habilite en infinitas ocasiones [17, 18]. La Figura §5.3 muestra el sistema de transición estándar asociado a este programa y su correspondiente transformación *finitary*. Fíjese que la conspiración antes mencionada ocurre ya que versión *finitary* no obliga a que la transición $\tau 1$ ocurra en el estado $x = 0, y = 1$.

Merece la pena destacar que algunos valores de k pueden resolver esta conspiración, pero otra contra $\tau 5$ surge. Por ejemplo, cuando $k = 2$ la transición $\tau 4$ no puede ocurrir en el estado $x = 0, y = 1$ ya que rechazaría la transición $\tau 1$ dos veces consecutivas. En este caso, si $\tau 2$ es ejecutada inicialmente, entonces las únicas alternativas que son ejecutadas son $\tau 1$ y $\tau 3$ en este orden. Así, una ejecución en la que las interacciones sean seleccionada de acuerdo al patrón $[\tau 2\tau 1\tau 3]^{\infty}$ es posible, y prohíbe a $\tau 5$ habilitarse y a $\tau 4$ ejecutarse.

Aunque no conocemos ninguna transformación de nivel *finitary* que además sea *strong*, este resultado no podría resolver otras conspiraciones. El motivo es que la transición $\tau 3$ depende de que la transición $\tau 1$ sea ejecutada cuando el sistema se encuentre en el estado $x = 0, y = 1$, pero esto no es obligatorio. Fíjese que esto no es contradictorio con el nivel *strong* ya que la transición $\tau 1$ se habilita en infinitas ocasiones y se ejecuta en infinitas ocasiones cuando el

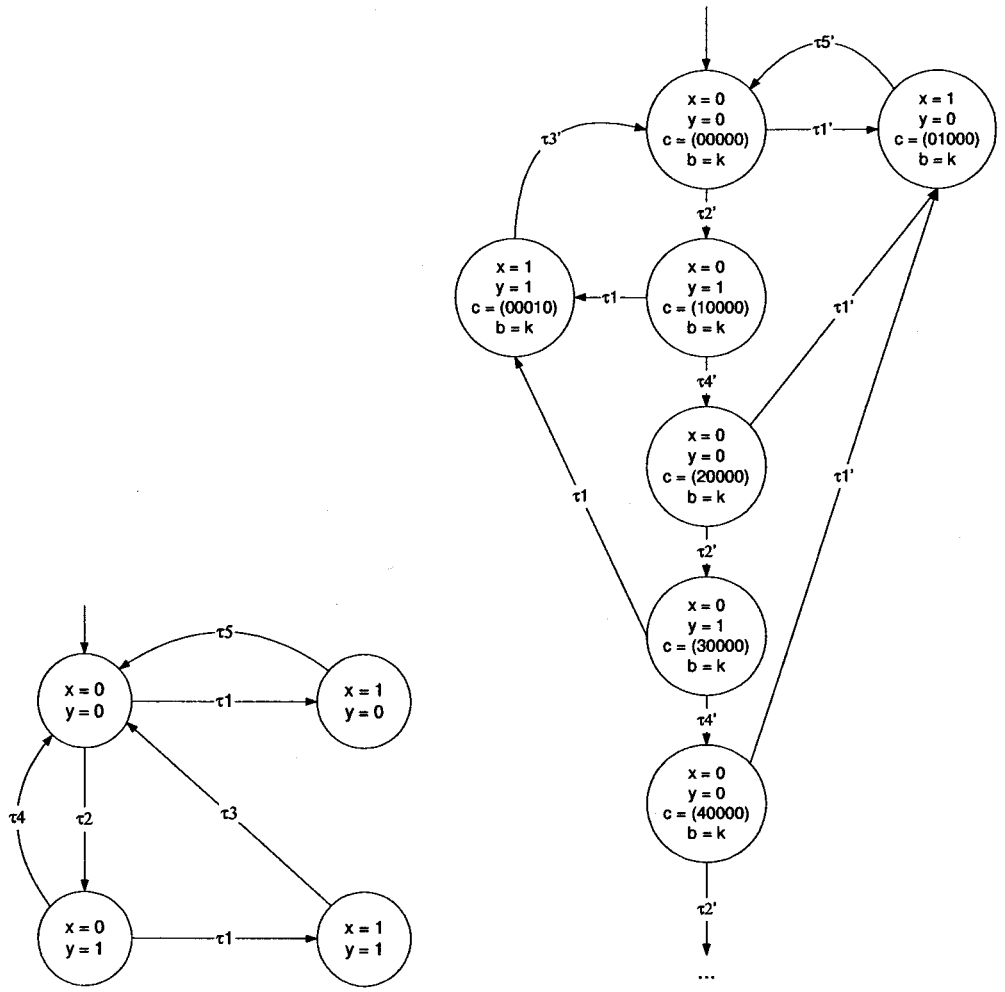
estado del sistema es $x = 0, y = 0$.

5.5 Discusión

A partir de las secciones anteriores, se puede concluir que una noción capaz resolver los problemas de la finitud justa y las conspiraciones es desable desde un punto de vista práctico; por desgracia, las propuestas actuales de justicia sólo se centran en uno de estos problemas (y con algunas limitaciones en el caso de las conspiraciones). La Tabla §5.1 resume todas las propuestas que hemos considerado y su capacidad de resolver la finitud justa y las conspiraciones. La tabla no es completa, pero las nociones que hemos dejado fuera son caso especiales. Por ejemplo, el nivel *weak* es un caso especial del *strong*. La última fila muestra las propiedades del nivel k-*conspiracy-free*, que es la noción que hemos definido en esta tesis doctoral. Como demostraremos, es la única que resuelve el problema de la fintud justa y las conspiraciones al mismo tiempo. El nivel *strong* k-*fair* fue el primer intento, pero éste podía aliviar sólo algunas conspiraciones no inherentes.

Nosotros pensamos que las propuestas actuales no resuelven ambos problemas simultáneamente ya que sus autores se centraban en aspectos distintos. Por ejemplo, el trabajo de Alur *et al.* sobre el nivel *finitary* se centra en los programas concurrentes que no están basados en interacciones multipartitas, y el concepto de conspiración no es habitual en este área. En la bibliografía aparece sólo en el contexto de las instrucciones alternativas múltiples, redes de Petri y interacciones multipartitas [17, 18, 10].

Attie *et al.* se centra en las interacciones multipartitas, y sientan las bases del nivel *hyperfair*; por desgracia, sus ideas no fueron desarrolladas en toda su extensión ya que su propuesta tiene muchas limitaciones prácticas. Sin embargo, tienen un interés especial ya que fueron los primeros en identificar claramente el problema y proponer una solución. Recientemente, en las Referencias [78, 79], Lamport considera el nivel *hyperfair* y el ∞-*fair* como la piedra angular de los programas concurrentes basados en acciones y reconoce la necesidad de investigar en este tema. El nivel *hyperfair* no resuelve el problema de la finitud justa ya que los autores se centran en diseñar una noción que preserve la propiedad de robustez, algo obligatorio si queremos que la noción sea adecuada según el criterio AFK [8, 52]. Se demostró que las conspiraciones son el mayor obstáculo para preservar esta propiedad, y hacerlo era la principal motivación de Atti *et al.*'. Fíjese que de acuerdo a la definición, ninguna ejecución finita puede contener conspiraciones.



(a) Sistema original (b) Versión *finitarily weak*

Figura 5.4: *Sistema de transiciones estándar para el programa de la Figura §5.3.*

Noción	Descripción	Fin. justa	Conspir.
Economy	Ninguna interacción es ejecutada menos de los que debiera conforme a un criterio cuantitativo.	Si	No
Strong	Ninguna interacción que se habilita en infinitas ocasiones es siempre rechazada.	No	No
∞-*fairness*	Toda interacción que puede habilitarse se habilita en infinitas ocasiones y se ejecuta en infinitas ocasiones.	No	Si
Hyperfairness	Las interacciones resistentes a conspiraciones se habilitan en infinitas ocasiones y son ejecutadas en infinitas ocasiones.	No	Si
Strong-k-*bounded*	Ninguna subsecuencia de una ejecución en la que una interacción se ha habilitado k veces no contiene una ocurrencia de dicha interacción. k debe conocerse de antemano.	Si	No
Finitary-strong	Ninguna subsecuencia de una ejecución en la que una interacción se ha habilitado k veces no contiene una ocurrencia de dicha interacción. k debe existir, pero no es impuesto de antemano.	No	No
Strong k-*fairness*	Ninguna interacción es seleccionada más de k veces son analizar el estado de las interacciones con las que ésta está en conflicto.	Si	Alivia
k-*conspiracy-free*	Ninguna interacción es seleccionada más de k veces son analizar el estado de las interacciones con las que puede llegar a estar en conflicto.	Si	Si

Tabla 5.1: *Comparación entre las nociones tratadas.*

Capítulo 6

Nuestro marco de trabajo de interacciones multipartitas

We dissect nature along lines laid down by our native language, which is not simply a reporting device for experience, but a defining framework for it.

Benjamin L. Whorf, 1897–1941
Business specialist

Debido a la complejidad inherente de los modelos basados en interacciones multipartitas, es importante definir un marco conceptual en que cada término sea definido con precisión. Nuestro objetivo en este capítulo es presentar el marco que hemos desarrollado para tratar las interacciones multipartitas que nos proporciona IP_{CORE}. *En la Sección §6.1, presentamos una introducción informal a través de un mapa conceptual; en la Sección §6.2, presentamos la definición formal; explicamos cómo definir propiedades safety, liveness y algunas nociones clásicas de selección justa en la Sección §6.3.*

6.1 Introducción

Aunque la idea de las interacciones multipartitas es conceptualmente simple, existen muchos detalles que deben ser consideramos. Esto argumenta la necesidad de una marco conceptual en el que poder describir con precisión y sin malentendidos qué es lo que puede pasar. El marco que hemos desarrollado es simple, pero no trivial, y este es el motivo por el que nos parece importante presentar una idea intuitiva antes de definirlo. El mapa conceptual de la Figura §6.1 puede facilitar su compresión. Gira en torno a los cuatro conceptos que a continuación explicamos.

Un participantes es un elemento software que puede ejecutar cálculo local sobre so propio estado de forma independiente al resto de elementos, aunque necesati cooperar con estos de vez en cuando. Un participantes puede estar en tres estados distintos:*working*, *waiting* o *finished*. Un participante puede producir eventos de ofrecimiento de la forma $p.\chi$ para hacer saber que no puede seguir a menos que coopere con otros participantes en una interacción del conjunto χ. También consume eventos de sincronización de la forma x que le permite cooperar cono otros a través de la interacción x. Fíjese que los eventos de ofrecimientos son controlados por la semántica del lenguaje subyacente, y que los eventos de sincronización son controlados por la semántica del modelo de interacción. Merece la pena destacar que un evento de la forma $p.\emptyset$ deshabilita a p para ejecutar cualquier acción, lo que puede identificarse como su terminación ya que llegaría a un punto fijo en el que ni puede ejecutar cálculo local ni puede interactuar con otros procesos.

Una interacción multipartita es una abstracción que permite a un fijo y conocido de participantes ejecutar una seria de acciones de intercambio de datos de forma atómica y coordinada. Se dice que una interacción está estable cuando ninguno de sus participantes está *working*; de lo contrario se dice que está inestable. Una interacción estable está habilitada si todos sus participantes la han ofrecido; estaré deshabilitada cuando no está habilitada. En conjunto de interacciones en las que un proceso puede participar se dice que está estáticamente enlazado, y el conjunto en de interacciones en las que un proceso está interesado en participar en tiempo de ejecución se dice que está dinámicamente enlazado. Fíjese que toda interacción está enlazado con ella misma.

Un sistema basado en interacciones multipartitas esta compuesto de un conjunto fijo de participantes e interacciones. Cuando se ejecuta produce una ejecución, que puede ser vista como una secuencia de transiciones entre configuraciones que son disparadas por una ocurrencia de un evento de ofrecimiento o un evento de sincronización. Una configuración es un objeto que

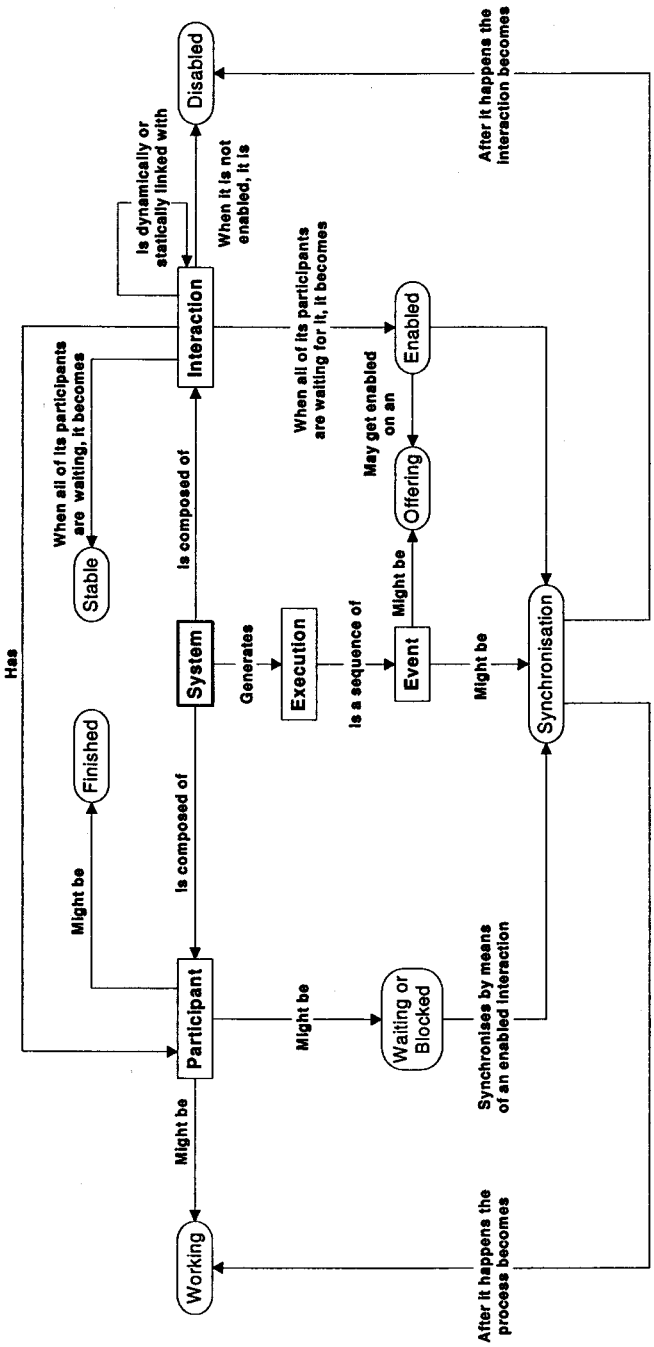


Figura 6.1: *Mapa conceptual de un sistema basado en interacciones multipartitas.*

captura los datos y el estado de ejecución de cada participante del sistema. En teoría, una ejecución puede ser finita o infinita, aunque en la práctica deba de ser finita (véase Section §5.2). Para que una ejecución sea correcta es necesario que satisfaga las siguiente propiedades *safety*:

i. Cuando un participante ofrece participar en un subconjunto de interacciones, debe permanecer bloqueado hasta que una de ellas se ejecute, o, de lo contrario, permanecerá bloqueado para siempre.

ii. Cuando un participante termina no puede volver a ofrecer participar en ninguna interacción. Además, las interacciones en las que puede participar ya no se vuelven a ejecutar.

iii. Una sincronización a través de una interacción ocurre si y sólo si ésta está habilitada. Después de la ejecución, las interacciones con las que está enlazada se deshabilitan, y los procesos participantes en ella comienzan a trabajar de nuevo.

6.2 Definición formal

En esta sección, presentamos una formalización completa de marco presentado. Primero describiremos las suposiciones sobre las que se construye y luego presentaremos un conjunto de definiciones que constituyen el núcleo del mismo; algunas definiciones útiles se presentarán después, y serán clasificadas en dos grupos, dependiendo de si son o no implementables; por último formalizamos las propiedades *safety* que debe cumplir toda ejecución basada en interacciones multipartitas.

6.2.1 Suposiciones

Las suposiciones habituales en el contexto de los sistemas concurrentes se dividen en dos grupos: las suposiciones físicas y las lógicas. El primer grupo permite abstraernos de los detalles relativos a posibles fallos de la máquina, y el segundo en las propiedades de más alto nivel de los elementos que componen el sistema.

Para definir nuestro marco, necesitamos realizar las siguientes suposiciones físicas:

i. La máquina no falla y todo participante que está listo para ejecutar cálculo lo hace en un tiempo finito. Esto también se conoce como propiedad fundamental de progreso, y merece la pena destacar que no implica que exista una cota conocida del tiempo máximo que un proceso necesita para progresar.

ii. Las líneas de comunicación no fallan y todo evento es detectado en un tiempo finito por los participantes del sistema. De nuevo el tiempo necesario para transmitir información debe ser finito pero no exista una cota conocida del mismo.

Además, necesitamos considerar las siguientes suposiciones lógicas:

i. El comportamiento de todo participante es autónomo, y el código que éste ejecuta es independiente del resto.

ii. Las interacciones multipartitas son el único mecanismo a través del cual los participantes se pueden sincronizar e intercambiar información los unos con los otros.

iii. Existe una cota superior del tiempo que todo participantes está ejecutando cálculo local. Fíjese que la cota existe, pero no tiene por qué conocerse de antemano.

Ninguna de estas restricciones son descabelladas en en contexto de nuestro trabajo, ya que no nos centramos en la tolerancia a fallos y parece realista pensar que la tecnología actual nos permita ejecutar cálculo local en un tiempo razonable. Además, son normales en la bibliografía.

6.2.2 Núcleo de definiciones

El núcleo de nuestro framework es un conjunto de definiciones a través de las cuales podemos definir rigurosamente los conceptos presentados en la Figura §6.1. Utilizaremos el sistema de los filósofos comensales del Programa §2.1 para ilustrar algunos conceptos.

Definición 6.2.1 (Sistema) *Un sistema Σ es una tupla de la forma (P_Σ, I_Σ) en la que $P_\Sigma \neq \emptyset$ es un conjuto finito de participantes e $I_\Sigma \neq \emptyset$ es un conjunto finito de interacciones. Denotaremos a los participantes que pueden ofrecer participar en la interacción x como $\mathbb{P}(x)$ y al conjunto de interacciones en las que un participante p puede participar como $\mathbb{I}(p)$.*

En nuestro ejemplo tenemos un sistema compuesto de N participantes filósofos llamados P_i y N participantes tenedor llamados F_i ($i \in [1, N]$). Se sincronizan a través de N interacciones Get_i para coger los tenedores y N interacciones Rel_i para soltarlos. El conjunto de participantes de la interacción Get_i es $\mathbb{P}(Get_i)=\{F_{i-1}, P_i, F_i\}$, y el conjunto de interacciones en las que F_i puede participar es $\mathbb{I}(F_i)=\{Get_i, Rel_i, Get_{i+1}, Rel_{i+1}\}$.

Definición 6.2.2 (Eventos) *Un evento es un acontecimiento que induce al sistema a transitar de una configuración a otra. Una configuración es un objeto que puede ser visto como fotografía del sistema en tiempo de ejecución. En nuestro modelo, distinguimos entre eventos de ofrecimiento y eventos de sincronización. Los primeros son denotados como* $p.\chi$ *y indican que el participante* p *esta listo para participar en cualquier interacción del conjunto* χ*; el segundo se denota como el nombre de una interacción e indica que ésta ha sido seleccionada para su ejecución. Una evento de la forma* $p.\emptyset$ *indica que el participante* p *ha terminado.*

Por ejemplo, cuando el filósofo P_i ofrece participar en la interacción Get_i, un evento de la forma $P_i.\{Get_i\}$ ocurre; análogamente, cuando P_i coge sus tenedores, un evento de la forma Get_i ocurre y sincroniza la ejecución de P_i, F_i y F_{i-1} ($i \in [1, N]$).

Definición 6.2.3 (Ejecucion) *Una ejecución es una 3-tupla de la forma* (C_0, α, β)*, donde* C_0 *es una configuración inicial,* $\alpha = [C_1, C_2, C_3, \ldots]$ *es una traza maximal de configuraciones y* $\beta = [e_1, e_2, e_3, \ldots]$ *es una traza maximal de eventos. Obviamente,* α *y* β *deben tener la misma longitud. Dada una ejecución* λ*, denotaremos su traza de configuraciones como* λ_α *y su traza de eventos como* λ_β*.*

Consideremos, por ejemplo, la siguiente ejecución:

$$\lambda = (C_0, \alpha, \beta)$$
$$\alpha = [C_1 C_2 C_3 C_4 \ldots]$$
$$\beta = [P_1.\{Get_1\}\, F_5.\{Get_5, Get_1\}\, F_1.\{Get_1, Get_2\}\, Get_1 \ldots]$$

En ella, el filósofo P_1 ofrece participación en Get_1, luego el tenedor F_5 ofrece participación en Get_5 o Get_1 y el tenedor F_1 ofrece participación en Get_1 o Get_2. La interacción Get_1 se habilita en la configuración C_3 y, en este caso concreto, resulta seleccionada y la ejecución continua.

Definición 6.2.4 (Semántica) *Nos referiremos a la regla que captura la semántica que controla las transiciones entre dos configuraciones consecutivas como* $\longrightarrow_L$. *Por ejemplo,* $C_i \xrightarrow{e_{i+1}}_L C_{i+1}$ *indica que el sistema ha transitado desde la configuración* C_i *hasta la* C_{i+1} *por la ocurrencia del evento* e_{i+1}. *Así, dada una ejecución de la forma* $(C_0, [C_1, C_2, C_3, \ldots], [e_1, e_2, e_3, \ldots])$, *la escribiremos como:*

$$C_0 \xrightarrow{e_1}_L C_1 \xrightarrow{e_2}_L C_2 \xrightarrow{e_3}_L \cdots$$

Nuestros ejemplos están implementados en IP_{CORE}, así $\longrightarrow_L$ equivale a $\longrightarrow_{\text{IP}_{\text{CORE}}}$. Consulte la Referencia [53] para una descripción detallada de la semántica de IP_{CORE} o el Apéndice §B para una breve introducción.

Definición 6.2.5 (Estados de un participante) *Un participante estará esperando a un conjunto no vacío de interacciones* $\Upsilon \subseteq \mathbb{I}(p)$ *en la* i*-ésima configuración de la ejecución* λ *si y sólo si ha ofrecido participar en un subconjunto de interacciones* $\chi \supseteq \Upsilon$ *y ninguna interacción de* χ *ha sido seleccionada desde ese momento. Un participante* p *ha terminado en la configuración* i*-ésima de la ejecución* λ *si y sólo si ha ofrecido participar en un conjunto vacío de interacciones con anterioridad. Está trabajando si y sólo si no está ni esperando ni terminado.*

$$\begin{aligned}
\text{Waiting}(\lambda, p, \Upsilon, i) &\iff \exists \chi \supseteq \Upsilon, k \in [1, i] \cdot \\
&\qquad (\lambda_\beta(k) = p.\chi \wedge \nexists j \in (k, i] \cdot (\lambda_\beta(j) = x \wedge x \in \Upsilon)) \\
\text{Finished}(\lambda, p, i) &\iff \exists k \in [1, i] \cdot \lambda_\beta(k) = p.\emptyset \\
\text{Working}(\lambda, p, i) &\iff \neg\, \text{Finished}(\lambda, p, i) \wedge (\nexists \Upsilon \subseteq I_\Sigma \cdot \text{Waiting}(\lambda, p, \Upsilon, i))
\end{aligned}$$

En nuestro ejemplo, el filósofo P_j está esperando una interacción del conjunto $\{Get_j\}$ en la configuración C_i si un evento de la forma $P_j.\{Get_j\}$ ocurren antes que C_i y la interacción Get_j no ha sido seleccionada desde ese momento ($i \geq 1, j \in [1, N]$). Nunca terminan, y empiezan a trabajar inmediatamente después de ejecutar la interacción.

Definición 6.2.6 (Estados de una interacción) *La interacción* x *esta estable en la* i*-ésima configuración de la ejecución* λ *si y sólo si sus participantes han terminado o están esperando que una interacción sea ejecutada. Está habilitada si y sólo si todos sus participantes están esperando a que se ejecute ella; en otro caso, está deshabilitada.*

$$\begin{aligned}
\text{Stable}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot \\
&\qquad (\text{Finished}(\lambda, p, i) \vee \exists \Upsilon \subseteq I_\Sigma \cdot \text{Waiting}(\lambda, p, \Upsilon, i)) \\
\text{Enabled}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot \text{Waiting}(\lambda, p, \{x\}, i) \\
\text{Disabled}(\lambda, x, i) &\iff \neg\, \text{Enabled}(\lambda, x, i)
\end{aligned}$$

La interacción Get_j está habilitada en la i-ésima configuración si y sólo si todos sus participantes (P_j, F_j y F_{j-1}) han ofrecido participar en ella en esa configuración. Además, Rel_j está estable en esa configuración ya que sus participantes están esperando a que Get_j sea ejecutada. Fíjese que habilitación implica estabilización, pero no al contrario.

