

14 994 - 21
TESIS - 1 A 2



CINVESTAV-IPN
Biblioteca de Ingeniería Eléctrica



FB00000988

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.

Departamento de Ingeniería Eléctrica
Sección Computación



Tesis de Maestría:

*“ Especificación y diseño del núcleo básico de un sistema operativo
mediante CSP “*

Que para obtener el grado de Maestro en Ciencias en la especialidad de
Ingeniería Eléctrica con Opción Computación

Presenta:

Manuel Aguilar Cornejo¹

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

Dirigida por:
Dr. Héctor Ruíz Barradas²

México, D. F., Mayo de 1997

¹ Becario de CONACYT y Prof. asociado de tiempo completo de la UAM Iztapalapa.

² Prof. investigador de tiempo completo del área de Sistemas Digitales del Depto. de Ing. Electrónica de la UAM Azcapotzalco

XM

CLASIF.:	97.6
ADQUIS.:	RI-14994
FECHA:	9-Sep-97
PROCED.:	TESIS-1997
	↓

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.

Departamento de Ingeniería Eléctrica
Sección Computación

Tesis de Maestría:

*“ Especificación y diseño del núcleo básico de un sistema operativo
mediante CSP “*

Que para obtener el grado de Maestro en Ciencias en la especialidad de
Ingeniería Eléctrica con Opción Computación

Presenta:

Manuel Aguilar Cornejo¹

Dirigida por:
Dr. Héctor Ruíz Barradas²

México, D. F., Mayo de 1997

¹ Becario de CONACYT y Prof. asociado de tiempo completo de la UAM Iztapalapa.

² Prof. investigador de tiempo completo del área de Sistemas Digitales del Depto. de Ing. Electrónica de la UAM Azcapotzalco

A mis padres:

Que gracias a sus cuidados, consejos y apoyo dieron sentido a mi vida, convirtiéndose así, en mis primeros maestros.

A mis hermanos:

Que gracias a su compañía y apoyo han contribuido en la felicidad y alegría que han acompañado mi vida, siendo así mis primeros compañeros.

Agradecimientos

Quiero agradecer a un sinnúmero de personas, familiares e instituciones por el apoyo y confianza que me han brindado, me disculpo de antemano si es que omito a alguna persona o a varias. Agradezco sinceramente a:

Mi familia. Gracias por el apoyo económico, moral, y por la confianza que me han brindado a lo largo de mi vida.

CONACYT. Gracias por el apoyo económico que nos brinda a muchos estudiantes de posgrado.

UAM Iztapalapa. Gracias por haberme liberado de muchas de mis labores académicas para realizar mis estudios de maestría. Gracias por haberme brindado el equipo necesario para concluir mi tesis y gracias a mis alumnos por su comprensión cuando esporádicamente falte a clases.

Mi asesor de Tesis. Gracias a ti Héctor por haber aceptado dirigir mi tesis, gracias por tu paciencia y apoyo, gracias por compartir un poco de tus conocimientos con nosotros, tus alumnos, gracias por ser una excelente persona y un excelente investigador.

A mis sinodales. Gracias a la Dra. Elizabeth Pérez C., al Dr. Ju Shiguang y nuevamente al Dr. Héctor Ruiz B. por aceptar formar parte del jurado asumiendo lo que esto implica. Agradezco a cada uno de ellos, a Héctor por haber dirigido mi tesis; a la Dra. Elizabeth por su paciencia en la revisión de mi tesis, por su apoyo, por sus consejos y por ser una excelente persona; y al Dr. Ju por haber aceptado ser parte del jurado, por sus consejos y por ser un gran amigo.

A mis compañeros. Agradezco a todos los compañeros que tuve en la sección de computación por haber hecho de mi estadía en la maestría, una de las etapas más felices y alegres de mi vida, conservaré en la memoria y en medios de almacenamiento los momentos más felices que pasamos juntos. En particular agradezco a: Alfredo Yañez, Alicia, Carlos Campos, Carlos Hdz., Cesar Vargas, David Pérez, Eduardo Camargo, Efrén López, Enrique Sánchez, Estela Canchola, Francisco Mtz., Francisco Zaragoza, Gonzalo Avilés, Heberto Ferreira, Ignacio Vega, Jesús Naro, José Mitre, Manuel Díaz, Marco, Marcos Marván, Marco Rueda, Martha Cordero, Nuri Custodio, Raúl Hdz., Ricardo Díaz, Samuel Cortés y Tanit Cruz. Gracias por haber sido excelentes compañeros. Gracias por su amistad.

A mis amigos. Les doy las gracias por su amistad y apoyo a: Javier Delgado A., Emiliano Zapata M., Emelio Campos F., Anabel Díaz, Sofía Reza, Flor Córdova y muy en particular a Sandra Díaz S. por las palabras de aliento, y por ser una excelente amiga. Agradezco también al resto de mis amigos que por fines de espacio no menciono aquí.

Sinceramente
Manuel Aguilar C.

Resumen

En este trabajo presentamos, a través de un caso de estudio, el desarrollo de sistemas mediante métodos formales. El caso de estudio es el núcleo básico de un sistema operativo multitareas y el método formal utilizado es la teoría de Procesos Secuenciales Comunicantes (CSP).

Comenzamos este trabajo dando una muy breve explicación de lo que son los métodos formales en general, para después presentar de manera particular el método formal de CSP, así como un demostrador automático de refinamientos CSP, denominado FDR (Refinamientos Falla-Divergencia). Posteriormente desarrollamos el sistema seleccionado mediante CSP

Comenzamos el desarrollo de nuestro caso de estudio con la especificación inicial del núcleo básico del sistema operativo, en donde sólo expresamos lo que queremos realizar sin dar detalles de implantación. Los detalles de implantación los damos de manera gradual en lo se denominan *pasos de refinamiento*, en cada uno de ellos refinamos una propiedad del sistema. Cada refinamiento debe conservar las propiedades esenciales de la especificación inicial, para asegurarnos de ello realizamos pruebas formales, por un lado usando el álgebra de procesos CSP y por otro lado usando FDR. Una vez que obtuvimos un refinamiento con los suficientes detalles de implantación derivamos un programa en pseudocódigo.

Terminamos la exposición de este trabajo con nuestras conclusiones y un par de apéndices. En el primer apéndice mostramos las leyes CSP utilizadas en el texto, en el segundo apéndice damos un listado de la especificación inicial y de los refinamientos del núcleo básico del sistemas operativo.

Palabras clave:

Método formales, CSP (Procesos Secuenciales Comunicantes), FDR (Refinamientos Falla-Divergencia), pasos de refinamiento, lenguaje de especificación formal, sistema de inferencia, proceso, evento, paralelismo, concurrencia, determinismo, no determinismo, sistemas críticos, corrección.

Abstract

In this thesis is presented, through a case study, the development of systems by the use of formal methods. The case study is the basic kernel of a multitasking operating system, and the formal method used is the Communications Sequential Processes (CSP).

The thesis begins by giving a brief general explanation of formal methods, and then describes the CSP formal method and an automatic refinements proof for CSP, called FDR (Failures-Divergence Refinements). Later we present the development of the selected system in CSP.

The development of the case study commences with the initial specification of the basic operating system kernel, where we just express what we want to do, without implementation details. Such implementation details are given in a gradual form in the named "*refinement steps*", in each of which, we refine a property of the system. All refinements have to conserve the essential properties of the initial specification. To ensure that such properties are conserved, we give formal proofs, using the algebra of processes CSP, and also FDR. Once a refinement with enough implementation details is obtained, we derive a program in pseudocode.

We finish the exposition of this work with our own conclusions and a couple of appendixes. In the first appendix we show the CSP laws used in the text, in the second one we give a listing about the initial formal specification and the refinements of the basic operating systems kernel.

key words:

Formal methods, CSP (Communications Sequential Processes), FDR (Failures-Divergence Refinement), refinement steps, formal specification language, inference systems, processes, event, parallelism, concurrence, determinism, non determinism, critical systems, correctness.

Capítulo 1. Introducción	4
1.1 Introducción	4
1.2 Métodos formales	6
1.3 El lenguaje de especificación formal	7
1.4 El sistema de Prueba	9
1.5 Desarrollo de sistemas usando métodos formales	9
1.6 Estructura de la tesis	11
Capítulo 2. C.S.P. (Procesos Secuenciales Comunicantes)	13
2.1 Procesos	15
2.1.1 Prefijo	16
2.1.2 Recursión	17
2.1.3 Elección	19
2.1.4 Recursión mutua	20
2.1.5 Trazas	21
2.1.5.1 Operación sobre trazas	21
2.1.6 Trazas de un proceso	22
2.1.6.1 Leyes sobre trazas de un proceso	23
2.2 Concurrencia	24
2.2.1 Trazas	26
2.3 No Determinismo	27
2.3.1 No determinismo interno	27
2.3.2 No determinismo externo	28
2.3.3 Rechazos	28
2.3.4 Fallas	29
2.3.5 Ocultamiento	30
2.3.6 Entrelazamiento	30
2.3.7 Divergencia	31
2.4 Comunicación	32
2.4.1 Introducción	32
2.4.2 Canales de Comunicación	33
2.4.3 Tubos de comunicación	34
2.4.4 Subordinación	35
2.5 Procesos Secuenciales	37
2.5.1 Interrupciones	38
2.5.1.1 Catástrofe	38
2.5.1.2 Reinicio	38
2.5.1.3 Puntos de rearmar	39
2.6 Refinamientos CSP	39
2.7 Refinamiento a implantación ejecutable	40

Capítulo 3. FDR (Failures Divergence Refinement, Refinamientos del modelo Falla-Divergencia)	42
3.1 Características de FDR2	43
3.2 Refinamientos CSP	43
3.3 Usando Refinamientos	45
3.4 Estructura e Interfaz	46
3.4.1 Estructura	46
3.4.2 Interfaz	46
3.4.2.1 Pantalla principal de FDR2	46
3.4.2.2 Opciones de la barra del menú principal	48
3.4.2.2.1 Submenú File	48
3.4.2.2.2 Submenú Assert	48
3.4.2.2.3 Submenú Process	49
3.4.2.2.4 Submenú Options	49
3.4.2.2.5 Ayuda en línea	49
3.4.2.3 Comandos de la barra de herramientas	50
3.4.2.4 La lista de afirmaciones	51
3.4.2.5 La lista de procesos	52
3.4.2.6 El selector de refinamientos	52
3.4.2.7 El depurador de procesos	53
3.4.2.7.1 Comandos del menú del depurador	53
3.4.2.7.2 La ventana de comportamiento de procesos	54
3.5 Formato de entrada de especificaciones CSP a FDR2	56
3.5.1 Lenguaje de bajo nivel	56
3.5.2 Lenguaje de alto nivel	58
3.5.3 Declaración de variables y canales	59
3.5.4 Operadores disponibles	60
Capítulo 4. Desarrollo del núcleo básico del sistema operativo multitarea	63
4.1 Introducción	63
4.2 Especificación inicial	64
4.3 Refinamientos de la especificación inicial	65
4.3.1 Primer refinamiento	65
4.3.1.1 Especificación del primer refinamiento	65
4.3.1.2 Prueba manual del primer refinamiento	66
4.3.1.3 Prueba automática del primer refinamiento	68
4.3.1.4 Motivación al segundo refinamiento	70
4.3.2 Segundo refinamiento	71
4.3.2.1 Especificación del segundo refinamiento	71
4.3.2.2 Prueba manual del segundo refinamiento	71
4.3.2.3 Prueba automática del segundo refinamiento	73
4.3.2.4 Motivación al tercer refinamiento	74

4.3.3 Tercer refinamiento	75
4.3.3.1 Especificación del tercer refinamiento	75
4.3.3.2 Prueba manual del tercer refinamiento	76
4.3.3.3 Prueba automática del tercer refinamiento	81
4.3.3.4 Motivación al cuarto refinamiento	84
4.3.4 Cuarto refinamiento	84
4.3.4.1 Especificación del cuarto refinamiento	84
4.3.4.3 Prueba automática del cuarto refinamiento	85
4.3.4.4 Motivación al quinto refinamiento	87
4.3.5 Quinto refinamiento	88
4.3.5.1 Especificación del quinto refinamiento	88
4.3.5.2 Prueba automática del quinto refinamiento	89
4.4 Hacia la Implantación	92
4.4.1 Subprocesos del núcleo básico del sistema operativo	101
4.4.2 Comportamiento del núcleo básico del sistema operativo	101
4.4.3 Mapeo de los subprocesos a pseudocódigo	102
Capítulo 5. Conclusiones	106
Apéndice I. Leyes CSP	109
A1. Introducción	109
A2. Composición Paralela	109
A.2.1 Alfabetos iguales	109
A.2.2 Alfabetos Diferentes	111
A.3 Entrelazamiento	114
A.4 No determinismo	116
A.4.1 Elección interna	116
A.4.2 Elección externa	118
A.5 Ocultamiento	120
A.6 Después (after)	122
A.7 Renombramiento	124
A.8 Etiquetamiento	126
A.9 Composición secuencial	128
A.10 Comunicación	131
A.11 Subordinación	132
Apéndice II. Script FDR2 de los refinamientos del núcleo básico del sistema operativo	133
Bibliografía	139

1.1 Introducción

Un requisito de la calidad de un programa es su corrección. A pesar de tal requisito actualmente hay muy pocos programas complejos sin errores o anomalías. Desafortunadamente muchos de esos defectos son extremadamente sutiles y aparecen en circunstancias muy inusuales. El problema es que estas circunstancias inusuales, aunque sean pocas las veces, se presentan.

La falta de calidad en los programas, no recae solamente en la gente responsable de producir el *software*, sino también en los métodos inadecuados que emplean para desarrollarlos. La respuesta para resolver esta situación ha sido la utilización de métodos como:

- administración de proyectos,
- análisis de requerimientos,
- descomposición de problemas de manera descendente (top-down),
- programación estructurada,
- tipos de datos abstractos,
- ocultamiento de información,
- pruebas rigurosas, etc.

Estos métodos nos han llevado a mejoras considerables en la calidad de programas. Sin embargo, la falta de corrección permanece aún como el problema principal. Los programadores consideran como inevitable el que sus programas contengan errores. En efecto, los programadores consideran que de un tercio a la mitad de los esfuerzos de programación se centran en la “depuración” de los programas[Dromey89]. Peor aún, en el acto de eliminar un error frecuentemente aparecen muchos más.

Esta situación resulta desagradable, debido a que en nuestra vida diaria dependemos cada vez mas del uso de sistemas críticos, tal como: sistemas de control de tráfico aéreo, sistemas de control de reactores nucleares, sistemas de comunicación y control, etc.

A continuación mencionamos algunos errores de *software* que se han presentado en tiempos recientes, y que han capturado la atención del público.

- La primer versión del software de navegación F-16 invierte la posición de los aviones siempre que se cruza el ecuador.
- Una versión del software del Apollo II tiene la gravedad de la luna como repulsiva en vez de atractiva.
- Un error de cómputo en un barco de guerra causó que disparara en dirección opuesta a su objetivo durante un ejercicio militar en San Francisco en 1982.
- En febrero de 1984, las máquinas de cambio de dos bancos de Estados Unidos tuvieron serios problemas de compatibilidad, debido a que uno registraba el cambio de año y el otro no.
- En 1983, el índice de precios y cotizaciones de la bolsa de Vancouver se encontró en 725 puntos, en vez de 960 puntos, debido a errores acumulativos en la forma en que el índice fue calculado.
- Una accidental serie de sobredosis de radiación fue administrada por máquinas terapeutas de cáncer en Georgia, Estados Unidos, en 1986 debido a un error en el programa que controla la máquina.

Tales errores son sólo una muestra de los que se han presentado. La pregunta que nos formulamos es: ¿Cómo construir software libre de errores? Muchas de las dificultades para construir software libre de errores se deben a la complejidad de las tareas que son programadas.

Los errores pueden presentarse en un sistema debido a tres posibles causas:

- defectos en los requerimientos del sistema, lo cual provoca que el sistema falle debido a aspectos del ambiente que no fueron considerados.
- defectos de diseño, que resultan al diseñar una solución insatisfactoria a los requerimientos establecidos.
- defectos de implantación, que resultan de los errores de implantación para satisfacer los detalles de diseño.

Los errores de requerimientos y los errores de diseño se propagan de manera directa al nivel de implantación. Esos errores son mucho más frecuentes y difíciles de corregir, que los de implantación.

La pregunta es, ¿dada la especificación correcta de los requerimientos del sistema, podemos construir un programa y asegurar que éste es correcto con respecto a su especificación? La respuesta es “sí”, pero requerimos cambiar de manera substancial nuestro enfoque para construir sistemas [Dromey89]. Es aquí donde los métodos formales muestran

sus bondades y nos ofrecen el enfoque para construir sistemas de manera razonada y metódica con las características solicitadas. Si el método formal es bien aplicado, entonces el programa generado cumplirá forzosamente con las propiedades especificadas al inicio del desarrollo del sistema.

El objetivo de la presente tesis es mostrar la viabilidad del desarrollo de sistemas utilizando métodos formales. Para ello, pensamos que lo mejor sería desarrollar un sistema con el grado de complejidad de los descritos arriba. Sin embargo, el desarrollo de tales sistemas sale de nuestro alcance, debido a limitaciones de tiempo, como económicas. Por tal razón decidimos desarrollar un sistema que tenga fines didácticos, y que además tenga la cualidad de interaccionar continuamente con su medio ambiente, tal como lo hacen los sistemas reactivos¹ (los sistemas descritos arriba son casos particulares de sistemas reactivos). El sistema con el cual ilustraremos la viabilidad del desarrollo de sistemas mediante métodos formales es el núcleo básico de un sistema operativo multitareas.

Antes de hacer una descripción de los tópicos contenidos en este trabajo revisemos algunos conceptos utilizados posteriormente.

1.2 Métodos formales

Los métodos formales usados en el desarrollo de sistemas de cómputo son técnicas basadas en las matemáticas para describir propiedades de sistemas. Tales métodos proporcionan un marco dentro del cual es posible especificar, desarrollar y verificar sistemas de una manera sistemática.

Un método es formal si tiene bases matemáticas sólidas, típicamente dadas por un lenguaje de especificación formal. El método formal proporciona los medios para probar que una especificación de un sistema, escrita en el lenguaje de especificación formal, puede ser llevada a una implantación. También nos ayuda a determinar las propiedades de un sistema sin necesidad de ejecutarlo para determinar su comportamiento.

Se recomienda que principalmente los diseñadores de sistemas usen métodos formales para especificar el comportamiento deseado de un sistema y sus propiedades estructurales. Aunque generalmente cualquiera involucrado en el desarrollo de un sistema puede hacer uso de los métodos formales, ya que pueden ser usados en las declaraciones iniciales del cliente o usuario, a lo largo del diseño del sistema, en la implantación, prueba, depuración, mantenimiento, verificación y evaluación.

Los métodos formales son usados para detectar ambigüedad, incompletez e inconsistencia en un sistema. Cuando se usan al inicio del proceso de desarrollo del sistema, éstos pueden evidenciar los errores de diseño que en otro caso pueden ser descubiertos sólo durante las costosas fases de prueba y depuración. Cuando se usan posteriormente, éstos pueden ayudar a

¹ Un sistema reactivo es un programa de cómputo que interacciona continuamente con su medio ambiente.

determinar la corrección de la implantación de un sistema y la equivalencia de distintas implantaciones.

Una especificación sirve como un contrato, una invaluable pieza de documentación, y un medio de comunicación entre el cliente, el especificador y el implantador. Debido a sus bases matemáticas, las especificaciones formales son más precisas y concisas que las informales.

Puesto que un método formal, es un método y no sólo un programa de computadora o lenguaje de programación, éste puede o no tener herramientas de soporte. Un método formal consta de un lenguaje de especificación formal y de un sistema de prueba. El lenguaje de especificación permite describir las propiedades o el comportamiento de un programa a diferentes niveles de abstracción, y el sistema de prueba nos permite probar, de manera formal, que las propiedades o comportamientos descritos en cada nivel, satisfacen las especificaciones del nivel más abstracto.

1.3 El lenguaje de especificación formal

Un lenguaje de especificación formal nos permite describir de manera precisa y no ambigua las propiedades de un sistema.

Definición. Un lenguaje de especificación formal es una tripleta $\langle Sin, Sem, Sat \rangle$, donde Sin y Sem son conjuntos y $Sat \subseteq Sin \times Sem$ es una relación entre ellos. Sin es llamado el dominio sintáctico del lenguaje. Sem , el dominio semántico, y Sat la relación a satisfacer.

En otros términos, un lenguaje de especificación formal proporciona una notación (su dominio sintáctico), un universo de objetos (su dominio semántico), y una regla precisa que define qué objetos satisfacen qué especificación. Una especificación es una sentencia escrita en términos de elementos del dominio sintáctico, ésta denota un conjunto de especificandos, un subconjunto del dominio semántico. Un especificando es un objeto que satisface una especificación. La relación a satisfacer proporciona el significado o interpretación para los elementos sintácticos.

Los métodos formales difieren entre ellos, debido a que sus lenguajes de especificación tienen diferentes dominios sintácticos y/o semánticos. Aunque ellos tengan dominios sintácticos y semánticos idénticos, pueden tener diferentes relaciones a satisfacer.

Definición. Dado un lenguaje de especificación $\langle Sin, Sem, Sat \rangle$, si $Sat(sin, sem)$, entonces sin es una especificación de sem , y sem es un especificando de sin .

Usualmente definimos el dominio sintáctico del lenguaje de especificación en términos de un conjunto de símbolos (por ejemplo, constantes, variable y conectivos lógicos) y un conjunto de reglas gramaticales para combinar esos símbolos a una sentencia bien formada.

Definición. Dado un lenguaje de especificación, $\langle Sin, Sem, Sat \rangle$, el conjunto de especificandos de una especificación *sin* en *Sin* es el conjunto de todos los especificandos *sem* en *Sem* tal que $Sat(sin, sem)$.

Los lenguajes de especificación difieren mucho en la elección del dominio semántico. Los siguientes son algunos ejemplos:

- Lenguajes de especificación con tipos de datos abstractos son usados para especificar álgebras, teoremas y programas. Sin embargo, las especificaciones escritas en ese rango de lenguajes sobre diferentes dominios semánticos, ofrecen una vista sintácticamente similar.
- Lenguajes de especificación de sistemas concurrentes y distribuidos. Son usados para especificar secuencias de estados, secuencias de eventos, secuencias de estados y transiciones, flujos y máquinas de estados.
- Lenguajes de programación son usados para especificar funciones que mapean entradas a salidas.

La forma de Backus-Naur es un ejemplo de un lenguaje de especificación formal, con una gramática como su dominio sintáctico y un conjunto de cadenas como su dominio semántico. Cualquier cadena generada por la gramática es un especificando. Cualquier conjunto de especificandos es un lenguaje formal.

Cada lenguaje de programación (con su semántica formal bien definida) es un lenguaje de especificación, pero lo inverso no, debido a que las especificaciones en general no tienen por que ser ejecutables en alguna máquina, mientras los programas sí. Al usar un lenguaje de especificación mas abstracto, ganamos la ventaja de no estar restringidos a expresar funciones computables solamente. Es perfectamente razonable en una especificación expresar sentencias tal como "Para toda x en el conjunto A , existe una y en el conjunto B , tal que la propiedad P se cumple sobre x y y , donde A y B deben ser conjuntos finitos"

Los programas, sin embargo, son objetos formales, susceptibles de manipulaciones formales (por ejemplo, la compilación y ejecución). Así, los programadores no pueden escaparse de métodos formales. La pregunta es: ¿si ellos trabajan con requerimientos informales y programas formales, o bien, si ellos usan formalismos adicionales para asistirse durante la especificación de requerimientos?

Cuando el dominio semántico de un lenguaje de especificación son programas o sistemas de programas, el término *implanta* es usado para la relación a satisfacer, y el término *implantación* es usado para un especificando en *Sem*. Una implantación *prog* es correcta con respecto a una especificación dada *spec*, si *prog* implanta *spec*. Más formalmente, tenemos:

Definición. Dado un lenguaje de especificación $\langle Sin, Sem, Sat \rangle$, una implantación *prog* en *Sem* es correcta con respecto a una especificación dada *spec* en *Sin* si y sólo si $Sat(spec, prog)$.

Propiedades de las Especificaciones

Un lenguaje de especificación define sentencias bien formadas, no ambiguas.

Definición. Dado un lenguaje de especificación $\langle Sin, Sem, Sat \rangle$, una especificación sin en Sin es no ambigua si y sólo si, Sat relaciona sin exactamente a un especificando.

Es decir, una especificación es no ambigua si y sólo si tiene exactamente un significado. Otra propiedad necesaria de una especificación es la satisfacibilidad.

Definición. Dado un lenguaje de especificación, $\langle Sin, Sem, Sat \rangle$, una especificación sin en Sin es satisficible, si y sólo si, Sat relaciona sin a un conjunto no vacío de especificandos.

En otros términos, una especificación es satisficible si y sólo si el conjunto de especificandos es no vacío. En términos de programas, la satisfacibilidad es importante por que ésta significa que hay alguna implantación que satisface la especificación.

1.4 El sistema de prueba

Los métodos formales son definidos en términos de un lenguaje de especificación formal y de un sistema lógico de inferencia bien definido. El sistema de inferencia se utiliza para probar propiedades de las especificaciones. Esto es, tomamos la especificación como un conjunto de hechos y derivamos nuevos hechos a través de la aplicación de las reglas de inferencia. Si el usuario obtiene que las propiedades solicitadas no se conservan con la aplicación de las reglas de inferencia, entonces, la especificación no fue bien diseñada.

Un método formal con un sistema de inferencia definido explícitamente tiene la ventaja de poder ser mecanizado e implantado en computadora, ventaja que no tienen los métodos formales sin sistema de inferencia definido explícitamente.