Definición 6.2.7 (Interacciones enlazadas) *Se dice que dos interacciones están estáticamente enlazadas si y sólo si tienen algún participante en común; y estarán dinámicamente enlazadas en la i-ésima configuración de la ejecución λ si y sólo si existe un participante que está esperando a que alguna de ellas se ejecute.*

$$\begin{aligned} \mathtt{StaLinked}(x) &= \{y \in I_\Sigma \mid \mathbb{P}(x) \cap \mathbb{P}(y) \neq \emptyset\} \\ \mathtt{DynLinked}(\lambda, x, i) &= \{y \in \mathtt{StaLinked}(x) \mid \exists p \in P_\Sigma \cdot \mathit{Waiting}(\lambda, p, \{x, y\}, i)\} \end{aligned}$$

6.2.3 Miscelánea de conjuntos implementables

En aras de la compresión, merece la pena definir varios conjuntos de indeices que pueden ayudar a definir otros conceptos con mayor facilidad. Para toda ejecución λ e interacción x, se definen los siguientes conjuntos en la i-ésima configuración:

i. El conjunto de índices i que identifican las configuraciones en las que la interacción x está habilitada (fíjese que ninguna interacción puede estar habilitada en la configuración inicial):

$$\mathtt{EnaSet}(\lambda, x, i) = \{k \in [1, i] \mid \mathtt{Enabled}(\lambda, x, k)\}$$

ii. El conjunto de índices i que identifican las configuraciones en las que la interacción x está estable:

$$\mathtt{StaSet}(\lambda, x, i) = \{k \in [1, i] \mid \mathtt{Stable}(\lambda, x, k)\}$$

iii. El conjunto de índices i que identifican las configuraciones en las que la interacción x es ejecutada:

$$\mathtt{ExeSet}(\lambda, x, i) = \{k \in [1, i] \mid \lambda_\beta(k) = x\}$$

iv. El conjunto de índices i que identifican las configuraciones en las que el participante p ofrece participar en un subconjunto de interacciones $\Upsilon \subseteq \mathbb{I}(p)$.

$$\mathtt{OffSet}(\lambda, \Upsilon, p, i) = \{k \in [1, i] \mid \lambda_\beta(k) = p.\chi \wedge \Upsilon \subseteq \chi\}$$

v. El conjunto de índices i que identifican las configuraciones en las que p está esperando a que la interacción x sea ejecutada:

$$WaitSet(\lambda, x, p, i) = \{k \in [1, i] \mid Waiting(\lambda, p, \{x\}, i)\}$$

6.2.4 Miscelánea de predicados no implementables

En la sección anterior, definimos algunos conjuntos de utilidad. En algunas ocasiones, también es necesario conocer si estos son finitos o no ya que tiene bastante importancia teórica. Los siguientes predicados pueden ser útiles para definir algunas propiedades, aunque claramente no son implementables, por lo que deben ser utilizadas con precaución.

$$\begin{aligned}
InfEna(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |EnaSet(\lambda, x, |\lambda|)| < k \\
InfSta(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |StaSet(\lambda, x, |\lambda|)| < k \\
InfExe(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |ExeSet(\lambda, x, |\lambda|)| < k \\
InfOff(\lambda, x, p) &\iff \nexists k \in \mathbb{N} \cdot |OffSet(\lambda, \{x\}, p, |\lambda|)| < k \\
InfWait(\lambda, x, p) &\iff \nexists k \in \mathbb{N} \cdot |WaitSet(\lambda, x, p, |\lambda|)| < k
\end{aligned}$$

6.2.5 Propiedades *safety*

Obviamente, cualquier combinación de eventos y configuraciones no es correcta. Las propiedades *safety* que toda ejecución basada en interacciones multipartitas debe cumplir son definidas a continuación, donde λ es cualquier ejecución generada por un sistema de la forma $\Sigma = (I_\Sigma, P_\Sigma)$.

Definición 6.2.8 (Ofrecimientos seguros) *Si un participante* p *ofrece participar en una interacción del conjunto* χ *en la* i*-ésima configuración, entonces ese participante no puede ejecutar ninguna acción hasta que una de estas interacciones sea ejecutada; si ninguna de ellas es ejecutada, entonces* p *permanecerá bloqueado para siempre.*

$$\begin{aligned}
SafeOffers(\lambda) \iff \forall i > 0, p \in P_\Sigma, \chi \subseteq I_\Sigma \cdot \lambda_\beta(i) = p.\chi \Rightarrow \\
((\nexists j > i, x \in \chi \cdot \lambda_\beta(j) = x \wedge \nexists k > i, \Upsilon \subseteq I_\Sigma \cdot \lambda_\beta(k) = p.\Upsilon) \vee \\
(\exists j > i, x \in \chi \cdot \lambda_\beta(j) = x \wedge \nexists k \in (i, j), \Upsilon \subseteq I_\Sigma \cdot \lambda_\beta(k) = p.\Upsilon))
\end{aligned}$$

Definición 6.2.9 (Terminaciones seguras) *Si un participante p ofrece participar en un conjunto vacío de interacciones en la i-ésima configuración, entonces ese participante no podrá ejecutar nunca más ninguna acción, y las interacciones en las que él puede participar ya no se ejecutarán más.*

$$\begin{aligned}&\mathtt{SafeTerms}(\lambda) \iff \forall i > 0, p \in P_\Sigma \cdot \lambda_\beta(i) = p.\emptyset \Rightarrow \\ &(\nexists j > i, \chi \subseteq I_\Sigma \cdot \lambda_\beta(j) = p.\chi \wedge \nexists j > i, x \in \mathbb{I}(p) \cdot \lambda_\beta(j) = x)\end{aligned}$$

Definición 6.2.10 (Sincronizaciones seguras) *Si una interacción x se ejecuta en la i-ésima configuración, entonces debe estar habilitada en esa configuración, las interacciones que están estáticamente enlazadas con ellas se deshabilitan en la siguiente configuración (fíjese que este conjunto incluye a x) y todos los participantes que se han sincronizado a través de x se ponen a trabajar en la siguiente configuración.*

$$\begin{aligned}&\mathtt{SafeSyncs}(\lambda) \iff \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \\ &(\mathtt{Enabled}(\lambda, x, i-1) \wedge \\ &\forall y \in \mathtt{StaLinked}(x) \cdot (\mathtt{Disabled}(\lambda, y, i) \wedge \neg\, \mathtt{Stable}(\lambda, y, i)) \wedge \\ &\forall p \in \mathbb{P}(x) \cdot \mathtt{Working}(\lambda, p, i))\end{aligned}$$

Fíjese que las definiciones anteriores no son mínimas ya que son redundantes, es decir, $\neg\, \mathtt{Stable}(\lambda, y, i)$ implica $\mathtt{Disabled}(\lambda, y, i)$, pero hemos preferido no minimizar ya que pensamos que así la formulación es más intuitiva.

Como ejemplo, considere una ejecución cuya traza de eventos es de la forma $[F_5.\{Get_5\}\, F_5.\{Get_1\}\, Get_1\, F_2.\emptyset\, F_2.\{Get_2\} \ldots]$. No cumple ninguna propiedad *safety* ya que F_5 ofrece participación en dos conjuntos distintos sin ejecutar ninguna interacción del primero de ellos, Get_1 es ejecutada sin llegar a esta habilitada y F_2 ofrece participación en Get_2 justo después de su finalización.

6.3 Beneficios

En esta sección, mostramos cómo usar este marco para definir algunas propiedades *safety* y *liveness* usuales, algunas nociones y métricas de ejecución, lo que demuestra que es lo suficientemente genérico para ser utilizado en otros contextos distintos a los de esta tesis doctoral. Sin embargo, no es universal ya que, por ejemplo, no puede definir nociones basadas en el oráculo. Como quiera que sea, no es nuestro objetivo producir un marco que sea universal

En los siguiente ejemplos, suponemos que λ es una ejecución de un sistema $\Sigma = (P_\Sigma, I_\Sigma)$. A menos que se diga lo contrario, asumiremos que λ es infinita.

Ejemplo 6.3.1 (Interbloqueo) *Sea λ una ejecución finita, se tendrá un interbloqueo si sólo si todas las interacciones I_Σ están estables en la última ejecución, pero ninguna de ellas está habilitada.*

$$\mathtt{Deadlock}(\lambda) \iff \forall x \in I_\Sigma \cdot (\mathtt{Stable}(\lambda, x, |\lambda|) \wedge \neg\, \mathtt{Enabled}(\lambda, x, |\lambda|))$$

Ejemplo 6.3.2 (Conspiración) *Se dice que una ejecución λ es conspiratoria si existe alguna interacción de I_Σ cuyos participantes están listos para ejecutarla en infinitas ocasiones pero sólo se habilita en un número finito de ocasiones. Fíjese que está caracterización no distingue entre las conspiraciones que son inherentes de las no inherentes ya que se requeriría analizar el código fuente del sistema que genera la ejecución λ.*

$$\mathtt{Conspiratorial}(\lambda) \iff \exists x \in I_\Sigma \cdot (\forall p \in \mathbb{P}(x) \cdot \mathtt{InfWait}(\lambda, x, p) \wedge \neg\, \mathtt{InfEna}(\lambda, x))$$

Ejemplo 6.3.3 (Unconditional fairness) *Se dice que la ejecución λ es de nivel unconditional si y sólo si toda interacción que está continuamente habilitada es ejecutada en infinitas ocasiones.*

$$\mathtt{UF}(\lambda) \iff \forall x \in I_\Sigma \cdot (\forall i \geq 1 \cdot \mathtt{Enabled}(\lambda, x, i)) \Rightarrow \mathtt{InfExe}(\lambda, x)$$

Ejemplo 6.3.4 (Weak fairness) *Se dice que una ejecución λ es de nivel weak si y sólo si toda interacción está habilitada desde un punto en adelante se ejecuta en infinitas ocasiones.*

$$\mathtt{WF}(\lambda) \iff \forall x \in I_\Sigma \cdot (\exists i \geq 1 \cdot \forall j \geq i \cdot \mathtt{Enabled}(\lambda, x, j)) \Rightarrow \mathtt{InfExe}(\lambda, x)$$

Fíjese que los requisitos de UF y WF nunca puede ser satisfechos de acuerdo a la semántica de nuestro marco. El motivo radica en que una ejecución correcta debe satisfacer la propiedad SafeSyncs, lo que implica que después de que una interacción se ejecuta ésta se deshabilita. Véase Sección §3.3.2 para más detalles.

Ejemplo 6.3.5 (Strong fairness) *Se dice que la ejecución λ es de nivel strong si y sólo si toda interacción que se habilita en infinitas ocasiones se ejecuta en infinitas ocasiones.*

$$\mathtt{SF}(\lambda) \iff \forall x \in I_\Sigma \cdot (\mathtt{InfEna}(\lambda, x) \Rightarrow \mathtt{InfExe}(\lambda, x))$$

Ejemplo 6.3.6 (Índice de rechazo) *Sea λ una ejecución finita, el índice de rechazo de una interacción x se define como uno menos el cociente entre el número de veces que x se ejecuta y el número de veces que se habilita.*

$$\mathtt{RejectionRatio}(\lambda, x) = 1 - \frac{|\ \mathtt{ExeSet}(\lambda, x, |\lambda|)\ |}{|\ \mathtt{EnaSet}(\lambda, x, |\lambda|)\ |}$$

Fíjese que si la índice de rechazo tiende a 0, se tiene que la interacción x ha sido ejecutada la mayoría de la veces que estaba habilitada; por el contrario, si tiende a 1, se tiene que se ha habilitado muchas veces pero se se ha ejecutado en pocas ocasiones.

Ejemplo 6.3.7 (Grado de habilitación) *Sea λ una ejecución finita, el grado de habilitación de la interacción x se define como el cociente entre el número de veces que la interacción se ejecuta y el número medio de veces que sus participantes la ofrecen.*

$$\mathtt{EnablementDegree}(\lambda, x) = \frac{|\ \mathtt{ExeSet}(\lambda, x, |\lambda|)\ |}{\left|\bigcup_{p \in\, \mathbb{P}(x)} \mathtt{OffSet}(\lambda, \{x\}, p, |\lambda|)\right| / |\,\mathbb{P}(x)|}$$

En esta expresión, el numerado cuenta el número de veces que la interacción x se ha ejecutado y el denominador cuenta el número medio de veces que sus participantes la ofrecen. Así, tiene que encontrarse en el intervalo $[0, 1]$, a saber: si tiende a 1 se tienen que los participantes tienen que insistir poco para participar en x; en caso contrario, se tiene que insistir mucho para poder ejecutar x.

Ejemplo 6.3.8 (Grado en enlace estático) *El grado de enlace estático de un participante p se define como el cociente entre el número de sus interacciones que están enlazadas con otras interacciones y el máximo número de interacciones en las que éste puede participar.*

$$\mathtt{StaLinkageDegree}(p) = \frac{\left|\bigcup_{q \in P_\Sigma \setminus \{p\}} \mathbb{I}(p) \cap \mathbb{I}(q)\right|}{|\,\mathbb{I}(p)|}$$

Si esta expresión tiende a 1 se tiene que el participante p comparte la mayoría de sus interacciones con otros procesos; por el contrario, si tiende a 0 se tiene que comparte pocas interacciones.

Capítulo 7

Una nueva noción de selección justa

Equal opportunity means that everyone will have a fair chance at being incompetent.

Laurence J. Peter, 1919–1988
Educator and writter

La finitud justa y las conspiraciones son los problemas a resolver en esta tesis. En este capítulo, presentamos la noción de selección justa que hemos desarrollado para resolver estos problemas. En la Sección §7.1, presentamos una idea general para ayudar al lector a entender la definición de la noción en la Sección §7.2; En la Sección §7.3, demostramos que nuestra noción resuelve los problemas mencionados y que preserva su nivel strong en las ejecuciones infinitas.

7.1 Introducción

La selección justa k-*conspiracy-free* es una nueva noción cuyos requisitos no son cumplidos trivialmente por cualquier ejecución y que, además, evita que las ejecuciones contengan conspiraciones. La idea detrás ella es identificar las situaciones de conspiración potencial y poner una cota en el número de veces que se permite que éstas ocurra. Fíjese que las conspiraciones sólo ocurren cuando una ejecución es infinita, pero es posible detectar las configuraciones en las que una conspiración se empieza a gestar. En pocas palabras, esto ocurre cuando las interacciones se ejecutan estando enlazadas con otras que aún no están estables. Así, las conspiraciones pueden ser solucionadas si la ejecución de estas interacciones es retrasada.

Nuestro primer intento de desarrollar una noción capaz de resolver los problemas de la finitud justa y las conspiraciones fue la noción llamada *strong* k-*fairness* [115]. En ella, considerábamos sólo las interacciones que estaban enlazadas en tiempo de ejecución, lo que permitía aliviar las conspiraciones pero no resolverlas totalmente (véase Apéndice §A). por contra, la noción k-*conspiracy-free* considera las interacciones enlazadas estáticamente lo que permite eliminar las conspiraciones totalmente.

La restricción que esta noción impone a los sistemas para poder ser aplicada es muy ligera, a saber: las suposiciones físicas y lógicas presentadas en la Sección §6.2.1 deber ser satisfechas por la máquina y el compilador y los sistemas deben estar totalmente conectados. Intuitivamente, un sistema está totalmente conectado cuando cualquier par de participantes se puede comunicar directamente a través de una interacción o indirectamente a través de varias interacciones, es decir, el grafo que conecta interacciones y participantes es bipartito y conexo. La Figura §7.1 muestra varios sistemas conectados. Fíjese que son disjuntos ya que no comparten ninguna interacción, así, es imposible para un participante de un sistema totalmente conectado comunicarse con otro participante de otro sistema totalmente conectado, lo que implica que las ejecuciones de unos y otros son independientes.

7.2 Selección justa k-*conspiracy-free*

De acuerdo a la definición de Attie *et al.* [10], una conspiración contra una interacción ocurre si y sólo si sus procesos participantes la ofrecen en infinitas ocasiones, pero las interacción sólo se habilita un número finito de veces porque alguno de sus participantes no confirma la habilitación.

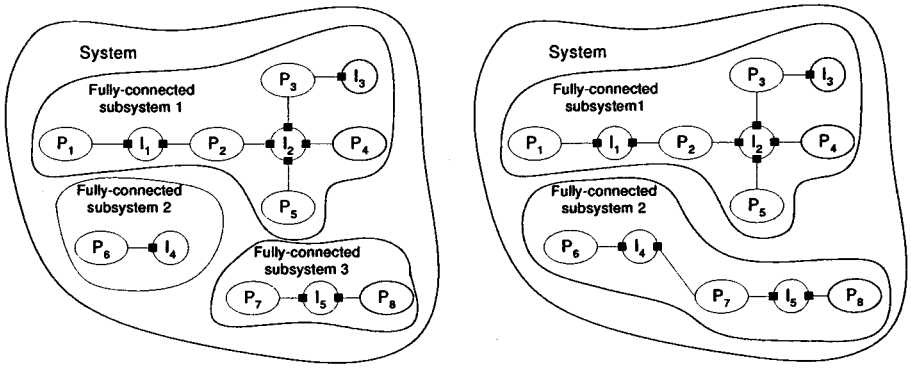


Figura 7.1: *Sistemas totalmente conectados.*

Por lo tanto, podemos prevenir las conspiraciones asegurando que cada vez que una interacción se ejecuta, todas las interacciones con las que está enlazada están estables, dándoles una oportunidad de que habiliten. Formalmente, cuando la siguiente formula se satisface, la ejecución λ no contiene conspiraciones:

$$\forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \neg \, \mathrm{PotConsp}(\lambda, x, i) \quad (7.1)$$

donde $\mathrm{PotConsp}(\lambda, x, i)$ se satisface cuando existe alguna interacción enlazada con x que no está estable en la configuración i-ésima. Formalmente,

$$\mathrm{PotConsp}(\lambda, x, i) \iff \exists y \in \mathrm{StaLinked}(x) \cdot \neg \, \mathrm{Stable}(\lambda, y, i)$$

Fíjese que la Fórmula §7.1 previene que ocurran las conspiraciones, pero no evita que una interacción que se habilita en infinitas ocasiones sea siempre rechazada. Se podría pensar que endurecer la formula como sigue es suficiente:

$$\mathrm{SF}(\lambda) \wedge \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \neg \, \mathrm{PotConsp}(\lambda, x, i) \quad (7.2)$$

donde $\mathrm{SF}(\lambda)$ fue definido en el Ejemplo §6.3.5 y se satisface cuando λ es de nivel *strong*. Sin embargo, esto no resolvería el problema de la finitud justa ya que SF se satisface trivialmente para cualquier ejecución finita. Una idea mejor de prevenir que una interacción que se habilita en infinitas ocasiones sea siempre rechaza es considerando sus edad, de forma que una interacción resulta seleccionada cuando es las más vieja de entre las interacciones que

están habilitadas. La edad de una interacción puede ser interpretada como el tiempo que transcurre desde las última vez que se ejecutó, pero dado que no estamos teniendo en cuenta restricciones de tiempo real, es suficiente definirla como el número de configuraciones que pasan desde la última vez que fue ejecutada.

En consecuencia, la condición de la Fórmula §7.1 puede ser redefinida de la siguiente forma:

$$\forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow (\neg \mathrm{PotConsp}(\lambda, x, i) \wedge \mathrm{OldestEna}(\lambda, x, i)) \quad (7.3)$$

donde OldestEna se define:

$$\mathrm{OldestEna}(\lambda, x, i) \iff \forall y \in I_\Sigma \setminus \{x\} \cdot \mathrm{Enabled}(\lambda, y, i) \Rightarrow \mathrm{Age}(\lambda, y, i) < \mathrm{Age}(\lambda, x, i)$$

$$\mathrm{Age}(\lambda, x, i) = \begin{cases} i - \max \mathrm{ExeSet}(\lambda, x, i-1) & \text{if } \mathrm{ExeSet}(\lambda, x, i-1) \neq \emptyset \\ i + \Phi(x) & \text{otherwise} \end{cases}$$

La edad inicial de que una interacción tiene es un problema importante a resolver. Intuitivamente, parece ser infinito, pero esto nos impediría comparar las edades iniciales de las interacciones a fin de encontrar la más vieja. Para resolver esto, hemos supuesto que existe un orden total arbitrario $\prec$ sobre las interacciones que nos ayuda a definir el mapa Φ de la siguiente forma:

$$\Phi(x) = |\{y \in I_\Sigma \mid y \prec x\}| + 1$$

Fíjese que $\Phi(x) \in [1, |I_\Sigma|]$ para todo $x \in I_\Sigma$, y esto garantiza que inicialmente no existen dos interacciones distintas con la misma edad. Como demostraremos después, esta edad inicial aleatorio no afecta a la corrección de la noción.

La condición de la Fórmula §7.3 previene que las conspiraciones ocurran y garantiza que toda interacción que se habilita en infinitas ocasiones se ejecute en infinitas ocasiones, pero tiene un punto débil: requiere que todo el conjunto de interacciones enlazas con la interacción seleccionada esté estable; cuando esto ocurre naturalmente el criterio es bueno, pero, de lo contrario, se fuerza a que la ejecución de algunas interacciones sea retrasada hasta que se estabilicen sus enlazadas. Esto puede provocar una ineficiencia que puede ser resuelta introduciendo un umbral de conspiraciones que acote el número máximo de veces que una interacción puede ser ejecuta en situación de conspiración potencial. Este umbral nos puede ayuda a ajustar el criterio a las características del sistema objeto de estudio, a saber: si es cero la velocidad a

la que un grupo de interacciones enlazadas progresa en determinada por la interacción más lenta; por el contrario, si es grande, las interacciones puede ser seleccionada tan pronto se habilitan.

Estas observaciones nos conducen a formular nuestra noción de selección justa como sigue:

Definición 7.2.1 (Selección justa k-*conspiracy-free*) *Sea λ una ejecución de un sistema completamente conectado de la forma $\Sigma = (P_\Sigma, I_\Sigma)$. Se dice que λ es de nivel* k*-conspiracy-free si y sólo si las interacciones son seleccionadas cuando no están en situación de conspiración potencial y son las más viejas o cuando están en situación de conspiración potencial pero no han sido seleccionada más de* k *veces en esta situación.*

$$\mathcal{CFF}_k(\lambda) \iff \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow$$
$$(\neg \mathrm{PotConsp}(\lambda, x, i) \wedge \mathrm{OldestEna}(\lambda, x, i) \vee \sum_{j=0}^{i} \mathrm{PotConsp}(\lambda, x, j) < k)$$

Merece la pena destacar que la implementación de k-*conspiracy-free* no equivale a una propiedad *void* ya que no toda ejecución finita satisface trivialmente sus requisitos. Toda ejecución en la que una interacción es ejecutada más de k veces en presencia de interacciones enlazadas no estables puede ser detectada en un tiempo finito. Así, es posible para un planificador deteminar cuando una ejecución viola o no el criterio.

7.3 Corrección

Nuestro objetivo en esta sección es demostrar que la selección k-*conspiracy-free* resuelve los problemas de la finitud justa y las conspiraciones y que preserva el nivel *strong*.

Lema 7.3.1 *Sea λ una ejecución infinita* k*-conspiracy-free para algún $k \in \mathbb{N}$ y z una interacción que es ejecutada en un número finito de veces. Existe una configuración $C_{r_1^*}$ en λ a partir de la que cada vez que se ejecuta una interacción enlazada con z, z está estable.*

Prueba Fíjese que el conjunto de interacciones puede ser particionado en dos conjuntos disjuntos I y F tal que I es el subconjunto de interacciones que se

ejecutan en infinitas ocasiones y F el subconjunto de interacciones que se ejecutan un número finito de veces. Está claro que $I \neq \emptyset$ ya que de otra forma λ sería finita y que $F \neq \emptyset$ ya que z se ejecuta un número finito de veces por la hipótesis de partida. Formalmente,

$$I_\Sigma = I \cup F, I \cap F = \emptyset$$
$$\forall x \in I \cdot \mathrm{InfExe}(\lambda, x)$$
$$\forall x \in F \cdot \neg\, \mathrm{InfExe}(\lambda, x)$$

Además, existe un índice p tal que ninguna interacción en F puede ejecutarse desde la configuración $\lambda_\alpha(p)$ en adelante. Esto implica que z no puede habilitarse después de esa configuración o nuestra hipótesis de partida sería violada.

El conjunto I también puede ser particionada en dos subconjuntos L y N de forma que el primero contenga las interacciones enlazadas con z y el segundo aquellas que no lo están. Fíjese que $L \neq \emptyset$ ya que, de lo contrario, el sistema no estaría completamente conectado y k-*conspiracy-free* no podría ser aplicado. Formalmente,

$$I = L \cup N, L \cap N = \emptyset$$
$$\forall x \in L \cdot x \in \mathrm{StaLinked}(z)$$
$$\forall x \in N \cdot x \notin \mathrm{StaLinked}(z)$$

L también puede ser particionado en dos subconjuntos, por un lado las interacciones que nunca exceden el umbral de conspiraciones k y por otro el resto. Formalmente,

$$L = L_N \cup L_E, L_N \cap L_E = \emptyset$$
$$\forall x \in L_N, q \geq p \cdot \sum_{j=0}^{q} \mathrm{PotConsp}(\lambda, x, j) < k$$
$$\forall x \in L_E \cdot \exists q \geq p \cdot \sum_{j=0}^{q} \mathrm{PotConsp}(\lambda, x, j) = k$$

Ahora, podemos estudiar dos casos:

i. Para todo $x \in L_N$, existe un índice $p_x \geq p$ de forma que a partir de la configuración $\lambda_\alpha(p_x)$ en adelante, x siempre se ejecuta en situación de no conspiración y siendo la más vieja de entra las habilitadas. Sea p_1 igual a $\max_{x \in L_N}\{p_x\}$.

ii. Para todo $x \in L_E$, existe un índice p_2 tal que $q \geq p_2$, $\sum_{j=0}^{q} \text{PotConsp}(\lambda, x, j) = k$.
Esto implica que cada vez que x se ejecuta no está en situación de conspiración y es la más vieja, lo que conlleva que z esta estable.

Por consiguiente, cualquier índice $r_1^* \geq \max\{p_1, p_2\}$ satisface el lema. □

Lema 7.3.2 *Sea* λ *una ejecución infinita* k*-conspiracy-free para algún* $k \in \mathbb{N}$ *y* z *una interacción que se ejecuta un número finito de veces. Existe una configuración* $C_{r_2^*}$ *en* λ *a partir de la que cada vez que una interacción que no está enlazada con* z *se ejecuta, no está en situación de conspiración y es la más vieja de entre las interacciones habilitadas.*

Prueba De la demostración del Lema §7.3.1, se tiene que existe un conjunto de de interacciones N de las interacciones que son ejecutas en infinitas ocasiones y que no están enlazadas con z. N puede ser particionada en dos subconjuntos N_N y N_E tal que

$$N = N_N \cup N_E, N_N \cap N_E = \emptyset$$
$$\forall x \in N_N, q \geq p \cdot \sum_{j=0}^{q} \text{PotConsp}(\lambda, x, j) < k$$
$$\forall x \in N_E \cdot \exists q \geq p \cdot \sum_{j=0}^{q} \text{PotConsp}(\lambda, x, j) = k$$

Ahora, vamos a diferencia nuevamente dos casos:

i. Para todo $x \in N_N$, se tiene que existe un índice $p_x \geq p$ de forma que a partir de la configuración $\lambda_\alpha(p_x)$ en adelante, el contador de situaciones de conspiración potencial no cambia, o de lo contrario x pertenecería a N_E. Sea p_1 igual a $\max_{x \in N_N}\{p_x\}$.

ii. Para todo $x \in N_E$, existe un índice p_2 tal que para todo $q \geq p_2$ se tiene que $\sum_{j=0}^{q} \text{PotConsp}(\lambda, x, j) = k$.

En consecuencia, cualquier índice $r_2^* \geq \max\{p_1, p_2\}$ satisface el lema ya que ninguna interacción en N se ejecuta en situación de conspiración potencial desde esta configuración en adelante. □

Teorema 7.3.1 (Finitud justa) *Dado un* $k \in \mathbb{N}$, *no toda ejecución finita satisface el criterio* k-*conspiracy-free.*

Prueba Demostrar este teorema es simple ya que basa con encontrar una ejecución que no sea k-*conspiracy-free* para algún $k \in \mathbb{N}$. Consideremos, por ejemplo, el siguiente programa, que está completamente conectado:

S :: [P || Q || R], **where**
P :: *[A⟨⟩ → **skip**];
Q :: *[A⟨⟩ → **skip** [] B⟨⟩ → **skip**];
R :: *[B⟨⟩ → **skip**].

Sea λ cualquiera de las posibles ejecución cuya traza de eventos es de la forma $[P.\{A\}\,Q.\{A,B\}\,A]^n \frown [R.\{B\}\,Q.\{A,B\}\,B\,P.\{A\}\,Q.\{A,B\}\,A]^m$. No es k-*conspiracy-free* para ningún $k < n$ ya que la interacción A se ejecuta más de k cuando la interacción B (enlazada con ella) no está estable. □

Teorema 7.3.2 (Conspiraciones) *Ninguna ejecución que satisface los requisitos de* k-*conspiracy-free para algún* $k \in \mathbb{N}$, *puede adolecer de otras conspiraciones que no sean inherentes.*

Prueba En otras palabras, tenemos que demostrar que si existe una ejecución λ tal que existe una interacción z cuyos participantes la esperan en infinitas ocasiones pero ésta sólo se habilita un número finito de veces y que satiface los requisitos de k-*conspiracy-free* para algún k, entonces la conspiración contra z tiene que ser inherente.

De la prueba del Lema §7.3.1, se tiene que toda interacción en L que es ejecutada a partir de la configuración r_1^*-ésima, z tiene que estar estable. Por consiguiente, la única posibilidad para que esta interacción no se habilite es que exista una conspiración inherente contra ella que prohíba que sus participantes la ofrezcan al mismo tiempo en infinitas ocasiones. Fíjese que cada vez que una interacción de N se ejecuta a partir de la configuración r_1^*-ésima, z no tiene que estar estable, pero estas interacciones no pueden conspirar contra ella ya que no están enlazadas. □

Teorema 7.3.3 (Strong fairness) *Sea* λ *una ejecución. Si satisface los requisitos de* k-*conspiracy-free, entonces también satisface los requisitos del nivel strong.*

Prueba Demostraremos este teorema por reducción al absurdo, ya que llegaremos a una contradicción si suponemos que existe una interacción z que

se habilita en infinitas ocasiones pero que sólo se ejecuta un número finito de veces.

De las demostraciones de los Lemas §7.3.1 y §7.3.2, se tiene que para todo índice $r \geq \max\{r_1^*, r_2^*\}$, las interacciones que están enlazadas estáticamente con z se ejecutan cuando s se estabiliza, y el resto de interacciones se ejecutan en situación de no conspiración. Consideremos un índice $r^* \geq r$ tal que toda interacción que se ejecuta de la configuración $\lambda_\alpha(r^*)$ en adelante ya ha sido ejecutada al menos una vez. Este índice tiene que existir o la ejecución sería finita. En otras palabras, r^* tiene que satisfacer que

$$\forall i \geq r^*, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow (x \in I \wedge \exists j < i \cdot \lambda_\beta(j) = x)$$

Sea $p \geq r^*$ cualquiera de los infinitos índices que hacen referencia a una configuración en la que z está habilitada. Se pueden distinguir los siguientes casos:

i. Si z nunca ha sido ejecutada antes de la configuración $\lambda_\alpha(p)$, se tiene que

$$-\Phi(z) < \max \mathrm{ExeSet}(\lambda, x, p) \text{ for all } x \in I,$$

ya que $\Phi(x) \in [1, |I_\Sigma|]$ para todo $x \in I_\Sigma$ y $\mathrm{ExeSet}(\lambda, x, p) \geq 1$ porque toda interacción que se ejecuta a partir de la configuración $\lambda_\alpha(p^*)$ ha sido ejecuta una vez con anterioridad. Por consiguiente, se tiene que

$$\begin{aligned} &p - -\Phi(z) > p - \max \mathrm{ExeSet}(\lambda, x, p), \text{ then} \\ &p + \Phi(z) > p - \max \mathrm{ExeSet}(\lambda, x, p), \text{ then} \\ &\mathrm{Age}(\lambda, z, p) > \mathrm{Age}(\lambda, x, p) \end{aligned}$$

Sea x la interacción que fue ejecutada en la p-ésima configuración. Si estaba enlazada con z, entonces esto es una contradicción ya que z estaba estable según el Lema §7.3.1 y no era la más vieja; de lo contrario, el Lema §7.3.2 garantiza que x ha tenido que ser ejecutada en una situación en la que no era conspiratoria y era la más vieja, lo que contradice la hipótesis de partida de nuevo.

ii. Si z fue ejecutada antes de que se habilitará en la configuración $\lambda_\alpha(p)$, se tiene que

$$\forall x \in X_I \cdot \mathrm{Age}(\lambda, x, p) < \mathrm{Age}(\lambda, z, p)$$

ya que toda interacción de I debe haber sido ejecutada al menos una vez después de que z fuese ejecutada por última vez, es decir, z es más vieja que el resto de las interacciones, lo que contradice la decisión de ejecutar otra interacción según los Lemas §7.3.1 y §7.3.2.

Estas contradicciones concluyen la prueba. □

Capítulo 8

Implementación del marco de trabajo y de la noción

If it's your idea, you get to implement it.

Leland E. Modesitt, Jr., 1943–
Science fiction and fantasy writer

Apesar de que nuestro marco teórico es muy expresivo, no sería práctico a menos que no proporcionáramos una implementación finita. En este capítulo, presentamos nuestros resultados y los aplicamos a la implementación de un planificador k-conspiracy-free. *En la Sección §8.1, hacemos una breve introducción al capítulo; en la Sección §8.2 describimos la implementación del marco y demostramos que ésta es correcta; en la Sección §8.3, describimos cómo implementar* k-conspiracy-free *sobre la implementación previa.*

8.1 Introducción

Nuestro marco de trabajo es muy expresivo, y permite describir propiedades que dependen una de ejecución parcial de un programa, así como, propiedades que dependo de la ejecución completa. Obviamente, las segundas no son implementables en general ya que requieren conocimiento de futuro, pero las primeras puede ser eficientemente implementadas manteniendo incrementalmente un conjunto de estructuras de datos y almacenado información sobre eventos pasados.

Describimos la implementación haciendo uso de reglas de transición de Plotkin, ya que está probado que esta técnica es simple y potente [107]. Fíjese que nuestro objetivo es demostrar que el marco de trabajo puede ser implementado utilizando maquinas de estados finitas, sin tener en cuenta las muchas optimaciones posibles. Sólo destacaremos algunos de los aspectos que pueden optimar la implementación, pero no entraremos en los detalles ya que estos se salen del alcance de esta tesis. En la Referencia [113], detallamos una implementación $\jmath^\sharp$ que incluye bastante optimaciones.

8.2 Implementación del marco

En las siguientes secciones, primero describimos las configuraciones que implementan el marco y cómo éstas son actualizadas cuando ocurren eventos de ofrecimiento, sincronización y terminación. Utilizaremos `letras de máquina de escribir` para distinguir entre las funciones o predicados teóricos de sus respectivas implementaciones. Después, demostraremos que esta implementación es correcta y que preserva las propiedades *safety* que caracterizan a una ejecución basada en interacciones multipartitas. Por último, demostraremos que no es necesario asumir ningún tipo de justicia en la implementación.

8.2.1 Configuraciones

Necesitaremos configuraciones de la forma (C, φ, ϑ), a saber: C es una configuración $\mathrm{IP}_{\mathrm{CORE}}$ cuya estructura dependa de la semántica de este lenguaje [53]; φ es un mapa de ofrecimientos que relaciona a cada interacción el subconjunto de participantes que actualmente la están ofreciendo, es decir, $\varphi\colon \mathrm{I}_\Sigma \to 2^{\mathrm{P}_\Sigma}$, $\mathrm{dom}\,\varphi = \mathrm{I}_\Sigma$; ϑ es el conjunto de participantes que han finalizado su ejecución, es decir, $\vartheta \subseteq 2^{\mathrm{P}_\Sigma}$.

Merece la pena destacar que nuestras configuraciones no almacenan ninguna información sobre qué interacciones se han ejecutado o las configuraciones en las que habilitan. Así, no implementamos la miscelánea de conjuntos de la Sección §6.2.3 ya que estos requieren mantener unas estructuras cuyo tamaño se incrementaría de forma monótona conforme el tamaño de la ejecución crece. Como se dijo, estos conjunto son definidos en areas de la compresión, pero no son estrictamente necesarios.

Utilizaremos las siguientes funciones para actualizar estas configuraciones:

$$\begin{aligned}
\texttt{AddOffer}(\varphi, p, \chi) &= \{x \mapsto \varphi(x) \mid x \in I_\Sigma \setminus \chi\} \cup \{x \mapsto \varphi(x) \cup \{p\} \mid x \in \chi\} \\
\texttt{RemoveOffer}(\varphi, x) &= \{x \mapsto \varphi(x) \setminus \mathbb{P}(x) \mid x \in \operatorname{dom} \varphi\} \\
\texttt{AddFinished}(\vartheta, p) &= \vartheta \cup \{p\}
\end{aligned}$$

$\texttt{AddOffer}(\varphi, p, \chi)$ se utiliza cada vez que el participante p ofrece participación en un conjunto no vacío de interacciones χ, y devuelve un nuevo mapa en el que p se añade al conjunto de participantes que están esperando a las interacciones de χ; por contra, $\texttt{RemoveOffer}(\varphi, x)$ se utiliza cada vez que una interacción es seleccionada, y elimina sus participantes de φ; $\texttt{AddFinished}(\vartheta, p)$ se utiliza cada vez que un participante termina para añadirlo a ϑ.

8.2.2 Predicados y funciones

Los predicados y funciones de una marco de trabajo, excepto para los conjuntos misceláneos, son fácilmente mapeados a sus implementaciones utilizando la información contenida en una configuración. Sea (C, φ, ϑ) una configuración del marco de trabajo, entonces:

$$\begin{aligned}
\texttt{Waiting}(\varphi, p, \Upsilon) &\iff \forall x \in \Upsilon \cdot p \in \varphi(x) \\
\texttt{Finished}(\vartheta, p) &\iff p \in \vartheta \\
\texttt{Working}(\varphi, \vartheta, p) &\iff \neg \texttt{Finished}(\vartheta, p) \wedge \nexists x \in I_\Sigma \cdot \texttt{Waiting}(\varphi, p, \{x\}) \\
\texttt{Stable}(\varphi, \vartheta, x) &\iff \forall p \in \mathbb{P}(x) \cdot \\
&\qquad (\texttt{Finished}(\vartheta, p) \vee \exists y \in I_\Sigma \cdot \texttt{Waiting}(\varphi, p, \{y\})) \\
\texttt{Enabled}(\varphi, x) &\iff \varphi(x) = \mathbb{P}(x) \\
\texttt{Disabled}(\varphi, x) &\iff \neg \texttt{Enabled}(\varphi, x) \\
\texttt{StaLinked}(x) &= \{y \in I_\Sigma \mid \mathbb{P}(y) \cap \mathbb{P}(x) \neq \emptyset\} \\
\texttt{DynLinked}(\varphi, x) &= \{y \in \texttt{StaLinked}(x) \mid \varphi(y) \cap \varphi(x) \neq \emptyset\}
\end{aligned}$$

Fíjese que `Working` y `Stable` requieren analizar todo el conjunto de interacciones para determinar si un proceso esta esperando o no a alguna de ellas. En aras de la eficiencia, la implementación actual evita examinar todo este conjunto secuencialmente manteniendo un mapa que asocia a cada proceso el conjunto de interacciones que está esperando. Por ejemplo, si este mapa es de la forma $\omega: P_\Sigma \to 2^{I_\Sigma}$, entonces `Working` puede ser reformulado como sigue:

$$\texttt{Working}(\omega, \vartheta, p) \iff \neg\texttt{Finished}(\vartheta, p) \wedge \omega(p) = \emptyset$$

Análogamente, la evaluación de `StaLinked` puede ser mejorada manteniendo un caché con los conjuntos de interacciones estáticamente enlazadas cuando son calculadas por primera vez. Obviamente, estas decisiones de diseño dependen completamente de las estructuras de datos utilizadas para implementar los mapas y el tamaño de I_Σ. Este es el motivo por el que no entraremos en más detalles, aunque estos pueden ser encontrados en la Referencia [113].

8.2.3 Semántica

La semántica de nuestro marco de trabajo permite monitorizar la ejecución de un programa almacenando toda la información que necesitemos. La configuración inicial es de la forma (φ_0, ϑ_0), donde $\varphi_0(x) = \emptyset$ para todo $x \in I_\Sigma$ y $\vartheta_0 = \emptyset$. Desde esta configuración en adelante, el sistema es controlado por las reglas de ofrecimiento, terminación y sincronización que a continuación presentamos.

La regla de ofrecimiento describe cómo nuestro marco reacciona bajo la ocurrencia de un evento de la forma $p.\chi$ con $\chi \neq \emptyset$. Es este caso, necesitaremos actualizar el mapa de ofrecimientos para añadirlo de forma que se habiliten una o más interacciones. Formalmente,

$$\frac{C_i \xrightarrow{p.\chi}_L C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{p.\chi}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})} \text{ SI } \chi \neq \emptyset$$

DONDE $\varphi_{i+1} \triangleq \texttt{AddOffer}(\varphi_i, p, \chi)$

$\vartheta_{i+1} \triangleq \vartheta_i$

La regla de terminación describe cómo nuestro marco de trabajo reacciona bajo la ocurrencia de un evento de la forma $p.\emptyset$. En estos casos, sólo tendremos que añadir p al conjunto de procesos finalizados a través de la función `AddFinished`. Formalmente,

$$\frac{C_i \xrightarrow{p.\emptyset}_L C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{p.\emptyset}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})}$$

DONDE $\varphi_{i+1} \triangleq \varphi_i$
$\vartheta_{i+1} \triangleq \texttt{AddFinished}(\vartheta_i, p)$

La regla de sincronización describe cómo nuestro marco de trabajo reacciona cuando se lleva a cabo la ejecución de una interacción. En este caso, necesitaremos eliminar los participantes de la interacción seleccionada de la imagen del mapa de ofrecimientos, lo que deshabilitará a algunas interacciones. Formalmente,

$$\frac{C_i \xrightarrow{x}_L C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{x}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})} \text{ SI } \texttt{Enabled}(\varphi_i, x)$$

DONDE $\varphi_{i+1} \triangleq \texttt{RemoveOffer}(\varphi_i, x)$
$\vartheta_{i+1} \triangleq \vartheta_i$

Fíjese que la implementación de nuestro marco de trabajo no controla la ejecución, aunque almacene información sobre ella que es necesario para implementar nuestra noción y otras propiedades. Por ejemplo, no podemos decidir cuándo un proceso producirá un evento de ofrecimiento o terminación ya que esto depende de la semántica del lenguaje subyacente, el cual es representado a través de la regla genérica de transición $\longrightarrow_L$. En este caso, $\longrightarrow_L$ equivale a $\longrightarrow_{IP_{CORE}}$, pero el marco puede ser fácilmente acomodado a otros lenguajes que soporten el modelo de interacción de IP_{CORE}. Esto hace nuestra implementación más abstracta y genérica, y muy atractiva ya que se puede, por ejemplo, implementar con $J^\sharp$ más la librería adecuada para soportar interacciones multipartitas.

8.2.4 Corrección de la implementación

En esta sección, demostraremos que los predicados y funciones que hemos presentado en la Sección §8.2.2 son implementaciones correctas en las configuraciones actuales de los predicados y funciones teóricos de la Sección §6.2.2.

Para comprender qué entendemos por "implementación correcta en la configuración actual", supongamos cualquiera de las ejecuciones que nuestro marco puede generar. En general, estás son de la forma $\lambda = (D_0, [D_1\, D_2\, D_3 \ldots],$

$[e_1\, e_2\, e_3 \ldots])$ donde cada D_i es una 3-tupla de la forma $(C_i, \varphi_i, \vartheta_i)$ para todo $i \geq 0$. Definimos su proyección sobre IP_{CORE} como $\lambda|_{\text{IP}_{\text{CORE}}} = (C_0, [C_1\, C_2\, C_3 \ldots], [e_1\, e_2\, e_3 \ldots])$. Dada esta definición, decimos que el predicado de nuestro marco teórico $\texttt{pred}(x_1, x_2, \ldots, x_n)$ es implementado en la configuración D_i por el predicado $\texttt{pred}(y_1, y_2, \ldots, y_m)$ si y sólo si las siguientes condiciones son satisfechas para todos los parámetros x_i, y_j $(i \in [1, n], j \in [1, m])$:

Si $\texttt{pred}(x_1, x_2, \ldots, x_n)$ se satisface en C_i, entonces
$\texttt{pred}(y_1, y_2, \ldots, y_m)$ se satisface en D_i
Si $\neg\texttt{pred}(x_1, x_2, \ldots, x_n)$ se satisface en C_i, entonces
$\neg\texttt{pred}(y_1, y_2, \ldots, y_m)$ se satisface en D_i

Análogamente, la función $\texttt{func}(x_1, x_2, \ldots, x_n)$ de nuestro marco teórico es implementado por la funcion $\texttt{func}(y_1, y_2, \ldots, y_m)$ si y sólo si la siguiente condición se satisface para todos los parámetros x_i, y_j $(i \in [1, n], j \in [1, m])$ resulta X:

Si $\texttt{func}(x_1, x_2, \ldots, x_n) = X$, entonces $\texttt{func}(y_1, y_2, \ldots, y_m) = X$

Teorema 8.2.1 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el predicado* $\textit{Waiting}(\lambda, p, \Upsilon, i)$ *es el implementado por* $\texttt{Waiting}(\varphi_i, p, \Upsilon)$.

Prueba Demostraremos este teorema por reducción al absurdo. Primero, supondremos que el predicado $\textit{Waiting}(\lambda, p, \Upsilon, i)$ se satisface y que, en cambio, $\texttt{Waiting}(\varphi_i, p, \Upsilon)$ no lo hace. En otras palabras, existe una interacción $y \in \Upsilon$ tal que $p \notin \varphi_i(y)$. Según la Definición §6.2.5, si $\textit{Waiting}(\lambda, p, \Upsilon, i)$ se satisface, entonces p ofreció participación en el conjunto de interacción $\chi \supseteq \Upsilon$ en alguna configuración previa C_j $(j < i)$, y que ninguna interacción de χ ha sido ejecutada desde entonces. La regla de ofrecimiento obliga al mapa φ_j a actualizarse a través de `AddOffer` de forma que $p \in \varphi_{j+1}(x)$ para todo $x \in \chi$ y $\Upsilon \subseteq \chi$. La única forma de que p sea eliminado del mapa de ofrecimientos es a través de la función `RemoveOffer`, y esto sólo puede ocurrir si y sólo si una regla de sincronización es aplicada. Esto, obviamente, contradice la hipótesis de partida ya que $\textit{Waiting}(\lambda, p, \Upsilon, i)$ no se sería satisfecho.

Ahora, supongamos que el predicado $\textit{Waiting}(\lambda, p, \Upsilon, i)$ no se cumple pero $\texttt{Waiting}(\varphi_i, p, \Upsilon)$ sí. Esto implica que $p \in \varphi_i(x)$ se verifica para todo $x \in \Upsilon$, lo cual implica que existe una configuración D_j $(j < i)$ sobre la que la regla de ofrecimiento ha sido aplicada bajo la ocurrencia de un evento de la forma $p.\chi$, $\chi \supseteq \Upsilon$. Además, ninguna interacción del conjunto χ ha sido ejecutada ya que, de lo contrario, la regla de sincronización habría sido aplicada y habrá eliminado p del mapa de ofrecimientos para toda interacción

de χ. Esto vuelve a contradecir la hipótesis inicial ya que esto implica que $\mathtt{Waiting}(\lambda, p, \Upsilon, k)$ no sería satisfecha para ningún k en el intervalo $[j, i]$. □

Corolario 8.2.2 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces* $\varphi_i(x)$ *contiene los participantes que están esperando a* x *para todo* $x \in I_\Sigma$.

Teorema 8.2.3 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el predicado* $\mathtt{Finished}(\lambda, p, i)$ *es implementado por* $\mathtt{Finished}(\vartheta_i, p)$.

Prueba Demostraremos este teorema por reducción al absurdotambién. Primero, suponemos que el predicado $\mathtt{Finished}(\lambda, p, i)$ se satisface y que, en cambio, $\mathtt{Finished}(\vartheta_i, p)$ no lo hace, es decir, $p \notin \vartheta_i$. Según la Definición §6.2.5, si $\mathtt{Finished}(\lambda, p, i)$ se satisface, entonces p ofreció participación en un conjunto vacío de interacciones en alguna configuración previa C_j $(j < i)$. Así, la regla de terminación fue aplicada, y añadió p a ϑ_{j+1} a través de la función `AddFinished`. Ninguna otra regla cambia el valor del conjunto de procesos finalizados. Por consiguiente, es imposible para p no pertenecer a ϑ_i, lo que contradice nuestra hipótesis de partida ya que $\mathtt{Finished}(\lambda, p, i)$ no sería satisfecho.

Ahora, asumamos que $\mathtt{Finished}(\lambda, p, i)$ no se satisface y que, en cambio, $\mathtt{Finished}(\vartheta_i, p)$ sí, es decir, $p \in \vartheta_i$. Esto implica que el proceso p nunca a producido un evento de la forma $p.\emptyset$ en alguna configuración pasada C_i. Entonces, el marco de trabajo no ha podido detectar este suceso, y p no puede pertenecer al conjunto ϑ_i ya que $\vartheta_0 = \emptyset$ y ninguna otra regla que no sea la de Terminación puede añadir un participante en el conjunto de finalizados. □

Teorema 8.2.4 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el predicado* $\mathtt{Working}(\lambda, p, i)$ *es implementado por* $\mathtt{Working}(\varphi_i, \vartheta_i, p)$.

Prueba Según la Definición §6.2.5, $\mathtt{Working}(\lambda, p, i)$ se satisface si y sólo si p no ha terminado y no está esperando a ejecutar ninguna interacción en la configuración C_i. La corrección de esta implementación es inmediata a partir de los Teoremas §8.2.1 y §8.2.3 ya que la definición de `Working` dependo sólo de `Waiting` y `Finished`. □

Teorema 8.2.5 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el predicado* $\mathtt{Enabled}(\lambda, x, i)$ *es implementado por* $\mathtt{Enabled}(\varphi_i, x)$.

Prueba Demostraremos este teorema por reducción al absurdotambién. Primero, supondremos que el predicado $\texttt{Enabled}(\lambda, x, i)$ se satisface y que, en cambio, $\texttt{Enabled}(\varphi_i, x)$ no lo hace, es decir, $\varphi_i(x) \neq \mathbb{P}(x)$. Según la Definición §6.2.6, si $\texttt{Enabled}(\lambda, x, i)$ se satisface, entonces todos los participantes de x están esperándola, es decir, todos sus participantes han ofrecido x en el pasada, pero ninguno de ellos ha ejecutado ninguna interacción desde entonces. Cada vez que un participante genera un evento de la forma $p.\chi$ con $x \in \chi$, la regla de ofrecimiento es aplicada y añade p al mapa de ofrecimientos de todas las interacciones de χ. Así, si $\texttt{Enabled}(\lambda, x, i)$ se satisface, es porque todos los participantes de x están en el mapa de ofrecimiento, lo que contradice la hipótesis inicial ya que $p \in \varphi_i(x)$ debería satisfacerse para todo $p \in \mathbb{P}(x)$.

Ahora, supongamos que el predicado $\texttt{Enabled}(\lambda, x, i)$ no se satisface, y que $\texttt{Enabled}(\varphi_i, p)$ si, es decir, $\varphi_i(x) = \mathbb{P}(x)$. Fíjese que esto implica que existe al menos un participante p de la interacción x que aún no la ha ofrecido. Así, si no se ha producido un evento de la forma $p.\chi$ con $x \in \chi$, es imposible que p pertenezca a $\varphi_i(x)$, lo que contradice la hipótesis de partida ya que esto implicaría que $\varphi_i(x) \neq \mathbb{P}(x)$. □

Teorema 8.2.6 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración inicial, entonces el predicado* $\texttt{Disabled}(\lambda, x, i)$ *es implementado por* $\texttt{Disabled}(\varphi_i, x)$.

Prueba La demostración es inmediata a partir del Teorema §8.2.5 ya que la definición de `Disabled` sólo se apoya en `Enabled`. □

Teorema 8.2.7 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el predicado* $\texttt{Stable}(\lambda, x, i)$ *es implementado por* $\texttt{Stable}(\varphi_i, \vartheta_i, x)$.

Prueba La demostración es inmediata a partir de los Teoremas §8.2.1 y §8.2.3 ya que la definición de `Stable` sólo depende de `Waiting` y `Finished`. □

Teorema 8.2.8 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el conjunto* $\texttt{StaLinked}(x)$ *es implementado por* $\texttt{StaLinked}(x)$.