1.5 Desarrollo de sistemas usando métodos formales

El desarrollo de sistemas usando métodos formales es similar al desarrollo de sistemas convencional, pero hay características distintivas en la primer aproximación. Muchos métodos de desarrollo pueden ser representados por un modelo de cascada en el que los primeros estados producen como resultado los estados posteriores. Una secuencia típica de pasos en el desarrollo de sistemas con métodos formales es:

- Análisis y captura de requerimientos.
- Especificación formal de requerimientos informales.

- Prueba de que la especificación es consistente y completa con los requerimientos.
- Los pasos de diseño involucran una repetición de:
 - refinamiento de la especificación hacia una forma más orientada hacia la implantación y
 - verificación (prueba) de que la especificación refinada tiene las propiedades esenciales de la especificación anterior.
- Implantación de la especificación de nivel más bajo.
- Validación (prueba) de que la implantación satisface la especificación de nivel más bajo.

El uso de métodos formales requiere de especificaciones formales en todos los pasos, para asegurar una interpretación única. El hecho de que las especificaciones sean formales significa que las transformaciones serán efectuadas preservando la corrección. Un diseñador usa su intuición y experiencia para derivar el siguiente nivel de la especificación. Puesto que hay un riesgo en introducir errores en este proceso, la especificación de nivel más bajo debe ser formalmente verificada. Esta prueba requiere que las propiedades esenciales de la especificación anterior se conserven en la nueva.

El número de refinamientos depende de la “distancia” existente entre la especificación de alto nivel y el ambiente de implantación de bajo nivel. Para producir una implantación en ensamblador o en lenguaje máquina es prudente tener un número de pasos relativamente grande. Para producir una implantación en un lenguaje de alto nivel serán necesarios menos pasos que en el caso anterior. Durante el diseño, puede ser necesario reconsiderar pasos anteriores que nos han llevado a soluciones no adecuadas.

Una especificación es refinada adicionando detalles de implantación y eliminando la libertad de implantar detalles no deseados. El refinamiento puede ser sugerido por transformaciones predefinidas, por principios generales de diseño, o por heurísticas específicas al problema. El tamaño del paso de refinamiento no puede ser más grande del que pueda ser manipulado intelectualmente o por herramientas. Probar la validez de un refinamiento no es un ejercicio trivial puesto que el refinamiento puede introducir detalles y características no presentadas en la especificación original, o puede eliminar la libertad permitida en la original.

Los métodos formales nos ayudan a estructurar o descomponer el sistema, facilitándonos el análisis y diseño. Además pueden tratar con sistemas en diferentes niveles de abstracción, permitiendo que una especificación de alto nivel sea transformada en una de bajo nivel. La biblioteca de componentes predefinidos puede ser especificada y verificada. Combinando componentes conocidos de formas conocidas, un diseñador puede tener seguridad en los resultados del diseño descendente. Finalmente las técnicas de verificación permiten verificar si un diseño propuesto satisface los requerimientos especificados.

A pesar de que el desarrollo de un sistema completo usando el enfoque formal tiene enormes beneficios, pueden haber razones prácticas por lo cual no es posible realizarlo de manera completa con métodos formales. El esfuerzo involucrado en la especificación y verificación puede ser muy alto. Sin embargo el costo extra de los métodos formales es desplazado por la reducción en errores. El costo de resolver un problema, cuando un sistema ha sido distribuido e instalado, es alrededor de 1000 veces el costo de resolver el mismo problema si éste es identificado en el estado de la especificación [Kenneth93]. Esto significa que el enfoque formal es particularmente benéfico en costo en el desarrollo de sistemas.

La posibilidad de verificar la corrección de una especificación es atractiva, de acuerdo a lo anterior. En sistemas críticos es esencial realizarlo, considerando los costos si no lo hacemos. La verificación es, desafortunadamente, una actividad intelectual difícil, en donde las herramientas de *software* actuales pueden darnos soporte limitado o nulo. Es por ello que al desarrollar la presente investigación buscamos un método formal que tuviera herramientas automatizadas para que nos auxiliara en las pruebas de los refinamientos. CSP es un método formal que tiene herramientas automatizadas (FDR), además de un gran número de operadores que nos ayudaron en el desarrollo de nuestro trabajo, éstas fueron las principales razones que nos orillaron a tomar la decisión de utilizar CSP

1.6 Estructura de la tesis

Los pasos que seguimos para exponer nuestra investigación son los siguientes:

- En el capítulo dos damos una descripción del lenguaje de especificación formal que utilizamos para este trabajo, CSP (Procesos Secuenciales Comunicantes), así como una descripción de los principales operadores CSP, ilustrándolos con ejemplos pequeños y claros.
- En el capítulo tres describimos la herramienta automatizada para demostrar refinamientos CSP, denominada FDR (de sus siglas en inglés, Refinamientos Falla-Divergencia). En este capítulo describimos las características de FDR, tipos de propiedades que puede verificar, su estructura y la interfaz con el usuario. Además describimos como usar esta herramienta.
- En el capítulo cuatro presentamos el desarrollo del núcleo básico del sistema operativo multitareas usando el método formal CSP. Este capítulo comienza con la especificación inicial del núcleo del sistema operativo. Las siguientes secciones contienen los pasos de los refinamientos. En los primeros cuatro pasos de refinamiento se tiene:
 - i) La especificación del refinamiento.
 - ii) La demostración manual de que el refinamiento conserva las propiedades originales del sistema.
 - iii) La demostración usando FDR de que el refinamiento conserva las propiedades originales del sistema.
 - iv) La motivación al siguiente refinamiento.

En las últimas secciones de este capítulo presentamos la especificación del quinto refinamiento, la prueba de su corrección utilizando sólo FDR y su implantación en pseudocódigo.

En el capítulo cinco presentamos las conclusiones de nuestro trabajo. Terminamos esta tesis con dos apéndices: el primero con la mayoría de las leyes CSP, y el segundo con el script FDR de la especificación inicial y todos los refinamientos de nuestro sistema.

C. S. P. (Procesos Secuenciales Comunicantes)

En el capítulo anterior introducimos los conceptos básicos sobre métodos formales, esto sirve como base para el buen entendimiento de este trabajo. En este capítulo describimos de manera detallada el método formal de CSP. Comenzamos describiendo lo que es un proceso y un evento, posteriormente revisamos los operadores CSP sobre procesos y sobre eventos. Cada operador es descrito en una sección o una subsección dependiendo de su complejidad e importancia.

En la sección uno describimos algunos de los operadores de procesos y eventos. En la sección dos describimos el operador de concurrencia, el cual opera sobre procesos. En la sección tres describimos el operador de no determinismo, esta sección contiene varias subsecciones en donde se describen los operadores de: no determinismo interno, no determinismo externo, rechazos, ocultamiento, entrelazamiento, y divergencia. En la sección cuatro describimos los operadores de comunicación. En la sección cinco describimos los operadores de secuencialidad de procesos. En la sección seis describimos como obtener refinamientos CSP y en la sección siete describimos como llevar el último refinamiento de un sistema a una implantación.

Antes de pasar a ver la descripción de CSP, veamos algunas de sus generalidades.

Conforme el *software* se ha incrementado en tamaño y complejidad, la academia y la industria han girado su atención a la verificación de la corrección de tales sistemas. En los últimos treinta años, muchas técnicas han evolucionado en la especificación y verificación de sistemas de cómputo complejos. Las técnicas que actualmente gozan de mayor popularidad son los llamados “Métodos formales”, los cuales fueron esbozados en el capítulo anterior.

Con el advenimiento de la concurrencia, se ha complicado la verificación de la corrección de los sistemas, debido a que los sistemas concurrentes son inherentemente más complejos, pues ofrecen la posibilidad de cómputo paralelo y distribuido.

Los métodos estándar para especificar y razonar sistemas secuenciales, no son adecuados para los sistemas concurrentes. Ellos no toman en cuenta los efectos laterales, la ocurrencia de múltiples eventos simultáneos, la sincronización que en ocasiones requieren los procesos para asegurar la integridad de los datos, etc. Sin embargo, las técnicas basadas sobre los métodos formales han probado ser más adecuadas que otras técnicas como las estructuradas (informales) y las basadas en lógica temporal [Hinchey95 a].

El método formal de Hoare, CSP [Hoare85], representa un paso importante en el estudio de la concurrencia, y en el dominio de la complejidad inherente de los sistemas concurrentes.

Los sistemas concurrentes, incluyendo los de tiempo real y los distribuidos, (los cuales consideramos como casos especiales de concurrencia), son típicamente grandes y complejos, generalmente reaccionan a entradas de varias fuentes y satisfacen restricciones de sincronización (frecuentemente dentro límites temporales estrictos). Para entender completamente los requerimientos de un sistema y la forma en que interactúan sus componentes, debemos incrementar los niveles de abstracción y posponer lo relacionado con la distribución y ejecución de los procesos. Esto obviamente no puede ser ignorado indefinidamente, dado que queremos una implantación que sea razonablemente eficiente.

Al asumir altos niveles de concurrencia, podemos reducir la complejidad del sistema. Claramente, la ejecución concurrente de este sistema, puede no ser factible en el ambiente de implantación (debido a restricciones económicas o de hardware), por lo cual se requiere algún medio de reducir los niveles de concurrencia. Es importante, sin embargo, reducir los niveles no comprometiendo el comportamiento que ha sido especificado. Por lo cual, esta reducción de niveles debe ser basada sobre alguna forma de transformaciones que preserven la corrección.

CSP soporta un modelo de eventos que facilita la descripción de entidades. Esto nos permite especificar sistemas que ejecuten varias acciones en algún orden en particular, y (usando su variante temporal) podemos expresar restricciones temporales entre esas acciones y sobre la sincronización de varios componentes del sistema.

Una especificación CSP pueden describir un sistema en cualquier nivel de abstracción. Una especificación es vista como un sistema de procesos ejecutándose independientemente, los cuales se comunican a través de canales unidireccionales sin almacenamiento, y se sincronizan con eventos particulares. Esta comunicación, (denominada paso de mensajes), no crea restricciones sobre la implantación actual de la comunicación, pues evita los problemas de exclusión mutua, y nos permite centrarnos en la esencia del diseño y en la especificación del sistema.

Las especificaciones pueden ser manipuladas a través de la aplicación de un número de leyes algebraicas y combinadas por medio de un pequeño número de operadores. Ello nos

permite reducir los niveles de abstracción, al proporcionarnos las herramientas para probar que una especificación conserva las propiedades de otra más abstracta.

Finalmente, cuando el sistema ha sido expresado en un nivel apropiado de abstracción, éste puede ser ejecutado (casi directamente) sobre un transputer como un programa OCCAM, o en un lenguaje de programación convencional, tal como Ada [Hinchey95 a].

2.1 Procesos

Si nos abstraemos por un momento de términos computacionales, y observamos el mundo que nos rodea, podremos darnos cuenta que el mundo está constituido por objetos con los cuales interactuamos. Más aún, si observamos uno de esos objetos, podremos darnos cuenta que éste ejecuta eventos de acuerdo a las circunstancias en que se encuentre. Habrá circunstancias que no le afecten, o bien que le afecten demasiado, al grado de alterar su comportamiento normal.

Al conjunto de eventos que ejecuta un objeto P , los cuales se consideran relevantes para su descripción particular, se le denomina el "*alfabeto de P* " y se denota por el símbolo " αP ". De tal forma, que un objeto puede ser descrito por el conjunto observable de eventos en que se "*compromete*" (o ejecuta). Es lógicamente imposible para un objeto comprometerse con un evento que no pertenece a su alfabeto; por ejemplo, una máquina que vende refrescos no podrá repentinamente entregar juguetes, o cualquier cosa distinta de refrescos.

La elección de un alfabeto usualmente involucra una simplificación deliberada: la decisión de ignorar propiedades y acciones que carecen de interés para nuestros propósitos. Por ejemplo, el color, peso y geometría de la máquina vendedora, así como la falta de refrescos en la máquina o si está llena la caja de dinero.

La ocurrencia de cada evento en la vida de un objeto debe ser considerada como una acción instantánea y atómica (sin duración). Otro detalle importante es ignorar de manera deliberada el tiempo exacto de duración de los eventos. La ventaja de ello es que nos permite, de manera resumida, el razonamiento y diseño de sistemas tanto físicos como de cómputo.

En la elección del alfabeto no haremos distinción entre los eventos iniciados por el objeto mismo y aquellos los cuales son iniciados por agentes externos, (al conjunto de agentes externos se le define como el medio ambiente). El evitar el concepto de causalidad nos llevará a simplificaciones considerables en la teoría y su aplicación.

Ahora definamos lo que es un proceso tomando como base lo explicado de los objetos. Un proceso es definido como el patrón de comportamientos de un objeto. El patrón de comportamientos de un objeto es formado por los elementos de su alfabeto. En otras palabras, un proceso es definido como el conjunto observable de eventos en los que el objeto modelado se compromete.

Como ejemplo de un proceso, considérese aquel que describe el comportamiento de una máquina vendedora de refrescos. La máquina recibe como entrada (del medio ambiente, cliente), una moneda de un peso, una segunda moneda de un peso, una moneda de 50 centavos y después da como salida una lata de refresco. De tal forma que el alfabeto del proceso *Máquina Vendedora* es el conjunto $\{\text{introduce } 1\text{ peso}, \text{ introduce } 1\text{ peso}, \text{ introduce } 50\text{ centavos}, \text{ proporciona } 1\text{ lata}\}$.

El proceso con alfabeto A , el cual nunca se compromete con evento alguno de A , es llamado $STOP_A$. Este describe el comportamiento de un objeto descompuesto: debido a que está equipado con la capacidad física de comprometerse en los eventos de A , pero nunca ejerce esa capacidad.

2.1.1 Prefijo

Sea x un evento y P un proceso, el operador de prefijo, denotado por el símbolo “ \rightarrow ”, opera entre un evento y un proceso de la siguiente manera:

$(x \rightarrow P)$ (se lee “ x entonces P ”)

describe un objeto el cual primero se compromete a ejecutar el evento x y después se comporta como el proceso P . En donde el proceso $(x \rightarrow P)$ debe tener el mismo alfabeto que el proceso P . Formalmente, tenemos:

$$\alpha(x \rightarrow P) = \alpha P \quad \text{con } x \in \alpha P$$

Ejemplos:

E1. Un torniquete de los que se ubican en el metro de la ciudad de México, el cual acepta como entrada un boleto antes de descomponerse.

$(\text{Boleto} \rightarrow STOP_{\text{alMetro}})$ □

E2. Una máquina vendedora de chocolates, la cual despacha dos chocolates antes de descomponerse.

$(\text{moneda} \rightarrow (\text{chocolate} \rightarrow (\text{moneda} \rightarrow (\text{chocolate} \rightarrow STOP_{\text{alM}}))))$

Inicialmente la máquina espera la inserción de una moneda antes de ofrecer un chocolate. Una vez insertada la moneda, la ranura donde fue insertada se cierra hasta que ha ofrecido un chocolate; una vez tomado éste, entonces se puede depositar la segunda moneda y retirar el segundo chocolate. La máquina entonces deja de funcionar. □

En el futuro, omitiremos los paréntesis en la definición de un proceso, en caso de secuencias lineales de eventos, tal como ocurre en el ejemplo anterior, dado que el operador de

prefijo es asociativo por la derecha. Esto se realiza siempre que el último elemento de la secuencial lineal sea un proceso, en caso contrario la expresión está sintácticamente mal escrita.

Nótese que el operador \rightarrow siempre toma un proceso del lado derecho y un solo evento del lado izquierdo. Si P y Q son procesos es sintácticamente incorrecto escribir:

$$P \rightarrow Q$$

Similarmenete si x y y son eventos es sintácticamente incorrecto escribir

$$x \rightarrow y$$

La forma correcta es: $(x \rightarrow (y \rightarrow STOP))$

Es importante distinguir el concepto de evento y de proceso así como la sintaxis correcta de los operadores sobre éstos.

2.1.2 Recursión

La notación en prefijo puede ser usada para describir el comportamiento completo de un proceso que eventualmente se detiene. Pero sería extremadamente tedioso escribir el comportamiento completo, por ejemplo, de una máquina vendedora de chocolates, para su diseño de vida máximo. Por lo cual, necesitamos una notación más corta para describir comportamientos repetitivos. La recursión permite la definición de un proceso como la solución de una ecuación.

Considere un reloj, el cual sólo hace *ticks*. Por lo cual

$$aReloj = \{tick\}$$

Considere a continuación un objeto que se comporta como un *Reloj*, excepto que éste primero emite un simple *tick*

$$(tick \rightarrow Reloj)$$

El comportamiento de este objeto es indistinguible del reloj original. Este razonamiento nos lleva a la formulación de la ecuación

$$Reloj = (tick \rightarrow Reloj)$$

Esto puede ser considerado como una definición implícita del comportamiento del *Reloj*. La ecuación para el reloj tiene resultados obvios, los cuales son derivados por sustituir iguales por iguales.

$$\begin{aligned}
\text{Reloj} &= (\text{tick} \rightarrow \text{Reloj}) \\
&= (\text{tick} \rightarrow (\text{tick} \rightarrow \text{Reloj})) \\
&= (\text{tick} \rightarrow (\text{tick} \rightarrow (\text{tick} \rightarrow \text{Reloj})))
\end{aligned}$$

ecuación original
sustituimos *Reloj* por su definición
similarmemente.

La ecuación puede ser reescrita tantas veces como se desee, y la posibilidad de desarrollarla aún más, se preserva. El comportamiento potencial ilimitado del reloj es definido como:

$$\text{tick} \rightarrow \text{tick} \rightarrow \text{tick} \rightarrow \text{tick} \rightarrow \dots \rightarrow \text{Reloj}$$

Esta autoreferenciación, o definición recursiva de proceso, trabajará adecuadamente sólo si el lado derecho de la ecuación inicia con al menos un evento prefijando a todas las ocurrencias recursivas del nombre del proceso. Por ejemplo la ecuación recursiva

$$X = X$$

no define nada, pues cualquier proceso es solución a esa ecuación. A la descripción de un proceso, que comienza con un prefijo, se le conoce como un proceso con guardia; en donde el prefijo es justamente la guardia. Si $F(X)$ es una expresión con guardia que contiene el nombre del proceso X , y A es el alfabeto de X , entonces decimos que la ecuación

$$X = F(X)$$

tiene una solución única con alfabeto A . En algunas ocasiones es conveniente denotar la solución por la expresión

$$\mu X : A . F(X)$$

En donde X es un nombre local (variable acotada) y puede ser cambiada, por lo cual,

$$\mu X : A . F(X) = \mu Y : A . F(Y)$$

Esta igualdad es justificada por el hecho de que una solución para X en la ecuación

$$\mu X : A . F(X)$$

es también una solución para Y en

$$\mu Y : A . F(Y)$$

Con la finalidad de hacer más entendible nuestro trabajo, en el futuro utilizaremos, ya sea, la definición del proceso o bien la notación μ . Frecuentemente omitiremos la mención explícita del alfabeto A , cuando sea obvio del contenido o contexto del proceso.

2.1.3 Elección

Un proceso que denota un solo patrón de comportamiento no es muy común. Por esa razón introducimos el operador de elección que nos permite definir comportamientos alternativos. Por ejemplo, el proceso P se define como:

$$P = (e_1 \rightarrow Q_1 \mid e_2 \rightarrow Q_2)$$

El proceso P tiene la elección de primero comprometerse con e_1 y después comportarse como Q_1 , o bien comprometerse con e_2 y después comportarse como Q_2 . El ambiente determina cual evento es elegido y por lo tanto el comportamiento futuro del proceso. Ejemplos:

E1. Una máquina que vende chocolates o caramelos en cada transacción

$$SMV = \mu X. (moneda \rightarrow (chocolate \rightarrow X \mid caramelo \rightarrow X))$$

E2. Una máquina más compleja aún, que vende chocolates “pequeños” o “grandes”, la cual regresa el cambio de acuerdo a lo comprado

$$\begin{aligned} MVC = & (in2pesos \rightarrow (ChocolateGrande \rightarrow MVC \\ & \mid ChocolatePequeño \rightarrow out1peso \rightarrow MVC) \\ & \mid in1peso \rightarrow (ChocolatePequeño \rightarrow MVC \\ & \mid in1peso \rightarrow (ChocolateGrande \rightarrow MVC))) \end{aligned}$$

La definición de elección puede fácilmente ser extendida a más de dos alternativas, esto es:

$$(x \rightarrow P \mid y \rightarrow Q \mid z \rightarrow R)$$

Nótese que el símbolo de elección “|” no es un operador entre procesos; sería sintácticamente incorrecto escribir $P|Q$, en donde P y Q son procesos. En general, si B es cualquier conjunto de eventos y $P(x)$ es una expresión que define un proceso para cada x distinta de B , entonces

$$(x : B \rightarrow P(x))$$

define un proceso el cual primero ofrece la elección de algún evento $x \in B$, y después se comporta como $P(x)$. Esto se lee como “ x de B entonces P de x ” El conjunto B define el menú inicial del proceso, puesto que contiene el conjunto de acciones iniciales que éste puede ejecutar. Ejemplo.

E3. Un proceso el cual todo el tiempo puede comprometerse con cualquier evento de su alfabeto A

$$\alpha Run_A = A$$

$$Run_A = (x : A \rightarrow Run_A)$$

En el caso especial de que el menú inicial contenga sólo el evento e , tenemos que:

$$(x : \{e\} \rightarrow P(x)) = (e \rightarrow P(e))$$

Puesto que e es el único evento inicial posible. Otro caso especial lo tenemos cuando el menú inicial está vacío, esto es, que nada puede ocurrir

$$(x : \{\} \rightarrow P(x)) = (y : \{\} \rightarrow Q(y)) = STOP.$$

2.1.4 Recursión mutua

La recursión permite la definición de un proceso simple como la solución de una ecuación. La técnica es fácilmente generalizada a la solución de conjuntos de ecuaciones simultáneas en donde más de una es desconocida. La recursión mutua es la definición de varios procesos mediante igual número de ecuaciones para que esto funcione adecuadamente, todos los lados derechos deben tener guardias, y cada uno de los procesos desconocidos debe aparecer exactamente una vez del lado izquierdo de una de las ecuaciones. Ejemplo:

E1. Un expendedor de bebidas tiene dos botones etiquetados con *NARANJA* y *LIMÓN*. Las acciones de presionar los botones son *EstableceNaranja* y *EstableceLimón*. Las acciones de despachar una bebida son *naranja* y *limón*. La elección de la bebida se lleva a cabo presionando el botón correspondiente y después ésta es despachada. No es despachada ninguna bebida si antes no se ha presionado ningún botón.

A continuación mostramos las ecuaciones del alfabeto y del comportamiento del proceso expendedor de bebidas (*EB*). La ecuación usa dos definiciones auxiliares N y L , las cuales son mutuamente recursivas.

$$\alpha EB = \alpha N = \alpha L = \{EstableceNaranja, EstableceLimón, naranja, limón\}$$

$$EB = (EstableceNaranja \rightarrow N \mid EstableceLimón \rightarrow L)$$

$$N = (naranja \rightarrow N \mid EstableceLimón \rightarrow L \mid EstableceNaranja \rightarrow N)$$

$$L = (limón \rightarrow L \mid EstableceNaranja \rightarrow N \mid EstableceLimón \rightarrow L)$$

Informalmente, después del primer evento, el expendedor se encuentra en uno de los estados N o L . En cada estado, servirá la bebida adecuada o bien conmutará (establecerá) el otro estado. Se podrá presionar el botón del mismo estado, pero sin efecto alguno. \square

2.1.5 Trazas

Una *traza* del comportamiento de un proceso, es una secuencia finita de símbolos que registra los eventos en los cuales se compromete dicho proceso en algún instante de tiempo. Imaginemos que existe un observador que vigila el comportamiento de un proceso y registra los eventos que ejecuta en una libreta. Si dos eventos ocurren simultáneamente, entonces el observador escribe uno y después el otro si importan el orden.

Una traza es denotada por una secuencia de símbolos, separada por comas y encerrada entre los símbolos “<” y “>”.

- $\langle x, y \rangle$ consiste de dos eventos, x seguido de y .
- $\langle x \rangle$ es una secuencia que contiene sólo el evento x .
- $\langle \rangle$ es la secuencia vacía, la cual no contiene eventos.

Ejemplos.

E1. La traza de la Máquina Vendedora Simple *MVS* (2.1.3 E1) al momento que ha atendido a un cliente

$$\text{Trazas}(MVS) = \{ \langle \text{moneda}, \text{chocolate} \rangle, \langle \text{moneda}, \text{caramelo} \rangle \} \quad \square$$

E2. Las trazas posibles de la Máquina Vendedora Simple *MVS* (2.1.3 E1) después de atender a dos clientes

$$\begin{aligned} \text{Trazas}(MVS) = \{ & \langle \text{moneda}, \text{chocolate}, \text{moneda}, \text{chocolate} \rangle, \\ & \langle \text{moneda}, \text{chocolate}, \text{moneda}, \text{caramelo} \rangle, \\ & \langle \text{moneda}, \text{caramelo}, \text{moneda}, \text{chocolate} \rangle, \\ & \langle \text{moneda}, \text{caramelo}, \text{moneda}, \text{caramelo} \rangle \} \quad \square \end{aligned}$$

2.1.5.1 Operaciones sobre trazas.

Las trazas juegan un rol central en la descripción y entendimiento del comportamiento de los procesos. En esta sección exploramos las principales operaciones con trazas, para ello usamos la siguiente convención.