Prueba Esta prueba es trivial ya por el hecho de que `StaLinked` sólo depende de las interacciones y procesos que componen el sistema, que son constantes. □

Teorema 8.2.9 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual, entonces el conjunto* $\texttt{DynLinked}(\varphi_i, x)$ *es la implementación de* $\texttt{DynLinked}(\lambda, x)$.

Prueba El Corolario §8.2.2 demuestra que $\varphi_i(x)$ denota el conjunto de participantes que están esperando a la interacción x; así, $\varphi_i(x) \cap \varphi_i(y)$ denota el conjunto de participantes que están esperando a las dos interacciones x y y en la i-ésima configuración. Por consiguiente, $\{y \in \texttt{StaLinked}(x) \mid \varphi_i(y) \cap \varphi_i(x) \neq \emptyset\}$ denota el conjunto de todas las interacciones que comparten algún participante con x en tiempo de ejecución. □

8.2.5 Preservación de las propiedades *safety*

En esta sección, demostraremos que las reglas de transición a través de la que hemos definido la semántica de nuestra implementación, preserva las propiedades *safety* de la Sección §3.2 si $\longrightarrow_L$ las preserva, que es absolutamente necesario ya que, de lo contrario, la semántica del lenguaje de programación sería incorrecta.

Teorema 8.2.10 $\texttt{SafeOffers}(\lambda|_{IP_{CORE}})$ *se satisface para cualquier ejecución* λ *generada por la implementación de nuestro marco.*

Prueba Según la Definición §6.2.8, la propiedad SafeOffers requiere que todo participante que ofrezca participación en un subconjunto $\chi \subseteq I_\Sigma$ permanezca bloqueado hasta que una de esa interacciones sea seleccionada para su ejecución. Supongamos que $\lambda|_{IP_{CORE}}$ no satisface este requisito, lo que implica que existen dos configuraciones $D_i = (C_i, \varphi_i, \vartheta_i)$ y $D_j = (C_j, \varphi_j, \vartheta_j)$ tal que

$$\texttt{Working}(\varphi_{i-1}, \vartheta_{i-1}, p) \wedge \exists \Upsilon \subseteq I_\Sigma \cdot \texttt{Waiting}(\varphi_i, p, \Upsilon) \wedge \neg \texttt{Waiting}(\varphi_j, p, \Upsilon)$$

y no existe ninguna configuración $D_k = (C_k, \varphi_k, \vartheta_k)$ tal que

$$k \in (i, j) \wedge C_k \xrightarrow{x}_L C_{k+1} \wedge x \in \Upsilon$$

Fíjese que $\texttt{Working}(\varphi_{i-1}, \vartheta_{i-1}, p) \wedge \exists \Upsilon \subseteq I_\Sigma \cdot \texttt{Waiting}(\varphi_i, p, \chi)$ implica que el participante p ofrece participación en un subconjunto de interacciones $\chi \supseteq \Upsilon$ en la i-ésima configuración. Así, la regla de ofrecimiento añade p a $\varphi_i(x)$ para todo $x \in \chi$, y la única regla que puede eliminarlo es la de sincronización. Por consiguiente, si $\neg \texttt{Waiting}(\varphi_j, p, \Upsilon)$ se satisface, se tiene que existe una configuración $k \in (i, j)$ en la que la regla de sincronización ha sido aplicada, lo que contradice la hipótesis de partida. □

Teorema 8.2.11 $\texttt{SafeTerms}(\lambda|_{IP_{CORE}})$ *se satisface para cualquier ejecución* λ *generada por nuestra implementación.*

Prueba Según la Definición §6.2.9, la propiedad SafeTerms requiere que todo participante que termina no vuelve a ofrece interacción y que, además, las interacciones en la que puede participar nunca se vuelven a habilitar.

Fíjese que cuando un participante p termina, se produce un evento de la forma $p.\emptyset$ en la i-ésima configuración, y la regla de terminación es entonces aplicada añadiendo p al conjunto de finalizados. Ninguna otra regla puede cambiar este conjunto. Como asumimos que $\longrightarrow_L$ satisface esta propiedad se tiene que ningún otro evento de la forma $p.\chi$ o x para todo $x \in \mathbb{I}(p)$ puede ser detectado por el marco y, así, $p \notin \varphi_j(x)$ para todo $j > i, x \in I_\Sigma$. □

Teorema 8.2.12 $\mathtt{SafeSyncs}(\lambda|_{IP_{CORE}})$ *se satisface para toda ejecución* λ *generada por nuestra implementación.*

Prueba Según la Definición §6.2.9, la propiedad SafeSyncs requiere que una interacción no sea ejecutada a menos que esté habilitada, y, que después de su ejecución, todas las interacciones con las que está enlazada (incluida ella) se deshabilitan y los participantes de x empiezan a trabajar. Sea x una interacción que se ejecuta en la i-ésima configuración. Ya que suponemos que $\longrightarrow_L$ satisface esta propiedad, chequear la habilitación en la regla de sincronización es redundante, aunque esto garantiza que ninguna interacción sea ejecutada a menos que esté habilitada. Fíjese que el consecuente de esta regla garantiza que todos los participantes de x son eliminados de φ_i, es decir, $\mathtt{Enabled}(\varphi_{i+1}, y)$ no puede satisfacerse para ningún $y \in \mathbb{I}(p)$, $p \in \mathbb{P}(x)$. Por consiguiente, ninguna interacción con la que x comparta un participante puede estar habilitada. Además, como ningún participante de x pertenece a $\varphi_{i+1}(y)$ para ningún $y \in I_\Sigma$, $\mathtt{Working}(\varphi_{i+1}, \vartheta_{i+1}, p)$ tiene que satisfacerse para todo $p \in \mathbb{P}(x)$. □

8.2.6 Requisitos de justicia en la implementación

El siguiente teorema demuestra que nuestra implementación no requiere que el sistema en tiempo de ejecución subyacente implemente ningún criterio de selección justa, lo que implica que la decisión de seleccionar qué regla aplicar en caso de conflicto puede ser completamente arbitraria.

Teorema 8.2.13 $\longrightarrow_{FW}$ *no requiere suponer ningún criterio de selección justa en el sistema en tiempo de ejecución.*

Prueba Por un lado, las reglas de ofrecimiento y terminación no pueden ser continuamente aplicadas 'puesto que el número de participantes que ofrecer participación o terminar es finito; así, después de un número finito de aplicaciones, la regla de sincronización tiene que ser necesariamente aplicada o el sistema se interbloquea. Por otro lado, la regla de sincronización tampoco puede ser aplicada continuamente ya que es necesario que para que una interacción se habilite, sus participantes tengan la oportunidad de ofrecerla.

Estos son los motivos por los que la implementación de nuestro marco de trabajo teórico no necesita apoyarse en ningún criterio de selección de reglas conflictivas, lo que concluye la demostración. □

8.3 Implementación de k-*conspiracy-free*

En esta sección, mostramos cómo implementar nuestra noción haciendo uso de la implementación mostrada del marco de trabajo. Primero presentaremos las configuración sobre las que nuestro algoritmo trabaja, y luego lo describimos a través de reglas de Plotkin. Por último, demostraremos que la implementación es correcta.

8.3.1 Configuraciones

Las configuraciones de nuestro algoritmo son 3-tuplas de la forma (D, τ, ρ), a saber: $D = (C, \varphi, \vartheta)$ es la configuración del marco de trabajo; τ es una cola en la ordenaremos las interacción en función a su edad, es decir, $\tau: \mathbb{N} \to I_\Sigma$, $\operatorname{ran} \varphi = I_\Sigma$; y ρ es un mapa que cuenta el número de veces que las interacciones son ejecutas en situación de conspiración potencial, es decir, $\rho: I_\Sigma \to \mathbb{N}$, $\operatorname{dom} \rho = I_\Sigma$.

Merece la pena destacar que estas configuraciones no nos permite qué edad tienen las interacciones, aunque si la edad relativa de unas respecto a otras. Fíjese que una implementación en la que se almacenara la edad de una interacción necesariamente sería incompleta cuando el rango de los contadores no permitiese almacenar la edad de las interacciones. Nuestras configuraciones nos permiten determinar si una interacción es más vieja que otra examinando sus posiciones en la cola τ ya que la semántica de nuestro algoritmo garantiza que una interacción x es más vieja que otra interacción y si $\tau^{-1}(x) < \tau^{-1}(y)$.

Las configuraciones son actualizadas y consultadas a través de las siguientes funciones y predicados:

$$\begin{aligned}
\texttt{MoveRear}([x_1,\ldots,x_{j-1},x_j,x_{j+1},\ldots x_n],x_j) &= [x_1,\ldots,x_{j-1},x_{j+1},\ldots x_n,x_j]\\
\texttt{IncreaseConsp}(\rho,x) &= \{z \mapsto \rho(z) \mid z \in I_\Sigma \setminus \{x\}\} \cup \{x \mapsto \rho(x)+1\}\\
\texttt{OldestEna}(\tau,\varphi) = x &\iff \texttt{Enabled}(\varphi,x) \Rightarrow \forall y \in I_\Sigma \setminus \{x\} \cdot \tau^{-1}(y) > \tau^{-1}(x)\\
\texttt{PotConsp}(\varphi,\vartheta,x) &\iff \exists y \in \texttt{StaLinked}(x) \cdot \neg\, \texttt{Stable}(\varphi,\vartheta,y)
\end{aligned}$$

$\texttt{MoveRear}(\tau,x)$ se utiliza cada vez que una interacción x es ejecutada, moviéndola al final de la cola τ y preservando el orden relativo de las restantes interacciones. $\texttt{IncreaseConsp}(\rho,x)$ incrementa el contador de situaciones de conspiración de la interacción x cuando esta es seleccionada en presencia de interacciones enlazadas que no están estables. $\texttt{OldestEna}(\tau,\varphi)$ devuelve la primera interacción habilitada de la cola τ, es decir, aquella cuya edad es mínima y está habilitada. $\texttt{PotConsp}(\varphi,\vartheta,x)$ se satisface si la interacción x se ejecuta en situación de conspiración potencial, lo que ocurre si existe al menos una interacción estáticamente enlazada con ella que no está estable.

8.3.2 Semántica

El objetivo de nuestro algoritmo es monitorizar las transiciones del marco de trabajo para no permitir la ejecución de una interacción que no satisfaga los requisitos de la selección k-*conspiracy-free*. La configuración inicial es de la forma (D_0,τ_0,ρ_0), donde D_0 es la configuración inicial del marco de trabajo, $\operatorname{ran}\tau_0 = I_\Sigma$ (fíjese que cualquier permutación I_Σ es válida) y $\rho(x) = 0$ para todo $x \in I_\Sigma$. Desde esta configuración en adelante, el sistema es controlado por la regla de tránsito, la de sincronización conspiratoria y la de sincronización estable. Teniéndose:

La regla de tránsito describe cómo nuestro algoritmo reacciona en la ocurrencia de un evento de la forma $p.\chi$. En este caso, lo único que tenemos que hacer el conservar los cambios producidos en la configuración del marco. Formalmente,

$$\frac{D_i \xrightarrow{p.\chi}_{FW} D_{i+1}}{(D_i,\tau_i,\rho_i) \xrightarrow{p.\chi}_{CFF_k} (D_{i+1},\tau_{i+1},\rho_{i+1})}$$

DONDE $\tau_{i+1} \triangleq \tau_i$

$\rho_{i+1} \triangleq \rho_i$

Fíjese que ambos casos, $\chi \neq \emptyset$ y $\chi = \emptyset$, son tratados homogéneamente ya que no es necesario ni cambiar la cola ni los contadores de conspiración.

La regla de sincronización conspiratoria describe cómo nuestro algoritmo reacciona cuando una interacción puede ser ejecutada en situación de conspiración potencial, aunque no ha ejecutado más de k veces en esta situación. En estos casos, el algoritmo no permite que la interacción se ejecute, aunque graba este hecho.

$$\frac{D_i \xrightarrow{x}_{FW} D_{i+1}}{(D_i, \tau_i, \rho_i) \xrightarrow{x}_{CFF_k} (D_{i+1}, \tau_{i+1}, \rho_{i+1})} \quad \text{SI } \texttt{PotConsp}(\varphi_i, \vartheta_i, x) \wedge \rho_i(x) < k$$

DONDE $D_i \triangleq (C_i, \varphi_i, \vartheta_i)$
$\tau_{i+1} \triangleq \texttt{MoveRear}(\tau_i, x)$
$\rho_{i+1} \triangleq \texttt{IncreaseConsp}(\rho_i, x)$

Fíjese que la interacción ejecutada el desplazada al final de la cola τ_i, lo que la convierte en la más joven, y su contador de conspiraciones es incrementado.

La regla de sincronización estable describo el comportamiento de nuestro algoritmo cuando una interacción habilitada esta lista para ser ejecutada en una situación de no conspiración, y es la más vieja de entre las interacciones habilitadas. En estos casos, puede ser ejecutada inmediatamente.

$$\frac{D_i \xrightarrow{x}_{FW} D_{i+1}}{(D_i, \tau_i, \rho_i) \xrightarrow{x}_{CFF_k} (D_{i+1}, \tau_{i+1}, \rho_{i+1})} \quad \text{SI } \neg\texttt{PotConsp}(\varphi_i, \vartheta_i, x) \wedge x = \texttt{OldestEna}(\tau_i, \varphi_i)$$

DONDE $D_i \triangleq (C_i, \varphi_i, \vartheta_i)$
$\tau_{i+1} \triangleq \texttt{MoveRear}(\tau_i, x)$
$\rho_{i+1} \triangleq \rho_i$

Fíjese que x también es desplazada al final de τ_i, pero ρ_i no cambia ya que esta interacción se ha ejecutado cuando todas sus interacciones enlazadas estaban estables.

8.3.3 Corrección de los predicados y las funciones

En esta sección, nuestro objetivo es demostrar que tanto `OldestEna` como `PotConsp` implementan OldestEna y PotConsp en la configuración actual. También demostramos que $\rho_i(x)$ es una implementación correcta de la expresión $\sum_{j=0}^{i} \text{PotConsp}(\lambda, x, j)$ para todo $x \in I_\Sigma$.

Teorema 8.3.1 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual del marco de trabajo y* $E_i = (D_i, \tau_i, \rho_i)$ *es la configuración actual de nuestro algoritmo,* $\mathtt{OldestEna}(\lambda, x, i)$ *es implementado po* $\mathtt{OldestEna}(\tau_i, \varphi_i)$.

Prueba Según la definición de la Sección §7.2, $\mathtt{OldestEna}(\lambda, x, i)$ se satisface si y sólo si la interacción x es la más vieja de entre las habilitadas en la configuración C_i. Recalquemos que x es más viaja que y si y sólo si la primera fue ejecutada antes o, de lo contrario, el orden inicial arbitrario $\prec$ sobre el que la definición de $\mathtt{Age}$ se basa hace y más vieja por definición.

Fíjese que cada vez que las reglas de sincronización estable o conspiratoria son aplicadas, la interacción seleccionada se desplaza al final de la cola τ y el orden relativo de las restantes interacciones permanece igual. Esto implica que cuanto más cerca está una interacción del final, menos tiempo hace que se ha ejecutado. Además, $\prec$ es un orden total, lo que implica que cualquier configuración inicial τ_0 es válida e implementa este orden arbitrario.

Por consiguiente, `OldestEna` no puede devolver otra interacción que no sea la más vieja entre las habilitadas en la configuración D_i. □

Teorema 8.3.2 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual del marco de trabajo y* $E_i = (D_i, \tau_i, \rho_i)$ *la configuración actual de nuestro algoritmo, entonces el predicado* $\mathtt{PotConsp}(\lambda, x, i)$ *es implementado por* $\mathtt{PotConsp}(\varphi, \vartheta, x)$.

Prueba La demostración se apoya en los Teoremas §8.2.8 y §8.2.7, donde demostramos que las implementaciones de `StaLinked` y `Stable` son correctas. □

Teorema 8.3.3 *Si* $D_i = (C_i, \varphi_i, \vartheta_i)$ *es la configuración actual del marco de trabajo y* $E_i = (D_i, \tau_i, \rho_i)$ *es la configuración actual de nuestro algoritmo, entonces* $\rho_i(x)$ *es una implementación correcta de* $\sum_{j=0}^{i} \mathtt{PotConsp}(\lambda, x, j)$ *en la configuración actual.*

Prueba Inicialmente, $\rho_0(x) = 0$ para todo $x \in I_\Sigma$. Desde la configuración inicial en adelante, la única regla que puede cambiar el valor de este mapa es la de sincronización conspiratoria, que incrementa en una unidad el contador de la interacción seleccionada. Ya que esta es la única regla que requiere que PotConsp se satisfaga y el Teorema §8.3.2 demuestra que este predicado es implementado correctamente por `PotConsp`, se tiene que $\rho_i(x)$ almacena el número de veces que la interacción x ha sido ejecutada en situación de conspiración potencial. □

8.3.4 Corrección y completitud de los algoritmos

En esta sección, demostramos que el conjunto de ejecuciones que nuestro algoritmo produce coincide con el conjunto de ejecuciones que son de nivel k-*conspiracy-free*. Como en el caso anterior, si λ es una ejecución producida por nuestro algoritmo, $\lambda|_{IP_{CORE}}$ denota su proyección sobre IP_{CORE}.

Teorema 8.3.4 (Correción) *Si* λ *cualquier ejecución producida por nuestro algoritmo, entonces* $\lambda|_{IP_{CORE}}$ *es* k-*conspiracy-free.*

Prueba Procederemos por reducción al absurdo. Supongamos que existe un índice $p^* \geq 0$ tal que nuestro algoritmo produce una prefijo de la forma

$$\Omega = E_0 \xrightarrow{e_1}_{CFF_k} E_1 \xrightarrow{e_2}_{CFF_k} \cdots \xrightarrow{e_{p^*}}_{CFF_k} E_{p^*} \xrightarrow{e_{p^*+1}}_{CFF_k} E_{p^*+1}$$
$$E_i = (D_i, \tau_i, \rho_i) \text{ para todo } i \in [0, p^* + 1]$$
$$D_i = (C_i, \varphi_i, \vartheta_i) \text{ para todo } i \in [0, p^* + 1]$$

Sin perdida de generalidad, podemos suponer que este prefijo es minimal, es decir, E_{p^*} es la primera configuración en la que se puede detectar que la ejecución no es k-*conspiracy-free* ya que la ejecución del evento e_{p^*+1} violaría el criterio. Esto implica que $e_{p^*+1} = x$ para algún $x \in I_\Sigma$, lo que implica que podremos distinguir los siguientes casos:

i. El umbral de conspiración puede ser excedido. Esto no puede ocurrir porque la regla de sincronización conspiratoria no sería aplicable ya que las implementaciones PotConsp y $\sum_{j=0}^{i} \text{PotConsp}(\lambda, x, j)$ son correctas según los Teoremas §8.3.2 y §8.3.3.

ii. Las interacciones enlazadas con ellas están estable, pero no es la más vieja. Esto es imposible ya que las implementaciones de OldestEna y PotConsp son correctas según los Teoremas §8.3.1 y §8.3.2.

Estas contradicciones concluyen la demostración porque implica que ni la regla de sincronización estable y conspiratoria pueden permitir que una interacción sea ejecutada a menos que el nuevo prefijo generado satisfaga los requisitos de k-*conspiracy-free*. □

Teorema 8.3.5 (Completitud) *Sea* λ *cualquier ejecución* k*-conspiracy-free para algún* $k \in \mathbb{N}$. *Nuestro algoritmo puede producir una ejecución* Ω *tal que* $\Omega|_{IP_{CORE}} = \lambda$.

Prueba Procederemos por reducción al absurdo. Supongamos que existe una ejecución λ k*-conspiracy-free* que no puede ser generada por nuestro algoritmo. Esto implica que existe un índice $p^* \geq 0$ tal que

$$\lambda = C_0 \xrightarrow{e_1}_L C_1 \xrightarrow{e_2}_L \cdots \xrightarrow{e_{p^*}}_L C_{p^*} \xrightarrow{e_{p^*+1}}_L C_{p^*+1}$$

pero el algoritmo produce

$$\begin{aligned}
&\Omega = E_0 \xrightarrow{e_1}_{CFF_k} E_1 \xrightarrow{e_2}_{CFF_k} \cdots \xrightarrow{e_{p^*}}_{CFF_k} E_{p^*} \\
&E_i = (D_i, \tau_i, \rho_i) \text{ para todo } i \in [0, p^*] \\
&D_i = (C_i, \varphi_i, \vartheta_i) \text{ para todo } i \in [0, p^*]
\end{aligned}$$

y se hace la "encerrona" en la configuración E_{p^*} o la transición a la siguiente transición es de la forma $E_{p^*} \xrightarrow{e'_{p^*+1}}_{CFF_k} E'_{p^*+1}$, pero $e'_{p^*+1} \neq e_{p^*+1}$ o E'_{p^*+1} es de la forma $((C'_{p^*+1}, \varphi'_{p^*+1}, \vartheta'_{p^*+1}), \tau'_{p^*+1}, \rho'_{p^*+1})$ y $C'_{p^*+1} \neq C_{p^*+1}$. Demostraremos que esto conduce a una contradicción por inducción estructural sobre el evento e_{p^*+1}, a saber:

i. Si $e_{p^*+1} = p.\chi$ para algún $p \in P_\Sigma$, $\chi \subseteq I_\Sigma$, entonces la regla de tránsito habría sido aplicada en la configuración E_{p^*} y E_{p^*+1} sería $(D_{p^*+1}, \tau_{p^*+1}, \rho_{p^*+1})$ con $D_{p^*+1} = (C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1})$.

ii. Si $e_{p^*+1} = x$ para algún $x \in I_\Sigma$, entonces podemos distinguir un par de casos, a saber: si C_{p^*} satisface $\neg \mathtt{PotConsp}(\lambda, x, i) \wedge \mathtt{OldestEna}(\lambda, x, i)$, entonces la regla de sincronización conspiratoria habría sido aplicada y E'_{p^*+1} sería $((C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1}), \mathtt{MoveRear}(\tau_{p^*}, x), \rho_{p^*})$ ya que las implementaciones de `PotConsp` y `OldestEna` son correctas según los Teoremas §8.3.1 y §8.3.2. Si $\sum_{j=0}^{i} \mathtt{PotConsp}(\lambda, x, j) < k$ se satisface, entonces el Teorema §8.3.3 demuestra que este predicado es implementado por $\rho_i(x) < k$ en E_i, lo que implica que E_{p^*+1} sería $(D_{p^*+1}, \mathtt{MoveRear}(\tau_{p^*}, x), \mathtt{IncreaseConsp}(\rho_{p^*}, x))$ donde la configuración D_{p^*+1} es $(C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1})$.

En ambos casos, existe una clara contradicción ya que el algoritmo debería reaccionar ante el mismo evento y conduciría a la misma configuración C_{p^*+1}. □

Capítulo 9

Estudio experimental

Experience teaches slowly
at the cost of mistakes

James A. Froude, 1818–1894
English historian

La mayoría de la implementaciones de las nociones de selección justa no han sido analizadas desde el punto de vista empírico. En este capítulo, presentamos un estudio detallado en el que comparamos nuestra implementación de k-conspiracy-free fairness *con la mayoría de algoritmos que implementan* strong fairness *e* hyperfairness *en el contexto de las interacciones multipartitas. En la Sección §9.1, introducimos el capítulo y definimos algunos conceptos importantes; en la Sección §9.2, mostramos cómo se comportan los tiempos de ejecución y de selección conforme el número de participantes y la longitud de la ejecución crecen; en la Sección §9.3, estudiamos cómo los distintos algoritmos se comportan en situaciones de conspiración; en la Sección §9.4, presentamos un método simple para obtener un buen valor de* k *para el programa de los filósofos comensales.*

9.1 Introducción

Para evaluar la implementación de nuestro marco de trabajo teórico y nuestra noción, hemos medido su rendimiento y efectividad usando el problema de los filósofos comensales como banco de pruebas. Para la implementación hemos utilizado el lenguaje de programación J♯, un dialecto eficiente de JAVA para la plataforma .NET [119]. Hemos ejecutado los pruebas bajo WINDOWS XP PROFESSIONAL en una máquina equipada con microprocesador AMD ATHLON XP a 2.0 Ghz y 512 MB de memoria DDR a 266 Mhz. Durante las pruebas, la máquina ha estado desconectada de la red y los servicios de WINDOWS fueron reducidos al mínimo con el objeto de evitar el máximo de interferencias posibles. No desactivamos el recolector de basura de la plataforma .NET ya que esta interferencias son inevitables en la vida real. Los detalles de esta implementación pueden encontrarse en la Referencia [113].

En este capítulo, comparamos nuestra propuesta (de ahora en adelante referida como $\mathcal{CFF}_k$) con la mayoría de la nociones para implementar justicia en el contexto de la interacciones multipartitas presentadas en el Capítulo §4, a saber: la propuesta de Francez y Forman [53] para el nivel *strong* (referida como *Fair*), la propuesta incremental de Corchuelo [34] (referida como *IncFair*), la propuesta transformacional de Attie *et al.* [10] (referida como *HyperFair*) y la propuesta aleatoria de Joung [67] (referida como *TB*). Fíjese que la implementación del marco de trabajo no incorpora justicia pero merece la pena compararlo ya que muestra como un sistema se comporta cuando la justicia no es implementada, es decir, nos permite analizar el impacto en la eficiencia que las anteriores propuestas introduce (referida como *Random*). Los algoritmos de Best, Fagin y Williams o Olderog y Apt [17, 18, 48, 98] son tan similares a los de Francez y Forman que hemos decidido no mostrarlos explícitamente.

Hemos ejecutado los algoritmos en varios escenarios para mostrar el impacto sobre las conspiraciones y la eficiencia, los cuales han sido interpretados como una función del tiempo necesario para ejecutar un experimento y el tiempo medio de selección; también hemos evaluado el impacto de k en el comportamiento del programa. Fíjese que la transformación de Attie *et al.* no puede ser aplicada a un programa a menos que éste se encuentre en forma normal y todas sus interacciones sean resistentes a conspiraciones. Los resultados que mostramos son los producidos por el Programa §9.1. Fíjese que la interacción $Lunch_i$ simula le ejecución de la secuencia de interacciones Get_i y Rel_i ya que de lo contrario las interacciones *Rel* no sería resistentes a conspiraciones y no podríamos aplicar la transformación.