- s, t, u se usarán para el manejo de trazas
- S, T, U se usarán para el manejo de conjuntos de trazas
- f, g, h se usarán para el manejo de funciones

Concatenación: construye una traza con un par de operandos (trazas) s y t , simplemente uniendo s seguida de t ; el resultado es denotado por:

$$s \cdot t$$

Por ejemplo

$$\langle \text{moneda, chocolate} \rangle \cdot \langle \text{moneda, caramelo} \rangle = \langle \text{moneda, chocolate, moneda, caramelo} \rangle$$

Restricción: la expresión $(t \uparrow A)$ denota la traza cuando es restringida a los símbolos del conjunto A ; ésta es formada por t omitiendo los símbolos fuera de A . Por ejemplo

$$\langle \text{moneda, chocolate, moneda, caramelo} \rangle \uparrow \{\text{chocolate, caramelo}\} = \langle \text{chocolate, caramelo} \rangle$$

Head y Tail: si s es una secuencia no vacía, su primer símbolo es denotado por s_0 , y el resultado de eliminar su primer símbolo es s' . Por ejemplo

$$\begin{aligned} \langle x, y, z \rangle_0 &= x \\ \langle x, y, z \rangle' &= \langle y, z \rangle \end{aligned}$$

Longitud: La longitud de una traza t es denotada por $\#t$. Por ejemplo

$$\# \langle x, y, z \rangle = 3$$

Composición: sea \surd , el símbolo que denota la terminación satisfactoria de un proceso. Por lo cual, este símbolo sólo puede aparecer al final de una traza. Sea t una traza que registra una secuencia de eventos, los cuales inician cuando s ha terminado satisfactoriamente. La composición de s y t es denotada por $(s ; t)$. Si \surd , no ocurre en s , entonces t no puede iniciar.

2.1.6 Trazas de un Proceso

Una *traza* de un proceso es un registro secuencial de su comportamiento hasta cierto instante de tiempo. Antes de que el proceso inicie, no se conoce cual de sus posibles trazas será registrada: la elección depende de el ambiente que controla al proceso. Sin embargo, el conjunto de todas las posibles trazas de el proceso P puede conocerse de antemano, definamos ese conjunto como las *Trazas*(P).

Ejemplos:

E1. La única traza del comportamiento de el proceso *STOP* es $\langle \rangle$

$$\text{Trazas}(\text{STOP}) = \{ \langle \rangle \}$$

E2. Hay solamente dos trazas en una máquina que acepta una moneda y después se descompone.

$$\text{Trazas}(\text{moneda} \rightarrow \text{STOP}) = \{ \langle \rangle, \langle \text{moneda} \rangle \} \quad \square$$

E3. Las trazas de un reloj, son:

$$\text{Trazas}(\mu X. \text{tick} \rightarrow X) = \{ \langle \rangle, \langle \text{tick} \rangle, \langle \text{tick}, \text{tick} \rangle, \langle \text{tick}, \text{tick}, \text{tick} \rangle, \dots \} \quad \square$$

2.1.6.1 Leyes sobre trazas de procesos

En esta sección mostraremos como calcular el conjunto de trazas de un proceso, usando las notaciones introducidas hasta aquí. Como se mencionó anteriormente, *STOP* tiene solamente una traza

$$L1. \text{Trazas}(\text{STOP}) = \{ t \mid t = \langle \rangle \} = \{ \langle \rangle \}$$

Una traza de $(c \rightarrow P)$ puede estar vacía, debido a que $\langle \rangle$ es una traza de cualquier proceso hasta el momento en que se compromete en su primer acción. Cualquier traza no vacía deberá comenzar con c y el resto debe ser una posible traza de P .

$$L2. \text{Trazas}(c \rightarrow P) = \{ t \mid t = \langle \rangle \vee (t_0 = c \wedge t' \in \text{Trazas}(P)) \} \\ = \{ \langle \rangle \} \cup \{ \langle c \rangle \wedge t \mid t \in \text{Trazas}(P) \}$$

Un proceso que ofrece una elección entre sus posible primeros eventos, contiene una alternativa de trazas.

$$L3. \text{Trazas}(c \rightarrow P \mid d \rightarrow Q) = \\ \{ t \mid t = \langle \rangle \vee (t_0 = c \wedge t' \in \text{Trazas}(P)) \vee (t_0 = d \wedge t' \in \text{Trazas}(Q)) \}$$

Las anteriores tres leyes pueden ser resumidas en la siguiente ley general.

$$L4. \text{Trazas}(x:B \rightarrow P(x)) = \{ t \mid t = \langle \rangle \vee (t_0 \in B \wedge t' \in \text{Trazas}(P(t_0))) \}$$

Descubrir el conjunto de trazas de un proceso definido recursivamente es un poco más complejo. Un proceso definido recursivamente es la solución a una ecuación como:

$$X = F(X)$$

Primero definimos la iteración de la función F por inducción

$$F^0(X) = X$$

$$\begin{aligned}
 F^{n+1}(X) &= F(F^n(X)) \\
 &= F^n(F(X)) \\
 &= F(\dots(F(F(X)))\dots)
 \end{aligned}$$

entonces, dado que F tiene guardias, podemos definir

$$L5. \text{Trazas}(\mu X:A. F(X)) = \bigcup_{n \geq 0} \text{Trazas}(F^n(\text{STOP}_A))$$

$$L6. \langle \rangle \in \text{Trazas}(P)$$

$$L7. s \wedge t \in \text{Trazas}(P) \Rightarrow s \in \text{Trazas}(P)$$

$$L8. \text{Trazas}(P) \subseteq (\alpha P)^*$$

en donde $*$, denota la concatenación de cero o más veces el elemento, o conjunto de elementos, que tiene precedido.

2.2 Concurrencia

Cuando dos procesos son reunidos para ejecutarse concurrentemente, es para que interactúen uno con otro. Esas interacciones pueden ser eventos que requieran la participación simultánea de ambos procesos. Por el momento centremos nuestra atención sobre tales eventos e ignoremos los demás. Así tenemos que los alfabetos de los dos procesos son los mismos. Por ejemplo un chocolate puede ser extraído de la máquina vendedora, sólo si, el cliente quiere extraerlo y, si además, la máquina vendedora está preparada para ofrecerlo. Si P y Q son procesos con el mismo alfabeto, tenemos que

$$P \parallel Q, \quad \alpha P = \alpha Q = \alpha(P \parallel Q)$$

denota el comportamiento del proceso (P concurrente con Q) compuesto de los procesos P y Q interactuando de manera síncrona paso a paso.

Ejemplo

E1. Un cliente fraudulento de una máquina vendedora, generalmente revisa primero si puede obtener un chocolate o un caramelo sin pagarlo. Si no es así, entonces deposita la moneda y obtiene el producto que desea. El comportamiento del proceso cliente es

$$\text{Cliente} = (\text{caramelo} \rightarrow \text{Cliente} \mid \text{chocolate} \rightarrow \text{Cliente} \mid \text{moneda} \rightarrow (\text{chocolate} \mid \text{caramelo}))$$

Por otro lado la máquina vendedora no permite sacar un producto, si es que antes no se ha depositado el importe de éste. El comportamiento de la máquina es

$$\text{SMV} = (\text{moneda} \rightarrow \text{chocolate} \mid \text{moneda} \rightarrow \text{caramelo})$$

El comportamiento del sistema completo esta constituido por la composición de los subprocesos *Cliente* y *SMV*, tal como se expresa a continuación

$$(\text{Cliente} \parallel \text{SMV}) \quad \alpha \text{Cliente} = \alpha \text{SMV} = \alpha(\text{Cliente} \parallel \text{SMV})$$

en donde un producto de la máquina no puede ser extraído sin antes pagarlo. □

El operador descrito en el párrafo anterior puede ser generalizado al caso cuando los operandos *P* y *Q* tienen diferentes alfabetos ($\alpha P \neq \alpha Q$). Cuando los procesos son reunidos a ejecutarse concurrentemente, los eventos que están en ambos alfabetos requieren la participación simultánea de ambos procesos. Por otro lado, los eventos dentro del alfabeto de *P* y fuera del de *Q*, son ejecutados de manera independiente por *P*, siempre que *P* se comprometa con ellos. Similarmente, *Q* puede comprometerse con eventos dentro de su alfabeto y ajenos a *P*, de manera independiente. Así el conjunto de todos los eventos que son lógicamente posibles, es la unión de los alfabetos de los procesos componentes.

$$\alpha (P \parallel Q) = \alpha P \cup \alpha Q$$

Ejemplo

E.2 Sea $\alpha MVA = \{\text{moneda, chocolate, klik, klok, caramelo}\}$, el alfabeto de una máquina vendedora automática donde, *klik*, es el sonido que emite la máquina al insertar una moneda y *klok* es el sonido que emite cuando una transacción ha concluido. Los eventos con los cuales se compromete esta máquina son

$$MVA = (\text{moneda} \rightarrow \text{klik} \rightarrow \text{chocolate} \rightarrow \text{klok} \rightarrow MVA)$$

en donde no aparece caramelo. El cliente de esta máquina prefiere un caramelo; justamente al momento de introducir su moneda, se da cuenta de que la máquina no tiene caramelos, por lo cual emite un grito de enojo, y molesto recoge el chocolate dado por la máquina.

$$\alpha \text{Cliente} = \{\text{moneda, chocolate, grito, caramelo}\}$$

$$\text{Cliente} = (\text{moneda} \rightarrow (\text{caramelo} \rightarrow \text{Cliente} \mid \text{grito} \rightarrow \text{chocolate} \rightarrow \text{Cliente}))$$

El resultado de la actividad concurrente de estos dos procesos es:

$$(\text{Cliente} \parallel MVA) = \mu X. (\text{moneda} \rightarrow (\text{klik} \rightarrow \text{grito} \rightarrow \text{chocolate} \rightarrow \text{klok} \rightarrow X \mid \text{grito} \rightarrow \text{klik} \rightarrow \text{chocolate} \rightarrow \text{klok} \rightarrow X))$$

Note que *klik* puede ocurrir antes del *grito*, o bien, puede el cliente primero emitir el *grito* y después se escucha el *klik*. No importa el orden en el cual sean registrados. Note también que la fórmula matemática no representa el hecho de que el cliente prefiera un *caramelo* mas que

expresar un *grito*. La fórmula es una expresión de la realidad, La cual ignora emociones humanas y se concentra sólo en describir las posibilidades de ocurrencia y no ocurrencia de eventos dentro del alfabeto del proceso. \square

2.2.1 Trazas

Si $\alpha P = \alpha Q$:

Dado que cada acción de $(P||Q)$ requiere la participación simultánea de ambos proceso, P y Q , cada secuencia de acciones es posible para ambos procesos, por lo cual:

$$\text{Trazas}(P||Q) = \text{Trazas}(P) \cap \text{Trazas}(Q)$$

Si $\alpha P \neq \alpha Q$:

Sea t una traza de $(P||Q)$, entonces cualquier evento en t , dentro del alfabeto de P , ha sido un evento en la vida de P ; y cualquier evento en t , que no esté en el alfabeto de P , ha ocurrido sin la participación de éste. Así $(t \uparrow \alpha P)$ es una traza, la cual contiene únicamente eventos en los cuales P ha participado, y es por lo tanto, una traza de P . Por argumentos similares $(t \uparrow \alpha Q)$ es una traza de Q . Por lo tanto, cualquier evento en t debe estar entre αP o αQ . Esto nos sugiere que

$$\text{Trazas}(P||Q) = \{t \mid (t \uparrow \alpha P) \in \text{Trazas}(P) \wedge (t \uparrow \alpha Q) \in \text{Trazas}(Q) \wedge t \in (\alpha P \cup \alpha Q)^*\}$$

Ejemplo

E1. Ver el ejemplo E2 de la sección 2.2.1 (2.2.1 E2).

Sea $t1 = \langle \text{moneda}, \text{clik}, \text{grito} \rangle$

entonces $t1 \uparrow \alpha MVA = \langle \text{moneda}, \text{clik} \rangle$

el cual está en $\text{Trazas}(MVA)$

y $t1 \uparrow \alpha \text{Cliente} = \langle \text{moneda}, \text{grito} \rangle$

el cual está en $\text{Trazas}(\text{Cliente})$

Por lo tanto, $t1 \in \text{Trazas}(MVA || \text{Cliente})$.

Razonamientos similares muestran que

$$\langle \text{moneda}, \text{grito}, \text{clik} \rangle \in \text{Trazas}(MVA || \text{Cliente})$$

Esto muestra que *grito* y *clik* pueden ser registrados uno después del otro en cualquier orden \square

2.3 No Determinismo

2.3.1 No determinismo interno

Si P y Q son procesos, entonces

$$P \amalg Q \quad (P \circ Q)$$

denota un proceso el cual se comporta como P , o bien como Q . La selección es arbitraria, sin el conocimiento o control del ambiente externo. Los alfabetos de ambos procesos son los mismos, esto es

$$\alpha(P \amalg Q) = \alpha P = \alpha Q$$

Ejemplos

E1. Una máquina que proporciona monedas fraccionarias en cantidad similar a la cantidad insertada. Esta máquina ofrece cambio en una de dos posibles maneras

$$MC5P = (in5p \rightarrow ((out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow MC5P) \amalg (out2p \rightarrow out1p \rightarrow out2p \rightarrow MC5P))) \quad \square$$

E2. $MC5P$ puede dar de manera diferente cambio en cada ocasión de uso. Una máquina que siempre da la misma combinación, pero no conocemos de manera inicial de que forma lo da es:

$$MC5A = (in5p \rightarrow out1p \rightarrow out1p \rightarrow out1p \rightarrow out2p \rightarrow MC5A) \\ MC5B = (in5p \rightarrow out2p \rightarrow out1p \rightarrow out2p \rightarrow MC5B)$$

$$MC5E = MC5A \amalg MC5B$$

Claro, después que la máquina da cambio por primera vez, el comportamiento de ésta es totalmente predecible. □

Trazas de procesos no determinísticos

Si s es una traza de P , entonces s es también una posible traza de $(P \amalg Q)$, esto es, en el caso de que P sea seleccionado. Similarmente si s es una traza de Q , ésta es también una posible traza de $(P \amalg Q)$. Análogamente, cada traza de $(P \amalg Q)$ es una traza de alguna de las dos alternativas. Es decir

$$Trazas(P \amalg Q) = Trazas(P) \cup Trazas(Q)$$

2.3.2 No determinismo externo

El ambiente de $(P \amalg Q)$ no tiene control ni conocimiento de la elección que es hecha entre P y Q , o el tiempo en el cual es hecha. Debido a ello, $(P \amalg Q)$ no es un forma adecuada para combinar procesos, ya que el ambiente debe estar preparado para tratar con P o Q . Por lo tanto una operación alternativa es $(P \sqcap Q)$, en la cual el ambiente controla cual proceso es seleccionado; ese control es ejercido por el ambiente desde la primera acción. Si esa primera acción, no es la primera acción de P , entonces Q es seleccionado; pero si Q no se compromete inicialmente con esa acción, P es seleccionado. Por otro lado, si esta acción es la primera de P y también de Q , entonces la elección entre ellas es de manera no determinística $(P \amalg Q)$. (Claro, si el evento es imposible para P , y para Q , un bloqueo ocurre). Como es usual

$$\alpha(P \sqcap Q) = \alpha P = \alpha Q$$

En el caso en que los eventos iniciales de P y Q son distintos, el operador de elección no determinista es equivalente al operador de elección como se muestra en la siguiente expresión

$$(c \rightarrow P \sqcap d \rightarrow Q) = (c \rightarrow P \mid d \rightarrow Q) \quad \text{si } c \neq d$$

Si los eventos iniciales son los mismos, $(P \sqcap Q)$ degenera en $(P \amalg Q)$.

$$(c \rightarrow P \sqcap d \rightarrow Q) = (c \rightarrow P \amalg d \rightarrow Q) \quad \text{si } c = d.$$

Trazas

Cualquier traza de $(P \sqcap Q)$, es una traza de P o de Q y viceversa.

$$\text{Trazas}(P \sqcap Q) = \text{Trazas}(P) \cup \text{Trazas}(Q)$$

2.3.3 Rechazos

La distinción entre $(P \amalg Q)$ y $(P \sqcap Q)$ es verdaderamente sutil, dado que no pueden ser distinguidos por sus trazas, pues cada traza de uno resulta ser también del otro. Sin embargo, es posible introducirlos en un ambiente en el cual $(P \amalg Q)$ puede ir a un bloqueo en su primer paso y $(P \sqcap Q)$ no. Por ejemplo, sea $x \neq y$ y

$$P = (x \rightarrow P), \quad Q = (y \rightarrow Q), \quad \alpha P = \alpha Q = \{x, y\}$$

$$\text{entonces} \quad (P \sqcap Q) \parallel P = (x \rightarrow P) \\ = P$$

$$\text{Pero} \quad (P \amalg Q) \parallel P = (P \parallel P) \amalg (Q \parallel P) \\ = P \amalg \text{STOP}$$

Esto muestra que dentro del ambiente P , y $(P \parallel Q)$ pueden alcanzar un bloqueo pero $(P \square Q)$ no. Claro que $(P \parallel Q)$ puede no necesariamente ir a un bloqueo, si es así, no habrá forma de que nosotros sepamos que un bloqueo pudo ocurrir. Pero la mera posibilidad de ocurrencia de un bloqueo es suficiente para distinguir entre $(P \parallel Q)$ y $(P \square Q)$.

En general, sea X un conjunto de eventos los cuales son ofrecidos inicialmente por el ambiente a un proceso P , con el alfabeto del ambiente igual al de P . Si es posible para P ir a un bloqueo con cualquier primer evento que ejecute del conjunto X , decimos que X es un rechazo de P . El conjunto de todos los rechazos de P es denotado por

$refusals(P)$

Note que los rechazos constituyen una familia de conjuntos de símbolos. La introducción del concepto de rechazo permite una distinción mas clara entre los procesos deterministas y los no deterministas. Un proceso se dice determinista si este no puede rechazar algún evento con el cual se comprometió. En otras palabras, un conjunto es un rechazo de un proceso determinista, sólo si éste conjunto no contiene los eventos que inicialmente ejecuta el proceso; más formalmente, tenemos

P es determinista $\Rightarrow (X \in refusals(P) \equiv (X \cap P^0 = \{\}))$
 en donde $P^0 = \{x \mid \langle x \rangle \in Trazas(P)\}$

Esa condición se aplica no sólo en el paso inicial, sino también después de cualquier secuencia de acciones de P

Por otro lado, un proceso no determinista no disfruta de estas propiedades, esto es, en algún momento puede existir algún evento con el cual se compromete; pero también puede rechazar ese mismo evento (como resultado de una elección no determinista interna).

2.3.3 Fallas

En ocasiones es insuficiente conocer sólo los rechazos iniciales de un proceso P . Por lo que es necesario también tomar en cuenta que P puede rechazar eventos después de haberse comprometido en una traza arbitraria s . Por lo que definimos las Fallas de un proceso como una relación (un conjunto de parejas) definida por:

$$f(P) = failures(P) = \{(s, X) \mid s \in trazas(P) \wedge X \in refusals(P/s)\}$$

Si (s, X) es una falla de P , eso significa que P puede comprometerse en la secuencia de eventos registrados por s , y después rechazar cualquier evento del conjunto X . Las fallas de un proceso nos revelan más información que los simples rechazos o las trazas, los cuales pueden ser definidos en términos de las fallas

$$trazas(P) = \{s \mid \exists X. (s, X) \in f(P)\}$$

$$\text{refusals}(P) = \{X \mid \langle \cdot, X \rangle \in f(P)\}$$

2.3.4 Ocultamiento

En general, el alfabeto de un proceso contiene solamente los eventos que son considerados relevantes. Para describir el comportamiento interno de un mecanismo, necesitamos considerar eventos que representan transiciones internas de éste. Tales eventos denotan las comunicaciones e interacciones entre los componentes del mecanismo, los cuales actúan concurrentemente. Después de la construcción del mecanismo, ocultamos la estructura de sus componentes, y también ocultamos todas las acciones internas de éste, de tal forma que no puedan ser observadas por algún observador o controladas por el ambiente del proceso. Si C es un conjunto de eventos que queremos ocultar de esa forma, entonces

$$P \setminus C$$

es un proceso el cual se comporta como P , excepto que los eventos dentro de C son ocultados. Como consecuencia, tenemos

$$\alpha(P \setminus C) = \alpha(P) - C$$

Ejemplo

E.1. Máquina vendedora automática MVA del ejemplo 2.2.1.E.2 puede ser introducida en una caja a prueba de ruido, con lo cual, tenemos

$$MVA \setminus \{\text{clik}, \text{clok}\}$$

lo que forma una Máquina sin Ruido □

2.3.5 Entrelazamiento

En ocasiones es necesario ejecutar procesos concurrentes con el mismo alfabeto sin que interactúen o se sincronicen. En este caso, cada acción del sistema, es acción de exactamente un proceso. Si un proceso se compromete con una acción, ésta la ejecuta independientemente de los procesos que se ejecuten concurrentemente con él. Si dos procesos concurrentes se comprometen con la misma acción se elige alguno de ellos de manera no determinista para que la ejecute primero y posteriormente el otro la ejecuta. Esta forma de combinación es denotada por

$P \parallel Q$ (P entrelazado con Q)

en donde su alfabeto se define como

$$\alpha(P \parallel Q) = \alpha P = \alpha Q$$

Ejemplo

E1. Una máquina vendedora que acepta hasta dos monedas antes de despachar hasta dos productos entre chocolate y caramelo, (2.1.3 E1).

$$(VMS \parallel VMS) = VMS2$$

si sustituimos VMS, por su definición, tenemos:

$$VMS2 = (moneda \rightarrow chocolate \mid moneda \rightarrow caramelo) \\ \parallel (moneda \rightarrow chocolate \mid moneda \rightarrow caramelo)$$

puesto que cada proceso ejecuta sus eventos de manera independiente, el primer proceso puede vender cualquiera de sus dos productos sin interactuar con el segundo proceso, y viceversa. Por lo que el conjunto de eventos entrelazados de los dos procesos puede aparecer en cualquier orden, siempre, claro, respetando el orden de ejecución de eventos de cada proceso. \square

2.3.6 Divergencia

En la sección 2.1.2 mencionamos que las ecuaciones recursivas con guardias tienen soluciones únicas, desafortunadamente, la introducción del ocultamiento (2.3.4) dentro de estas ecuaciones, provoca comportamientos caóticos en éstas. Por ejemplo, considere la ecuación

$$X = c \rightarrow (X \setminus \{c\})$$

describe al proceso que primero se compromete con el evento c y después se comporta de manera caótica, pues ejecuta una serie infinita de eventos internos, no conocidos por el ambiente. Para explicar el comportamiento caótico mostraremos el caso más simple, el cual resulta también el peor caso, esto es, la recursión infinita

$$\mu X.X$$

Cualquier proceso es una solución de la ecuación recursiva

$$X = X$$

Consecuentemente $\mu X . X$ puede comportarse como cualquier proceso, este es el mayor proceso no determinista, el menos predecible, el menos controlable, en resumen, el peor. Este proceso es definido como:

$$CHAOS_A = \mu X : A . X$$

Una divergencia de un proceso es definida como cualquier traza del proceso después del cual, éste, se comporta caóticamente. El conjunto de todas las divergencias esta definido como:

$$d(P) = \text{divergencias}(P) = \{s \mid s \in \text{Trazas}(P) \wedge (P/s) = CHAOS_{\alpha P}\}$$

De lo anterior concluimos que

$$d(P) \subseteq \text{Trazas}(P)$$

2.4 Comunicación

2.4.1 Introducción

En las secciones previas hemos introducido e ilustrado el concepto general de evento como una acción sin duración, cuya ocurrencia puede requerir la participación simultánea de más de un proceso. En esta sección nos concentraremos en una clase especial de evento conocido como *comunicación*. Una comunicación es un evento que es descrito por el par

$c.v$

en donde c es el nombre del canal sobre el cual la comunicación tendrá a lugar y v es el valor del mensaje a pasar. El conjunto de todos los mensajes que P puede comunicar por el canal c está definido por

$$\alpha_c(P) = \{v \mid c.v \in \alpha P\}$$

También se definen las funciones que extraen los componentes *canal* y *mensaje* de la comunicación

$$\text{channel}(c.v) = c \quad \text{message}(c.v) = v$$

Sea v un miembro de $\alpha_c(P)$. Un proceso el cual primero emite v por el canal c , y después se comporta como P es definido por

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Este proceso solamente está preparado a comprometerse de manera inicial con el evento de comunicación $c.v$

Un proceso el cual esta inicialmente preparado a recibir cualquier valor x que sea comunicado por el canal c , para después comportarse como $P(x)$, es definido por

$$(c?x \rightarrow P(x)) = (y : \{y \mid channel(y) = c\} \rightarrow P(message(y)))$$

Ejemplo

E1 Construyamos un buffer de un solo dato binario (*CopyBit*), para ello necesitamos esperar recibir un elemento por un canal de entrada, y cuando sea solicitado el elemento, lo emitimos por el canal de salida. Este comportamiento esta dado por la siguiente expresión

$$CopyBit = \mu X. (in?x \rightarrow (out!x \rightarrow X))$$

en donde $\alpha_{in}(CopyBit) = \alpha_{out}(CopyBit) = \{0,1\}$ □

E2 Un proceso que copia cualquier mensaje de su entrada a su salida.

$$\alpha_{left}(COPY) = \alpha_{right}(COPY)$$

$$COPY = \mu X (left?x \rightarrow right!x \rightarrow X)$$
 □

2.4.2 Canales de Comunicación

Sean P y Q procesos, y sea c un canal usado para emisión por P y para recepción por Q . Así el conjunto que contiene todos los eventos de comunicación de la forma $c.v$ esta dentro de la intersección de los alfabetos de P y Q . Cuando esos procesos son compuestos concurrentemente en el sistema ($P \parallel Q$), una comunicación $c.v$ puede ocurrir solamente cuando ambos procesos se comprometen simultáneamente en ese evento. i.e., siempre que P emita un valor v sobre el canal c y Q simultáneamente recibe el mismo valor. Un proceso que recibe un valor por un canal, debe estar preparado para aceptar cualquier valor comunicable. El proceso emisor es el que determina el valor del mensaje a comunicar en cada ocasión.

Así la emisión es considerada como un caso especializado del operador de prefijo, y la recepción un caso especial de elección; esto nos lleva a lo siguiente

$$(c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)) = c!v \rightarrow (P \parallel Q(v))$$

Ejemplo

E1 Sea P un proceso que recibe por el canal izq un valor y emite por el canal $inter$ el cuadrado del valor recibido, y sea Q un proceso que recibe por el canal $inter$ un valor y emite ese valor multiplicado por 10 por el canal der . La composición concurrente (Z) de estos procesos recibe por el canal izq un valor, y emite el cuadrado de éste multiplicado por 10 por el canal der . Esto se ilustra en las siguientes expresiones.

$$Z = (P \parallel Q)$$

$$P = (izq?x \rightarrow inter!(x*x) \rightarrow P)$$

$$Q = (inter?y \rightarrow der(y*10) \rightarrow Q)$$

□

E2 Dos flujos de números son introducidos por $izq1$ e $izq2$. Por cada x leída por $izq1$ y cada y leída de $izq2$, emitiremos el número $(a*x + b*y)$ por el canal $derecho$. Los requerimientos de rapidez sugieren que la multiplicación se ejecute concurrentemente. Por lo tanto definimos dos procesos y después los componemos concurrentemente.

$$AX = (izq1?x \rightarrow inter!(a*x) \rightarrow AX)$$

$$BY = (izq2?y \rightarrow inter?z \rightarrow derecho!(z + b*y) \rightarrow BY)$$

$$Z = (AX \parallel BY)$$

□

2.4.3 Tubos de comunicación

En este apartado centramos nuestra atención en los procesos con sólo dos canales en su alfabeto, denominados canal *izquierdo* o de recepción y canal *derecho* o de emisión. Tales procesos son llamados *tubos de comunicación*, y se ilustran a continuación



Los procesos P y Q pueden ser unidos conectando el canal derecho de P con el canal izquierdo de Q , y la secuencia de mensajes emitidos por P y recibidos por Q sobre ese canal interno es oculto de su ambiente común. El resultado de esta conexión es denotado por

$$P \blacktriangleright Q$$

y puede ser ilustrada como la siguiente serie



Este diagrama muestra el ocultamiento de el canal de conexión entre los procesos, al no mostrar el nombre de el canal. El diagrama también muestra que todos los mensajes introducidos por el canal izquierdo de $(P \bowtie Q)$ son introducidos a P , y todos los mensajes emitidos por el canal derecho de $(P \bowtie Q)$ son emitidos por Q . Finalmente $(P \bowtie Q)$ es en si mismo un tubo de comunicación, y puede ser utilizado para formar tubos de comunicación más complejos, como

$(P \bowtie Q) \bowtie R$, $(P \bowtie Q) \bowtie (R \bowtie S)$, etc.

2.4.4 Subordinación

Sean P y Q procesos con

$$\alpha P \subseteq \alpha Q$$

En la combinación $(P||Q)$, cada acción de P puede ocurrir sólo cuando Q lo permita; mientras tanto Q puede comprometerse independientemente con las acciones $(\alpha Q - \alpha P)$, sin el permiso y sin el conocimiento de P . Así P , sirve como proceso esclavo o subordinado, mientras Q actúa como proceso maestro o principal. Cuando las comunicaciones entre el proceso subordinado y el principal se ocultan de su ambiente común, usaremos la notación asimétrica

$$P // Q$$

Usando el operador de ocultamiento, esto es definido como

$$P // Q = (P || Q) \setminus \alpha P$$

Esta notación es solamente usada cuando $\alpha P \subseteq \alpha Q$; y entonces

$$\alpha(P // Q) = (\alpha Q - \alpha P)$$

Usualmente es conveniente dar un nombre al proceso subordinado, digamos m , el cual es usado por el proceso principal para todas las interacciones con éste. La forma de nombrar los componentes de comunicación del proceso subordinado es de la forma $m.c$, en donde m es el nombre del proceso y c es el nombre de uno de sus canales. Cada comunicación sobre ese canal es una tripleta

$$m.c.v$$

donde $\alpha m.c(m:P) = \alpha c(P)$ y $v \in \alpha c(P)$

En la construcción de $(m:P//Q)$, Q se comunica con P a través de los canales con nombres compuestos de la forma $m.c$ y $m.d$; mientras que P usa la notación simple correspondiente a los canales c y d para la misma comunicación. Así por ejemplo

$$(m: (c!v \rightarrow P) // (m.c?x \rightarrow Q(x))) = (m:P // Q(v))$$

Dado que todas las comunicaciones se ocultan al medio ambiente, el nombre m nunca puede ser detectado fuera; por lo tanto sirve como nombre local para el proceso subordinado.

La subordinación puede ser anidada, por ejemplo

$$(n: (m: P//Q) //R)$$

En este caso, todas las ocurrencias de eventos que involucran el nombre de m son ocultadas antes de que el nombre de n sea ligado a el resto de los eventos, los cuales están en el alfabeto de Q , y no en el de P . No hay forma de que R pueda comunicarse directamente con P , ni de que conozca su existencia o su nombre.

Ejemplos

E1.

$$doble:DOBLE // Q$$

$$DOBLE = \mu X(izq?x \rightarrow der!(x+x) \rightarrow X)$$

El proceso subordinado actúa como una simple subrutina llamada desde el proceso principal Q . Dentro de Q , el valor de $2*e$ puede ser obtenido al emitir el argumento e a través del canal izq de $doble$ y el resultado es obtenido por el canal der de $doble$, esto es

$$Q = doble.izq!e \rightarrow doble.der?resultado$$

□

E2. Un proceso puede usar otro proceso subordinado, tantas veces como se desee.

$$CUÁDRUPLE = (doble:DOBLE // (\mu X. izq?x \rightarrow doble.izq!x \rightarrow \\ doble.der?y \rightarrow doble.izq!y \rightarrow \\ doble.der?z \rightarrow der!z \rightarrow X))$$

Este mismo proceso puede también ser usado como subrutina

$$cuádruple: CUÁDRUPLE // Q$$

□

2.5 Procesos Secuenciales

El proceso *STOP* fue definido como un proceso que nunca se compromete con acción alguna. Este es un proceso escasamente utilizado, y probablemente resulte de un candado mortal (bloqueo) o de algún error de diseño, más que de alguna elección deliberada del diseñador. Sin embargo, existe la posibilidad de que un proceso no ejecute nada más, debido a que ha realizado todo para lo que fue diseñado. Tal proceso se dice que ha terminado satisfactoriamente. Para distinguir la terminación satisfactoria de *STOP*, es conveniente considerar la terminación satisfactoria como un evento especial, denotado por el símbolo \checkmark . Un proceso secuencial es definido como un proceso que contiene el símbolo \checkmark en su alfabeto; y naturalmente ese puede ser solamente el último evento en el cual se compromete. Se estipula que \checkmark no puede ser una alternativa en el constructor de elección

$(x: B \rightarrow P(x))$ es incorrecto si $\checkmark \in B$

$SKIP_A$ es definido como un proceso el cual no hace nada y termina de manera satisfactoria

$$\alpha SKIP_A = A \cup \{\checkmark\}$$

Ejemplo

E1. Una máquina vendedora que pretende atender solamente a un cliente, ya sea con un chocolate o con un caramelo para después terminar de manera satisfactoria.

$$MV1 = (\text{moneda} \rightarrow (\text{chocolate} \rightarrow SKIP \mid \text{caramelo} \rightarrow SKIP)) \quad \square$$

Al diseñar un proceso que resuelva una tarea compleja, frecuentemente partimos la tarea en dos subtareas, una de las cuales debe terminar satisfactoriamente antes de que la otra comience. Si P y Q son procesos secuenciales con el mismo alfabeto, Su composición secuencial

$$P ; Q$$

es un proceso el cual se comporta como P , cuando P termina satisfactoriamente, $(P;Q)$ continua comportándose como Q . Si P nunca termina $(P;Q)$ tampoco lo hará.

Ejemplo

E2. Una máquina vendedora diseñada para atender exactamente a dos clientes, uno después de otro.

$$MV2 = MV1 ; MV1 \quad \square$$

2.5.1 Interrupciones

En esta sección definimos una clase de composición secuencial ($P \nabla Q$), la cual no depende de la terminación satisfactoria de P . El proceso P es interrumpido con la ocurrencia del primer evento de Q , con lo cual, P nunca concluirá. Esto nos indica que una traza de ($P \nabla Q$) es justamente una traza de P hasta un punto arbitrario en donde la interrupción ocurre, seguida de cualquier traza de Q .

$$\alpha(P \nabla Q) = \alpha P \cup \alpha Q$$
$$\text{trazas}(P \nabla Q) = \{s \wedge t \mid s \in \text{trazas}(P) \wedge t \in \text{trazas}(Q)\}$$

Para evitar problemas, \surd no podrá estar en αP

2.5.1.1 Catástrofe

\surd (llámese, “catástrofe”) es un evento especial que denota un evento catastrófico o inesperado. El proceso ($P \nabla \surd Q$) se comporta como el proceso P hasta que el evento \surd ocurre, después de ese punto se comporta como Q .

Claramente el operador puede ser definido en términos de el operador de interrupción como

$$(P \nabla \surd Q) = P \nabla (\surd \rightarrow Q)$$

En los lenguajes de programación, \surd corresponde a la ocurrencia de una excepción, mientras que el proceso Q corresponde a la rutina que atiende esa excepción

2.5.1.2 Reinicio

\widehat{P} es un proceso que se comporta como P , pero cuando \surd ocurre, éste vuelve a comportarse como P desde el inicio. Nuevamente, este proceso puede ser definido en términos del operador de interrupción:

$$\widehat{P} = P \nabla \surd P$$
$$= P \nabla (\surd \rightarrow P)$$

2.5.1.3 Puntos de re arranque

Sea P un proceso que describe el comportamiento de un gran sistema de bases de datos. Si éste comienza a ejecutarse y después de tiempo, de manera súbita aparece una catástrofe, una de las peores respuestas es restablecer P en su estado inicial, perdiendo los datos que se han recabado. Es mucho mejor regresar a algún estado reciente del sistema el cual se conoce que es correcto. Tal estado se conoce como punto de re arranque, denotado por \odot . Cuando el evento catastrófico ocurre, el punto de re arranque más reciente es restablecido; o si no hay punto de re arranque, el estado inicial es restablecido. Dado que \odot y el evento catastrófico no deben estar en el alfabeto de P , definimos $Ch(P)$ como el proceso que se comporta como P , pero respondiendo de la manera adecuada a estos dos eventos.

2.6 Refinamientos CSP

La finalidad de una especificación abstracta es facilitarnos el entendimiento y establecer una forma no ambigua en la cual el sistema se ejecuta.

Altos niveles de abstracción facilitan el razonamiento y manipulación de las especificaciones. Tales especificaciones no contienen detalles de implantación, y cualquier implantación que satisfaga la especificación es aceptable. Por otro lado, la especificación y la implantación no pueden estar estrictamente separadas [Swartout82]. Cualquier especificación, no importa que tan formal o abstracta sea, es esencialmente la implantación de alguna especificación de mas alto nivel. La especificación más abstracta es el valor booleano falso, puesto que una premisa falsa admite cualquier comportamiento.

La finalidad del desarrollo de un sistema es implantar la especificación, en un lenguaje de programación, de manera que satisfaga todos los requerimientos funcionales (dados en la especificación) e idealmente los requerimientos no funcionales, tales como la eficiencia, facilidad de uso, etc.

La tarea de transformar una especificación de alto nivel a una implantación de bajo nivel, involucra reducir los niveles de abstracción, eliminar el no determinismo, elegir las estructuras de datos que sean más apropiadas para el ambiente de implantación, etc. Este proceso es llamado refinamiento.

El refinamiento no es un proceso directo. Excepto para problemas muy triviales, intentar llevar una especificación muy abstracta de manera directa a una implantación ejecutable es ingenuo, más nunca imposible.

Eliminar el no determinismo y reducir los niveles de abstracción es un proceso iterativo. En cada estado del proceso de refinamiento derivamos una especificación de nivel mas bajo que implanta la de nivel mas alto.

En CSP denotamos que un proceso $P1$ implanta al proceso P , o es un refinamiento del proceso P , de la siguiente manera:

$P \sqsubseteq P1$

Existen varios modelos en los cuales se puede probar si un proceso refina o implanta a otro. Los tres modelos más importantes son:

El modelo de trazas: en donde sólo se verifica que las trazas del proceso refinado se encuentren en las trazas del proceso a refinar.

El modelo de Fallas: en donde se verifica que las trazas del proceso refinado estén incluidas en el proceso a refinar y además se verifica que las fallas del proceso refinado sean un subconjunto del proceso a refinar.

El modelo Falla-Divergencia: en donde se revisan que las trazas, fallas y divergencias del proceso refinado estén incluidas en las del proceso a refinar.

Una explicación más amplia de estos modelos y sobre las propiedades que comprueba es dado en el siguiente capítulo.

Así pues, se dice que un proceso $P1$ refina a un proceso P , siempre que las siguientes 3 condiciones se cumplan:

- 1.- $\alpha P1 = \alpha P$
- 2.- $\mathbf{f}P1 \subseteq \mathbf{f}P$ (las fallas del proceso que refina son menos que las del proceso refinado)
- 3.- $\mathbf{d}P1 \subseteq \mathbf{d}P$ (las divergencias del proceso que refina son menos que las del proceso refinado)

Si las anteriores condiciones se cumplen, entonces cualquier comportamiento de $P1$ es también un comportamiento de P , pero en algunos casos $P1$ es mejor ---i.e. es menos probable de ir a un candado mortal, es menos no-determinista, mas concreto, etc.--- Por lo cual, $P1$ puede ser usado en lugar de P en cualquier ambiente en el cual P es aceptado.

2.7 Refinamiento a código ejecutable

En muchos casos, es deseable mostrar que un sistema de procesos es refinado por otros, esto con la finalidad de reemplazar unos por otros, y realizar así un especificación mas concreta. Sin embargo, también queremos ser capaces de derivar programas ejecutables de nuestra especificaciones abstractas.

Hemos visto que mediante un proceso iterativo, podemos eliminar el no determinismo y reducir los niveles de abstracción. En el desarrollo de un sistema real necesitamos que el producto final de este proceso sea código ejecutable que sea correcto con respecto a su especificación. El estado final de el proceso de refinamiento involucra la traducción de la

especificación CSP menos abstracta a código ejecutable. Esto es posible al transcribir la especificación CSP a código *ocamm* [Inmos84 e Inmos88].

El desarrollo paralelo de CSP y *ocamm* como medio para reducir la complejidad de la concurrencia ha producido enormes similitudes entre los dos lenguajes [Hoare91]; cada ley CSP tiene su equivalente en *ocamm* [Rascoe88]. Sin embargo, mientras la traducción a *ocamm* es simple, *ocamm* es un lenguaje muy especializado el cual se ejecuta sobre transputers. Por lo cual si queremos llevar nuestra implantación a un lenguaje de programación diferente, tendremos que refinar aún más nuestra especificación, hasta llevarla a un nivel tan concreto que podamos mapear los procesos CSP a código fuente. Para este último paso existe un camino formal para el lenguaje de programación ADA 9X [Hinchey95 a], pero para ello tendremos que utilizar una extensión a la teoría CSP, conocida como “Teoría receptiva de proceso” Sin embargo, para los demás lenguajes de programación no existe tal extensión de la teoría CSP, e intentar realizarla sale del ámbito del presente trabajo; por lo cual, el mapeo final lo haremos de manera semiformal a pseudocódigo.

FDR (FAILURES DIVERGENCE REFINEMENT, REFINAMIENTOS DEL MODELO FALLA DIVERGENCIA).

El objetivo fundamental de este capítulo es ilustrar las características y forma de uso de la herramienta automatizada FDR versión 2.0 (FDR2).

El sistema FDR2 (Failures Divergence Refinement) desarrollado por Formal Systems, es una herramienta que permite la prueba automática de refinamientos y la exploración interactiva de procesos. Este sistema está basado en la teoría matemática de Procesos Secuenciales Comunicantes (CSP), desarrollada en la Universidad de Oxford y posteriormente aplicada de manera satisfactoria en diferentes aplicaciones industriales. Sistemas con alrededor de 10^6 estados pueden ser analizados en unos cuantos minutos sobre una estación de trabajo.

En la sección 3.1 mencionamos las principales características de FDR2. En la sección 3.2 detallamos las relaciones de refinamiento que FDR2 puede probar, y para qué es usada cada relación de refinamiento. La sección 3.3 describe algunas ideas de cómo postular refinamientos. La sección 3.4 describe cómo está estructurado FDR2, su interfaz con el usuario y cómo utilizar la herramienta. Por último, la sección 3.5 describe una notación para transcribir en ASCII un subconjunto del lenguaje CSP, que forma el lenguaje aceptado por FDR2.

3.1 Características de FDR2

FDR2 compara sistemas usando los modelos de: trazas, fallas y fallas-divergencia de CSP. CSP es un lenguaje expresivo el cual puede representar muchas de las descripciones usuales de sistemas de estado finito, tal como: sistemas de transición, redes de Petri, muchas clases de autómatas y -en algunas aplicaciones- lenguajes tal como VHDL. CSP puede describir composiciones jerárquicas de elementos en los cuales la ejecución de sus acciones pueden ser sincronizadas, entrelazadas, renombradas u ocultadas. CSP proporciona una base matemática sólida para un número de técnicas de verificación. La teoría puede ser extendida a sistemas de estados infinitos y a análisis en tiempo real. CSP también proporciona la base para el lenguaje de programación *ocamm*. FDR2 permite que los sistemas sean descritos como archivos de texto de acuerdo al lenguaje de entrada.

La principal función de FDR2 es probar que relaciones de refinamientos entre descripciones de sistemas se cumplan. Esto permite comparar descripciones concretas de diseño con especificaciones abstractas para validar la corrección de los refinamientos. Además de permitir la comparación directa de diferentes implantaciones de los componentes de un sistema, FDR2 permite elegir niveles de comparación para validar tipos específicos de propiedades de manera independiente; de esta manera, se pueden probar propiedades de seguridad (en donde aseguramos que ciertos eventos no ocurren), propiedades de vivacidad (en donde aseguramos la ejecución de algunos eventos) y propiedades de estabilidad (el cual define la circunstancia, de cuando un estado estable es alcanzado). Características comunes de sistemas como el estar libre de bloqueos (*deadlock*'s), o de ciclos internos infinitos (*livelock*'s), por ejemplo, pueden ser probados de las especificaciones.

Si una comparación entre procesos, o entre un proceso y una especificación falla, FDR2 proporciona los medios para descomponer el comportamiento erróneo del proceso en sus subcomponentes. Un número de comandos interactivos permiten que esos subcomponentes sean descompuestos para permitir así, investigar los eventos ejecutados previos al error.

3.2 Refinamientos CSP

La noción de refinamiento es un concepto particularmente usado en diversas actividades de la ingeniería. Si podemos establecer la relación entre componentes de un sistema, de tal forma que capturen el hecho de que un componente satisface la mismas condiciones que el otro, entonces podemos reemplazar el peor componente por uno mejor sin degradar las propiedades del sistema. Obviamente la noción de refinamiento debe conservar las propiedades importantes del sistema: en la construcción de un puente puede ser aceptable reemplazar una viga de aluminio (más débil) por una de acero (más fuerte), pero si el peso es crítico, digamos en un avión, este no es un refinamiento válido.

En la descripción de sistemas de cómputo reactivos, CSP ha mostrado ser una herramienta adecuada. La relación de refinamiento puede ser definida de diferentes maneras, dependiendo

de las propiedades que se desean refinar. Existen principalmente tres relaciones de refinamiento, los cuales son:

Refinamiento de Trazas. Es la relación mas usada, está basada sobre la secuencia de eventos que un proceso puede ejecutar (las trazas de un proceso). Un proceso Q es un refinamiento de trazas de otro proceso, P , si todas las posibles secuencias de eventos de Q pueden ser también posibles para P . Esta relación es escrita $P \mathcal{J}_t Q$. Si consideramos a P como una especificación que determina posibles estados seguros de un sistema, entonces podemos pensar del refinamiento $P \mathcal{J}_t Q$, que Q es una implantación segura: ningún evento erróneo es permitido. El refinamiento de trazas sin embargo no nos dice nada acerca de lo que ocurre. El proceso $STOP$, el cual nunca ejecuta evento alguno, es un refinamiento de cualquier proceso, y satisface cualquier especificación segura.

Refinamiento de Fallas. Una distinción más fina entre procesos puede ser hecha al restringir los eventos que bloquean la implantación, también como aquellos que ejecuta. El refinamiento de Fallas, \mathcal{J}_f es definido como la prueba de que: las fallas de un proceso refinado están incluidas en las fallas de el proceso a refinar, esto es:

$$P \mathcal{J}_f Q \equiv Failures(Q) \subseteq Failures(P).$$

Refinamientos Falla-Divergencias. Para poder probar propiedades de vivacidad y seguridad, es necesario tener un modelo semántico más refinado que el modelo de fallas. El modelo falla-divergencia reúne esos requerimientos, y también captura la noción de falla total de un componente: la divergencia. Esta representa un conjunto de trazas después del cual el proceso puede comportarse caóticamente. Esto nos da dos características importantes: podemos analizar sistemas propensos a fallas, y asegurarnos de que éstas no ocurran en situaciones consideradas, o podemos usar la divergencia en especificaciones para describir situaciones “no importa” La relación \mathcal{J}_{fa} es formalmente definida como sigue:

$$P \mathcal{J}_{fa} Q \equiv Failures(Q) \subseteq Failures(P) \wedge \\ divergences(Q) \subseteq divergences(P)$$

Las divergencias de un proceso están incluidas en sus trazas: si s es un miembro de las $divergences(P)$ entonces (s, X) es una falla en $Failures(P)$ para cualquier rechazo en $refusal(X)$. Naturalmente, para procesos libres de divergencias, lo cual incluye una gran cantidad de sistemas prácticos, \mathcal{J}_{fa} es equivalente a \mathcal{J}_f .

Las tres formas de refinamiento son soportadas por FDR2 y normalmente se esperaría que sean usadas en el siguiente contexto:

- Las trazas de refinamiento son usadas para probar propiedades de seguridad.

- El refinamiento falla-divergencia es usado para probar propiedades de seguridad, vivacidad y combinación de ambas, y también para establecer relaciones de refinamiento e igualdad entre sistemas.
- El refinamiento de fallas es normalmente usado para probar refinamientos de procesos que se conoce de antemano que están libres de divergencias.

En lo sucesivo, por razones de simplicidad, utilizaremos el símbolo de refinamiento J en vez de J_d .

3.3 Usando Refinamientos

Un sistema formal que soporta refinamientos puede ser usado de distintas maneras:

- Podemos desarrollar sistemas mediante una serie de pasos de refinamiento. Iniciamos con la especificación de un proceso, y gradualmente refinamos ésta a una implantación. Puesto que los operadores de alto nivel de CSP, (éstos se explicaran mas adelante) conservan la noción de refinamiento, no es necesario aplicar reglas de refinamiento en los niveles más altos del proceso. Por ejemplo, si la composición paralela de P y Q refina una especificación S , esto es,

$$S \ JP \parallel Q,$$

entonces podemos desarrollar el sistema refinando P y Q de manera separada: si $P \ JP'$ y $Q \ JQ'$, entonces la composición de P' y Q' también refina a S .

$$S \ JP' \parallel Q'$$

sin tener que probar esta condición explícitamente.

- Existe la misma observación acerca del refinamiento de componentes, significa que siempre es posible reemplazar cualquier componente del sistema por uno que lo refina, manteniendo las propiedades ya probadas.
- Una implantación propuesta puede ser comparada con la especificación del proceso. Esa especificación debe ser compleja y debe capturar el comportamiento completo de la implantación, o ser simple y capturar una simple propiedad deseada, tal como que el sistema este libre de bloqueos.
- Si probamos que dos procesos son un refinamiento falla-divergencia el uno del otro en ambos sentidos, entonces ambos procesos son equivalentes y por lo tanto intercambiables.

3.4 Estructura e Interfaz

En esta sección describimos la estructura general de FDR2 así como la interfaz con el usuario. FDR2 está estructurado por medio de una serie de programas los cuales son descritos en la primer sección. La segunda sección describe la interfaz con el usuario de FDR2, aquí describimos la pantalla principal de FDR2 y el uso de cada una de las opciones del menú principal y de sus submenús.

3.4.1 Estructura

FDR2 está diseñado como una combinación de los siguientes programas:

- **fdr2** es un script del shell que invoca al sistema.
- **fdr2tix** es la interfaz principal con el usuario y el probador de modelos.
- **state2** es el compilador de CSP

En uso normal, estos programas se invocan automáticamente cuando son requeridos. Por lo cual, el usuario solamente necesita ejecutar **fdr2**.

3.4.2 Interfaz

3.4.2.1 Pantalla principal de FDR2

La interfaz con el usuario es gráfica y está basada en el lenguaje TCL y en las bibliotecas TK de John Ousterhout's [FORMAL95]. Este diseño es familiar para los usuarios de Motif, SAA o aplicaciones del estilo de Microsoft Windows. Cuando FDR2 inicia, despliega una ventana como se muestra en la figura 1. La cual contiene cinco componentes ordenados verticalmente, los cuales se explican a continuación.

La Barra del Menú. En la parte superior de la ventana, se encuentra la barra de menú y contiene los encabezados que describen los grupos de comandos. Para desplegar el menú relacionado a un encabezado en particular, es necesario presionar el botón izquierdo del ratón sobre éste. Alternativamente se puede presionar la tecla ALT y el carácter subrayado del encabezado del submenú elegido.