A lo largo de este capítulo usaremos los siguiente términos:

Longitud de la ejecución: la definimos como el número de interacciones *Get*

CR-DINNER :: $[\|_{i=1}^{N} P_i \| F_i]$, **where**
P_i :: *[$\text{Lunch}_i\langle eat\rangle \rightarrow think$];
F_i :: *[$\text{Lunch}_i\langle\rangle \rightarrow$ **skip** [] $\text{Lunch}_{i+1}\langle\rangle \rightarrow$ **skip**].

Programa 9.1: *Una versión resistente a conspiraciones de los filósofos comensales en forma normal.*

que se ejecutan, y nos referimos a ella como L. Fíjese que una ejecución debe incluir eventos *Get* y *Rel*, pero el segundo depende de que el primero se ejecute antes; así, los eventos *Rel* no necesitan se tenidos en cuenta de forma explícita. En el caso de *HyperFair*, la longitud de la ejecución es definida como número de interacciones *Lunch* ejecutadas por dos.

Número de comidas: representan el número de interacciones *Get* que cada filósofo ejecuta. Lo denotamos como L_i, donde i es el índice correspondiente al filósofo i-ésimo.

Distribución de comidas: es una función discreta que relaciona cada ejecución de la interacción *Get* de su correspondiente filósofo. La representaremos como una nube de puntos en la que el eje 0X representa las ejecuciones de las interacciones *Get* y el eje 0Y representa su correspondiente filósofo.

Tiempo de ejecución: es el tiempo que transcurre desde el comienzo de la ejecución hasta que está alcanza una longitud predeterminada.

Tiempo de selección: es el tiempo medio que pasa desde que una interacción es ofrecida por un participante hasta que se selecciona o la ejecución termina, lo que ocurra antes.

A menudo escribimos que "el filósofo i come" para referir que éste ha ejecutado la interacción Get_i. Fíjese que las comidas no son atomicas ya que requieren que los filósofos participen en dos interacciones. Sin embargo, "comer" puede ser visto como una acción atómica ya que una vez que un filósofo o un tenedor ejecutan *Get*, lo único que ejecutan después es el correspondiente *Rel*. Hablaremos también de "comidas" para referirnos a la ejecución de las interacciones *Get*. Aunque es un abuso del lenguaje, pensamos que estás expresiones merecen la pena ya que hacen la escritura más fluida e intuitiva.

9.2 Tiempos

Para evaluar el comportamiento de nuestra implementación con respecto al tiempo, hemos utilizado un escenario en el que el tiempo que cada filósofo está pensando o comiendo es despreciable con respecto al tiempo necesario para detectar al habilitaciones, obtener la exclusión mutua y seleccionar las interacciones. De esta forma, todo filósofo debe ser capaz de comer tantas veces que sus compañeros durante una ejecución justa lo suficientemente larga.

Por un lado, medimos cuánto tiempo necesitan los algoritmos para distribuir 5000 comidas cuando el número de filósofos se encuentra entre 10 y 200 y crece de 10 en 10 unidades; por otro lado, medimos cuánto tiempo necesitan 10 filósofos para distribuir 100, 200, 300, ..., 5000 comidas. El primer experimento es útil para caracterizar el comportamiento de los algoritmos en función al número de participantes e interacciones. Como mostraremos más adelante, este comportamiento es polinomial de grado dos. Por el contrario, el segundo experimento se utiliza para determinar cómo la longitud de la ejecución afecta a los tiempos. En este caso, nuestra implementación se comporta linealmente.

Para reducir la significancia de los posibles *outliers* producidos por interferencias ocasionales con el sistema operativo, hemos tomado tiempos medios de 100 ejecuciones y hemos despreciado los valores máximos y mínimos de cada experimento.

9.2.1 Tiempo conforme el número de participantes se incrementa

La Figura §9.1 muestra que *Random* es el algoritmo más rápido, y que más lento es $\mathcal{CFF}_k$ con $k = 0$. Fíjese que el rendimiento de *IncFair* es mejor que el de *Fair* ya que, en promedio, el primero no necesita examinar todas las interacciones antes de tomar una decisión, pero peor que *Random* ya que necesita mantener una estructuras de datos más complejas y el tiempo de necesario para seleccionar una interacción depende de la probabilidad de encontrarla en la cola y la el tiempo necesario para que se estabilicen.

Los tiempos de $\mathcal{CFF}_k$ dependen del valor que le asignemos a k. Si k es mínima ($k = 0$), el rendimiento es peor que *Fair* ya que las interacciones están enlazadas dos a dos y esto implica que cuando una interacción es seleccionada, las restantes tienen que estar estables. Aunque los tiempos son similares a *Fair*, las estructuras de datos que tiene que mantener $\mathcal{CFF}_k$ son más complejas,

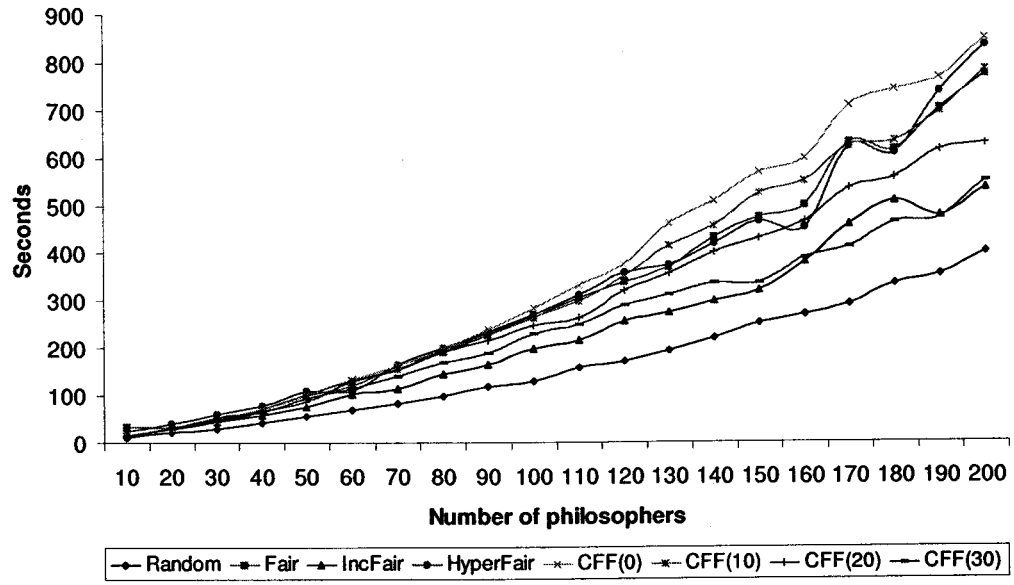


Figura 9.1: *Tiempo de ejecución conforme el número de participantes crece.*

N	*Random*	*Fair*	*HyperFair*	*IncFair*	$\mathcal{CFF}_k(0)$	$\mathcal{CFF}_k(10)$	$\mathcal{CFF}_k(20)$	$\mathcal{CFF}_k(30)$
10	10.79	33.87	23.61	16.82	18.18	16.46	15.67	14.38
20	21.82	34.20	40.38	28.10	29.34	31.18	29.27	31.22
30	28.31	50.87	60.47	43.92	46.52	49.74	52.50	47.38
40	42.86	72.36	77.81	57.81	69.29	72.08	67.28	65.35
50	55.43	99.87	108.53	76.26	99.91	102.44	86.62	94.39
60	68.74	120.90	108.62	101.53	134.23	131.19	128.24	113.45
70	81.46	161.95	164.56	114.16	161.95	156.16	154.26	140.98
80	97.42	191.88	199.54	144.72	198.02	195.10	190.65	169.78
90	118.46	227.25	234.26	164.57	239.24	231.91	215.91	189.91
100	130.00	263.41	269.43	198.52	283.87	266.10	248.15	228.48
110	157.95	304.81	312.58	216.83	331.69	297.44	262.61	248.73
120	171.99	339.18	359.12	256.53	376.28	351.95	321.46	290.69
130	193.41	370.85	373.64	274.27	460.99	413.58	357.14	311.53
140	221.27	432.38	419.86	299.49	508.19	456.67	400.00	336.62
150	252.16	474.73	465.00	319.70	568.60	524.12	430.04	335.96
160	269.13	499.09	451.20	379.97	596.29	550.34	466.21	390.48
170	292.04	629.60	621.01	459.16	708.32	624.70	535.05	411.42
180	334.91	614.70	607.76	507.15	742.70	632.66	557.76	464.12
190	353.69	702.02	737.38	475.83	766.19	694.71	615.29	471.16
200	401.64	773.87	833.13	533.68	848.39	782.33	629.01	550.10

Tabla 9.1: *Tiempos de ejecución conforme el número de participantes crece.*

Algoritmo	Polinomial	(R^2)	Exponencial	(R^2)
Random	$0.66x^2 + 6.02x + 6.67$	0.998	$10.31e^{0.16x}$	0.928
Fair	$1.30x^2 + 11.65x + 10.74$	0.995	$39.63e^{0.16x}$	0.946
IncFair	$0.84x^2 + 10.28x + 4.95$	0.989	$29.35e^{0.16x}$	0.928
HyperFair	$1.50x^2 - 8.00x + 24.00$	0.985	$39.73e^{0.16x}$	0.934
$\mathcal{CFF}_k(0)$	$1.31x^2 + 17.96x - 17.89$	0.996	$32.80e^{0.18x}$	0.920
$\mathcal{CFF}_k(10)$	$1.10x^2 + 17.33x - 10.99$	0.997	$33.79e^{0.17x}$	0.910
$\mathcal{CFF}_k(20)$	$0.71x^2 + 18.94x - 12.79$	0.997	$33.43e^{0.16x}$	0.896
$\mathcal{CFF}_k(30)$	$0.40x^2 + 18.67x - 8.02$	0.994	$34.88e^{0.15x}$	0.891

Tabla 9.2: *Tendencia de los tiempos de ejecución en función al número de participantes.*

lo que justifica que sea ligeramente peor. Sin embargo, si k es grande, el rendimiento de $\mathcal{CFF}_k$ es similar a *Random* si la ejecución no es demasiado larga. Fíjese que los Lemas §7.3.1 y §7.3.2 garantizan que existe una configuración a partir de la cual toda interacción es seleccionada en situaciones de no conspiración. Esto implica que a partir de ese momento, el comportamiento de $\mathcal{CFF}_k$ con $k > 0$ tiene a ser como el de $\mathcal{CFF}_k$ con $k = 0$.

Este es el comportamiento esperado ya que k restringe el número de veces que las interacciones que están enlazadas no tienen que esperarse unas a otras; así, cuanto más pequeño es k, más tendrán que esperar las unas a las otras. Por ejemplo, si tenemos 10 filósofos, el número medio de interacciones por segundos va desde 550.05 int/sec en el peor de los casos ($\mathcal{CFF}_k$ con $k = 0$) hasta 926.78 int/sec en el mejor de los casos (*Random*). Por el contrario, si tenemos 200 filósofos, el rendimiento cae hasta 11.78 int/sec y 24.89 int/sec. (Este número se obtiene dividiendo entre 10 000 (5000 interacciones *Get* y 5000 interacciones *Rel*) los tiempos mostrado en la Tabla §9.1.)

La Tabla §9.1 contiene los tiempos de ejecución de la Figura §9.1. Fíjese que *Random* ejecuta el test en 10.79 segundos y que $\mathcal{CFF}_k$, con $k = 0$, en 18.18 segundos cuando el sistema está compuesto por 10 filósofos. Este tiempo se incrementa con el número de número de filósofos. Hemos llevado a cabo un análisis experimental para determinar si este comportamiento es lineal, logarítmico, polinomial, potencial o exponencial, y los resultados son los que se muestran en la Tabla §9.2. De acuerdo al valor estándar de R^2 [94], podemos concluir que el comportamiento de todos los algoritmos es polinomial de grado dos.

La Figura §9.2 muestra el comportamiento de *TB* conforme el número de participantes se incrementa. Fíjese que estos tiempos son mucho mayores que los otros, y que no tienen una tendencia clara. Además, los tiempos son expre-

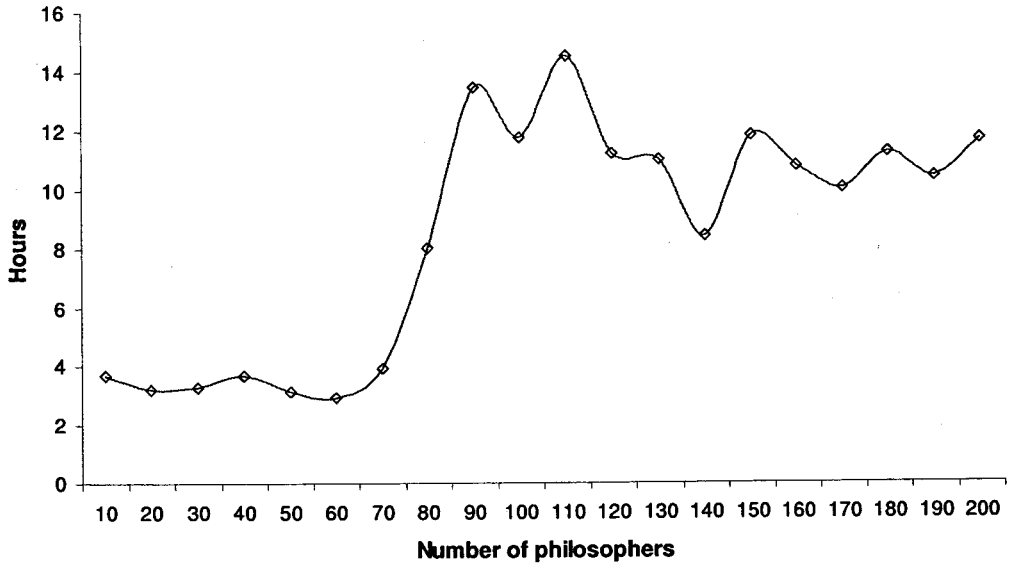


Figura 9.2: *Tiempo de ejecución de TB conforme el número de participantes crece.*

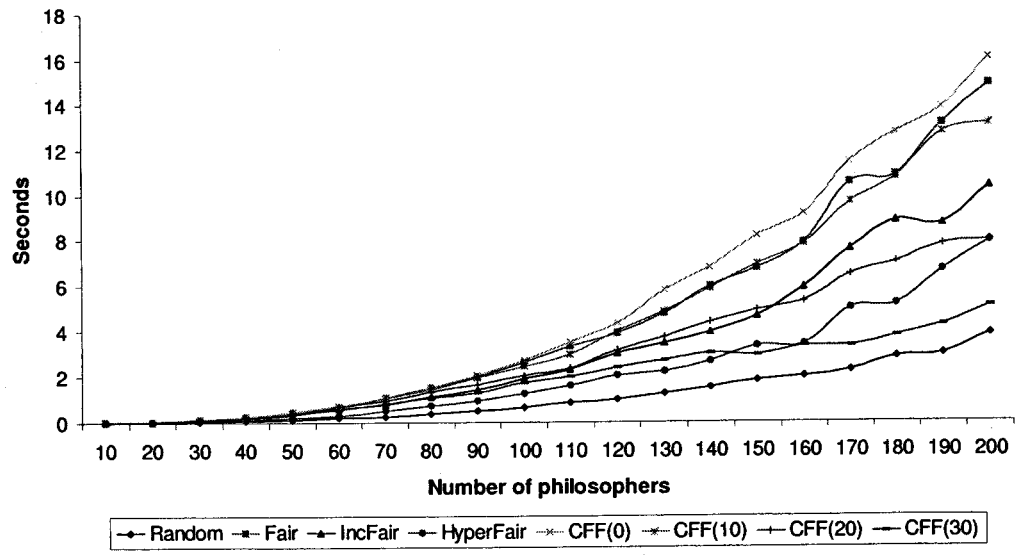


Figura 9.3: *Tiempo de selección conforme el número de participantes crece.*

sados en horas, teniendo un pico de 14 horas cuando el número de filósofos es 110. Estos resultado corroboran que *TB* carece de interés práctico, aunque sea el único algoritmo sin esperas que implemente el nivel *strong* con probabilidad uno.

La Figura §9.3 muestra los tiempos de selección. En el caso de $\mathcal{CFF}_k$ el comportamiento vuelve a depender del valor de k. Si k es mínimo, los tiempos son un poco peores que *Fair*, pero cuando k es grande, los tiempos se asemejan a los mejores de *Random* ya que como el número de veces que una interacción se ejecuta en situación de conspiración no alcanza k se tiene que éstas no tienen que esperarse entre si. Cuando tenemos 200 filósofos, el tiempo de selección en *Fair* es algo mayor de 14 segundos. Dependiendo del valor de k, el tiempo de selección de nuestra propuesta va desde 5 segundos cuando $k = 30$ hasta 16 segundos cuando $k = 0$. El tiempo de selección para *TB* va desde 30 segundos hasta 14 minutos por interacción, aunque omitimos el gráfico ya que no contribuye a la discusión.

9.2.2 Tiempos conforme la longitud de la ejecución crece

La Figura §9.4 muestra los tiempos de ejecución de *Random*, *Fair*, *IncFair*, *HyperFair* y $\mathcal{CFF}_k$. Es comportamiento es bastante bueno, ya que este es lineal. Los tiempos de *TB* los hemos omitido ya que son muy grandes y no contribuyen a la discusión.

Fíjese que el valor de k en este experimento no permanece constante y que es expresado como un porcentaje. La razón es que resulta adecuado definir el número de situaciones de conspiraciones permitidas como un porcentaje del número total de interacciones ejecutadas. Así, la significancia de permitir 10 situaciones de conspiración en una ejecución con 100 comidas es mayor que en una ejecución con 10 000 comidas, es decir, el 10% versus el 0.1% respectivamente.

La Tabla §9.3 muestra los tiempos obtenidos. Fíjese que *Random* tiene un tiempo de ejecución medio de 221 milisegundos para 100 comidas, y de 256 milisegundos con $\mathcal{CFF}_k$ para $k = 0$. Estos tiempos se incrementan con el número de comidas. Hemos llevado a cabo un análisis experimental para determinar cuál es la tendencia de este comportamiento, el cual se encuentra esquematizado en la Tabla §9.4. Como muestran las figuras, todos los algoritmos tienen un comportamiento lineal.

La Figura §9.5 muestra gráficamente los tiempos de selección de *Random*, *Fair*, *IncFair*, *HyperFair* y $\mathcal{CFF}_k$. Fíjese que permanecen casi constantes cuando el número de comidas se incrementa. La Figura §9.6 muestra que el único

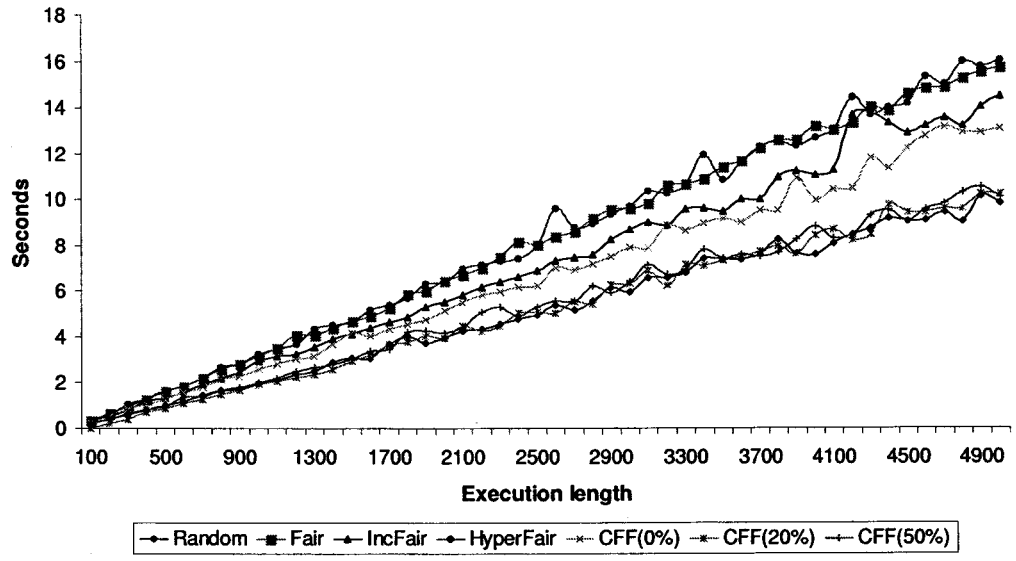


Figura 9.4: *Tiempo de ejecución conforme la longitud de la ejecución crece.*

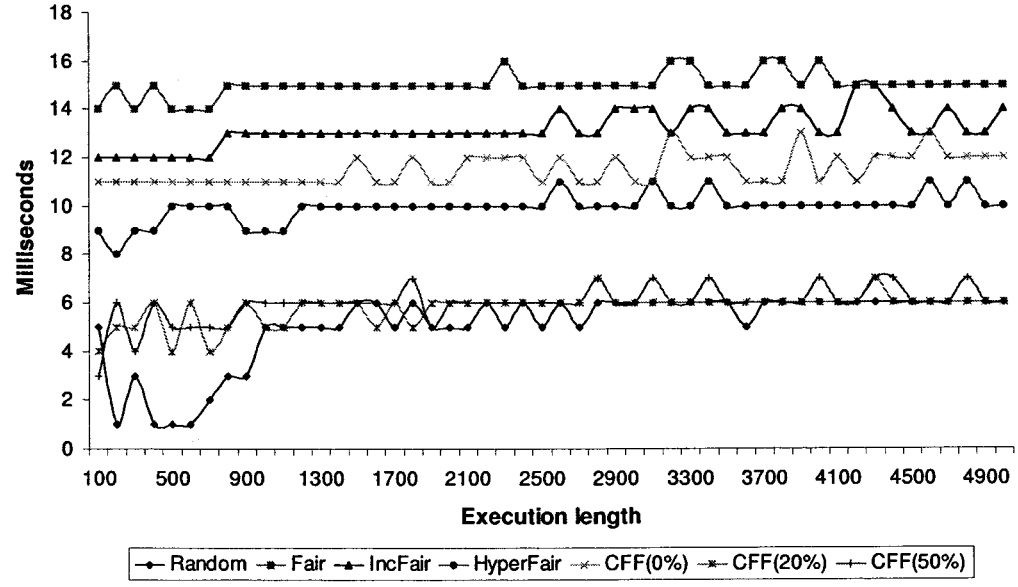


Figura 9.5: *Tiempo de selección conforme la longitud de la ejecución crece.*

N	*Random*	*Fair*	*HyperFair*	*IncFair*	$\mathcal{CFF}_k(0\%)$	$\mathcal{CFF}_k(20\%)$	$\mathcal{CFF}_k(50\%)$
100	221	336	367	271	256	216	215
200	386	677	601	571	526	415	401
300	611	951	1074	781	826	717	671
400	811	1262	1284	1167	1052	861	842
500	1017	1618	1683	1307	1267	1092	982
600	1167	1853	1840	1598	1588	1252	1347
700	1468	2179	2194	1913	1808	1497	1397
800	1668	2579	2674	2209	2163	1653	1678
900	1708	2794	2809	2444	2279	1923	1803
1000	1963	3160	3261	2925	2574	2068	2003
1100	2113	3521	3463	3180	2830	2244	2203
1200	2359	4072	3696	3250	3050	2384	2509
1300	2499	4072	4327	3556	3165	2599	2699
1400	2895	4412	4522	3911	3696	2950	2800
1500	3065	4682	4665	4137	4162	3220	3020
1600	3080	4902	5191	4382	4037	3651	3360
1700	3676	5253	5379	4657	4347	3786	3460
1800	4046	5885	5694	4877	4577	4046	4232
1900	3741	6030	6325	5308	4763	3957	4257
2000	3951	6450	6408	5544	5123	4492	4182
...	...	...	...	...	...	...	...
5000	9815	15735	16023	14492	13070	10501	10115

Tabla 9.3: *Tiempos de ejecución conforme la longitud de la ejecución crece.*

Algoritmo	Tendencia	R^2
Random	$0.20x + 0.02$	0.994
Fair	$0.31x - 0.02$	0.998
IncFair	$0.29x - 0.20$	0.991
HyperFair	$0.32x - 0.01$	0.995
$\mathcal{CFF}_k(0\%)$	$0.26x - 0.10$	0.994
$\mathcal{CFF}_k(20\%)$	$0.21x - 0.18$	0.994
$\mathcal{CFF}_k(50\%)$	$0.21x + 0.31$	0.993

Tabla 9.4: *Tendencia del tiempo de ejecución en función a la longitud de la ejecución.*

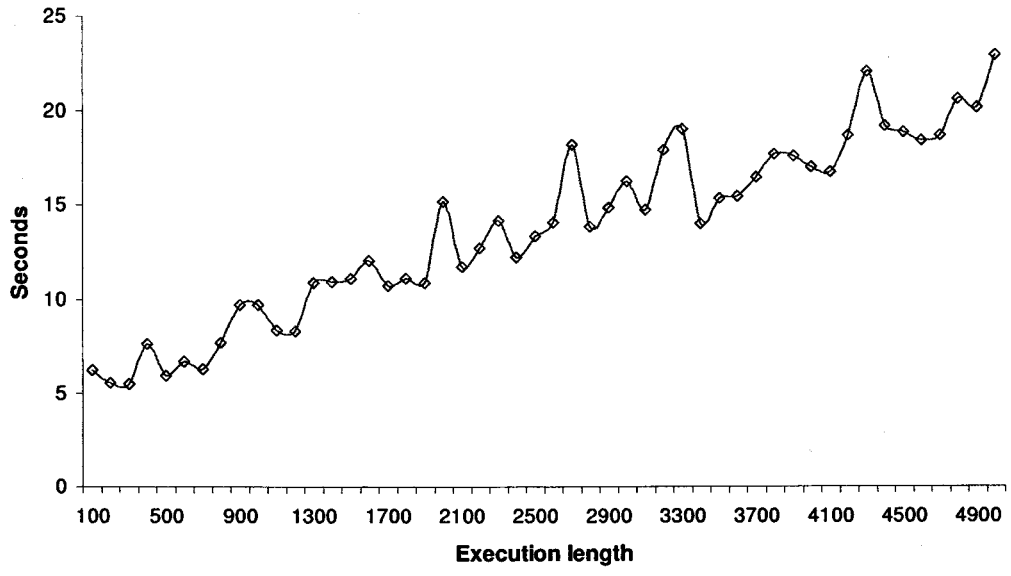


Figura 9.6: *Tiempo de selección de TB conforme la longitud de la ejecución crece.*

```
DINNER :: [||_{i=1}^{N} P_i || F_i], where
    P_i :: *[ Get_i⟨⟩ → eat; Rel_i⟨⟩; think ];
    F_i :: [ i mod 2 = 0 → sleep for one second ];
           *[ Get_i⟨⟩ → Rel_i⟨⟩ [] Get_{i+1}⟨⟩ → Rel_{i+1}⟨⟩ ].
```

Programa 9.2: *Un programa para forzar conspiraciones.*

algoritmo cuyo tiempo de selección se ve afectado por la longitud de la ejecución es *TB*. El motivo es que el tiempo que cada participante está monitorizando a otros participantes depende de muchos factores externos. Por ejemplo, un fallo de página o un pico de CPU pueden alargar la ventana de monitorización innecesariamente, y esto implica que las pequeñas interferencias que inevitablemente se producen durante una ejecución hagan que el tiempo de selección crezca con el tamaño de la ejecución. El autor propone reiniciar le tamaño de las ventanas de monitorización de vez en cuando, pero aún no existen resultados publicados que describan cómo hacerlo.