La barra de Herramientas. Bajo la barra del menú se encuentra la barra de herramientas. Esta consta de cuatro iconos, cada uno representando los comandos más usados, como los comandos para probar que un proceso: está libre de Bloqueos

(deadlock's), ciclos infinitos internos (livelock's), y si es determinista. El último icono sirve para abortar la ejecución de un comando.

Lista de Afirmaciones. Bajo la barra de herramientas se encuentra la lista de afirmaciones, la cual contiene una lista de afirmaciones hechas acerca de los refinamientos de los procesos, tal como: si un proceso es refinado por otro, si un proceso esta libre de *deadlock's* o *livelock's*, etc. Para cada declaración, FDR2 mostrará si ésta es verdadera, falsa, o bien, si aún no se ha probado. Cuando un archivo es cargado, la lista de afirmaciones incluidas al final de él, son adicionadas a la lista de afirmaciones, para que el usuario las pueda seleccionar y probar después.

Las afirmaciones desplegadas en esta lista pueden ser probadas, y si es falsa la afirmación seleccionada, entonces el depurador de procesos de FDR2 puede ser invocado, y éste mostrará contraejemplos (Ver. sección 3.4.2.7).

Lista de Procesos. Bajo la lista de afirmaciones, FDR2 despliega una lista con todos los procesos definidos en el archivo cargado. Los procesos de esta lista pueden ser seleccionados: uno como la especificación y otro como la implantación para la prueba de un refinamiento, o bien uno sólo para probar sus propiedades intrínsecas.

Selector de Refinamiento. La parte más baja de la ventana principal es usada para construir afirmaciones de refinamientos. Dos selectores de procesos definen el proceso que servirá como especificación y el que servirá como implantación en la prueba. Un tercer selector definirá que tipo de refinamiento será probado. Una vez seleccionado lo anterior, la prueba puede ser adicionada a la lista de afirmaciones o probada inmediatamente.

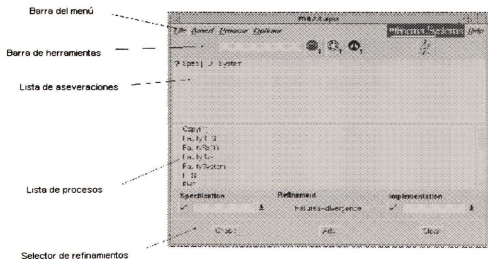


Figura 1. Ventana principal después de haber cargado el archivo mbuff.fdr2

3.4.2.2 Opciones de la barra del menú principal

El menú principal contiene 5 encabezados, en donde cada uno de ellos agrupa un conjunto de comandos. Los encabezados son: **File**, **Assert**, **Process**, **Options** y **Help**. Los comandos contenidos en cada submenú son descritos en las siguientes subsecciones.

3.4.2.2.1 Comandos del submenú FILE

Los comandos básicos para cargar y analizar sistemas de FDR2 son agrupados bajo el submenú **FILE**. Este contiene los comandos para cargar un archivo, volver a cargar un archivo, y salirse de FDR2.

El comando “Load” (Cargar)

Al seleccionar esta opción del submenú **File**, provoca que una caja de dialogo aparezca solicitando el nombre y ruta del archivo a cargar. Una vez que se ha proporcionado esta información, (de manera adecuada), el archivo es cargado, (si es que no contiene errores). Cuando el archivo se carga, las afirmaciones y los procesos que existían en el ambiente son eliminados, y las afirmaciones y los procesos del nuevo archivo son inicializados. Si un error de sintaxis o de algún otro tipo ocurre al cargar el archivo, el mensaje de error es almacenado en la bitácora interna de FDR2. Los mensajes son desplegados al seleccionar la opción **Show Status** del submenú **Options**.

El comando “Reload” (cargar nuevamente)

Al seleccionar esta opción, se carga nuevamente el archivo de uso actual incorporando posibles nuevos cambios realizados a éste. Si no hay archivo cargado y se selecciona esta opción, el sistema no ejecuta nada.

El comando “Exit” (salir)

Cuando este comando es seleccionado, **FDR2** despliega una caja de diálogo en donde se le pide al usuario que confirme si realmente desea terminar la sesión con **FDR2**. Si la respuesta es **Quit**, **FDR2** termina, en otro caso el comando **Exit** es ignorado.

3.4.2.2.2 Comandos del submenú Assert

El comando para verificar si un proceso es un refinamiento de otro (**Run**), y el comando que nos ayuda a depurar los errores del proceso refinado (**Debug**), se encuentran agrupados en el submenú **Assert**.

Run Inicia la revisión de la afirmación seleccionada, (Ver sección 3.4.2.4). Si la afirmación es correcta, es marcada en la lista de afirmaciones con un ✓, si es falsa, es marcada con un ✘, si no se pudo efectuar la prueba debido a errores en la compilación de la afirmación, es marcada con !. Los errores encontrados en la compilación de las afirmaciones son registrados en la ventana de estatus (ver comando **Show Status** de la siguiente sección).

Debug Si éste comando es seleccionado, aparece nueva ventana que nos permite investigar de manera interactiva los contraejemplos que hacen que el refinamiento falle. La pasos para usar esta ventana son dados en la sección 3.4.2.7

3.4.2.2.3 Comandos del submenú Process

Aún no están disponibles en las versiones actuales.

3.4.2.2.4 Comandos del submenú Options

El submenú **Options** permite el acceso a aspectos internos de las operaciones de FDR2. Dependiendo de la versión de FDR, puede tener o no disponibles las siguientes opciones:

Supercompilation Esta opción conmuta a FDR2 a usar una representación interna basada en máscaras para la composición de máquinas de estado finito. Este debe ser deshabilitado para todas las operaciones estándar. (No habilitado).

Messages Este submenú permite controlar el monto de retroalimentación que es adicionado a la ventana de status, cuando las pruebas sobre los procesos son efectuadas. El default, **Auto**, no reporta operaciones que tengan menos de 200 estados, e indica el progreso cada 100 estados después del estado 200 y hasta el 2000, y después los reportes se harán cada 1000 estados. **Full**, muestra los detalles de todas las operaciones; **None**, inhibe toda la información. Para ver esta información utilice el siguiente comando.

Show status Este comando provoca que FDR2 abra una ventana de texto la cual es actualizada en cuanto FDR2 genere información sobre la compilación o prueba de procesos. La ventana de status también recibe mensajes detallando errores sintácticos o semánticos detectados por el compilador de CSP.

Compilation Feedback Cuando esta opción es habilitada, la compilación de definiciones CSP de cualquier archivo cargado por FDR2 generará más información detallada, incluyendo los nombres y argumentos de los procesos que son explorados. Esa información es adicionada a la ventana de status descrita arriba.

3.4.2.2.5 Ayuda en línea


Para obtener información sobre el funcionamiento de FDR2, se puede seleccionar el menú de ayuda (**Help**) del lado derecho de la barra de menús (presionando el botón izquierdo del ratón). Esta opción nos ayuda a consultar información acerca de un rango de tópicos, como:

- Una introducción a FDR2 y CSP.

- Una guía para la interfaz del usuario.
- Un resumen del lenguaje de entrada CSP.
- Información acerca de la versión y configuración instaladas.

La información es desplegada en una ventana separada. Para moverse alrededor del texto se puede usar las barras laterales de la ventana. La información mostrada en hipertexto contiene ligas a ayuda adicional. Para seguir esas ligas basta colocar el ratón sobre el texto en realce y presionar el botón izquierdo. Para cerrar la ventana de ayuda presione el botón izquierdo del ratón sobre el mensaje “Close ... “ desplegado en la parte inferior de cada ventana.

3.4.2.3 Comandos de la barra de herramientas

Al presionar el icono  con el botón izquierdo del ratón, la prueba o compilación actual es abortada.

Los siguientes botones sobre la barra de herramientas invocan comandos que operan sobre el proceso seleccionado (para seleccionar un proceso de la lista de procesos, coloque el cursor del ratón sobre él y presione el botón izquierdo)



Prueba si el proceso seleccionado esta libre de bloqueos. Esto es, se prueba si el proceso puede alcanzar un estado en el cual no haya acción posible. Si un bloqueo ocurre, bastará una traza mostrada por el depurador para ilustrar ese hecho.



Prueba si el proceso seleccionado esta libre de *livelock's*. Esto es, prueba si un proceso puede alcanzar una serie infinita de estados en donde sólo acciones internas sean posibles, sin que ningún evento externo tenga lugar (divergencia CSP). Si tal secuencia es encontrada, esta será accesible desde el depurador (el cual permitirá el examen de los detalles interno involucrados)



Este comando determina si un proceso es determinista, i.e. si el conjunto de acciones posibles en cualquier estado, es siempre determinado de manera única por la historia previa de acciones visibles. Un proceso es no determinista si éste puede divergir (*livelock*), o si después de haberse comprometido con una traza, éste puede aceptar un evento y al mismo tiempo lo puede rechazar. En este último caso, el depurador presentará los dos comportamientos del

mismo proceso como un contraejemplo; uno en el cual el proceso se compromete en el evento, y otro en el cual el proceso rechaza el evento.

3.4.2.4 La lista de afirmaciones

Como lo mencionamos anteriormente, un archivo para FDR2 puede contener declaraciones sobre refinamientos. Esas declaraciones tienen la forma:

```
assert abstracta [X= concreta
```

donde *abstracta* y *concreta* son procesos, y X indica el tipo de refinamiento a probar entre éstos: T para trazas, F para fallas y FD para falla-divergencias. Cuando un archivo es cargado, cualquier afirmación de esta forma es adicionada a la lista de afirmaciones de FDR2. Inicialmente cada afirmación es marcada como no explorada, usando el símbolo “?”, (ver figura 1).

Una afirmación puede ser seleccionada presionando el botón izquierdo del ratón sobre ella, y ésta será realizada con otro color. La afirmación seleccionada puede ser sometida a prueba al seleccionar la opción **Run** del submenú **Assert**. FDR2 intentará probar la conjetura al compilar, normalizar, y probar el refinamiento. Mientras la prueba se realiza, la afirmación es marcada con el símbolo de “espera” (un pequeño reloj).

Cuando la prueba termina, o es detenida por el botón de interrupción, el símbolo asociado a la afirmación es actualizado para reflejar el posible resultado:

- ✓ Indica que la prueba ha terminado de manera satisfactoria; el refinamiento es correcto.
- ✘ Indica que la prueba ha terminado, pero se encontró uno o más contraejemplos a la propiedad cuestionada: el refinamiento no es correcto, y el depurador puede ser explorado para conocer las razones del por qué.
- ! Es usado para marcar cuando una prueba ha terminado incorrectamente por alguna razón: ya sea por un error de sintaxis, debido a que los recursos del sistema han sido agotados, o debido a que la prueba fue interrumpida. Si el proceso no pudo ser compilado, FDR2 lo indicará mediante una caja de diálogo.

Cuando una prueba ha terminado, el depurador de FDR2 puede ser invocado para investigar las causas del resultado de la prueba de la afirmación. Para invocar el depurador, es necesario seleccionar la opción **Debug** del submenú **Assert**. Esto abrirá una nueva ventana la cual permitirá que el proceso invocado sea examinado (Ver sección 3.4.2.7).

3.4.2.5 La lista de procesos.

La lista de procesos desplegada por FDR2, tiene dos propósitos fundamentales: permitir la elección de ellos por el selector de refinamientos, o bien, seleccionar un proceso para explorar sus propiedades intrínsecas.

Cada entrada en la lista consiste del nombre del proceso o función seguido del número de argumentos que requiere, ver figura 1.

La elección de algún proceso es similar a la elección de alguna afirmación (se coloca el cursor del ratón sobre el proceso y se presiona el botón izquierdo). El proceso seleccionado puede ser transferido al selector de refinamientos al presionar el botón izquierdo del ratón sobre la elección ✓, o ser usado con algún comando de la barra de herramientas.

3.4.2.6 El selector de refinamientos.

Cuando un archivo es cargado por FDR2, las afirmaciones contenidas en él son inicializadas en la lista de afirmaciones (ver fig. 2). Si el archivo no contiene afirmaciones o se requieren más de ellas, en la parte más baja de la ventana principal de FDR2 (en el selector de refinamientos), se le permite al usuario componer afirmaciones sobre refinamiento de propiedades entre procesos. El selector de refinamientos consiste de cuatro componentes: dos selectores de procesos que permiten elegir un proceso cada uno, un selector para elegir la relación de refinamiento CSP a probar, y tres botones más para el manejo de esas afirmaciones.

Cada selector de procesos opera de manera idéntica: cada uno consiste de tres elementos: un botón de selección ✓, un campo de texto y un botón que despliega la lista de procesos disponibles de manera descendente ↓. Cada uno de estos elementos puede modificar la definición del proceso desplegado en el campo de texto:

- Si presionamos el botón izquierdo del ratón cuando el cursor se encuentra sobre la opción ✓, entonces el proceso seleccionado de la lista de procesos se trasladará al campo de texto.
- Si presionamos el botón izquierdo del ratón cuando el cursor se encuentra sobre el campo de texto, entonces podremos introducir texto desde el teclado, modificando el ya existente.
- Si presionamos el botón izquierdo del ratón cuando el cursor se encuentra sobre el botón de despliegue descendente, ↓, entonces el sistema mostrará la lista de procesos disponibles a ser probados, al seleccionar uno, éste se trasladará al campo de texto.

Tres botones de comando completan el selector de refinamientos. Estos nos permiten que las pruebas sean registradas, probadas o descartadas. Los botones son:

Check Provoca que los valores actuales de los selectores de procesos sean tomados como las partes abstracta y concreta de la declaración de refinamiento CSP de la forma:

Especificación [= Implantación

El tipo de refinamiento es tomado del selector central, y el resultado de la afirmación es adicionado inmediatamente a la lista de afirmaciones y ejecutado.

Add Provoca que la afirmación formulada sea adicionada a la lista de afirmaciones, sin que la prueba sea inmediatamente iniciada.

Clear Al presionar este botón, ambos selectores de procesos se vaciarán, y estarán listos para que se introduzcan nuevos procesos.

3.4.2.7 El depurador de procesos

Cuando una prueba sobre un refinamiento ha terminado y se selecciona ésta para depurarla, FDR2 crea una nueva ventana, la cual contiene contraejemplos encontrados a lo largo de la prueba. La ventana del depurador se muestra en la figura 2 y consiste de tres partes:

La barra de Menú Al igual que la ventana principal, la barra de menú contiene encabezados describiendo los grupos de comandos o consultas del depurador. En la versión 2.0 solamente existen los encabezados para el grupo de comandos **File** y para el de **Help**.

Selector de Contraejemplos FDR2 incluye la capacidad de generar contraejemplos de una prueba de refinamiento dada, con el fin de que el usuario los analice. Debemos recordar, que la ocurrencia de un simple contraejemplo basta y sobra para que el comportamiento de la especificación concreta sea errónea.

Ventana del comportamiento del proceso. La porción más grande de la ventana del depurador está reservada para desplegar el comportamiento del proceso. Esta ventana muestra la estructura particular del proceso junto con la contribución de cada uno de sus componentes para un contraejemplo en particular. Cuando más de un proceso está involucrado en la prueba, se usarán los botones marcados con **0** y **1** para ver la especificación y la implantación respectivamente.

3.4.2.7.1 Comandos del menú del depurador

La versión actual de FDR2 sólo tiene dos grupos de comandos agrupados en **El menú File** y **El menú Help**, los cuales están en construcción.

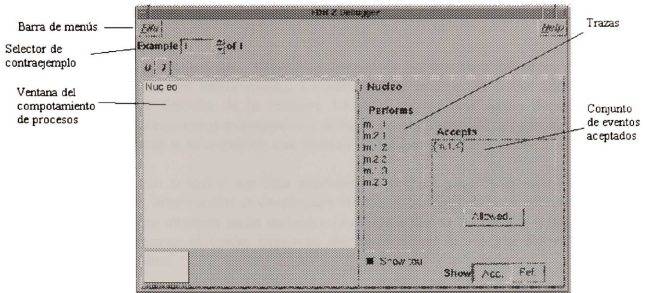


Figura 2. Depurador de FDR2

3.4.2.7.2 La ventana del comportamiento de procesos

La ventana del comportamiento de procesos está organizada en una serie de páginas indexadas por números, uno por cada proceso relevante a el contraejemplo seleccionado. Para cualquier contraejemplo particular, el sistema mantiene un registro de los procesos involucrados. Si la propiedad probada es intrínseca a un proceso como: estar libre de bloqueos o ciclos internos infinitos (*livelock's*) solamente habrá un proceso involucrado. En el caso de la prueba de un refinamiento, habrá dos procesos: una especificación y una implantación. En este caso, FDR2 desplegará por default el comportamiento de la implantación, pero si deseamos ver la especificación entonces debemos presionar el ratón sobre la etiqueta 0, en la parte de arriba de la ventana del comportamiento de procesos, (ver figura 2).

Cada página representa de esta manera un sólo proceso y está involucrado en un comportamiento en particular. Así la información es presentada en dos partes: una vista jerárquica de la estructura del proceso, y una serie de ventanas mostrando las contribuciones de una parte seleccionada del proceso o el comportamiento total.

La estructura del Proceso. La estructura del proceso es representada como un árbol, similar a los diagramas de la estructura de programas. El nodo raíz (en la parte alta del árbol) representa el proceso como un todo, y es inicialmente mostrado solo, sin detalles adicionales. Cualquier nodo que pueda arrojar más información puede ser expandido al posicionarse sobre él y presionando dos veces consecutivas el botón izquierdo de ratón.

Cuando un nodo es expandido, se adicionan al árbol tantas ramas como subcomponentes tenga el nodo en cuestión. Así un nodo etiquetado con el símbolo de composición paralela [|. .] se expandirá en dos componentes, representando los subprocesos que son combinados en paralelo. Cada componente puede tener su propia contribución al comportamiento erróneo que es examinado.

Información del comportamiento. Cuando examinamos la estructura del proceso, cualquier nodo en el árbol puede ser seleccionado. La información acerca del nodo seleccionado es desplegada en el área derecha de la ventana. La información desplegada dependerá de la naturaleza de los contraejemplos examinados y de las contribuciones hechas por el componente seleccionado. En general la información que pueden ser desplegada es:

- La traza del proceso actual y una lista conteniendo eventos prohibidos que hacen que el proceso falle. Esta información es desplegada sólo si la bandera **Allows** está activada. Por omisión, los eventos internos serán incluidos (eventos etiquetados por τ), pero pueden ser eliminados al presionar el botón izquierdo del ratón sobre la opción **Show Tau** de la ventana del depurador.
- Una traza sin errores, que tal vez nos lleve después a un error es desplegada cuando elegimos la opción **Performs** de la ventana del depurador.
- El conjunto de eventos no aceptados o rechazados por el proceso. Esta información puede ser expresada como un conjunto de eventos aceptados por el proceso (el cual será el más pequeño que la especificación permite) o como un rechazo máximo. Para conmutar entre esas opciones, se presiona el botón izquierdo del ratón sobre **Acc** (aceptaciones) o **Ref** (rechazos).
- Divergencia. Si el proceso diverge ilegalmente, será exhibido.
- Repetición de eventos visibles. Al descomponer el comportamiento de un proceso divergente, podemos descubrir una serie de eventos repetidos de manera infinita, y los cuales están ocultos de su último ambiente. Esa secuencia será etiquetada con la palabra **Repeat** y será desplegada de la misma manera que las demás trazas discutidas anteriormente.

Típicamente la siguiente información es desplegada para cada tipo de contraejemplo:

Successful Refinement: no hay información relevante sobre el comportamiento. Ninguna información es desplegada.

No direct contributions: existe una traza sin errores.

Refusal/acceptance failure: existe una traza sin errores, mas un rechazo/divergencia ilegal.

Divergence: la traza nos lleva a una divergencia.

Divergence (internally): tenemos una traza que nos lleva a una divergencia.

3.5 Formato de entrada de especificaciones CSP a FDR2.

El lenguaje para describir procesos interpretados por FDR2 consiste de tres elementos:

- El lenguaje de bajo nivel para la descripción de procesos.
- El lenguaje de alto nivel.
- El soporte del lenguaje matemático.

Esos niveles de descripción son escritos en un formato único, el cual conforma el estándar para representar especificaciones CSP en una forma entendible por una máquina.

Las siguientes secciones describen las características del lenguaje, además de proporcionar algunos ejemplos.

3.5.1 Lenguaje de bajo nivel

El principal propósito de los operadores de bajo nivel de FDR2, es permitir la definición de componentes de procesos relativamente pequeños. El comportamiento de esos procesos puede ser parametrizado. Cada operador de bajo nivel es descrito de la siguiente forma: colocamos al inicio en el lado izquierdo el nombre del operador en el formato FDR2, en la misma línea pero del lado derecho, escribimos el mismo operador en el formato de CSP, y posteriormente damos una breve explicación del operador.

Los procesos más simples, son los procesos atómicos estándar de CSP: *STOP* y *SKIP*

STOP (escrito como *STOP*)
Representa un proceso bloqueado, el cual no ejecuta ninguna acción (de esta manera puede rechazar cualquier acción)

SKIP (escrito como *SKIP*)
Representa al proceso que sólo se compromete con la terminación satisfactoria. La terminación satisfactoria se representa de la manera usual, como el evento \surd . (escrito como *tick*)

a->P (escrito como $a \rightarrow P$)
Este proceso se compromete con *a* y después se comporta como el proceso *P*. El evento *a* no puede ser rechazado inicialmente. El evento puede ser un simple nombre atómico (tal como *ack* o *a*), o un evento indexado, usado con un canal, tal como *chan.value*, por ejemplo.

chan ! exp -> P (escrito como $chan! exp \rightarrow P$)

Esta forma de especificar eventos corresponde a una emisión a través de un canal. La expresión exp es evaluada en el contexto actual, y el proceso ofrece ejecutar el evento $chan.v$, en donde v es el valor resultante. En efecto, la diferencia entre la expresión $c.exp \rightarrow P$ y $c!exp \rightarrow P$ es completamente sintáctica. Reflejando la definición usual CSP de emisión tenemos que $c!v \rightarrow P \equiv c.v \rightarrow P$.

$chan \ ? \ atom \ \rightarrow \ P$ (escrito como $c \ ? \ x \ \rightarrow \ P$)

El proceso $c \ ? \ x \ \rightarrow \ P$ inicialmente ofrece una elección no determinista de todos los eventos de la forma $c.v$, en donde v es el valor a emitir por el canal c , la elección no determinista es resuelta por el medio ambiente. Cuando la recepción ocurre, la variable x , la cual debe ser un simple identificador atómico, es asociada al valor elegido, y el proceso subsecuentemente se comporta como P dado ese valor para x .

Las nociones generales de recepción y emisión permiten estructurar eventos de manera que permiten modelar arreglos de canales y estructuras similares. Si i es un elemento de algún conjunto de índices, entonces:

$a \ i \ ! \ x \ \rightarrow \ P$

puede ser considerado como la emisión del valor x sobre el canal i del arreglo a . Esta estructura puede permitir comunicaciones mucho mas complejas.

$P \ [] \ Q$ (escrito como $P \ \square \ Q$)

El operador de elección determinista, ofrece al ambiente la elección de todos los eventos inicialmente ofrecidos por P o Q . Si el primer evento ejecutado fue visible sólo para P , entonces el proceso se comportará subsecuentemente como P , y similarmente para Q . Si el primer evento fue posible para ambos procesos el comportamiento subsecuente será no determinista.

$P \ | \sim \ Q$ (escrito como $P \ \Pi \ Q$)

El operador $| \sim$ representa una elección interna entre P y Q . El comportamiento observable puede ser de cualquier proceso.

Ambas formas de elección tienen extensiones que contemplan el uso de índices:

$[] \ x : \ X \ S \ @ \ P$ y $| \sim \ x : \ X \ S \ @ \ P$ (escrito como $\square_{x \in X \ S} P(x)$ y $\Pi_{x \in X \ S} P(x)$ respectivamente)

El operador de elección determinista y no determinista indexados respectivamente. La elección en el primer caso es determinista entre los eventos de x en $X \ S$ y en el segundo caso es no determinista entre P evaluado en un ambiente, con la variable x limitada a los valores del conjunto $X \ S$.

$P ; Q$ (escrito como $P ; Q$)

Este proceso se comporta como P , hasta que P termina (al comprometerse con $tick$, la terminación satisfactoria), y entonces se comporta tal como Q . Como mencionamos anteriormente, el proceso $SKIP$ señala la terminación satisfactoria con un evento $tick$. El

proceso *SKIP* es empleado por el operador de composición secuencial, “;”, para transferir el control a procesos subsecuentes.

El último constructor que describimos en esta sección es el operador de recursión. Mas que emplear la notación usual de CSP, FDR2 emplea el estilo ecuacional:

$X = P$ (escrito como $X = P$)

Un archivo de entrada para FDR2 consiste de una serie de definiciones que refieren a procesos. Todas las definiciones cargadas a una sesión FDR2 son tomadas en cuenta cuando los procesos son analizados, debido a que las definiciones no necesariamente están en orden; ellas pueden ser tomadas de diferentes fuentes de entrada. Las definiciones pueden tomar parámetros: una lista de identificadores separados por comas, los cuales son ligados a valores cuando las definiciones son expandidas.

Por ejemplo, un buffer de una localidad puede ser definido como

$\text{Buff1} = \text{in } ? x \rightarrow \text{Almacén}(x)$

$\text{Almacén}(x) = \text{out } ! x \rightarrow \text{Buff1}$

en donde Almacén es un proceso parametrizado el cual toma un solo valor ---el contenido actual del buffer---

Expresiones condicionales pueden ser usadas para escribir términos de procesos dependiendo de tales parámetros: la expresión

$\text{if } b \text{ then } P \text{ else } Q$

es equivalente a P si la condición b es evaluada a verdadero, y a Q si b es falsa.