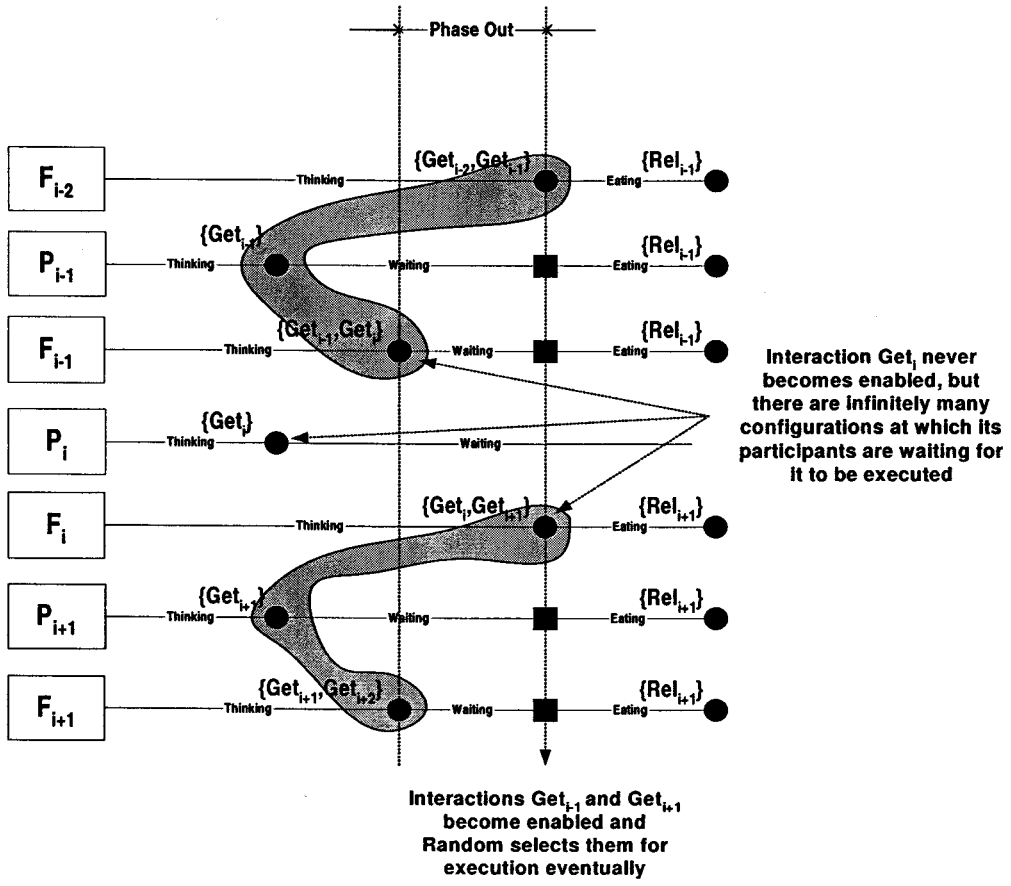


Figura 9.7: *Escenario para forzar conspiraciones.*

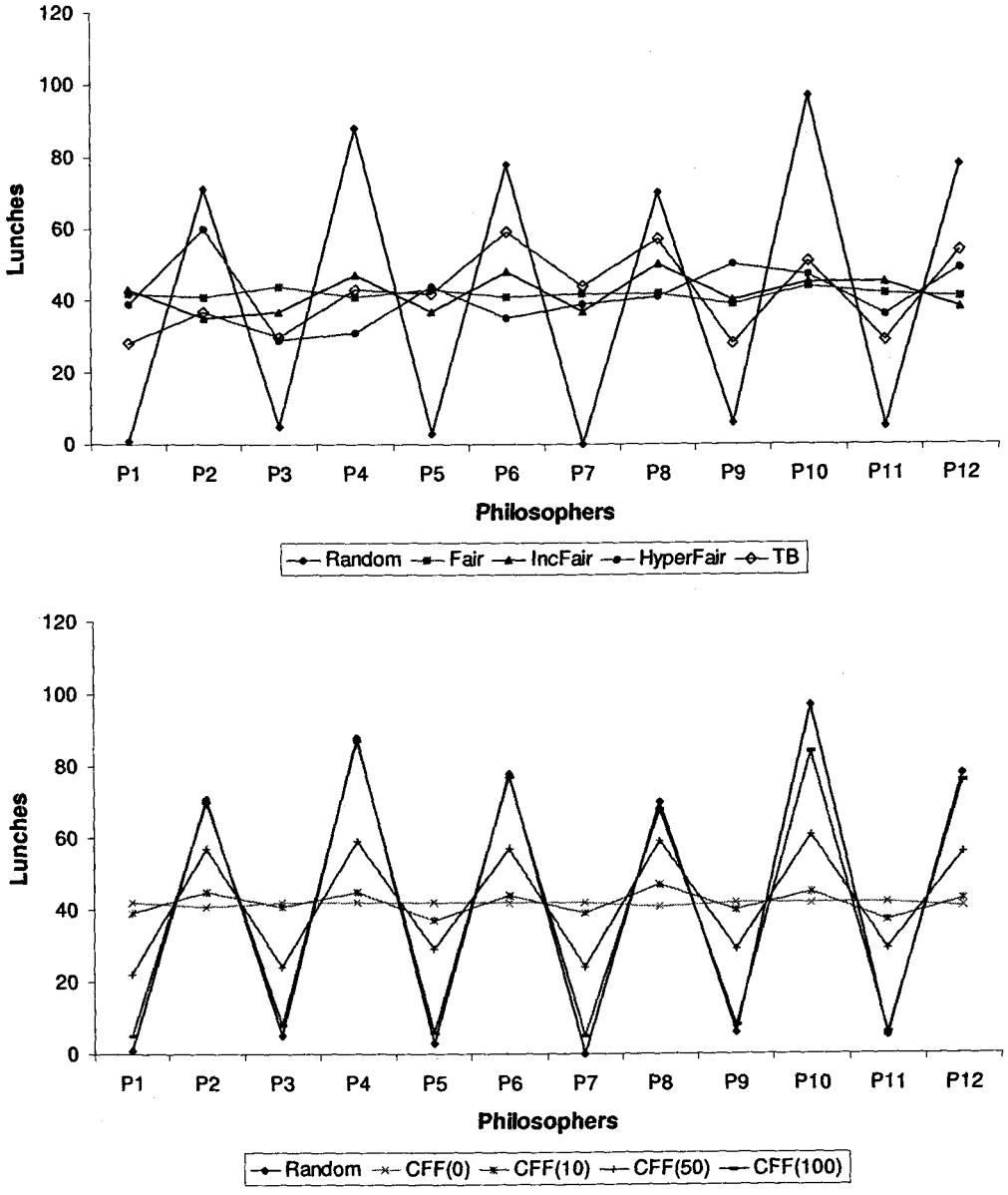


Figura 9.8: *Histogramas cuando las conspiraciones son forzadas.*

P_i	Random	Fair	HyperFair	IncFair	TB	$\mathcal{CFF}_k(0)$	$\mathcal{CFF}_k(10)$	$\mathcal{CFF}_k(50)$	$\mathcal{CFF}_k(100)$
P_1	1	42	39	43	28	42	39	22	5
P_2	71	41	60	35	37	41	45	57	70
P_3	5	44	29	37	30	42	41	24	8
P_4	88	41	31	47	43	42	45	59	87
P_5	3	43	44	37	42	42	37	29	6
P_6	78	41	35	48	59	42	44	57	77
P_7	0	42	39	37	44	42	39	24	5
P_8	70	42	41	50	57	41	47	59	68
P_9	6	39	50	40	28	42	40	29	8
P_{10}	97	44	47	45	51	42	45	61	84
P_{11}	5	42	36	45	29	42	37	29	6
P_{12}	78	41	49	38	54	41	43	56	76

Tabla 9.5: *Histogramas cuando las conspiraciones son forzadas.*

Algorithm	Execution time (ms)
Random	2033
Fair	14 108
HyperFair	12 042
IncFair	9855
TB	133 536
$\mathcal{CFF}_k(0)$	15 540
$\mathcal{CFF}_k(10)$	4054
$\mathcal{CFF}_k(50)$	3466
$\mathcal{CFF}_k(100)$	2007

Tabla 9.6: *Tiempos de ejecución cuando las conspiraciones son forzadas.*

9.3 Resolviendo conspiraciones

Para ver cómo $\mathcal{CFF}_k$ resuelve las conspiraciones, hemos estudiado el Programa §9.2. Es una versión de los filósofos comensales en el que el comportamiento de los tenedores ha sido cambiado para introducir un desfase entre los filósofos pares y los impares, a saber: los tenedores pares esperan un segundo antes de empezar a ofrecer interacciones, lo que implica que las interacciones de los filósofos impares siempre se habiliten antes que la de los impares. El diagrama temporal de la Figura §9.7 muestra como las interacciones Get_{i-1} y Get_{i+1} conspiran contra la interacción Get_i por el desfase inicial de un segundo ($i \in [1, N]$).

La Figura §9.8 muestra el histograma de comidas obtenido con este experimento y la Tabla §9.5 los valores concretos. Además, la Tabla §9.6 muestra el tiempo necesario para ejecutar los diferentes algoritmos. Fíjese que $\mathcal{CFF}_k$ para $k = 0$ resuelve las conspiraciones al igual que *Fair* pero el tiempo necesario es algo mayor; por el contrario, para $\mathcal{CFF}_k$ con $k = 10$ también se alivian los efectos negativos de las situaciones de conflicto al igual que *Fair* los hace (fíjese que la diferencia máxima de comidas entre dos filósofos es 8 con $\mathcal{CFF}_k$ y 5 con *Fair*) pero el tiempo de ejecución baja desde 15 540 hasta 4054 milisegundos.

Sin embargo, estas figuras no son concluyentes. Los gráficos de la Figura §9.9 muestran la distribución de comidas producida por los algoritmos objeto de estudio. A primera vista, podría parecer que el motivo por el que *Fair*, *Hyperfair* y *IncFair* obtienen una buena distribución de comidas es porque requieren que las interacciones estén estables (o congeladas con *HyperFair*) antes de seleccionar una de ellas para ejecución; sin embargo un estudio más detallado de los algoritmos revela que este comportamiento depende completamente de las características del generador de números aleatorios utilizado, a saber: su calidad y el rango de salida de los valores que produce. En los experimentos anteriores, hemos utilizado el generador de Casale [35] configurado para producir números en el intervalo $[0, 100]$. Cuando lo configuramos para producir números en el intervalo $[0, 2^{31} - 1]$ obtenemos la distribución de comidas mostradas en las Figuras §9.10.(a) y §9.10.(b). Estos resultados son claramente no deseables ya que las distribuciones de comidas son aún peores que si no implementásemos justicia (*Random*) consumiendo, además, mucho más tiempo. El comportamiento de *TB* parece más adecuado pero depende también del generador de números aleatorios. Si sustituimos éste por el generador que la plataforma .NET proporciona los resultados que obtenemos son los que se muestran en la Figura §9.10.(c). Esta distribución es similar a la que produce *Random*, pero su conste computacional es de muchísimo peor.

Al contrario que estas propuestas, la nuestra no depende del generador

```
DINNER :: [||_{i=1}^{N} P_i || F_i], where
    P_i :: *[ Get_i⟨⟩ → eat; Rel_i⟨⟩; wait for Think(i) units of time ];
    F_i :: *[ Get_i⟨⟩ → Rel_i⟨⟩ [] Get_{i+1}⟨⟩ → Rel_{i+1}⟨⟩ ].
```

Programa 9.3: *Un programa de ejemplo para sintonizar* k.

de números aleatorios. Si $k = 0$, no se permite que ocurra ninguna situación potencial de conspiración, así el sistema es controlado de forma que cada filósofo come tantas veces como el resto, lo cual era el comportamiento esperado. Conforme el valor de k se incrementa, más situaciones de conspiraciones potencial son permitidas, aunque éstas nunca pueden exceder el valor de k. Así, cambiando el valor de k podemos ajustar la noción a las características del sistema objeto de estudio.

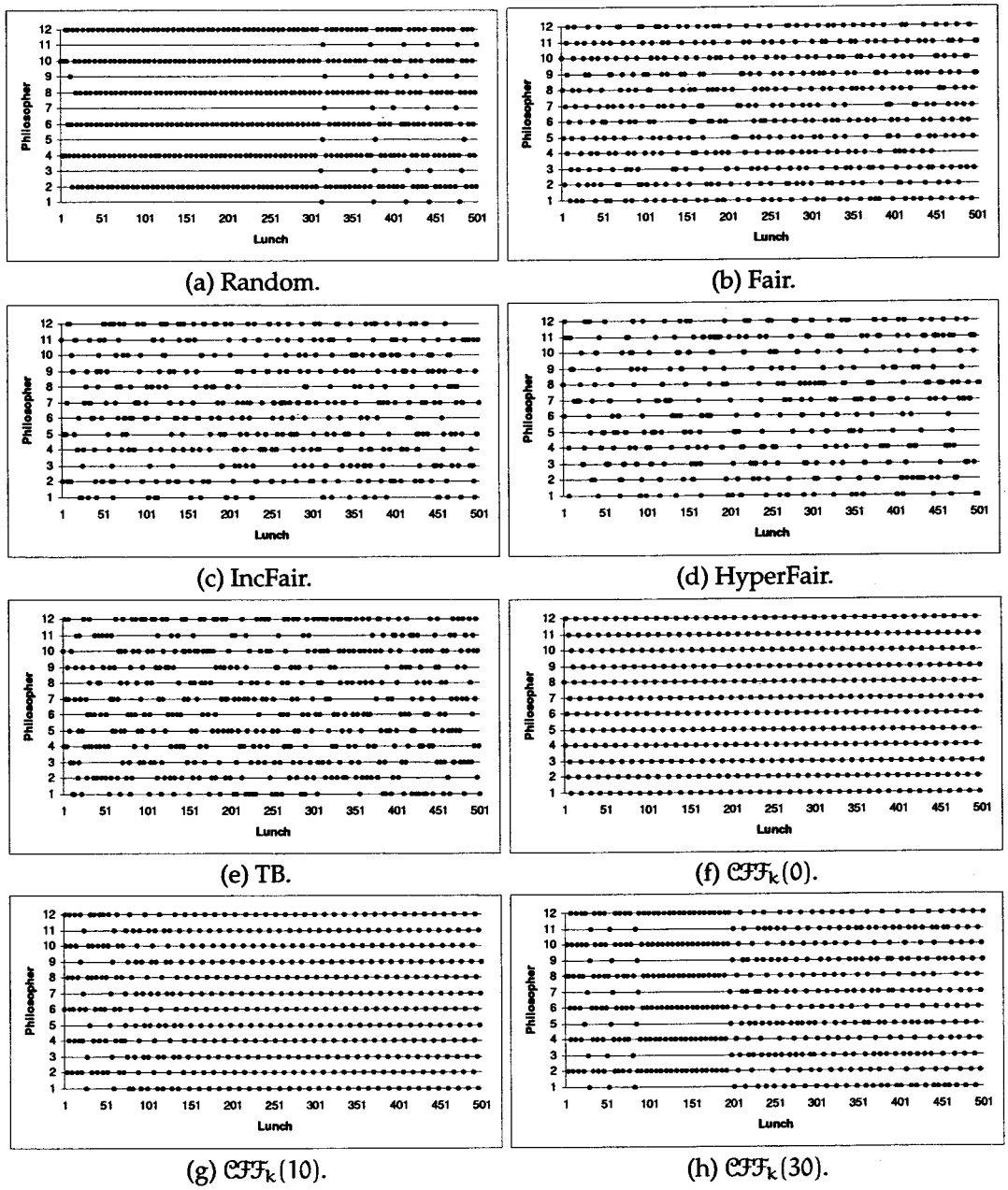
9.4 Ajuste de k para los filósofos comensales

Para ajustar el valor de k y analizar su influencia en la ejecución, hemos estudiado el Programa §9.3. Ahora, el tiempo que los filósofos pasan pensando viene dado por la función Think. En otras palabras, asumimos que el tiempo que los recursos son usados es despreciable respecto al tiempo que los procesos esperan a obtenerlos. Fíjese que $\frac{1}{\mathrm{Think}(i)}$ es la frecuencia con la que el filósofo i solicita sus tenedores.

9.4.1 Estudio analítico

Sea L_i el número de veces que el filósofo P_i come durante una ejecución, y sea L la longitud de esta ejecución; entonces, P_i está $L_i\mathrm{Think}(i)$ unidades de tiempo pensando y $\sum_{i=1}^{N} L_i = L$. Por ejemplo, si existen tres filósofos y $\mathrm{Think}(i) = i$, entonces el filósofo P_1 está L_1 unidades de tiempo pensando, P_2 piensa por $2L_1$ unidades de tiempo y el filósofo P_3 piensa $3L_1$ unidades de tiempo. Fíjese que suponiendo un entrelazado ideal y una ejecución lo suficientemente larga podemos asumir que todos los filósofos están pensando el mismo tiempo, es decir, $L_1 = 2L_1 = 3L_1$. Este razonamiento puede ser extendido para cualquier función Think y para todo par de filósofos, teniéndose

$$\forall i, j \in [1, N] \cdot (L_i\mathrm{Think}(i) = L_j\mathrm{Think}(j))$$



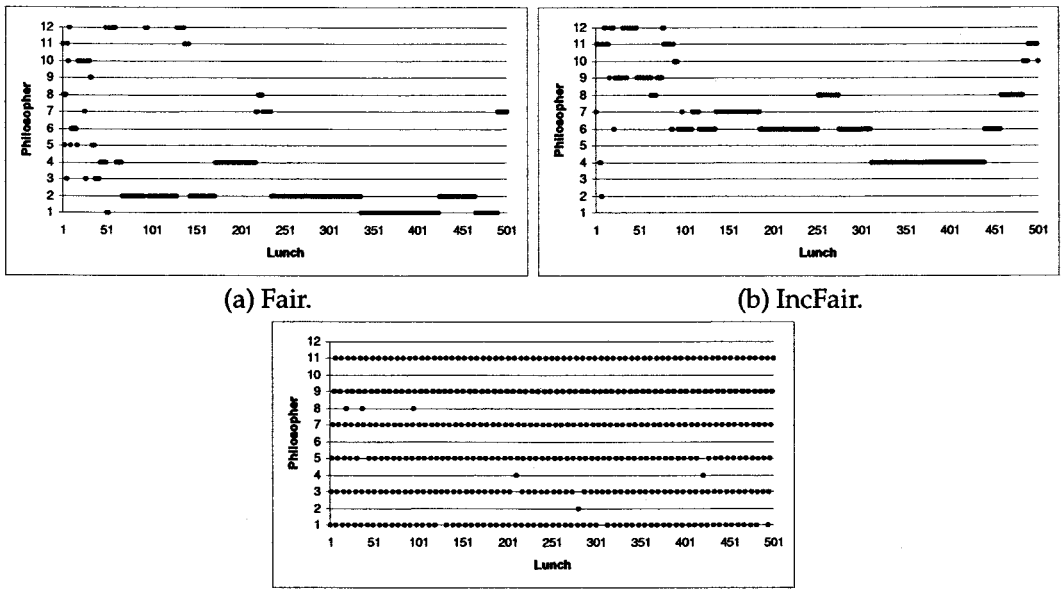
(a) Random. (b) Fair.

(c) IncFair. (d) HyperFair.

(e) TB. (f) $\mathcal{CFF}_k(0)$.

(g) $\mathcal{CFF}_k(10)$. (h) $\mathcal{CFF}_k(30)$.

Figura 9.9: *Distribución de comidas cuando las conspiraciones son forzadas.*



(a) Fair. (b) IncFair.

(c) TB.

Figura 9.10: *Distribución de comidas usando distintos generadores de números aleatorios.*

Esta formula puede escribirse como el siguiente sistema de ecuaciones:

$$\begin{aligned} L_1 Think(1) &= L_2 Think(2) \\ L_1 Think(1) &= L_3 Think(3) \\ &\vdots \quad \vdots \quad \vdots \\ L_1 Think(1) &= L_N Think(N) \end{aligned}$$

Así, podemos expresas las veces que P_i come como una función del número de veces que P_1 lo hace para despejar la variable L_1 ($i \in [1, N]$):

$$L_2 = L_1 \frac{Think(1)}{Think(2)}, L_3 = L_1 \frac{Think(1)}{Think(3)}, \ldots, L_N = L_1 \frac{Think(1)}{Think(N)}$$

Sumando en ambos lados se tiene:

$$\sum_{i=2}^{N} L_i = L_1 Think(1) \sum_{i=2}^{N} \frac{1}{Think(i)}$$

Degrees of freedom	Confidence level α				
	0.995	0.990	0.975	0.95	0.900
5	0.412	0.554	0.831	1.145	1.610
6	0.676	0.872	1.237	1.635	2.204
7	0.989	1.239	1.690	2.167	2.833
8	1.344	1.647	2.180	2.733	3.490
9	1.735	2.088	2.700	3.325	4.168
10	2.156	2.558	3.247	3.940	4.865
11	2.603	3.053	3.816	4.575	5.578
12	3.074	3.571	4.404	5.226	6.304
13	3.565	4.107	5.009	5.892	7.041
14	4.075	4.660	5.629	6.571	7.790
15	4.601	5.229	6.262	7.261	8.547
20	7.434	8.260	9.591	10.851	12.443
30	13.787	14.953	16.791	18.493	20.599
40	20.707	22.164	24.433	26.509	29.051
50	27.991	29.707	32.357	34.764	37.689
60	35.534	37.485	40.482	43.188	46.459
70	43.275	45.442	48.758	51.739	55.329
80	51.172	53.540	57.153	60.391	64.278
90	59.196	61.754	65.647	69.126	73.291
100	67.328	70.065	74.222	77.929	82.358
150	109.142	112.668	117.985	122.692	128.275
200	152.241	156.432	162.728	168.279	174.835

Tabla 9.7: *Algunos valores de* χ^2.

Como $\sum_{i=2}^{N} L_i = L - L_1$, L_1 puede ser expresado de la siguiente forma:

$$L_1 = \frac{L}{1 + \mathrm{Think}(1) \sum_{i=2}^{N} \frac{1}{\mathrm{Think}(i)}}$$

9.4.2 Un método simple para seleccionar k

Una vez que hemos calculado la distribución de comidas esperada, podemos utilizar un método de bondad de ajuste para calcular el primer valor de k que produce una ejecución cuya distribución de comidas no difiera demasiado de la esperada. Existen muchos métodos en la bibliografía [63, 94], pero nosotros hemos seleccionado el método de Chi cuadrado porque es simple, potente

y genérico. Puede ser aplicado a cualquier sistema con más de dos interacciones que se ejecuten al menos cinco veces, lo que supone unas restricciones muy débiles.

Utilizando este Test, la forma más simple de calcular el primer valor de k que permite ajustarse a la distribución esperada es:

Paso 0: iniciar k a 0 y α a un nivel de confianza estáqndar. Sea Exp_i el número de veces esperado que la interacción x_i se debe ejecutar para todo $i \in [1, N]$.

Paso 1: ejecutar una conjunto de experimentos y obtener el valor de Obs_i como la media aritmética de las veces que la interacción x_i se ejecuta para todo $i \in [1, N]$.

Paso 2: Calcular X^2 de la siguiente forma:

$$X^2 = \sum_{i=1}^{N} \frac{(Obs_i - Exp_i)^2}{Exp_i}$$

Paso 3: Comparar el valor de X^2 con $\chi^2(N-1, \alpha)$, donde χ^2 se obtiene de la tabla estándar parcialmente mostrada en la Tabla §9.7. Si $X^2 \geq \chi^2(N-1, \alpha)$ entonces la ejecución no es lo suficiente buena y necesitamos incrementar el valor de k y volver al Paso 1; en caso contrario, implicaría que hemos encontrado el primer valor de k que produce una ejecución que se ajusta a la esperada con un nivel de confianza α.

Por ejemplo, supongamos que $Think(i) = (i-6)^2 + 1$ y que el nivel de confianza que queremos es $\alpha = 0.995$. La distribución de comidas esperada en un programa con 12 filósofos y 500 comidas es la que se muestra en la segunda fila de la Tabla §9.8, teniendo que la Tabla §9.9 muestra un extracto de de las distintas distribuciones de comidas obtenidas para distintos valores de k y el valor de X^2.

Como tenemos 12 interacciones *Get*, necesitaremos iterar hasta que obtengamos el primer valor de X^2 menor o igual que $\chi^2(11, 0.995) = 2.603$, lo que ocurre cuando $k = 161$. Fíjese que este valor no es el mejor de los posibles ya que, por ejemplo, $X^2 = 0.37$ si $k = 170$, lo que implica que esta ejecuión se ajusta mejor. En general, no es de interés obtener el valor óptimo, sólo un valor que nos permita considerar que una ejecución se aproxima a lo esperado con un nivel de confianza alto, como es $\alpha = 0.995$.