3.5.2 Lenguaje alto nivel

Los operadores de alto nivel soportados por FDR2 son:

$P [X \parallel Y] Q$ (escrito como $P_{X|Y} Q$)

Esta combinación paralela “alfabetizada” modela la operación concurrente sincrónica. P puede ejecutar sólo los eventos dentro del conjunto X , y Q ejecuta sólo los eventos de Y . Los eventos en la intersección de X y Y se ejecutan de manera sincrónica.

$P [[X]] Q$ (escrito como $P_{|X} Q$)

La composición sincrónica de dos procesos puede también ser escrita usando el operador $[[X]]$, donde la sincronización es forzada sobre los eventos dentro del conjunto X . Los eventos fuera de ese conjunto pueden ocurrir de manera independiente.

$P ||| Q$

(escrito como $P ||| Q$)

El operador de entrelazamiento modela la operación concurrente y asíncrona. Cada proceso ejecuta sus eventos de manera independiente

$P \setminus A$

(escrito como $P \setminus A$)

Este operador oculta los eventos del conjunto A , al ambiente de P y permite que las transiciones internas de P , ocurran sin sincronización con su medio ambiente. El uso de este operador puede introducir no determinismo o divergencia.

$P [[a \leftarrow b, b \leftarrow a]]$

(escrito como $f(P)$ o $f^{-1}(P)$)

La operación de renombramiento implanta las operaciones de funciones y funciones inversas de CSP. Las expresiones encerradas en paréntesis cuadrados deben ir separadas por comas, y deben tener la forma $a \leftarrow b$, en donde a y b son eventos, canales o expresiones que evalúan canales o eventos. El efecto de la operación es una substitución simultánea de b por a en todas las parejas listadas en el renombramiento.

$P \wedge Q$

(escrito como $P \nabla Q$)

El operador de interrupción permite que los eventos del proceso Q se ejecuten, hasta que el primer evento de Q es disponible, entonces, los eventos de Q son ejecutados.

3.5.3. Declaración de variables y canales

Identificadores, tales como `Hello` y `xx112` son tratados como variables almacenando datos. Tales variables son ligadas al contexto en donde fueron definidas. Las formas de definir una variable son:

- Mediante declaraciones explícitas al principio del archivo. Después de una definición como: $Z = \{1,2,3\}$, el identificador Z es tratado como una variable almacenando un valor dentro del conjunto $\{1,2,3\}$.
- Mediante parámetros a los procesos. En el lado derecho de la definición $P(x, y, z) = \dots$, los identificadores x , y y z son variables ligadas a los parámetros del proceso.
- Mediante eventos de entrada. En el proceso $c ? x \rightarrow P$, x está ligada al proceso P . El valor asociado a x , es el valor de recibido por el canal c .

Variables con el mismo nombre pueden ser declaradas en distintos niveles. El alcance de una variable, en caso de múltiples declaraciones, es el nivel en el cual fue declarada. Si una

variable x es declarada más de una vez, y ésta es referenciada, entonces se toma la variable x de ese nivel, si es que fue definida en ese nivel, en caso contrario se toma la variable x global.

Un canal es declarado anteponiendo la palabra reservada `channel` seguido del nombre del canal, dos puntos y el tipo de datos que pueden ser emitidos por éste, tal como se muestra a continuación.

```
channel Canallzquierdo : Int
```

Los datos a emitir por un canal pueden ser enteros (`Int`), o ser datos definidos por el usuario. Un ejemplo de datos definidos por el usuario es:

```
datatype FRUTA = manzana | naranja | pera
```

FDR2 maneja una representación interna para estos datos definidos por el usuario. También se pueden definir datos que tomen valores de subconjuntos de enteros como:

```
datatype Subconjunto = {1,2,3,4,5}
```

de esta forma podemos definir canales que comuniquen cualquier dato.

3.5.4 Operadores disponibles

Operadores de lógicos

Los operadores de comparación de FDR2 son:

Expresión	Tipos de argumentos	Operador representado
<code>a == b</code>	(expresión, expresión)	<code>=</code>
<code>a != b</code>	(expresión, expresión)	<code>≠</code>
<code>a < b</code>	(expresión, expresión)	<code><</code>
<code>a > b</code>	(expresión, expresión)	<code>></code>
<code>not (b)</code>	valor de verdad	<code>¬</code>
<code>c and d</code>	(valor de verdad, valor de verdad)	<code>∧</code>
<code>c or d</code>	(valor de verdad, valor de verdad)	<code>∨</code>

Operadores aritméticos

Las operaciones aritméticas de FDR2 son: multiplicación (`*`), división (`/`), módulo (`%`), adición (`+`) y sustracción (`-`).

Operaciones sobre conjuntos

Las principales operaciones sobre conjuntos son definidas de la siguiente manera:

Operación	Tipo de Argumentos	Tipo del Resultado	Función
card(XS)	conjunto	entero	cardinalidad
empty(XS)	conjunto	valor de verdad	prueba de vacuidad
member(x,XS)	(elemento, conjunto)	valor de verdad	prueba de membresía
inter(XS,YS)	(conjunto, conjunto)	conjunto	intersección
union(XS,YS)	(conjunto, conjunto)	conjunto	unión
diff(XS,YS)	(conjunto, conjunto)	conjunto	diferencia

Operaciones sobre trazas

Las operaciones sobre trazas reconocidas por FDR2 de CSP, son:

Operación	Tipo de argumentos	Tipo del resultado	Función	Símbolo CSP
# (xs)	traza	entero	longitud de secuencia	#xs
nul(xs)	traza	valor de verdad	prueba de vacuidad	xs = <>
xs ^ ys	(traza, traza)	traza	concatenación	xs ^ ys
head(xs)	traza	elemento	primer elemento	xs ₀
tail(s)	traza	traza	traza sin el primer elemento	xs'

Hasta este punto hemos visto como esta estructurado el probador de refinamientos FDR2, hemos descrito la interfaz que ofrece al usuario y el lenguaje que reconoce. Antes de pasar al desarrollo del núcleo del sistema operativo, creemos que es necesario realizar la siguiente observación.

Cualquier técnica de verificación trabaja sólo sobre un número finito de estados alcanzables. Esta restricción obviamente también se aplica a sistemas analizados por FDR2. Sin embargo, generalmente no es posible determinar mecánicamente si la expansión específica de un proceso terminará en un número finito de estados. No hay forma así de que FDR2 prevenga al usuario de definir un proceso que genere un número infinito de estados. Si los siguientes puntos se tienen en mente al definir sistemas, tales situaciones deben ser evitadas fácilmente

- Los parámetros a procesos deben variar sobre un conjunto finito. El proceso contador

```
Cont = Cnt(0)
```

```
Cnt(0) = if n==0 then up -> Cnt(1)
```

```
        else up -> Cnt(n+1) [] down -> Cnt(n-1)
```

puede obviamente alcanzar un número ilimitado de estados.

- No deben aparecer llamadas recursivas sobre el lado izquierdo de una composición secuencial. El proceso palindromo²

$$\text{Pal} = (\text{left } ?x \rightarrow (\text{Pal} ; \text{right } !x \rightarrow \text{SKIP}))$$
$$\quad | \sim |$$
$$\quad \text{SKIP}$$

puede aceptar un número infinito de diferentes secuencias, y alcanzar un estado distinto después de cada una de ellas.

Otra fuente común de comportamientos infinitos en CSP es el uso de la composición paralela y la recursión para generar un número potencialmente ilimitado de procesos separados. La división del lenguaje de entrada a FDR2 en partes atómicas y compuestas hacen que tales procesos sean ilegales.

Teniendo en mente esta observación y después de haber revisado el formato del lenguaje CSP que acepta FDR2, procedamos a la implantación del núcleo de nuestro sistema operativo.

² Palindromo: palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda.

Especificación inicial y Pasos de refinamiento del núcleo básico de un sistema operativo

4.1 Introducción.

Una vez presentado un panorama muy general de los métodos formales, y una revisión de CSP, así como su herramienta de soporte, FDR, pasemos al cuerpo de nuestra investigación.

El presente capítulo contiene la especificación inicial del núcleo básico de un sistema operativo multitareas. Esta especificación inicial sólo contiene la descripción del entrelazamiento de acciones de programas de usuario. Después de explicar la especificación inicial, pasaremos a mostrar cómo podemos implantarla; esto lo haremos de manera gradual en las siguientes secciones, mediante pasos de refinamiento. En cada refinamiento indicaremos mayores detalles de implantación.

Comenzamos con un primer refinamiento, en donde modularizamos la especificación inicial y aparecen los programas de usuario. Probamos de manera manual este primer refinamiento y después verificamos nuestro resultado con el obtenido con FDR2. Terminamos esta sección dando una motivación al segundo refinamiento.

En el segundo refinamiento, aparece el primer programa de sistema, el cual es la especificación de un mecanismo de transferencia del control del procesador entre los programas de usuario. Una vez explicado el segundo refinamiento, pasamos a su prueba manual y después verificamos nuestro resultado con lo obtenido por FDR2, terminamos esta sección con una motivación al tercer refinamiento.

En el tercer refinamiento se refina el mecanismo de transferencia del control del procesador. Se prueba el refinamiento de manera manual y con FDR2 para concluir la sección con una motivación al cuarto refinamiento.

En el cuarto refinamiento generalizamos el sistema para que n programas de usuario puedan ser ejecutados dentro del núcleo del sistema operativo. Probamos la corrección del refinamiento con FDR2, para terminar la sección con una motivación al quinto refinamiento.

En el quinto refinamiento modelamos un mecanismo que le permita al núcleo del sistema operativo saber cuánto tiempo se ejecuta un proceso además de refinar nuevamente el mecanismo de transferencia del control del procesador. Después de revisar el refinamiento, mostramos su corrección solamente usando FDR2, pues la prueba manual es demasiado complicada. Terminamos el capítulo con la implantación en pseudocódigo del quinto refinamiento del núcleo del sistema operativo.

4.2 Especificación inicial.

La idea básica en nuestra especificación de un núcleo multitareas es la de *entrelazamiento de acciones*. En efecto, para implantar la ejecución concurrente de varios programas, el procesador entrelaza las acciones de los diferentes programas para dar la impresión que todos ellos se ejecutan en forma simultánea. Por lo tanto, nuestra especificación inicial consiste en observar una serie de eventos que corresponden al entrelazamiento de los eventos generados por dos programas denominados *Corutina1* y *Corutina2*. El hecho de sólo interesarnos a dos programas en la especificación no es restrictivo, ya que los mecanismos necesarios para realizar el entrelazamiento de las acciones de dos programas, son los mismos que se necesitan para entrelazar las acciones de n programas. Los programas *Corutina1* y *Corutina2* corresponden a lo que puede llamarse como “programas de usuario” en un sistema de cómputo.

Sea $\langle m_{1,1}, m_{1,2}, m_{1,3}, \dots \rangle$ y $\langle m_{2,1}, m_{2,2}, m_{2,3}, \dots \rangle$ dos secuencias de eventos generadas por los programas *Corutina1* y *Corutina2* respectivamente. Cada evento $m_{i,j}$ puede verse a cualquier nivel de abstracción, desde ser la ejecución de una micro instrucción hasta la ejecución de un subprograma o función. Por lo tanto, la especificación inicial en CSP es:

$Núcleo = Corrutinas_{1,1}$

$Corrutinas_{i,j} = m_{1,i} \rightarrow m_{2,j} \rightarrow Corrutinas_{i+1,j+1}$

En donde el alfabeto del proceso *Núcleo* es el conjunto de eventos que éste puede ejecutar, esto es:

$$\alpha Núcleo = \{m_{i,j} \mid i,j \geq 1\}$$

4.3 Refinamientos de la especificación inicial.

La especificación inicial del proceso *Núcleo* sólo nos indica cuales son los eventos observables de la ejecución concurrente de los programas *Corutina1* y *Corutina2*. Esto es intencional, pues la especificación inicial sólo debe concentrarse a lo que *se debe* hacer y no al *cómo* se hace. Para pasar a la implantación de una especificación en algún lenguaje de programación, debemos de *refinar* la especificación inicial, de modo que, además de decir lo que se debe hacer, se describa el cómo se hace. Sin embargo, el paso de una especificación inicial a una que contenga todos los detalles de implantación, debe hacerse en forma gradual a través de *pasos de refinamiento*. Cada paso de refinamiento se dedica a un detalle particular de la especificación. El refinamiento de una especificación debe hacerse en forma cuidadosa, de modo que no se alteren las propiedades fundamentales de la especificación inicial. Es aquí donde los métodos formales muestran su utilidad. Con la ayuda del sistema de prueba, se debe probar la corrección de cada refinamiento, de modo que la especificación que se refina no se vea alterada en cada paso de refinamiento. Los siguientes párrafos muestran los refinamientos de la especificación original.

4.3.1 Primer refinamiento.

4.3.1.1 Especificación del primer refinamiento

El primer refinamiento consiste en modularizar la especificación inicial mediante el enunciado explícito de dos nuevos procesos: *Corrutina1_1_i* y *Corrutina1_2_i*³. Estos modelan el comportamiento de los programas de usuario *Corutina1* y *Corutina2* respectivamente. La definición de cada uno de ellos es una definición mutuamente recursiva. Los subíndices *i* en la definición de los nuevos procesos, señalan el *siguiente* evento en que éstos se comprometen ---i.e. “se ejecutan”---. El primer refinamiento de la especificación inicial es:

$$\text{Núcleo1} = \text{Corrutina1_1}_{i_1}$$

$$\text{Corrutina1_1}_{i_1} = m_{1,i_1} \rightarrow \text{Corrutina1_2}_{i_1}$$

$$\text{Corrutina1_2}_{i_1} = m_{2,i_1} \rightarrow \text{Corrutina1_1}_{i_{1+1}}$$

En donde el alfabeto de *Núcleo1* es el mismo que el de *Núcleo* y está dado por la siguiente expresión:

$$\alpha\text{Núcleo1} = \{m_{i,j} \mid i,j \geq 1\}$$

³ Para evitar ambigüedad, cuando un proceso aparezca en más de un refinamiento seguiremos el siguiente formato: *ProcesoX_Y*, en donde *X* indica el número de refinamiento al cual pertenece el proceso, y *Y* indica el número de proceso dentro de la especificación, este campo sólo irá en caso de que existan varios procesos del mismo tipo en la misma especificación.

4.3.1.2 Prueba manual del primer refinamiento

A continuación probamos la corrección del primer refinamiento, esto lo llevamos a cabo utilizando leyes del álgebra de procesos CSP

Antes de pasar a la demostración, mencionemos algunos hechos.

- Dado que:

$$\text{Núcleo}1 = \text{Corrutina}1_1$$

y

$$\begin{aligned} \text{Corrutina}1_1 &= m_{1,i} \rightarrow \text{Corrutina}1_2 &<\text{definición de } \text{Corrutina}1_1 > \\ &= m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina}1_1 &<\text{sustituimos } \text{Corrutina}1_2, \\ & &\text{por su definición } > \end{aligned}$$

Por lo cual:

$$\text{Núcleo}1 = \text{Corrutina}1_1 = m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina}1_1 \quad \text{con } i=1.$$

- Dado lo anterior, observamos que *Núcleo* y *Núcleo1* fueron definidos de manera recursiva con guardias (Para ver la definición de procesos recursivos con guardias, refiérase a la sección 2.1.2), por lo cual tenemos:

$$\text{Núcleo} = \mu X_{i,j}:A. F(X_{i,j}) \quad <\text{sustituyendo, tenemos: } >$$

$$\text{Núcleo} = \mu X_{i,j}: \{m_{i,j} \mid i, j \geq 1\}. (m_{1,i+1} \rightarrow m_{2,j+1} \rightarrow X_{i,j})$$

$$\text{Núcleo}1 = \mu Y_i: B. G(Y_i) \quad <\text{sustituyendo, tenemos: } >$$

$$\text{Núcleo}1 = \mu Y_i: \{m_{i,j} \mid i, j \geq 1\}. (m_{1,i+1} \rightarrow m_{2,i+1} \rightarrow Y_i)$$

Debido a que ambos procesos son deterministas, para probar que *Núcleo1* es un refinamiento de *Núcleo*, debemos probar que (Ver sección 2.8.2 de [Hoare85]):

- $\alpha \text{Núcleo} = \alpha \text{Núcleo}1$
- $\text{trazas}(\text{Núcleo}) = \text{trazas}(\text{Núcleo}1)$

Pero de sus definiciones, observamos que la primer condición se cumple de manera obvia, pues ambos procesos tienen definidos sus alfabetos, y estos son los mismos. Por lo cual, sólo nos resta probar que las trazas de ambos procesos son las mismas.

Prueba.

- Por demostrar que:

$$\text{Trazas}(\text{Núcleo}) = \text{Trazas}(\text{Núcleo}1)$$

O bien que:

$Trazas(Corrutina_{1,1}) = Trazas(Corrutina1_I_1)$ < sustituyendo Núcleo y Núcleo1 por sus definiciones >

Esto equivale a probar que:

$Trazas(\mu X_{i,j}:A. F(X_{i,j})) = Trazas(\mu Y_i:B. G(Y_i))$ < de las definiciones de procesos recursivos con guardias >

Dado que:

$Trazas(\mu X_{i,j}:A. F(X_{i,j})) = \bigcup_{n \geq 0} Trazas(F^n(STOP_{\alpha Nucleo}))$ <ver la ley L5 de la sección 2.1.6.1>

y que:

$Trazas(\mu Y_i:B. G(Y_i)) = \bigcup_{n \geq 0} Trazas(G^n(STOP_{\alpha Nucleo1}))$ <ver la ley L5 de la sección 2.1.6.1>

entonces, esto equivale a probar que:

$\bigcup_{n \geq 0} Trazas(F^n(STOP_{\alpha Nucleo})) = \bigcup_{n \geq 0} Trazas(G^n(STOP_{\alpha Nucleo1}))$

Demostración:

La demostración la realizaremos por inducción sobre n , el número de iteraciones de los procesos recursivos con guardias, (Ver 2.1.6.1 L5)

i) Para $n=0$ tenemos:

$Trazas(F^0(STOP_{\alpha Nucleo})) = Trazas(G^0(STOP_{\alpha Nucleo1}))$
 $Trazas(STOP_{\alpha Nucleo}) = Trazas(STOP_{\alpha Nucleo1})$ < def. de $G^0(X)$ y $F^0(X)$ >
 $\{\langle \rangle\} = \{\langle \rangle\}$ < def. de trazas para $STOP$ >

ii) Suponemos cierta la igualdad, para $n-1$, esto es:

$Trazas(F^{n-1}(STOP_{\alpha Nucleo})) = Trazas(G^{n-1}(STOP_{\alpha Nucleo1}))$

iii) Ahora, en base a tal suposición demosntremos la igualdad para n iteraciones.

Por demostrar que:

$Trazas(F^n(STOP_{\alpha Nucleo})) = Trazas(G^n(STOP_{\alpha Nucleo1}))$.

Partamos del lado izquierdo de la igualdad:

$Trazas(F^n(STOP_{\alpha Nucleo}))$

$$\begin{aligned}
& \langle \text{Desarrollando } F^n(STOP_{\alpha Nucleo}) \text{ tenemos: } \rangle \\
& = \text{Trazas}(F(F^{n-1}(STOP_{\alpha Nucleo}))) \\
& \quad \langle \text{sustituyendo la definición de } F(F^{n-1}(STOP_{\alpha Nucleo})), \text{ tenemos: } \rangle \\
& = \text{Trazas}(m_{1,i-1} \rightarrow m_{2,j-1} \rightarrow F^{n-1}(STOP_{\alpha Nucleo})) \\
& \quad \langle \text{de la definición de trazas de procesos 2.1.6 L4, tenemos: } \rangle \\
& = \{ \langle \rangle, \langle m_{1,i-1} \rangle, \langle m_{1,i-1}, m_{2,j-1} \rangle \} \cup \text{Trazas}(F^{n-1}(STOP_{\alpha Nucleo})) \\
& \quad \langle \text{Por hipótesis de inducción, tenemos: } \rangle \\
& = \{ \langle \rangle, \langle m_{1,i-1} \rangle, \langle m_{1,i-1}, m_{2,j-1} \rangle \} \cup \text{Trazas}(G^{n-1}(STOP_{\alpha Nucleo1})) \\
& \quad \langle \text{formando un proceso con las trazas anteriores, tenemos: } \rangle \\
& = \text{Trazas}(m_{1,i-1} \rightarrow m_{2,j-1} \rightarrow G^{n-1}(STOP_{\alpha Nucleo1})) \\
& \quad \langle \text{de acuerdo a la definición de } G(G^{n-1}(STOP_{\alpha Nucleo1})), \text{ tenemos: } \rangle \\
& = \text{Trazas}(G(G^{n-1}(STOP_{\alpha Nucleo1}))) \\
& = \text{Trazas}(G^n(STOP_{\alpha Nucleo1})). \\
& \quad \langle \text{Por lo cual: } \rangle \\
& \text{Trazas}(F^n(STOP_{\alpha Nucleo})) = \text{Trazas}(G^n(STOP_{\alpha Nucleo1})).
\end{aligned}$$

Con esto terminamos la demostración por inducción, con lo cual podemos concluir que efectivamente *Núcleo1* es un refinamiento de *Núcleo*

4.3.1.3 Prueba Automática del primer refinamiento

Dadas las especificaciones de *Núcleo* y *Núcleo1*, transcribimos estas especificaciones al lenguaje de entrada de FDR2, visto en el capítulo anterior. FDR2 genera un sistema de transición para el sistema especificado y otro para el refinamiento, con el fin de compararlos [Roscoe94]. Los procesos especificados, *Núcleo* y *Núcleo1* nunca terminan pues fueron definidos de manera recursiva con guardias, de tal forma que FDR2 intentará generar un

sistema de transiciones, (con un número infinito de transiciones) para estas especificaciones, lo cual evidentemente es imposible, por lo que FDR2 emitirá un mensaje de error. Debido a lo anterior y dado que FDR2 no puede probar refinamientos de procesos con alfabetos infinitos [Roscoe94], debemos restringir el número de eventos ejecutados por *Núcleo* y *Núcleo1*. Esto es, el número de eventos de los programas de usuario $m_{i,j}$ con $i, j \geq 1$ presentes en ambas especificaciones, debe ser limitado a un conjunto menor, como por ejemplo, $m_{i,j}$ con $1 \leq i, j \leq n$, en donde n es un entero finito. En nuestras especificaciones para FDR2, n tendrá un valor de 4. El valor es pequeño y se eligió así, pues cuan más grande sea el número de eventos en los que se compromete un proceso más tardará su prueba en FDR2.

El limitar el número de eventos de los programas de usuario en los cuales se comprometen *Núcleo* y *Núcleo1* no es restrictivo, pues un programa de usuario nunca es infinito.

Una vez limitado el número de eventos de los programas de usuario en los que se comprometen *Núcleo* y *Núcleo1* procedemos a transcribir sus especificaciones al formato de entrada de FDR2.

```
-- =====
--                               Especificacion Inicial
-- =====
```

```
channel m: Int.Int
```

```
Nucleo = Corrutinas(1,1)
```

```
Corrutinas(i,j) = if i<4 then (m.1.i -> m.2.j -> Corrutinas(i+1,j+1))
                  else (SKIP)
```

```
-- =====
--                               Primer Refinamiento
-- =====
```

```
Núcleo = Corrutina1_1(1)
```

```
Corrutina1_1(i) = if i<4 then ( m.1.i -> Corrutina1_2(i)
                              else (SKIP)
```

```
Corrutina1_2(i) = m.2.i -> Corrutina1_1(i+1)
```

Las especificaciones anteriores fueron guardadas en un archivo tipo texto e introducidas a FDR2 como se mencionó en la sección 3.4.2.2.1. Después de cargar el archivo con los *scripts* se define que la especificación es el proceso *Núcleo* y la implantación el proceso *Núcleo1*. Posteriormente se elige el tipo de refinamiento a probar. Terminada la verificación del

refinamiento, FDR2 nos muestra la pantalla de la figura 4.1, en donde podemos constatar a través de la señal que *Núcleo [Núcleo1*.

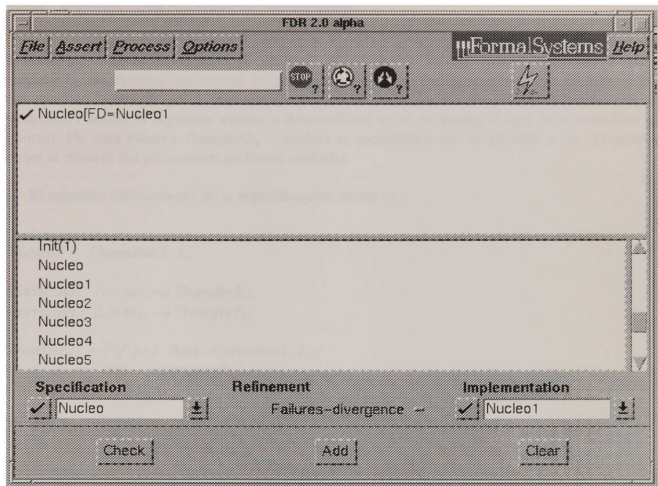


Figura 4.1 Prueba del primer refinamiento utilizando FDR2.

4.3.1.4 Motivación al segundo refinamiento

La modularización de la especificación inicial en el primer refinamiento, nos permite especificar el comportamiento del núcleo multitareas en función del comportamiento de los programas de usuario. Sin embargo, en este refinamiento no aparecen los “programas del sistema” que le permiten a los programas de usuario realizar el entrelazamiento de sus acciones ---i.e. compartir el uso de la CPU entre diferentes procesos---. Por lo tanto, en el segundo refinamiento, tenemos como objetivo la especificación de un mecanismo que, de manera *explícita*, le permita a un programa la transferencia de control del procesador; a este mecanismo le denominamos Transfer.