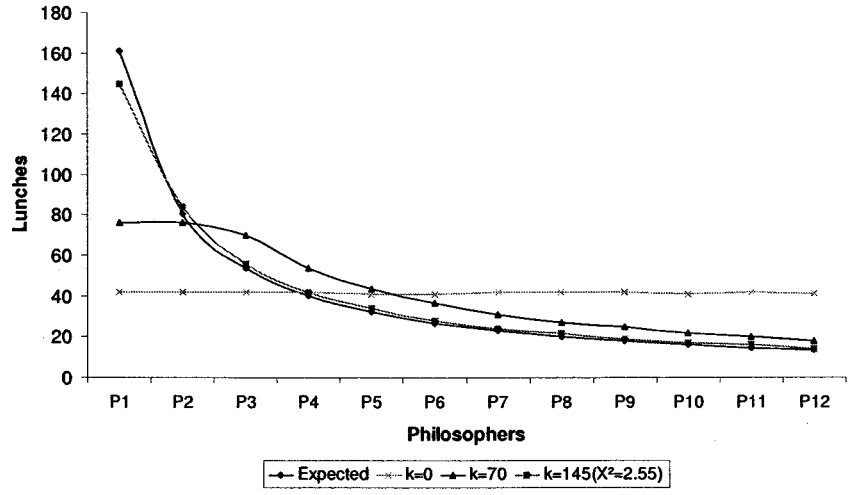
Las Figuras §9.11.(a), §9.11.(b), §9.12.(c) y §9.12.(d) muestran varias gráficas en las cuales hemos representado las distribuciones esperadas para varias funciones Think y varios valores de k.

Think(i)	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
i	161	80	53	40	32	26	23	20	17	16	14	13
$(i-6)^2+1$	7	10	18	35	89	177	89	35	18	10	7	5
$\frac{1}{\sqrt{(i)}}$	17	24	29	34	38	41	45	48	51	54	56	59
$\frac{i}{\sin\frac{(2i+1)\pi}{2}}+12$	16	20	14	25	12	32	11	45	10	75	9	226

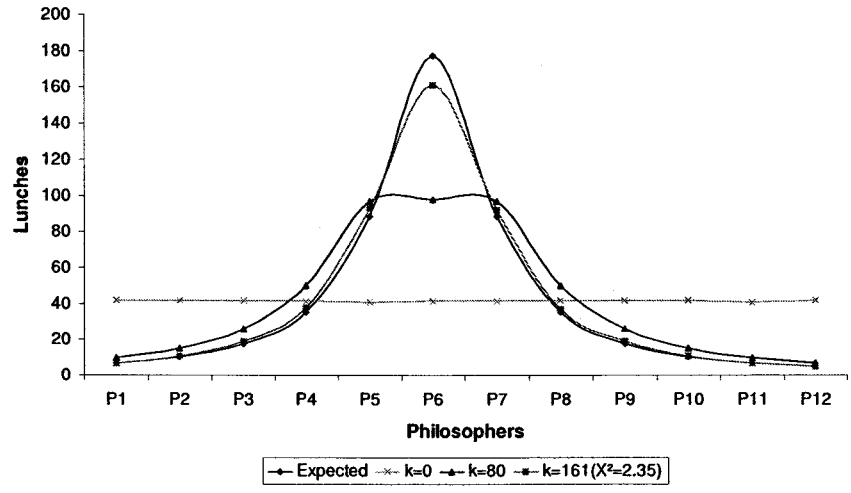
Tabla 9.8: *Número de comidas esperado para algunas funciones* Think.

k	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	X^2
0	42	42	42	42	41	42	42	42	42	42	41	42	1055.71
10	37	39	42	44	47	48	47	44	42	39	37	35	818.55
20	34	37	41	46	52	52	52	46	40	37	34	29	658.35
30	25	33	40	47	64	65	64	47	40	33	25	18	379.54
40	20	30	43	53	65	65	65	52	43	31	20	14	318.29
50	13	20	34	58	81	81	80	58	35	20	13	9	146.76
60	11	17	28	55	90	91	90	55	28	17	11	7	89.95
70	11	17	28	51	91	92	91	56	28	17	11	8	87.37
80	10	15	26	50	97	98	97	50	26	15	10	7	64.71
90	9	14	24	48	101	102	101	48	24	14	9	6	52.90
100	9	14	23	45	104	105	104	45	23	14	9	6	47.24
110	8	13	21	42	105	116	105	43	21	13	8	6	33.24
120	8	13	21	42	103	120	103	42	21	13	8	6	28.78
130	8	12	21	41	100	130	101	41	21	12	8	5	19.63
140	8	12	20	40	98	140	98	39	20	12	8	5	12.23
150	8	12	19	38	95	150	95	38	20	12	8	5	6.75
160	7	11	19	38	93	160	93	37	19	11	7	5	2.63
161	7	11	19	38	93	161	92	37	19	11	7	5	2.35
170	7	11	18	36	90	171	90	36	18	11	7	5	0.37

Tabla 9.9: *Encontrando un valor adecuado para* k *si* $\mathtt{Think}(i) = (i-6)^2+1$.

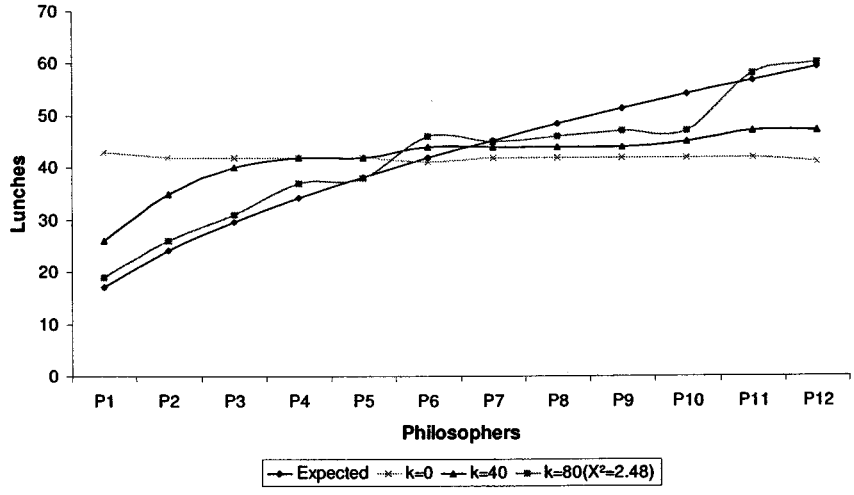


(a) Think(i) = i.

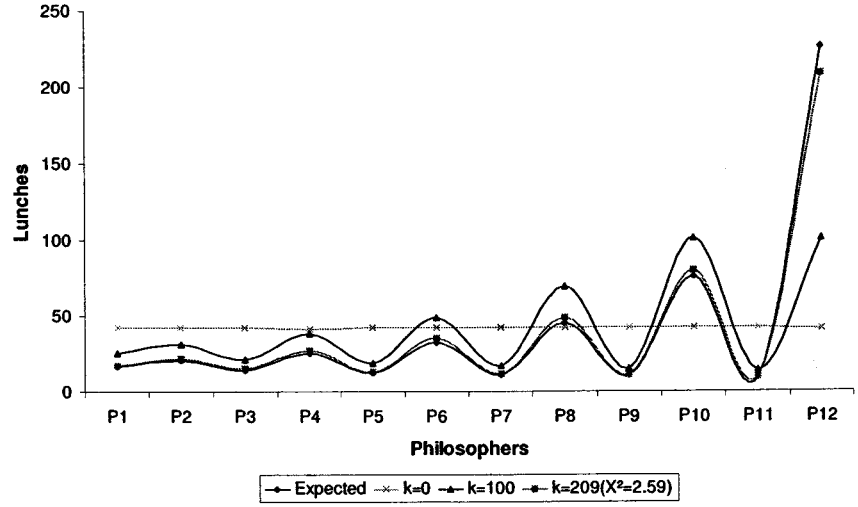


(b) Think(i) = $(i-6)^2 + 1$.

Figura 9.11: *Ajuste de* k-conspiracy-free fairness *(I).*



(c) Think(i) $= \frac{1}{\sqrt{i}}$.



(d) Think(i) $= \frac{i}{\sin \frac{(2i+1)\pi}{2}} + 12$.

Figura 9.12: *Ajuste de* k-conspiracy-free fairness *(II).*

Parte IV

Notas finales

Capítulo 10

Conclusiones y trabajo futuro

When people agree, it is only in their conclusions; their reasons are always different.

Jorge A.N. de Santayana, 1863–1952
Spanish philosopher

La justicia es el concepto clave que permite a los programadores abstraerse de los detalles de bajo nivel relativos a la implementación y la ejecución de sus programas. Sin embargo, las nociones habituales en el contexto de las interacciones multipartitas adolecen de dos problemas conocidos como finitud justa y conspiraciones, lo que constituye un gran obstáculo a la hora de su aplicación práctica. Las implementaciones de algunas nociones resuelven el problema de las conspiraciones cuando se utiliza un generador de números aleatorio adecuado, pero, en general, no pensamos que esto sea lo más adecuado ya que hace depender a nuestros programas del planificador utilizado para ejecutarlo.

Nuestro objetivo es esta tesis ha sido proporcionar una nueva noción de selección justa capaz de resolver ambos problemas simultáneamente, de forma que los programadores puedan pensar en cómo sus programa se ejecutarán de forma independiente a los detalles de bajo nivel que se empleen en la planificación del mismo. En esta tesis, hemos mostrado que nuestra noción puede ser implementada eficientemente. Además, nuestro estudio experimental demuestra que la noción puede adaptarse con facilidad al programa objeto de estudio. Como se dijo en el resumen, la selección justa implica que toda interacción sea tratada de acuerdo a las reglas establecidas en el programa al que pertenece. Esto significa que nuestro objetivo no es forzar a que toda interacción se ejecute tantas veces como el resto de interacciones, aunque esto

sea lo que pudiera parecer más "justo". De todas las nociones que hemos analizado en esta tesis doctoral, k-*conspiracy-free* es la única que se adapta a las características del sistema ajustando el valor de k de la forma adecuada, es decir, de forma independiente al planificador usado para implementar la justicia. Para mostrar la viabilidad de nuestra propuesta, hemos proporcionado un marco teórico para definirla rigurosamente, así como una implementación eficiente utilizando planificadores finitos.

De cualquier forma, los resultados mostrados en esta tesis doctoral no pueden ser visto como el final de un camino, sino como la motivación para otras líneas de investigación. De entre los distintos aspectos que quedan abiertos o que puede ser mejorados, pensamos que quizás el estudio de cómo k puede ser calculado en tiempo de ejecución es uno de los más interesantes. En la actualidad, es posible determinar cuál es el primer valor de k que mejor controla una ejecución finita cuya longitud se conoce de antemano. Sería interesante se capaz de predecir si una conspiración puede ocurrir y ajustar el valor de k dinámicamente para resolverla.

Parte V

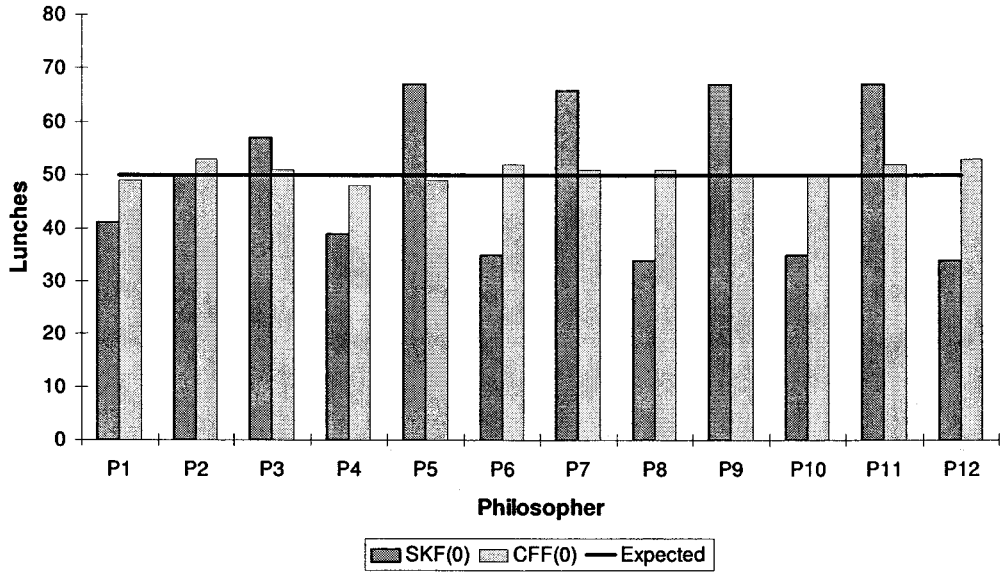
Apéndices

Apéndice A

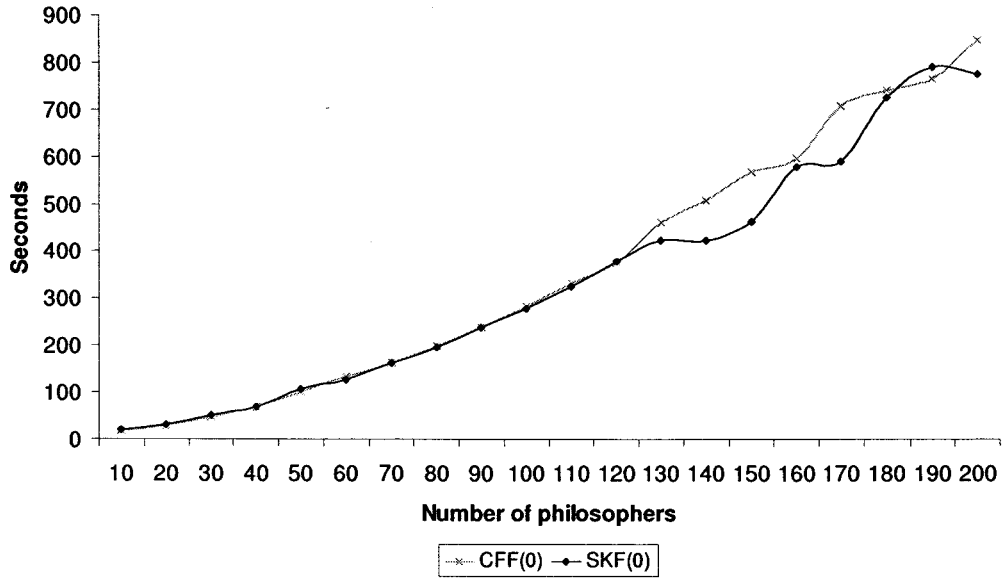
Strong k-*fairness*

Una versión preliminar de la noción k-*conspiracy-free* fue presentada en la conferencia Euro-Par'02 [115]. Allí, introdujimos una nueva noción llamada *strong* k-*fairness* que difiere de la propuesta actual en que detecta las situaciones de conspiración potencial analizando las interacciones que están enlazadas en tiempo de ejecución en lugar de las interacciones enlazadas estáticamente.

A pesar de ser similares, los resultados que se obtienen son muy distintos ya que k-*conspiracy-free* resuelve el problema de las conspiraciones y *strong* k-*fairness* sólo lo alivia, es decir, existen ejecuciones que son *strong* k-*fair* y contienen conspiraciones. La Figura §A.1 muestra los resultados obtenidos en una ejecución de los filósofos comensales en la que todos los filósofos deberían de tener la mismas oportunidades de comer. Las figuras muestran que k-*conspiracy-free* nos permite controlar esta ejecución mejor que *strong* k-*fairness* y que el efecto en el rendimiento no es significante.



(a) Distribución de comidas.



(b) Tiempo de ejecución.

Figura A.1: *Comparación entre* k-*conspiracy-free y strong* k-*fairness.*

Apéndice B

El lenguaje IP

B.1 Una idea general

El lenguaje IP proporciona una notación de alto nivel de abstracción para modelar y diseñar complejas cooperaciones en sistemas concurrentes y distribuido. Fue diseñado por Nissim Francez y Ira R. Forman, y está construido sobre la base SCRIPT y RADDLE [54, 51]. Ambas propuestas dieron lugar a IP cuando los autores se conocieron en el *MCC's Software Technology Program* [53].

IP proporciona tres constructores, a saber: interacciones multipartitas, equipos y superposición. Las interacciones multipartitas son adecuadas para describir cooperaciones complejas entre un conjunto fijo de procesos participantes; son construcciones de alto nivel que permiten a los programadores abstraerse de los detalles de bajo nivel concernientes a los protocolos necesarios para la sincronización y comunicación ya que estos pueden ser generados automáticamente. Los equipos nos permiten encapsular y reutilizar patrones de cooperación comunes en los que los procesos pueden participar bajo un papel que les permite interactuar con un número arbitrario de procesos. La superposición es un constructor que permite a un proceso observar y/o controlar la comportamiento de otros procesos; es la piedra angular del diseño en capas en el que las distintas partes puede ser reutilizadas fácilmente superponiendo un conjunto de procesos.

IP_{CORE} es un subconjunto de IP que proporciona sólo interacciones multipartitas, es decir, no proporcionan ningún mecanismo de modularización o abstracción mas que los procesos. En esta tesis, nos hemos centrado en IP_{CORE} en lugar de IP porque nuestro objetivo es estudiar la justicia en el contexto de

las interacciones multipartitas, y el modelo de interacción de IP_{CORE} es conceptualmente más simple que el modelo de IP, aunque ambos son muy similares. Los resultados obtenidos en esta tesis doctoral son fácilmente adaptables a IP o a cualquier lenguaje que soporte un modelo de interacción similar, pero estos detalles están fueran del alcance de nuestros objetivos.

Hasta lo que nosotros sabemos, existen sólo dos compiladores de IP disponibles. Una de ellos fue diseñado por Francez y Forman y se ejecuta sobre plataformas UNIX generando código para MEIKO, TRANSPUTER o i860. El otro fue diseñado por el autor de esta tesis doctoral como parte de su proyecto final de carrera y se ejecuta sobre plataformas UNIX y genera código SR [7]. Recientemente, varias implementaciones del modelos de interacción de IP han aparecido en la bibliografía. Por ejemplo, una propuesta basa en CORBA es presentada en la Referencia [105] y nosotros describimos nuestra implementación basada en $J^\sharp$ en la Referencia [113]. *Cf.* Sección §2.1 para más detalles.

B.2 La sintaxis

En IP_{CORE}, los sistemas son modelados como una conjunto de procesos secuenciales cooperativos cuyas relaciones están basadas en interacciones multipartitas. Cada proceso ejecuta un cuerpo que está compuesto de una secuencia de instrucciones. La forma general de un programa en IP es la siguiente:

SYSTEM :: $[P_1 \parallel P_2 \parallel \ldots \parallel P_n]$, **where**
P_1 :: *Lista de instrucciones*$_1$;
P_2 :: *Lista de instrucciones*$_2$;
⋮
P_n :: *Lista de instrucciones*$_n$.

Aunque no es normal, un programa puede estar compuesto de un único proceso. En este caso la cabecera puede ser omitida y el programa es escrito de la siguiente forma:

SINGLE_PROCESS :: *Lista de instrucciones.*

Si el nombre del procesos no es significante, el programa puede reducirse a la mínima expresión de la forma:

Lista de instrucciones.

Las instrucciones que IP_{CORE} proporciona son asignaciones, interacciones, alternativas e iterativas múltiples. Como es habitual, las asignaciones son de la forma $x := e$, donde x es una variable local y e es una expresión sobre el estado local del proceso que ejecuta la instrucción. La asignación vacía es referida

como **skip**, y puede utilizarse en cualquier sitio donde una lista de instrucciones vacía sea necesaria. Operadores y literales arbitrarios son permitidos para construir expresiones siempre y cuando su semántica esté bien definida en el contexto del programa. Las variables no necesitan ser declaradas explícitamente, pero es habitual hacerlo como un comentario entre llaves.

Las instrucciones de interacción son de la forma $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, donde A es el nombre de la interacción y $x_1 := e_1; \ldots; x_n := e_n$ es una secuencia opcional de asignaciones referida como parte de comunicación. Para que una interacción se habilite, todos los participantes tienen que ofrecerla a través de una instrucción de interacción. Una vez que una interacción se habilita, puede ser ejecutada tan pronto sus participante lo confirmen, es decir, que no hayan confirmado otra interacción. La parte de comunicación permite a los procesos obtener datos los unos de los otros a través de A, ya que cada e_i se refiere a variables locales. Varias mejoras a este mecanismo de comunicación se han propuesto en la bibliografía [34, 36, 37, 130].

Las instrucciones alternativas múltiples son de la forma

$[[]_{i=1}^{n}$ G_i $\rightarrow$ S_i$]$

donde cada G_i es un guarda y S_i es la lista de instrucciones que se ejecuta cuando el guarda se habilita. Los guardas son B &$A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, donde B es una expresión lógica y el resto una instrucción de interacción. Se habilitan, es decir, sus correspondientes instrucciones pueden ser ejecutadas, si y sólo si la expresión lógica es evaluada a verdadero y la interacción A se habilita. Si B & es omitido, entonces se interpreta como *true* &; si &a$\langle \overline{x := e} \rangle$ es omitido, se interpreta como &$\langle\rangle$, donde $\langle\rangle$ es una interacción local, unipartita y anónima. Fíjese que ambas partes del guardes no pueden ser omitidas al mismo tiempo.

La instrucciones iterativas múltiples son de la forma

*$[[]_{i=1}^{n}$ G_i $\rightarrow$ S_i]

La semántica consiste en repetir la instrucción alternativa múltiple hasta que ninguno de los guardas sea verdadero. Fíjese que pueden no terminar si alguno de los guardas es siempre evaluado a verdadero.

B.3 Algunos ejemplos

En esta sección, presentamos algunos ejemplos para ilustrar las características básicas de IP$_{\text{CORE}}$. Nuestro objetivo es proporcionar al lector un conjunto de programas que puedan ayudarle a entender las instrucciones IP$_{\text{CORE}}$ más

facilmente sin entrar en los detalles de la sincronización y comunicación multipartita. Véase la Sección §2.4 para ejemplos más elaborados.

Ejemplo B.3.1 *El siguiente programa almacena el valor máximo del vector* v *en la variable* result. *Fíjese que las variables aparecen en un comentario, aunque esto no es estrictamente necesario. Cómo la variable* v *no es inicializada, asumimos que sus valores iniciales son aleatorios. Ninguna hipótesis puede hacerse sobre estos valores.*

```
{ result, i: natural; v: array (1..10) of natural }
result := 0; i := 1;
*[ i ≤ 10 →
    [ v(i) > result → result := v(i) ];
   i := i + 1
].
```

Ejemplo B.3.2 *Le siguiente programa calcula el máximo común divisor de* X *e* Y *usando el algoritmo de Euclides. Fíjese que no especificamos los valores iniciales de* X *e* Y *ya que no son importantes en este contexto, pero ambas variables tienen que tener un valor antes de que el bucle principal comience a ejecutarse.*

```
X := a value;
Y := a value;
*[ X < Y → Y := Y - X
[] X > Y → X := X - Y
]
gcd := X;
```

Ejemplo B.3.3 *En el siguiente programa, tenemos dos procesos cuyo objetivo es intercambiar el valor de sus estados locales a través de la interacción* Exchange. *Fíjese que dos variables no pueden tener el mismo nombre, aunque pertenezcan a procesos distintos. El motivo es que* IP_{CORE} *sólo proporciona un espacio global de nombres para las variables.*

```
EXCHANGE :: [P || Q], where
        P :: { x: natural }
             x := a value;
             Exchange⟨x := y⟩;
        Q :: { y: natural }
             y := a value;
             Exchange⟨y := x⟩.
```

Fíjese que no es necesario utilizar una variable temporal ya que la semantica de la comunicación multipartita garantiza que no ocurren condiciones de carrera mientras una interacción está siendo ejecutada. Cuando la interacción Exchange *tiene lugar, un estado combinado formado por una copia de los estados locales de todos los participantes se forma. Las expresiones en la parte de comunicación sólo pueden leer de este*

estado combinado, es decir, los datos leídos son anteriores a la ejecución de la interacción. No hay forma de que un proceso conozca cómo cambia el estado de otro proceso durante la ejecución de una interacción, lo que previene que las condiciones de carrera ocurran.

Ejemplo B.3.4 *En el siguiente programa, hemos modelado un sistema típico con un productor y un consumidor que necesitan comunicarse indirectamente a través de un búfer de tamaño fijo. Este es un problema clásico de coordinación en la bibliografía.*

```
PROD-CONS :: [Producer || Consumer || Buffer], where
    Producer :: { x: Item }
                *[ true → x := Produce a new item; Produce⟨⟩ ];
    Consumer :: { y: Item }
                *[ true → Consume⟨y := Head(q)⟩; Consume item y ];
      Buffer :: { size: natural; MAX: natural; q: Queue of Item }
                size := 0; MAX := 10; q := NewQueue();
                *[ size < MAX & Produce⟨q := Enqueue(q, x)⟩ → size := size + 1
                [] size > 0 & Consume⟨q := Dequeue(q)⟩ → size := size - 1
                ].
```

Fíjese que cuando una interacción Consume *es ejecutada, tanto el consumidor como el búfer necesitan tener acceso a la cola: el primero para consultar su cabeza, y el segundo para eliminarla. Ya que ninguna expresión en la parte derecha de la instrucción de asignación de la parte de comunicación de la interacción es evaluada en el estado combinado, las condiciones de carrera tampoco pueden ocurrir en este ejemplo.*

B.4 Programas IP en forma normal

La sintaxis de IP_{CORE} es simple, aunque la capacidad de combinar todas instrucciones ortogonalmente puede dar lugar a programas más complejos que sean más difíciles de analizar. En estos casos, estos pueden ser transformados a su equivalente forma normal que tiene el mismo comportamiento, pero el cuerpo de los procesos es "aplanado" de la siguiente forma:

$$S_0; *[[]_{j=1}^{n_i} B_j \;\&\; a_j\langle v_j := e_j\rangle \rightarrow S_j]$$

donde S_0 es una instrucción de inicializacion que no contiene ni interacciones ni bucles, y cada S_j es una lista de instrucciones que sólo contiene cálculo local. Los autores de IP definen otras formas normales, pero nosotros nos centramos sólo en está ya que es requerida para la transformación *hyperfair*, véase Sección §4.3.