4.3.2 Segundo Refinamiento

4.3.2.1 Especificación del segundo refinamiento

El comportamiento del mecanismo Transfer lo modelamos a través del proceso $Transfer_{2,i,j}$. Este es utilizado en el comportamiento de $Corrutina2_{1,i}$ y $Corrutina2_{2,i}$ para modelar la transferencia de control del procesador entre los procesos de usuario. El subíndice j del proceso Transfer indica qué corrutina gana el control del procesador, mientras que el subíndice i señala el siguiente evento a desarrollarse en el programa al que se le transfiere el control. De esta manera $Transfer_{2,i,j}$ modela el mecanismo que le permite a un programa ceder el control del procesador en forma explícita.

El segundo refinamiento de la especificación inicial es:

$$Núcleo2 = Corrutina2_{1,i}$$

$$Corrutina2_{1,i} = m_{1,i} \rightarrow Transfer_{2,i,2}$$

$$Corrutina2_{2,i} = m_{2,i} \rightarrow Transfer_{2,i,1}$$

$$Transfer_{2,i,j} = (\text{if } j=1 \text{ then } Corrutina2_{1,i+1} \\ \text{else } Corrutina2_{2,i} \\)$$

En donde:

$$\alpha Núcleo2 = \{ m_{i,j} \mid i,j \geq 1 \}$$

4.3.2.2 Prueba manual del segundo refinamiento

Antes de probar que $Núcleo2$ es un refinamiento correcto de $Núcleo1$, reescribamos la definición de los procesos, para ello partamos de la definición de $Núcleo2$.

$$Núcleo2 = Corrutina2_{1,i} \quad i=1$$

<de la definición de $Corrutina2_{1,i}$, tenemos >

$$= m_{1,i} \rightarrow Transfer_{2,i,2} \quad i=1$$

<sustituyendo $Transfer_{2,i,2}$ por su definición>

$$= m_{1,i} \rightarrow Corrutina2_{2,i} \quad i=1$$

<de la definición de $Corrutina2_{2,i}$ tenemos: >

$$= m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Transfer}_{2,i,1} \quad i=1$$

< sustituyendo $\text{Transfer}_{2,i,1}$ por su definición >

$$= m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina2_1}_{i+1} \quad i=1$$

< Por lo tanto: >

$$\text{Núcleo2} = \text{Corrutina2_1}_i = m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina2_1}_{i+1} \quad i=1$$

En donde claramente se ve que Núcleo2 está definido en términos de un proceso recursivo con guardias.

Por otro lado tenemos de la definición de Núcleo1 que:

$$\text{Núcleo1} = \text{Corrutina1_1}_i \quad i=1$$

< de la definición de Corrutina1_1_i , tenemos: >

$$= m_{1,i} \rightarrow \text{Corrutina1_2}_i \quad i=1$$

< de la definición de Corrutina1_2_i tenemos: >

$$= m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina1_1}_{i+1} \quad i=1$$

< Por lo tanto: >

$$\text{Núcleo1} = \text{Corrutina1_1}_i = m_{1,i} \rightarrow m_{2,i} \rightarrow \text{Corrutina1_1}_{i+1} \quad i=1$$

En donde claramente se puede observar que Núcleo1 también está definido en términos de un proceso recursivo con guardias.

Prueba

Para probar que Núcleo2 refina a Núcleo1 debemos probar que:

- i) $\alpha \text{Núcleo1} = \alpha \text{Núcleo2}$
- ii) $\text{trazas}(\text{Núcleo1}) = \text{trazas}(\text{Núcleo2})$

Pero i) es evidente que se cumple, pues ambos alfabetos fueron definidos explícitamente y son los mismos, por lo cual, sólo debemos probar que las trazas de ambos procesos son las mismas, esto es:

$$\text{trazas}(\text{Núcleo1}) = \text{trazas}(\text{Núcleo2})$$

sustituyendo *Núcleo1* y *Núcleo2* por sus definiciones, tenemos:

$$\text{Trazas}(\text{Corrutina1_1}_i) = \text{Trazas}(\text{Corrutina2_1}_i)$$

Dado que ambos procesos fueron definidos de manera recursiva con guardias, entonces, lo que tenemos que probar es que:

$$\text{Trazas}(\mu X_i:A. F(X_i)) = \text{Trazas}(\mu Y_i:B. G(Y_i))$$

O bien que:

$$\begin{aligned} \text{Trazas}(\mu X_i:\{m_{i,j} \mid i,j \geq 1\} . (m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow X_i)) = \\ \text{Trazas}(\mu Y_i:\{m_{i,j} \mid i,j \geq 1\} . (m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow Y_i)) \end{aligned}$$

Dado que dos procesos recursivos con guardias son iguales si: tienen el mismo alfabeto, y sus ecuaciones recursivas con guardias son similares (ver. sección 2.1.3), por lo tanto la igualdad de arriba es correcta, con lo que nos permite concluir que *Núcleo2* es un refinamiento correcto de *Núcleo1*. □

4.3.2.3 Prueba Automática del segundo refinamiento

Dadas las definiciones de *Núcleo1* y *Núcleo2*, procedimos a transcribirlas a la notación aceptada por FDR2. Reducimos nuevamente el número de eventos ejecutados por los procesos de usuario, al mismo número que lo hicimos en el primer refinamiento. Dado que *Núcleo1* ya fue mostrado en el anterior script y dado que no sufrió modificaciones, entonces, por fines de espacio no aparecerá en este script. Siguiendo un procedimiento similar al realizado en el refinamiento automático anterior, tenemos:

```
-- =====
--                               Segundo Refinamiento
-- =====
```

```
Nucleo2 = Corrutina2_1(1)
```

```
Corrutina2_1(i) = if i<4 then ( m.1.i -> Transfer2(2,i) )
                  else (SKIP)
```

```
Corrutina2_2(i) = m.2.i -> Transfer2(1,i)
```

```
Transfer2(j,i) = if j==1 then Corrutina2_1(i+1)
                  else Corrutina2_2(i)
```

El resultado obtenido de FDR2 al cuestionarle si *Núcleo2* es un refinamiento de *Núcleo1* es afirmativo, y es ilustrado en la figura 4.2.

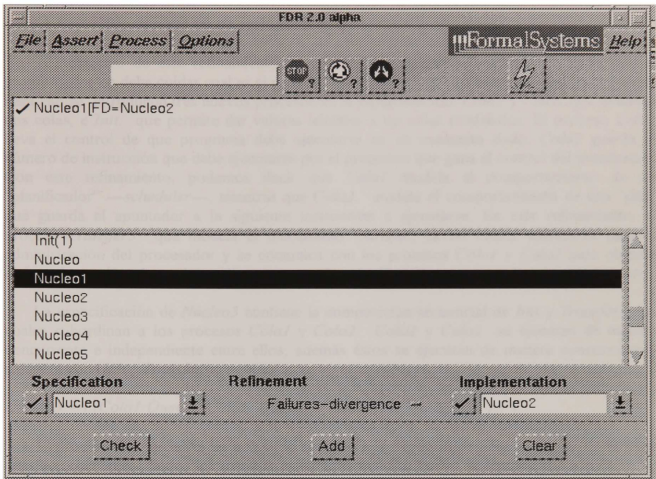


Figura 4.2. Prueba del segundo refinamiento utilizando FDR2.

4.3.2.4 Motivación al tercer refinamiento

La utilización de los subíndices i y j en la definición del mecanismo Transfer, provocan que el programador de una aplicación concurrente tenga en cuenta a cada momento que programa debe tomar el control de la CPU y a partir de donde éste debe reasumirse. La administración de esos subíndices aumenta la complejidad de una aplicación, a medida que ésta es compuesta de un número considerable de programas cooperantes. Esta restricción es resuelta en el siguiente refinamiento.

4.3.3 Tercer Refinamiento

4.3.3.1 Especificación del tercer refinamiento.

El objetivo de este refinamiento es refinar el mecanismo *Transfer*, de modo que el programador no deba cuidar cual es el programa que se ejecuta, ni a partir de cual instrucción. Para ello se introducen tres nuevos procesos en la especificación: *Cola1* y *Cola2* que modelan dos colas, e *Init*, que permite dar valores iniciales a las colas modeladas. El proceso *Cola1* lleva el control de que programa debe ejecutarse en un momento dado; *Cola2* guarda el número de instrucción que debe ejecutarse por el programa que gana el control del procesador. Con este refinamiento, podemos decir que *Cola1* modela el comportamiento de un “planificador” ---*scheduler*---, mientras que *Cola2*, modela el comportamiento de una “pila” que guarda el apuntador a la siguiente instrucción a ejecutarse. En este refinamiento, el proceso *Transfer3* que modela el mecanismo *Transfer*, ya no utiliza subíndices para la administración del procesador y se comunica con los procesos *Cola1* y *Cola2* para obtener dicha información. A continuación se presenta la especificación completa de este refinamiento.

La especificación de *Núcleo3* contiene la composición secuencial de *Init* y *Transfer3*, los cuales subordinan a los procesos *Cola1* y *Cola2*. *Cola1* y *Cola2* se ejecutan de manera concurrente e independiente entre ellos; además éstos se ejecutan de manera concurrente y sincrónica con (*Init* ; *Transfer3*):

$$Núcleo3 = ((Cola1:Queue || Cola2: Queue) // (Init_{<1,2>} ; Transfer3)) \ \checkmark$$

El proceso *Init* inicializa las colas (de acuerdo a la sublista que tiene asociada) y termina satisfactoriamente. Todos los eventos ejecutados por *Init* (salvo la terminación satisfactoria, \checkmark) son también eventos de las colas, por lo cual no pertenecen al alfabeto de *Núcleo3* (definición de procesos subordinados A11. //1).

$$Init_{<1,2>} = Cola1.left!x \rightarrow Cola2.left!1 \rightarrow Init_s \\ Init_{<1,2>} = SKIP$$

Las colas comienzan en un estado de “bloqueadas”, esperando recibir un valor por el canal izquierdo. Al recibir las colas algún valor por el canal izquierdo, lo guardan en el almacén que tienen asociado. Una vez que tienen al menos un valor almacenado, su comportamiento diferirá de entre los siguientes:

- i) Espera recibir un nuevo valor por el canal izquierdo, cuando llega, revisa si al intentar introducirlo, el tamaño del almacén no es excedido, si no es excedido, se guarda el valor, en otro caso espera a que alguien retire un elemento del almacén para poder guardar el valor recibido.
- ii) Espera que alguien retire un elemento por el canal derecho

la elección del comportamiento, lo decide el primer evento del ambiente.

$$\begin{aligned}
Queue &= S_{<>} \\
S_{<>} &= left?x \rightarrow S_{<x>} \\
S_{<x>} &= ((left?y \rightarrow \text{if } \#<x>^{\wedge} s^{\wedge} <y> < n \text{ then } S_{<x^{\wedge} s^{\wedge} y>} \\
&\quad \text{else } (right!x \rightarrow S_{<x>})) \\
&\quad) \\
&\quad \square \\
&\quad (right!x \rightarrow S_{<>}) \\
&)
\end{aligned}$$

Las rutinas de usuario permanecen sin cambio, a excepción de que ahora en vez de invocar a *Transfer2* invocan a *Transfer3*.

$Corrutina3_1_i = m_{1,i} \rightarrow Transfer3$

$Corrutina3_2_i = m_{2,i} \rightarrow Transfer3$

El proceso *Transfer3* siempre invoca a la corrutina cuyo identificador se encuentra al frente de *Cola1*, y transmite el número de evento a ejecutarse en esa corrutina; éste lo obtiene del frente de *Cola2*. El valor recibido de *Cola1* se vuelve a encolar, al igual que el de *Cola2*, pero este último incrementado en 1, para que se ejecute el siguiente evento.

$$\begin{aligned}
Transfer3 &= Cola1.right?j \rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!i+1 \rightarrow \\
&\quad \rightarrow (\text{if } j=1 \text{ then } Corrutina3_1_i \\
&\quad \quad \text{else } Corrutina3_2_i \\
&\quad)
\end{aligned}$$

Cabe señalar que en la definición de *Núcleo3* ocultamos el evento \surd , ya que no está dentro del alfabeto de la especificación anterior, y que el alfabeto de *Núcleo3* es exactamente el mismo que el de *Núcleo2*.

4.3.3.2 Prueba manual del tercer refinamiento.

Dado que ambos procesos son deterministas, para probar que *Núcleo3* es un refinamiento de *Núcleo2*, debemos probar que (Ver sección 2.8.2 de [Hoare85]):

- i) $\alpha Nucleo2 = \alpha Nucleo3$
- ii) $trazas(Nucleo2) = trazas(Nucleo3)$

Para demostrar lo anterior, procederemos tal como lo hicimos en la prueba del primer refinamiento (Sección 4.3.1.2), esto es, dado que sabemos de antemano que los alfabetos son los mismos, bastará probar la igualdad ii).

Antes de comenzar la prueba, veamos los siguientes hechos.

<de la definición de *Núcleo3*, tenemos: >

$$\text{Trazas}(\text{Núcleo3}) = \text{Trazas}((\text{Cola1} \parallel \text{Cola2}) // (\text{Init}_{\langle 1,2 \rangle} ; \text{Transfer3})) \setminus \sqrt{\quad} \quad \dots (1)$$

<Sustituyendo *Init*, *Cola1* y *Cola2* por su definición>

$$= \text{Trazas}((\text{Cola1.S}_{\langle \rangle} \parallel \text{Cola2.S}_{\langle \rangle}) // (\text{Cola1.left!1} \rightarrow \text{Cola2.left!1} \rightarrow \text{Init}_{\langle 2 \rangle} ; \text{Transfer3})) \setminus \sqrt{\quad}$$

<Sustituyendo *Cola1.S* y *Cola2.S* por su definición, tenemos: >

$$= \text{Trazas}((\text{Cola1.left?x} \rightarrow \text{Cola1.S}_{\langle x \rangle} \parallel \text{Cola2.left?x} \rightarrow \text{Cola2.S}_{\langle x \rangle}) // (\text{Cola1.left!1} \rightarrow \text{Cola2.left!1} \rightarrow \text{Init}_{\langle 2 \rangle} ; \text{Transfer3})) \setminus \sqrt{\quad}$$

<Dado que la comunicación se realiza de manera interna entre proceso subordinado y proceso subordinador, esta no será visible desde fuera (A11. //1), por lo cual tenemos: >

$$= \text{Trazas}((\text{Cola1.S}_{\langle 1 \rangle} \parallel \text{Cola2.S}_{\langle 1 \rangle}) // (\text{Init}_{\langle 2 \rangle} ; \text{Transfer3})) \setminus \sqrt{\quad}$$

<Sustituyendo *Init*, *Cola1.S* y *Cola2.S* por sus definiciones, tenemos: >

$$= \text{Trazas}((\text{Cola1.left?y} \rightarrow \text{if } \# \langle 1 \rangle \wedge \langle \rangle \wedge \langle y \rangle < n \text{ then } \text{Cola1.S}_{\langle 1 \rangle \wedge \langle y \rangle} \text{ else } (\text{Cola1.right!1} \rightarrow \text{Cola1.S}_{\langle y \rangle})) \parallel \text{Cola1.right!1} \rightarrow \text{Cola1.S}_{\langle \rangle})$$

```

    ( (Cola2.left?y → if #<I>^<I>^<y> < n then Cola2.S<I>^<I>^<y>
      else (Cola2.right!1 → Cola2.S<y>))
    )
    □
    (Cola2.right!1 → Cola2.S<y>)
  )
)
//
(Cola1.left!2 → Cola2.left!1 → SKIP ; Transfer3)
) \ (✓)
)

```

< Dado que la comunicación se realiza de manera interna entre proceso subordinado y proceso subordinador, esta no será visible desde fuera. Además dado que ✓, no volverá a aparecer, tenemos: >

=Trazas((Cola1.S<1,2> ||| Cola2.S<1,1>) // Transfer3)

< sustituyendo Cola1.S<1,2>, Cola2.S<1,1> y Transfer3 por sus definiciones, tenemos: >

```

=Trazas( ( ( ( Cola1.left?y → if #<1>^<2>^<y> < n then Cola1.S<1>^<2>^<y>
  else (Cola1.right!1 → Cola1.S<2>^<y>)
  )
  □
  (Cola1.right!1 → Cola1.S<2>)
  )
  |||
  ( (Cola2.left?y → if #<I>^<I>^<y> < n then Cola2.S<I>^<I>^<y>
    else (Cola2.right!1 → Cola2.S<I>^<y>)
  )
  □
  (Cola2.right!1 → Cola2.S<I>)
  )
  )
  //
  (Cola1.right?j → Cola2.right?i → Cola1.left!j → Cola2.left!i+1 →
    → (if j=1 then Corrutina3_1,
      else Corrutina3_2,
    )
  )
)
)

```

< Dado que la comunicación se realiza de manera interna entre proceso subordinado y proceso subordinador, esta no será visible desde fuera. Por lo cual, tenemos: >

$$= \text{Trazas}((Cola1.S_{<2,1>} \parallel Cola2.S_{<1,2>}) // \text{Corrutina3_1}_1) \quad \dots(2)$$

<Sustituyendo *Corrutina3_1₁* por su definición, tenemos: >

$$= \text{Trazas}((Cola1.S_{<2,1>} \parallel Cola2.S_{<1,2>}) // m_{1,1} \rightarrow \text{Transfer3})$$

<Aplicando la ley A11. //5, tenemos: >

$$= \text{Trazas}(m_{1,1} \rightarrow ((Cola1.S_{<2,1>} \parallel Cola2.S_{<1,2>}) // \text{Transfer3}))$$

<sustituyendo *Transfer3*, *Cola1.S_{<2,1>}* y *Cola2.S_{<1,2>}* por sus definiciones, tenemos: >

$$= \text{Trazas}(m_{1,1} \rightarrow ((((Cola1.left?y \rightarrow \text{if } \#<2 \wedge \neg y < n \text{ then } Cola1.S_{<2 \wedge \neg y>} \text{ else } (Cola1.right!2 \rightarrow Cola1.S_{<1>\wedge y}))))))$$

$$\quad \square$$

$$\quad (Cola1.right!2 \rightarrow Cola1.S_{<1>})$$

$$\quad)$$

$$\quad \parallel$$

$$\quad ($$

$$\quad \quad (Cola2.left?y \rightarrow \text{if } \#<1 \wedge \neg y < n \text{ then } Cola2.S_{<1 \wedge \neg y>} \text{ else } (Cola2.right!1 \rightarrow Cola2.S_{<2>\wedge y}))$$

$$\quad \quad)$$

$$\quad \quad \square$$

$$\quad \quad (Cola2.right!1 \rightarrow Cola2.S_{<2>})$$

$$\quad \quad)$$

$$\quad)$$

$$\quad //$$

$$\quad (Cola1.right!j \rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!i+1 \rightarrow$$

$$\quad \quad \rightarrow (\text{if } j=1 \text{ then } \text{Corrutina3_1}_i \text{ else } \text{Corrutina3_2}_i)$$

$$\quad)$$

$$\quad)$$

< Dado que la comunicación se realiza de manera interna entre proceso subordinado y proceso subordinador, esta no será visible desde fuera. Por lo cual, tenemos: >

$$= \text{Trazas}(m_{1,1} \rightarrow ((Cola1.S_{<1,2>} \parallel Cola2.S_{<2,2>}) // \text{Corrutina3_2}_1))$$

<Si aplicamos a esta expresión, de manera similar, los mismos pasos que fueron aplicados a la expresión marcada por (2), tenemos: >

$$= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow ((Cola1.S_{<2,1>} \parallel Cola2.S_{<2,3>}) // \text{Corrutina3_1}_2)) \quad \dots (3)$$

De (1), (2) y (3) tenemos que:

$$\begin{aligned} \text{Trazas}(\text{Núcleo3}) &= \text{Trazas}((\text{Cola1.S}_{<2,1>} \parallel \parallel \text{Cola2.S}_{<1,2>} // \text{Corrutina3}_{-1_1}) \\ &= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow (\text{Cola1.S}_{<2,1>} \parallel \parallel \text{Cola2.S}_{<2,3>} // \text{Corrutina3}_{-1_2})) \end{aligned}$$

Dado que los eventos de las colas permanecen ocultos, entonces las trazas de *Núcleo3* estarán constituidas solamente por las trazas de *Corrutina3_1_1*, y las trazas de *Corrutina3_1_1* estarán formadas por:

$$\text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{Corrutina3}_{-1_2})$$

Por lo cual tenemos que:

$$\begin{aligned} \text{Trazas}(\text{Núcleo3}) &= \text{Trazas}(\text{Corrutina3}_{-1_1}) \\ &= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{Corrutina3}_{-1_{i+1}}) \quad i=1 \quad \dots \quad (4) \end{aligned}$$

En donde las trazas solamente contienen eventos externamente visibles. De lo anterior vemos que *Núcleo3* esta definido en términos de un proceso recursivo con guardias, similar al de los refinamientos anteriores.

Por otro lado tenemos.

$$\begin{aligned} \text{Trazas}(\text{Núcleo2}) &= \text{Trazas}(\text{Corrutina2}_{-1_1}) && \text{<de la definición de Núcleo2>} \\ & && \text{<de la definición de Corrutina2}_{-1_1} \text{>} \\ &= \text{Trazas}(m_{1,1} \rightarrow \text{Transfer}_{2,1,2}) \\ & && \text{<sustituyendo Transfer}_{2,1,2} \text{ por su definición>} \\ &= \text{Trazas}(m_{1,1} \rightarrow \text{if } j=1 \text{ then } \text{Corrutina2}_{-1_2} \text{ else } \text{Corrutina2}_{-2_1}) \\ & && \text{<dado que } j=2 \text{>} \\ &= \text{Trazas}(m_{1,1} \rightarrow \text{Corrutina2}_{-2_1}) \\ & && \text{<sustituyendo Corrutina2}_{-2_1} \text{ por su definición >} \\ &= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{Transfer}_{2,1,1}) \\ & && \text{<sustituyendo Transfer}_{2,1,1} \text{ por su definición>} \\ &= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{if } j=1 \text{ then } \text{Corrutina2}_{-1_2} \text{ else } \text{Corrutina2}_{-2_1}) \\ & && \text{<dado que } j=1 \text{>} \\ &= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{Corrutina2}_{-1_2}) \end{aligned}$$

Observamos claramente que :

$$\text{Trazas}(\text{Núcleo2}) = \text{Trazas}(\text{Corrutina2}_{-1_1}) \quad i=1$$

$$= \text{Trazas}(m_{1,1} \rightarrow m_{2,1} \rightarrow \text{Corrutina2_}I_{i+1}) \quad i=1. \quad \dots(5)$$

En donde se ve claramente la definición del proceso recursivo con guardias.

Justamente sobre los procesos definidos de manera recursiva con guardias en (4) y (5) nos basaremos para mostrar que:

$$\text{Trazas}(\text{Núcleo3}) = \text{Trazas}(\text{Núcleo2})$$

Prueba.

1) Por demostrar que:

$$\text{Trazas}(\text{Núcleo2}) = \text{Trazas}(\text{Núcleo3})$$

O bien que:

$$\text{Trazas}(\text{Corrutina2_}I_{\alpha\text{Núcleo2}}) = \text{Trazas}(\text{Corrutina3_}I_{\alpha\text{Núcleo3}}) \quad < \text{de 4 y 5} >$$

Esto Equivale a probar que:

$$\text{Trazas}(\mu X_i: A. F(X_i)) = \text{Trazas}(\mu Y_i: B. G(Y_i)) \quad < \text{de las definiciones de procesos recursivos con guardias} >$$

O bien que:

$$\begin{aligned} \text{Trazas}(\mu X_i: \{m_{i,j} \mid i,j \geq 1\} . (m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow X_i)) = \\ \text{Trazas}(\mu Y_i: \{m_{i,j} \mid i,j \geq 1\} . (m_{1,i-1} \rightarrow m_{2,i-1} \rightarrow Y_i)) \end{aligned}$$

De manera similar a la prueba manual del anterior refinamiento, tenemos que dos procesos recursivos con guardias son iguales si: tienen el mismo alfabeto, y sus ecuaciones recursivas con guardias son similares (ver. sección 2.1.3), por lo tanto la igualdad de arriba es correcta, con lo que concluimos que *Núcleo3* es un refinamiento correcto de *Núcleo2*. □

4.3.3.3 Prueba automática del tercer refinamiento.

Al transcribir las especificaciones de *Núcleo2* y *Núcleo3* a la notación aceptada por FDR2, tomando en cuenta la limitante de eventos en los programas de usuario mencionada en los refinamientos anteriores, obtuvimos el siguiente *script*. En el *script* sólo mostraremos a *Núcleo3*, ya que *Núcleo2* ya fue presentado en la anterior prueba.

```
channel c1left, c1right, c2left, c2right:{0,1,2,3,4}
```

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
-- como el paralelismo entre procesos con sus eventos sincronos y ocultos.

```
Nucleo3 = ( ( Cola1(<>)|| Cola2(<> )
            [|{ c1left,c2left,c1right, c2right |}]
            (Init(<1,2>) ; Transfer3 )
            ) \ { c1left, c1right, c2left, c2right |}
```

```
Init(s) = if ( null(s) ) then (SKIP)
         else ( c1left!head(s) -> c2left!1 -> Init(tail(s)) )
```

-- El tamaño de las colas lo limitamos a 6 para que la prueba entre procesos fuera rápida.

```
Cola1(s) = if (s == <>) then ( c1left?x -> Cola1(<x> )
                             else
                             (
                               (
                                 (c1left?y -> if ((#s)+1 < 6) then Cola1(s^<y>)
                                           else (c1right!(head(s)) -> Cola1(tail(s)^<y>))
                                )
                               []
                                 (c1right!(head(s)) -> Cola1(tail(s)) )
                               )
                             []
                             SKIP
                             )
                             )
```

-- Definimos Cola2 en función de Cola1 renombrando sus eventos.

```
Cola2(s) = Cola1(s)[[ c1left.n <- c2left.n, c1right.n <- c2right.n | n<-{0..4}]]
```

-- Limitamos el número de eventos en los cuales se comprometen los programas de usuario

```
Corrutina3_1(i) = if i<4 then ( m.1.i -> Transfer3 )
                 else (SKIP)
```

```
Corrutina3_2(i) = m.2.i -> Transfer3
```

Suma1(x) = (x + 1) % 5

```
Transfer3 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i)->
  ( if j==1 then Corrutina3_1(i)
    else Corrutina3_2(i)
  )
```

-- Fin del 3er. refinamiento.