Apéndice C

Notación

C.1 Método de Plotkin

Hemos utilizado el popular método de Plotkin para definir nuestros algoritmos [107] ya que es simple y potente. Se basa en reglas de inferencia de la forma

$$\frac{\text{Antecedente}}{\text{Consecuente}} \text{ SI Aplicabilidad condiciones}$$

DONDE Definiciones

Para definir un algoritmo utilizando este método, es necesario identificar los datos sobre los que se trabaja y modelarlos como una tupla a la que habitualmente nos referimos como configuración del algoritmo.

Por ejemplo, para modelar el sistema del productor/consumidor, necesitaremos configuraciones de la forma (p, c, τ), donde p denota el estado del productor, c el estado del consumidor y τ la cola de tamaño fijo que almacena los elementos que están listos para ser consumidos. Obviamente, también necesitamos un par de reglas para referidas como $\longrightarrow_{\text{PROD}}$ y $\longrightarrow_{\text{CONS}}$ para describir como el productor y el consumidor cambian sus estados respectivamente. Estas reglas pueden dejarse sin especificar ya que al nivel de abstracción que nos movemos no es necesario profundizar en su semántica. Utilizando esta información, el sistema puede describirse con las siguientes reglas:

La regla de producción, que controla cómo los nuevos elementos son introducimos en la cola cuando hay espacio en ella. (Suponemos que MAX denota el número máximo de elementos que se pueden almacenar en τ.)

$$\frac{p \xrightarrow{i}_{\text{PROD}} p'}{(p, c, \tau) \longrightarrow_{\text{CONS}} (p', c', \tau')} \text{ SI } |\tau| < \text{MAX}$$

DONDE $c' \triangleq c$

$\tau' \triangleq \text{Enqueue}(\tau, i)$

$p \xrightarrow{i}_{\text{PROD}} p'$ significa que el productor produce un elemento i y que esto provoca una transición del estado p al estado p'. Fíjese que la regla puede aplicarse cuando $|\tau| < \text{MAX}$; de lo contrario, el productor tiene que esperar hasta que el consumidor elimine algún elemento de la cola.

La regla de consumición, que controla cómo los elementos son eliminados de la cola.

$$\frac{c \xrightarrow{i}_{\text{CONS}} c'}{(p, c, \tau) \longrightarrow_{\text{CONS}} (p', c', \tau')} \text{ SI } |\tau| > 0 \wedge i = \text{Head}(\tau)$$

DONDE $p' \triangleq p$

$\tau' \triangleq \text{Dequeue}(\tau)$

Las reglas de inferencia son de muy alto nivel, y permiten modelar algoritmos concurrentes con precisión, ya que pueden ser vistas como las descripción de las acciones atómicas que pueden ser ejecutadas en el sistema concurrente. El comportamiento o el algoritmo puede ser visto como el conjunto de todos los posibles entrelazados permitidos por las reglas de inferencia usadas para describirlo.

C.2 Autómatas de Büchi

Los autómatas de Büchi fueron introducidos en la Referencia [25] como una herramienta adecuada para reconocer palabras infinitas utilizando autómatas finitos. Pueden ser vistos como 4-tuplas de la forma (S, I, δ, F), donde S denota un conjunto finito de estados, $I \subseteq S$ un conjunto de estados iniciales, $\delta \subseteq S \times S$ una relación de transición y $F \subseteq S$ un conjunto de estados finales. Se dice que una secuencia de estados es aceptada si y sólo si contiene infinitos estados finales.

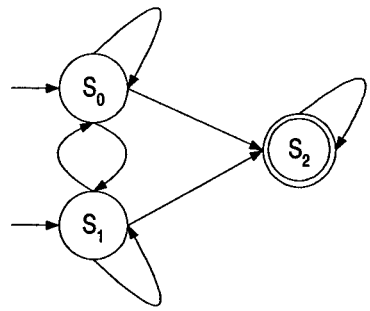


Figura C.1: *Autómata de Büchi para reconocer* $(S_0 + S_1)^* S_2^\omega$.

Por ejemplo, la Figura §C.1 muestra un autómata de Büchi simple y no determinista para reconocer $(S_0 + S_1)^* S_2^\omega$. Existen procedimientos para minimizar los autómatas de Büchi, pero es un resultado bien conocido que no existe un autómata de Büchi equivalente para todo autómata no determinista. El proceso de minimizar o comparar estos autómatas es generalmente exponencial.

Estos autómatas son importantes en las ciencias de la computación ya que se ha demostrado que existen procedimientos automáticos para transformar fórmulas de lógica temporal (lineal o de saltos) en sus correspondientes autómatas de Büchi [32]. En general, es más fácil estudiar propiedades sobre un autómata de Büchi que propiedades sobre sus correspondientes fórmulas temporales.

C.3 Notación del marco de trabajo

La Tabla §C.1 muestra la notación que hemos utilizado para definir el núcleo de nuestra marco de trabajo teórico y su implementación. La Tabla §C.2 muestra los predicados y la Tabla §C.3 las funciones.

C.4 Notación matemática

La Tabla §C.4 resume la notación matemática que hemos utilizado a los largo de la tesis doctoral.

Notación	Concepto
Σ	Sistema
P_Σ	Conjunto de procesos del sistema Σ
I_Σ	Conjunto de interacciones del sistema Σ
$P, Q, \ldots, p, q, \ldots$	Procesos de ejemplo
$A, B, \ldots, x, y, \ldots$	Interacciones de ejemplo
$\mathbb{P}(x)$	Procesos participantes de la interacción x
$\mathbb{I}(p)$	Interacciones en la que el procesos p puede participar
$p.\chi$	Evento en el que p ofrece participar en cualquier interacción de χ
$p.\emptyset$	Evento en el que p indica que ha finalizado su ejecución
x	Evento de sincronización en el que la interacción x es ejecutada
λ	Una ejecución
C_i, D_i	La configuración i-ésima de una ejecución
λ_α	La traza de configuraciones de λ
λ_β	La traza de eventos de λ
$\longrightarrow_L$	La regla que describe la semántica del lenguaje subyacente
$\longrightarrow_{FW}$	La regla que describe la implementación del marco de trabajo
$\longrightarrow_{CFF_k}$	La regla que describe la implementación de k-*conspiracy-free*

Tabla C.1: *Notación usada para definir el marco de trabajo.*

Notación	Concepto
$\mathtt{Waiting}(\lambda, p, \Upsilon, i)$	Se satisface si y sólo si el proceso p está esperando ejecutar alguna de las interacciones $\Upsilon \subseteq \mathbb{I}(p)$ en la configuración i-ésima de la ejecución λ
$\mathtt{Finished}(\lambda, p, i)$	Se satisface si y sólo si el proceso p ha finalizado en la configuración i-ésima de la ejecución λ
$\mathtt{Enabled}(\lambda, x, i)$	Se satisface si y sólo si la interacción x está habilitada en la configuración i-ésima de la ejecución λ
$\mathtt{Stable}(\lambda, x, i)$	Se satisface si y sólo si la interacción x está estable en la configuración i-ésima de la ejecución λ
$\mathtt{InfEna}(\lambda, x)$	Se satisface si y sólo si la interacción x se habilita en infinitas ocasiones en la ejecución λ
$\mathtt{InfSta}(\lambda, x)$	Se satisface si y sólo si la interacción x se estabiliza en infinitas ocasiones en la ejecución λ
$\mathtt{InfExe}(\lambda, x)$	Se satisface si y sólo si la interacción x se ejecuta en infinitas ocasiones en la ejecución λ
$\mathtt{InfOff}(\lambda, x, p)$	Se satisface si y sólo si el proceso p ofrece participar en la interacción x en infinitas ocasiones en la ejecución λ
$\mathtt{InfWait}(\lambda, x, p)$	Se satisface si y sólo si el proceso p espera a la interacción x en infinitas ocasiones en la ejecución λ
$\mathtt{SafeOffers}(\lambda)$	Se satisface si y sólo si los eventos de ofrecimiento en λ son seguros
$\mathtt{SafeTerms}(\lambda)$	Se satisface si y sólo si los eventos de terminación en λ son seguros
$\mathtt{SafeSyncs}(\lambda)$	Se satisface si y sólo si los eventos de sincronización en λ son seguros

Tabla C.2: *Predicados usados para definir el marco de trabajo.*

Notación	Concepto
StaLinked(x)	Interacciones estáticamente enlazadas con x
DynLinked(λ, x, i)	Interacciones dinámicamente enlazadas con x en la configuración i-ésima de λ
EnaSet(λ, x, i)	Índices de las configuraciones en la que x esta habilitada hasta la configuración i-ésima de λ
StaSet(λ, x, i)	Índices de las configuraciones en la que x esta estable hasta la configuración i-ésima de λ
ExeSet(λ, x, i)	Índices de las configuraciones en la que x se ejecuta hasta la configuración i-ésima de λ
OffSet(λ, ϒ, p, i)	Índices de las configuraciones en la que p ofrece participación en ϒ hasta la configuración i-ésima de λ
WaitSet(λ, x, p, i)	Índices de las configuraciones en la que p espera a x hasta la configuración i-ésima de λ

Tabla C.3: *Funciones para definir el marco de trabajo.*

Notación	Concepto
$\mathbb{N}$	Conjunto de números naturales
$\mathbb{N}^+$	Números naturales menos el cero
Ξ	Número natural aleatorio uniformemente distribuido
$[n_1, n_2]$	Intervalo cerrado
(n_1, n_2)	Intervalo abierto
$[n_1, n_2), (n_1, n_2]$	Intervalo semiabierto
$\{e_i\}_{i=1}^{n}$	Conjunto definido por enumeración
$\{e \in A \mid P(e)\}$	Conjunto definido por compresión
$\lvert A \rvert$	Números de elementos de A
$A \cap B$	Intersección de A y B
$\bigcap_{i=1}^{n} A_i$	Intersección distributiva de los conjuntos $A_1, A_2, \ldots, A_n$
$A \cup B$	Unión de A y B
$\bigcup_{i=1}^{n} A_i$	Unión distributiva de los conjutos $A_1, A_2, \ldots, A_n$
$A \setminus B$	B menos A
2^A	El conjunto potencia de A
$\{a_i \mapsto b_i\}_{i=1}^{n}$	Mapa definido por enumeración
$\mathrm{dom}\,\sigma$	Dominio del mapa σ
$\mathrm{ran}\,\sigma$	Imagen o rango del mapa σ
$\sigma(a)$	Imagen del elemento a
$[e_i]_{i=1}^{n}$	Secuencia definida por enumeración
$S_1 \frown S_2$	Concatenación de las secuencias S_1 y S_2
S^∞	Concatenación infinita $S \frown S \frown S \frown S \ldots$
$S(i)$	Elemento que se encuentra en la posición i-ésima de S
$\lvert S \rvert$	Número de elementos de S
$\rightarrowtail_R$	Regla de transición
$a_1 \rightarrowtail_R a_2 \rightarrowtail_R \ldots \rightarrowtail_R a_n$	Serie de transiciones
$\sum_{i=1}^{N} P(i)$	Cuenta el número de veces que P(i) se satisface cuando i está en el intervalo [1, N]
$\forall^\infty$	Para todos, pero en número finito
$\exists^\infty$	Existe un número infinito

Tabla C.4: *Notación matemática.*

Bibliografía

[1] N Abramson. The ALOHA system - Another alternative for computer communications. *AFIPS*, 37:281–285, 1970.

[2] MW Alford, JP Ansart, G Hommel, L Lamport, B Liskov, GP Mullery, and FB Schneider. *Distributed Systems - Methods and Tools for Specification - An Advanced Course*. Springer-Verlag, 1985.

[3] B Alpern and FB Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] B Alpern and FB Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] R Alur and TA Henzinger. Finitary fairness. *ACM Transactions on Programming Languages and Systems*, 20(6):1171–1194, 1998.

[6] E Andersen. *Conceptual Modeling of Objects: A Role Modeling Approach*. PhD thesis, University of Oslo, 1997.

[7] GE Andrews and RA Olson. *The SR Programming Language*. The Benjamin-Cummings Publishing Company, 1993.

[8] KR Apt, N Francez, and S Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[9] F Arbab. A channel-based coordination model for component composition. Technical Report SEN-R0203, CWI, 2002.

[10] PC Attie, N Francez, and O Grumberg. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.

[11] RJR Back and R Kurki-Suonio. Distributed co–operation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[12] RL Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, 1989.

[13] KE Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[14] B Bauer, J Müller, and J Odell. Agent UML: A formalism for specifying multiagent interaction. *Lecture Notes in Computer Science*, 1957:91–103, 2001.

[15] B Bauer, J Müller, and J Odell. Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[16] N Benton, L Cardelli, and C Fournet. Modern concurrency abstractions for C#. *Lecture Notes in Computer Science*, 2374:415–440, 2002.

[17] E Best. Erratum: Fairness and conspiracies. *Information Processing Letters*, 19(3):162, 1984.

[18] E Best. Fairness and conspiracies. *Information Processing Letters*, 18(4):215–220, 1984.

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

[20] T Bolognesi and E Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems, North-Holland*, 14:25–59, 1987.

[21] G Booch. *Object-Oriented Design with Applications*. The Benjamin-Cummings Publishing Company, Redwood City, California, 1990.

[22] D Box, D Ehnebuske, G Kakivaya, A Layman, N Mendelsohn, HF Nielsen, S Thatte, and D Winer. Simple object access protocol (SOAP 1.1).

[23] E Brinksma. *On the design of Extended LOTOS – A specification language for open distributed systems*. PhD thesis, Department of Computer Science, University of Twente, 1988.

[24] M Broy and G Nelson. Adding fair choice to Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 16(3):924–938, May 1994.

[25] JR 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 University Press, 1962.

[26] G Caire, F Leal, P Chainho, R Evans, F Garijo, J Gómez, J Pavón, P Kearney, J Stark, and P Massonet. Agent oriented analysis using MESSAGE/UML. *Lecture Notes in Computer Science*, 2222:119–126, 2002.

[27] N Carriero and D Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

[28] D de Champeaux. Object-oriented analysis and top-down software development. *Lecture Notes in Computer Science*, 512:360–375, 1991.

[29] KM Chandy and J Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[30] A Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, 1987.

[31] P Chrząstowski-Wachtel. Generalisation of liveness and fairness properties. In *Proceedings of the 8th Workshop on Applications and Theory of Petri Nets*, Zaragoza, Spain, 1987.

[32] EM Clarke, EA Emerson, and AP Sistla. Automatic verification of finite–state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 3(2):244–263, 1986.

[33] P Coad and E Yourdon. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, New Jersey, 1990.

[34] R Corchuelo. *Prototyping Constraint-Based Specifications of Distributed Systems*. PhD thesis, University of Seville, 1999.

[35] R Corchuelo and JA Pérez. Casale I: A new object–based language for discrete simulation. In *Proceedings of ECEC'98*, pages 310–312, 1998.

[36] R Corchuelo, JA Pérez, and A Ruiz-Cortés. Aspect-oriented interaction in multi-organizational web-based systems. *Computer Networks*, 41(4):385–406, 2003.

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

[38] R Corchuelo, JA Pérez, and M Toro. An order-based algorithm for implementing multiparty synchronisation. *Concurrency and Computation: Practice & Experience*, 2003. To be published.

[39] R Corchuelo, D Ruiz, M Toro, and A Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the IEEE Euromicro'99*, Milan, Italy, 1999. IEEE Computer Society Press.

[40] R de Simone. Higher-level synchronization devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[41] N Dershowitz, DN. Jayasimha, and S Park. Bounded fairness. *Lecture Notes in Computer Science*, 2772:To appear, 2003.

[42] EW Dijkstra. A class of allocation strategies inducing bounded delays only. *Manuscript, EWD 319*, 1971.

[43] EW Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[44] EW Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[45] EW Dijkstra. Position paper on "fairness". *Software Engineering Notes*, 3(2):18–20, 1988.

[46] DF D'Souza and AC Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Massachussets, 1999.

[47] R Englander. *Java and SOAPi*. O'Reilly & Associates, 2002.

[48] R Fagin and JH Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.

[49] M Fayad. E–Frame: A process-based, object-oriented framework for e-commerce. In *Proceedings of IC'01*, pages 124–128, Las Vegas, Nevada, 2001. CSREA Press.

[50] P Felber and MK Reiter. Advanced concurrency control in Java. *Concurrency and Computation: Practice & Experience*, 14(4):261–285, 2002.

[51] IR Forman. Design by decomposition of multiparty interactions in Raddle. 5^{th} *International Workshop on Software Specification and Design*, pages 2–10, 1989.

[52] N Francez. *Fairness*. Springer-Verlag, 1986.

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

[54] N Francez, BT Hailpern, and G Taubenfeld. Script: A communication abstraction mechanism and its verification. *Science of computer programming*, 6(1):35–88, 1986.

[55] N Francez and A Pnueli. A proof method for ciclyc programs. *Acta Informatica*, 9:133–157, 1978.

[56] DM Gabbay, A Pnueli, S Shelah, and J Stavi. On the temporal basis of fairness. In *Proceedings of ACM POPL'80*, pages 163–173. ACM Press, 1980.

[57] MS Day GC Roman. Multifaceted distributed systems specification using processes and event synchronization. In *Proceedings of the IEEE ICSE'84*, pages 44–55. IEEE Computer Society Press, 1984.

[58] SM German. A language for specifying synchronization. Technical Report TM-0509-02-92-152, GTE Laboratories, 1992.

[59] SJ Hartley. *Operating Systems Programming: The SR Programming Language*. Oxford University Press, 1997.

[60] GH Hilderink. Communicating Java threads reference manual. In *Proceedings of WoTUG'20*, pages 283–325, Twente, Netherlands, 1997. IOS Press, Netherlands.

[61] CAR Hoare. *Occam programming manual*. Prentice Hall, 1984.

[62] CAR Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[63] RV Hogg and AT Craig. *Introduction to Mathematical Statistics*. MacMillan, 5^{th} edition, 1994.

[64] DN Jayasimha and N Dershowitz. Bounded fairness. Technical Report 615, Center for Supercomputing Research and Development, University of Illinois, 1986.

[65] Y-J Joung. *On the design and implementation of multiparty interaction*. PhD thesis, University of New York, 1992.

[66] Y-J Joung. Characterizing fairness implementability for multiparty interaction. *Lecture Notes in Computer Science*, 1099:110–121, 1996.

[67] 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.

[68] Y-J Joung and SA Smolka. A completely distributed and message-efficient implementation of synchronous multiprocess communication. In *Proceedings of ICPP'90*, pages 311–318, Urbana-Champaign, Illinois, 1990. Pennsylvania State University Press.

[69] Y-J Joung and SA Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems*, 16(3):954–985, 1994.

[70] Y-J Joung and SA Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM*, 43(1):75–115, 1996.

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

[72] A Keen, T Ge, J Maris, and R Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the IEEE ICDCS'01*, pages 575–584, Los Alamitos, California, 2001. IEEE Computer Society.

[73] E. Kindler and W. van der Aalst. Liveness, fairness, and recurrence in petri nets. *Information Processing Letters*, 70(6):269–274, 1999.

[74] E Kindler and R Walter. Mutex needs fairness. *Information Processing Letters*, 62(1):31–39, 1997.

[75] D Kumar. An implementation of n-party synchronization using tokens. In *Proceedings of CDCS'90*, pages 320–327. IEEE Computer Society Press, 1990.

[76] MZ Kwiatkowska. Infinite behaviour and fairness in concurrent constraint programming. In *Semantics: foundations and applications*, number 666 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[77] L Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[78] L Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, 2000.

[79] L Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison–Wesley, 2003.

[80] G Le Lann. Distributed systems: Towards a formal approach. In *Proceedings of IFIP'77*, pages 155–160, 1977.

[81] D Lea. *Concurrent Programming Using Java: Design Principles and Pattern*. Addison-Wesley, 1999.

[82] D Lea, J Bowbeer, B Goetz, D Holmes, C McCorvey, and T Peierls. JSR 166: Concurrency Utilities, 2002.

[83] DJ Lehmann, A Pnueli, and J Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. *Lecture Notes in Computer Science*, 115:264–277, 1981.

[84] F Leymann and IBM Software Group. Web services flow language (WSFL 1.0). Available at `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[85] P Li. *BizTalk Server Developer's Guide*. Osborne McGraw–Hill, 2001.

[86] NA Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of ACM STC'80*, pages 70–81. ACM Press, April 1980.

[87] NA Lynch, M Merritt, WE Weihl, and A Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.

[88] TW Malone. What is coordination theory? Technical Report 2051-88, MIT Sloan School of Management, 1988.

[89] TW Malone and K Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26:87–119, 1994.

[90] Z Manna and A Pnueli. *The Temporal Logic of Reactive and Concurrent Systems-Specification*. Springer-Verlag, 1992.

[91] O Martín. *Quality-Aware Software Procurement*. PhD thesis, University of Seville, Under development.

[92] GJ Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–289, 1985.

[93] R Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[94] MJ Norusis. *SPSS 11.0 Guide to Data Analysis*. Prentice Hall, 2002.

[95] J Odell, HVD Parunak, and B Bauer. Representing agent interaction protocols in UML. *Lecture Notes in Computer Science*, 1957:121–140, 2001.

[96] US Department of Defense. Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A, US Goberment, 1983.

[97] S Older. Strong fairness and full abstraction for communicating processes. *Information and Computation*, 163(2):471–509, 2000.

[98] E Olderog and KR Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10(3):420–455, 1988.

[99] G Papadopoulos and F Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.

[100] J Peña, R Corchuelo, and JL Arjona. A top down approach for MAS protocol descriptions. In *Proceedings of SAC'03*, pages 45–49. ACM Press, 2003.

[101] J Peña. *The Joint Processes Specification: A Methodological Fragment*. PhD thesis, University of Seville, Under development.

[102] J Peña, R Corchuelo, and JL Arjona. Towards interaction protocol operations for large multiagent systems. *Lecture Notes in Computer Science*, 2699:79–91, 2002.

[103] J Peña, R Corchuelo, and JL Arjona. Towards interaction protocol operations for large multi-agent systems. *Lecture Notes in Computer Science*, 2699:79–91, 2002.

[104] J Peña, R Corchuelo, and JL Arjona. A top down approach for mas protocol descriptions. In *Proceedings of the ACM SAC'03*, pages 45–49, Melbourne, Florida, 2003. ACM Press.

[105] JA Pérez. *An aspect-oriented framework to describe the coordinated behaviour. An application to multiorgisational systems.* PhD thesis, Facultad de Informática y Estadística. Dpto. de Lenguajes y Sistemas Informáticos. University Of Seville, 2001.

[106] JA Pérez, R Corchuelo, D Ruiz, and M Toro. A framework for aspect–oriented multiparty coordination. In *DAIS*, volume 198 of *IFIP Conference Proceedings*, pages 161–173. Kluwer, 2001.

[107] GD Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[108] JP Queille and J Sifakis. Fairness and related properties in transition systems – A temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, 1983.

[109] MJ Quinn. *Designing Efficient Algorithms for Parallel Computers.* MacGraw Hill, 1987.

[110] I Rammer. *Advanced .NET Remoting.* APress, 2002.

[111] T Reenskaug, P Wold, and OA Lehne. *Working With Objects. The OOram Software Engineering Method.* Manning Publications Co., 1995.

[112] A Ruiz, R Corchuelo, JA Pérez, A Durán, and M Toro. An aspect–oriented approach based on multiparty interactions to specifying the behaviour of a system. In *Proceedings of PLI'99*, pages 56–65, Paris, France, 1999.

[113] D Ruiz and R Corchuelo. A J♯ framework to implement multiparty interactions. Technical Report 03-001, The Distributed Group, 2003.

[114] D Ruiz and R Corchuelo. A general, strongly hyperfair algorithm. In *Proceegings of PROLE'03*, page Submitted, 2003.

[115] D Ruiz, R Corchuelo, JA Pérez, and M Toro. An algorithm for ensuring fairness and liveness in non–deterministic systems based on multiparty interactions. *Lecture Notes in Computer Science*, 2400:563–572, 2002.

[116] A Ruiz-Cortés. *Una Aproximación Semicualitativa al Tratamiento Automático de Requisitos de Calidad.* PhD thesis, University of Sevilla, 2002.

[117] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. *Object–Oriented Modelling and Design.* Prentice Hall, Schenectady, New York, 1991.

[118] J Rumbaugh, I Jacobson, and G Booch. *The Unified Modelling Language Reference Manual.* Addison-Wesley, Reading, Massachussets, 1999.

[119] PG Sarang, E Adatia, and B Jouhier. *J#.* Wrox Press, Inc., 2002.

[120] F B Schneider and L Lamport. Another position paper on "fairness". *Software Engineering Notes*, 13(3):1–2, 1988.

[121] BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services version 1.1. Available at `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

[122] C Szyperski, D Gruntz, and S Murer. *Component Software.* Component Software Series. Addison–Wesley, 2 edition, 2002.

[123] AS Tanenbaum. *Computer Networks.* Prentice Hall, 1988.

[124] P Tang and Y Muraoka. Parallel programming with interacting processes. In *Proceedings of LCPC'99*, pages 201–218, 1999.

[125] P Tang and Y Muraoka. Parallel programming with interacting processes. *Lecture Notes in Computer Science*, 1863:201–218, 2000.

[126] S Thatte and Microsoft. XLANG: Web services for business process design. Available at `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

[127] YK Tsay and RL Bagrodia. Some impossibility results in interprocess synchronization. *Distributed Computing*, 6(4):221–231, 1993.

[128] R Wirfs-Brock and B Wilkerson. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[129] P Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[130] AF Zorzo and RJ Stroud. A distributed object-oriented framework for dependable multiparty interactions. *ACM Sigplan*, 34(10):435–446, 1999.

UNIVERSIDAD DE SEVILLA

Reunido el Tribunal constituido para juzgar la Tesis Doctoral de

en el día de la fecha

D. DAVID RUIZ CORTÉS

titulada SELECCIÓN JUSTA EN EL CONTEXTO DEL MODELO DE INTERACCIÓN ENTRE MÚLTIPLES PARTICIPANTES

acordó otorgarle la calificación de Sobresaliente cum laude

(por unanimidad)

Sevilla, 18 de DICIEMBRE 2003

El Presidente

El Vocal,

El Vocal,

El Vocal,

El Secretario,

El Doctorando,