El resultado obtenido de FDR2, fue que *Núcleo3* es un refinamiento correcto de *Núcleo2* tal como lo muestra la figura 4.3.

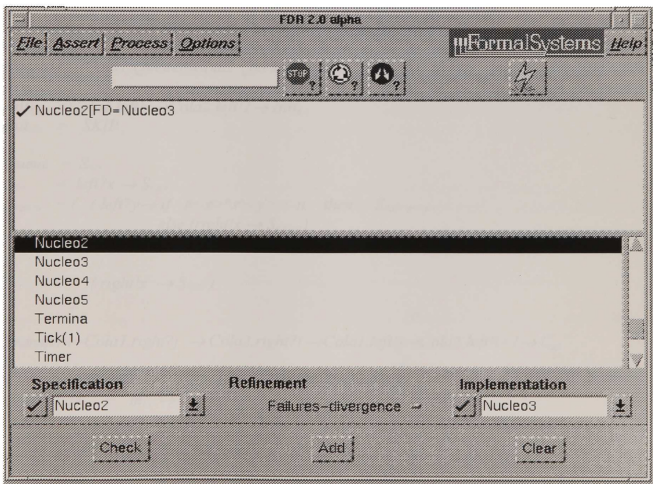


Figura 4.3 Prueba del tercer refinamiento utilizando FDR2.

4.3.3.4 Motivación al cuarto refinamiento

En este momento, tenemos la especificación de cuatro procesos que modelan los programas del sistema que permiten a un programador conmutar el procesador entre dos programas de usuario. El cuarto refinamiento tiene como objetivo permitir que un número mayor de programas de usuario puedan transferirse el control del procesador entre sí.

4.3.4 Cuarto Refinamiento

4.3.4.1 Especificación del cuarto refinamiento.

En este refinamiento se generaliza el mecanismo *Transfer*, de modo que no se restrinja la transferencia de control del procesador a sólo dos programas. Para ello se introduce la familia de procesos C_{ij} que modelan los diferentes programas de usuario en un sistema. El proceso *Transfer4* decide los valores de i y j después de comunicarse con los procesos *Cola2* y *Cola1* respectivamente. En este contexto, podemos pensar que j denota el "identificador" *--pid--* del programa que gana el control del procesador e i , el valor del contador de programa a partir de donde el programa seleccionado se ejecuta. La especificación de este refinamiento es:

$$\text{Núcleo4} = ((\text{Cola1:Queue} ||| \text{Cola2:Queue}) // (\text{Init}_{\langle 1,2, \dots, m \rangle}; \text{Transfer4})) \setminus \{m_{ij} \mid i > 2 \wedge j \geq 1\} \cup \{v\}$$

$$\begin{aligned} \text{Init}_{\langle x \rangle^s} &= \text{Cola1.left!x} \rightarrow \text{Cola2.left!1} \rightarrow \text{Init}_s \\ \text{Init}_{\langle \rangle} &= \text{SKIP} \end{aligned}$$

$$\begin{aligned} \text{Queue} &= S_{\langle \rangle} \\ S_{\langle \rangle} &= \text{left?x} \rightarrow S_{\langle x \rangle} \\ S_{\langle x \rangle^s} &= ((\text{left?y} \rightarrow \text{if } \# \langle x \rangle^s \wedge \# \langle y \rangle < n \text{ then } S_{\langle x \rangle^s \langle y \rangle} \\ &\quad \text{else } (\text{right!x} \rightarrow S_{\langle s \rangle})) \\ &\quad) \\ &\quad \square \\ &\quad (\text{right!x} \rightarrow S_{\langle s \rangle}) \\ &\quad) \end{aligned}$$

$$\text{Transfer4} = \text{Cola1.right?j} \rightarrow \text{Cola2.right?i} \rightarrow \text{Cola1.left!j} \rightarrow \text{Cola2.left!i+1} \rightarrow C_{j,i}$$

$$\begin{aligned} C_{j,i} &: \text{Corrutina4}_{j,i} \\ \text{Corrutina4}_{j,i} &= m_{j,i} \rightarrow \text{Transfer4} \end{aligned}$$

En donde el alfabeto de *Núcleo4* es exactamente el mismo que el de *Núcleo3*.

Nótese que el único proceso que cambió en este refinamiento es el proceso *Transfer4*, ahora en vez de invocar sólo a dos procesos de usuario invoca a m . Por otro lado podemos ver que en *Núcleo4* se ocultan los eventos de los procesos de usuario que fueron introducidos en

este refinamiento, esto tiene como finalidad facilitar la prueba de la corrección de este refinamiento, pero de ninguna manera implica que los programas de usuario no se ejecuten. El ocultamiento de estos eventos debe considerarse únicamente para fines de la prueba de corrección del refinamiento.

Dada la complejidad de este refinamiento no será presentada la prueba de manera manual, por lo cual, pasaremos de manera directa a probar su corrección mediante FDR2.

4.3.3.3 Prueba automática del cuarto refinamiento.

Dado que *Núcleo4* no se limita a atender sólo dos programas de usuario, sino que se generaliza a m programas de usuario. Esto nos lleva a serios problemas al intentar probar el refinamiento con FDR2, dado que el etiquetamiento no fue implantado como tal en FDR2. Al intentar utilizar el renombramiento se agotó la memoria virtual del sistema, lo cual nos impidió mostrar el refinamiento.

Por lo anterior, tuvimos que definir de manera explícita los procesos que implantan los programas de usuario, lo cual no es grave pues sólo se cambió una notación mas compacta para declarar procesos por una mas grande. Como el cuarto refinamiento es generalizado a atender m programas de usuario, instanciamos esa m a 5 y sólo declaramos de manera explicita esos 5 procesos que los implantan. Se observa de manera clara que ese número puede ser ampliado sin ningún problema al declarar mas procesos de usuario y adicionado invocaciones al proceso *Transfer4*.

Con lo explicado en el párrafo anterior hacemos notar que los cambios realizados no afectan en nada substancial la especificación de *Núcleo4*. Por lo cual, la especificación queda de la siguiente forma:

```
-- =====  
--                               Cuarto Refinamiento  
-- =====
```

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
-- como el paralelismo entre procesos con sus eventos sincronicos y ocultos.

```
Nucleo4 = ( ( Cola1(◇) ||| Cola2(◇) )  
           [|{ c1left, c2left, c1right, c2right }|]  
           (Init(<1,2,3,4,5>) ; Transfer4)  
           ) \ { | c1left, c2left, c1right, c2right, m.3, m.4, m.5 | }
```

-- Dado que el etiquetamiento no existe en FDR y el renombramiento cuando se utilizó agotó la
-- memoria virtual, tuvimos que buscar un camino alternativo para probar nuestro cuarto refi-

-- namiento. Por lo que invocamos corrutinas definidas de manera explícita.

```
Transfer4 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i) ->
  (if j==1 then Corrutina4_1(i)
   else (if j==2 then Corrutina4_2(i)
         else ( if j==3 then Corrutina4_3(i)
                else ( if j==4 then Corrutina4_4(i)
                       else Corrutina4_5(i)
                     )
              )
        )
  )
)
```

-- Debido a que el tiempo de prueba del refinamiento de FDR2 era muy grande limitamos el
-- número de programas de usuario a 5, aunque se observa de manera clara que este número
-- puede ser ampliado sin ningún problema.

```
Corrutina4_1(i) = if i<4 then ( m.1.i -> Transfer4 )
                  else (SKIP)
Corrutina4_2(i) = m.2.i -> Transfer4
Corrutina4_3(i) = m.3.i -> Transfer4
Corrutina4_4(i) = m.4.i -> Transfer4
Corrutina4_5(i) = m.5.i -> Transfer4
```

```
-- =====
--                               Fin del 4o. refinamiento.
-- =====
```

El resultado obtenido de FDR2 es que *Núcleo4* es un refinamiento correcto de *Núcleo3*, tal como lo muestra la figura 4.4.

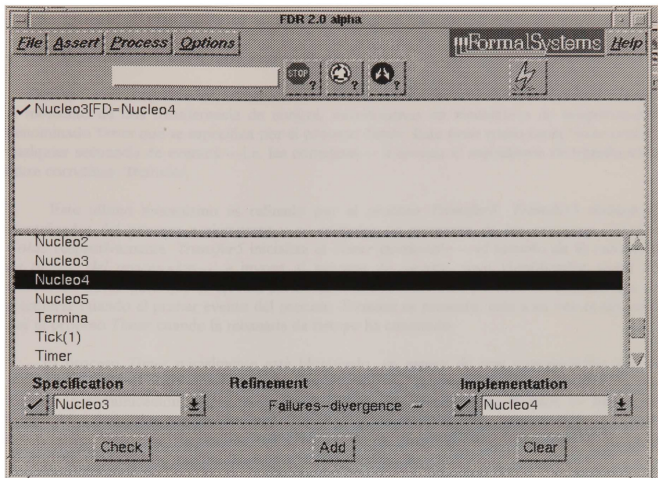


Figura 4.4 Prueba del cuarto refinamiento utilizando FDR2.

4.3.4.4 Motivación al quinto refinamiento

Tenemos hasta el momento la especificación de un mecanismo general que permite a los programadores transferir de manera *explícita* el control del procesador a cualquier programa. Este mecanismo sirve para implantar una facilidad de programación conocida como *corrutina*. Esta es una facilidad de programación similar al de subrutinas. La diferencia fundamental es que el llamado a una corrutina, transfiere el control de un programa a un punto de la corrutina, que no es necesariamente el inicio. El llamado a una subrutina siempre transfiere el control del programa al inicio de la subrutina.

La programación paralela mediante corrutinas se vuelve difícil a medida que el número de corrutinas aumenta, dado que el programador, además de tratar con la lógica de su problema, debe cuidar la manera en que el procesador se conmuta de corrutina a corrutina. Es en este sentido donde se propone el último refinamiento. El programador de una aplicación paralela, sólo se concentra a su problema y deja que los programas del sistema decidan a que programa se le transfiera el control del procesador y el momento de hacerlo.

4.3.5. Quinto Refinamiento

4.3.5.1 Especificación del quinto refinamiento

Para quitar al programador, la tarea de decidir qué corrutina toma el control del procesador al momento de una transferencia de control, introducimos un mecanismo de temporización denominado *Timer* que se especifica por el proceso *Timer*. Este tiene como tarea “interrumpir” cualquier secuencia de eventos ---i.e. las corrutinas--- e invocar el mecanismo de transferencia entre corrutinas *Transfer*.

Este último mecanismo es refinado por el proceso *Transfer5*. *Transfer5* obtiene el identificador del proceso a ejecutarse y su contador de programa de los procesos *Cola1* y *Cola2* respectivamente. *Transfer5* inicializa al *Timer* enviándole ---el tamaño de la rebanada de tiempo del procesador---, e invoca al proceso de usuario cuyo identificador acaba de extraer de *Cola1* para que se ejecute paralelamente con *Timer*. El proceso de usuario deja de ejecutarse cuando el primer evento del proceso *Termina* se presenta, éste a su vez es activado por el proceso *Timer* cuando la rebanada de tiempo ha concluido.

El proceso *Timer* inicialmente está bloqueado, en espera de una comunicación con el proceso *Transfer5*, para que le proporcione el valor del intervalo de interrupción. Una vez lograda la comunicación, el proceso *Timer* genera una secuencia de eventos --- *tick* --- de longitud igual al valor comunicado por *Transfer5*. Al término de dicha secuencia, la ejecución del proceso *Timer* es suspendida, en espera de una nueva comunicación con el proceso *Transfer5*, repitiéndose indefinidamente su comportamiento.

El proceso *Termina* es invocado por la comunicación del fin de “rebanada” de tiempo del proceso *Timer*, el cual es recibido por el canal *Timer.right*. El proceso *Termina* extrae el identificador del proceso que se ejecutó así como su contador de programa de los procesos *Cola1* y *Cola2* respectivamente para volver a encolarlos, no sin antes actualizar el contador del programa.

La especificación del quinto refinamiento es:

$$\text{Núcleo5} = ((Cola1:Queue || Cola2:Queue) // ((Init_{<1,2,...,m>} ; Transfer5) | (Timer))) \setminus \{ \checkmark, tick, Timer.left, Timer.right \} \cup \{ mi.j \mid i \geq 3, j \geq 1 \}$$

$$Init_{<x>} = Cola1.left!x \rightarrow Cola2.left!1 \rightarrow Init_s$$

$$Init_{<>} = SKIP$$

$$Queue = S_{<>}$$

$$S_{<x>} = left?x \rightarrow S_{<x>}$$

$$S_{<x>} = ((left?y \rightarrow \text{if } \#<x>^s \wedge \#<y> < n \text{ then } S_{<x>} \wedge \#<y> \text{ else } (right!x \rightarrow S_{<y>}))$$

)

□

$$(right!x \rightarrow S_{<x>})$$

)

$$\text{Transfer5} = (\text{Cola1.right?j} \rightarrow \text{Cola2.right?i} \rightarrow \text{Cola1.left!j} \rightarrow \text{Cola2.left!i+1} \rightarrow \text{Timer.left!1} \rightarrow \text{C}_{j,i} \nabla \text{Termina})$$

$$\text{C}_{j,i} : \text{Corrutina5}_{j,i}$$

$$\text{Corrutina5}_{j,i} = (m_{j,i} \rightarrow \text{tick} \rightarrow \text{tick} \rightarrow \text{Corrutina5}_{j,i+1})$$

$$\text{Termina} = \text{Timer.right?x} \rightarrow \text{Transfer5}$$

$$\text{Timer} = \text{left?i} \rightarrow \text{tick}_i$$

$$\text{tick}_i = (\text{if } i > 0 \text{ then } (\text{tick} \rightarrow \text{tick}_{i-1}) \\ \text{else } \text{right!0} \rightarrow \text{Timer})$$

La prueba manual del quinto refinamiento, al igual que la del cuarto refinamiento, no es presentada debido a su complejidad, por lo cual sólo mostraremos la prueba con FDR2.

4.3.5.2 Prueba automática del quinto refinamiento

Al transcribir las especificaciones de *Núcleo4* y *Núcleo5*, limitando el número de eventos de los procesos de usuario, como lo hicimos anteriormente, obtuvimos el siguiente *script*, en el cual sólo se muestra el quinto refinamiento.

```

=====
--
--                               Quinto refinamiento
--
=====
channel tick
channel TimerLeft: {0,1,2}
channel TimerRight : {0}

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
-- como el paralelismo entre procesos con sus eventos sincrosos y ocultos.

Nucleo5 = ( ( Cola1(<>) ||| Cola2(<>)
  [ | { c1left, c2left, c1right, c2right } | ]
  ( ( Init(<1,2,3,4,5>) ; Transfer5 )
    [ | { tick, TimerLeft, TimerRight } | ]
    Timer
  )
) \ { c1left, c2left, c1right, c2right, TimerLeft, TimerRight, tick, m.3, m.4, m.5 }

-- Proceso Transfer5, invoca al proceso que le corresponde ejecutarse, no si antes inicializar el
-- timer, Cuando se le termina su rebanada de tiempo la interrupción Termina realiza el cambio
-- de contexto e invoca nuevamente a Transfer5

```


El resultado obtenido de FDR2 fue que *Núcleo5* es un refinamiento correcto de *Núcleo4*, tal como lo muestra la figura 4.5.

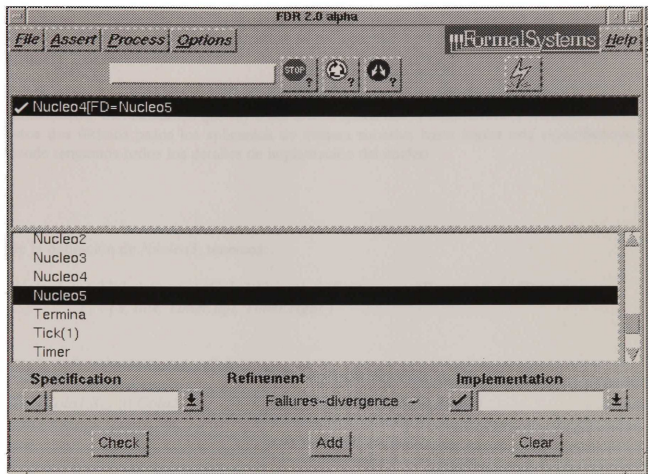


Figura 4.5 Prueba del quinto refinamiento utilizando FDR2.

Habiendo probado el quinto refinamiento, pasemos a su implantación

4.4 Hacia la Implantación

Para llevar el quinto refinamiento a una implantación en una máquina con un solo procesador, es necesario eliminar los operadores de entrelazamiento, subordinación, paralelismo, etc. Al eliminar los operadores de alto nivel de este refinamiento, tendremos sólo una secuencia de eventos. Esta secuencia de eventos formará justamente nuestro programa secuencial del núcleo básico del sistema operativo multitareas. Eliminamos los operadores de alto nivel mediante la aplicación sucesiva de las leyes del álgebra de procesos CSP, dadas en el apéndice 1 de esta tesis.

Para realizar lo explicado en el párrafo anterior, partimos de la especificación del quinto refinamiento, mencionamos que ley o definición aplicamos, y mostramos el resultado obtenido; estos dos últimos pasos los aplicamos de manera sucesiva hasta lograr una especificación en donde tengamos todos los detalles de implantación del núcleo.

De la definición de *Núcleo5*, tenemos:

$$\text{Núcleo5} = ((\text{Cola1:Queue} ||| \text{Cola2:Queue}) // ((\text{Init}_{\langle 1,2,\dots,m \rangle} ; \text{Transfer5}) \mid (\text{Timer}))) \setminus \{ \checkmark, \text{tick}, \text{Timer.left}, \text{Timer.right} \}$$

<de la definición de *Cola1* y *Cola2* tenemos: >

$$= ((\text{Cola1.S}_{\langle \rangle} ||| \text{Cola2.S}_{\langle \rangle}) // ((\text{Init}_{\langle 1,2,\dots,m \rangle} ; \text{Transfer5}) \mid \text{Timer})) \setminus \{ \checkmark, \text{tick}, \text{Timer.left}, \text{Timer.right} \}$$

< sustituimos *Cola1.S*_{<>}, *Cola2.S*_{<>} e *Init*_{<1,2,...,m>} por sus definiciones >

$$= ((\text{Cola1.left?x} \rightarrow \text{Cola1.S}_{\langle x \rangle} ||| \text{Cola2.left?x} \rightarrow \text{Cola2.S}_{\langle x \rangle}) // ((\text{Cola1.left!1} \rightarrow \text{Cola2.left!1} \rightarrow \text{Init}_{\langle 2,\dots,m \rangle} ; \text{Transfer5}) \mid \text{Timer})) \setminus \{ \checkmark, \text{tick}, \text{Timer.left}, \text{Timer.right} \}$$

< aplicando la ley A.2.2. | .4 y la A.10.comm.1, tenemos >

$$= (\text{Cola1.left.1} \rightarrow \text{Cola2.left.1} \rightarrow ((\text{Cola1.S}_{\langle 1 \rangle} ||| \text{Cola2.S}_{\langle 1 \rangle}) // ((\text{Init}_{\langle 2,\dots,m \rangle} ; \text{Transfer5}) \mid \text{Timer}))) \setminus \{ \checkmark, \text{tick}, \text{Timer.left}, \text{Timer.right} \}$$

< sustituimos *Cola1.S*_{<1>}, *Cola2.S*_{<1>} e *Init*_{<2,...,m>} por sus definiciones >

```

= ( Cola1.left.1 → Cola2.left.1 →
  ( ( ( Cola1.left?y → if #<I>^<>^<y> < n then Cola1.S<I>^<y>
    else ( Cola1.right!1 → Cola1.S<y> )
      )
    □
    (Cola1.right!1 → Cola1.S<>)
      )
    |||
    ( (Cola2.left?y → if #<I>^<>^ y < n then Cola2.S<I>^<y>
      else ( Cola2.right!1 → Cola2.S<y> )
        )
      □
      (Cola2.right!1 → Cola2.S<>)
        )
    )
  )
  //
  ( Cola1.left!2 → Cola2.left!1 → Init<3,...,m>; Transfer5 ) || Timer
)
) \ {√, tick, Timer.left, Timer.right }

```

<Aplicando la ley A.10.comm.1 sobre los eventos subrayados, tenemos: >

```

= ( Cola1.left.1 → Cola2.left.1 → Cola1.left.2 → Cola2.left.1 →
  ( (Cola1.S<1,2> ||| Cola2.S<1,1>) // ( (Init<3,...,m>; Transfer5 ) || Timer ) )
) \ {√, tick, Timer.left, Timer.right }

```

<Aplicando los dos pasos anteriores $m-2$ veces, con $m < n$, en donde m es el número de programas que atenderá el núcleo del sistema operativo, y n el tamaño máximo de las listas asociadas a los procesos $Cola1$ y $Cola2$, tenemos: >

```

= ( Cola1.left.1 → Cola2.left.1 → Cola1.left.2 → Cola2.left.1 → . . . Cola1.left.m →
  Cola2.left.1 →
  ( (Cola1.S<1,2,...,m> ||| Cola2.S<1,1,...,1>) // ( (Init<>; Transfer5 ) || Timer ) )
) \ {√, tick, Timer.left, Timer.right }

```

<Dado que el número de eventos que están prefijando al proceso se va a incrementar conforme apliquemos leyes y definiciones, entonces debemos buscar una manera de representar al sistema, de tal forma que sea fácil su lectura, por lo cual sustituimos los primeros eventos por el proceso **Inicializa** el cual ejecuta exactamente los mismos eventos, esto es:

```

Inicializa = Cola1.left.1 → Cola2.left.1 → Cola1.left.2 →
  Cola2.left.1 → . . . Cola1.left.m → Cola2.left.1 → SKIP

```

Dado que un proceso no puede prefijar otro, pues sintácticamente es incorrecto, los compondremos secuencialmente, además el evento *SKIP* no causa problemas pues se oculta al final del proceso.

Realizando lo anterior y sustituyendo *Init*<*y*> por su definición, tenemos: >

$$= (\mathbf{Inicializa} ; ((Cola1.S_{<1,2,\dots,m>} \parallel Cola2.S_{<1,1,\dots,1>}) // (SKIP ; Transfer5) \parallel Timer)) \setminus \{ \surd, tick, Timer.left, Timer.right \}$$

<dado que *SKIP* es la terminación satisfactoria, tenemos: >

$$= (\mathbf{Inicializa} ; ((Cola1.S_{<1,2,\dots,m>} \parallel Cola2.S_{<1,1,\dots,1>}) // (Transfer5 \parallel Timer))) \setminus \{ \surd, tick, Timer.left, Timer.right \} \quad \dots (1)$$

<sustituimos *Cola1.S*<*1,2,...,m*>, *Cola2.S*<*1,1,...,1*>, *Transfer5* y *Timer* por sus definiciones >

$$= (\mathbf{Inicializa} ; ((((Cola1.left?y \rightarrow \text{if } \#<I>^{\wedge}<2,\dots,m>^{\wedge}<y> < n \text{ then } Cola1.S_{<1>^{\wedge}<2,\dots,m>^{\wedge}<y>} \text{ else } (Cola1.right!1 \rightarrow Cola1.S_{<2,\dots,m>^{\wedge}<y>}))) \square (Cola1.right!1 \rightarrow Cola1.S_{<2,\dots,m>})) \parallel ((Cola2.left?y \rightarrow \text{if } \#<I>^{\wedge}<1,\dots,1>^{\wedge}<y> < n \text{ then } Cola2.S_{<1>^{\wedge}<1,\dots,1>^{\wedge}<y>} \text{ else } (Cola2.right!1 \rightarrow Cola2.S_{<1,\dots,1>^{\wedge}<y>}))) \square (Cola2.right!1 \rightarrow Cola2.S_{<1,\dots,1>})) \parallel ((Cola1.right?j \rightarrow Cola2.right?i \rightarrow Cola1.left!j \rightarrow Cola2.left!(i+1) \rightarrow Timer.left!1 \rightarrow C_{j,i} \forall Termina)) \parallel (Timer))) \setminus \{ \surd, tick, Timer.left, Timer.right \}$$

<Nótese que el tamaño de la sublista <*I*,...,*I*>, la cual aparecerá nuevamente, dependerá del contexto, pero jamás podrá exceder en tamaño al valor *m*.

Aplicamos la ley A.2.2. | |.4 sobre el operador | |, la ley A.3. | | |.1 sobre el operador ||| y la ley A.10.comm.1 en las comunicaciones, en cada uno de los eventos subrayados >

```
= ( Inicializa ;
  ( Cola1.right!1 → Cola2.right!1 →
    ( ( Cola1.S<2,...,m> ||| Cola2.S<1,...,l> )
      //
      ( ( Cola1.left!1 → Cola2.left!2 → Timer.left!1 → Cj,l ∇ Termina ) | | ( Timer ) )
    )
  )
) \ { √, tick, Timer.left, Timer.right }
```

<substituimos Cola1.S<2,...,m> y Cola2.S<1,...,l> por sus definiciones >

```
= ( Inicializa ;
  ( Cola1.right!1 → Cola2.right!1 →
    ( ( ( Cola1.left?y → if #<2>^<3,...,m>^<y> < n then Cola1.S<2>^<3,...,m>^<y>
      else ( Cola1.right!2 → Cola1.S<3,...,m>^<y> )
    )
    □
    (Cola1.right!2 → Cola1.S<3,...,m>)
  )
  |||
  ( ( Cola2.left?y → if #<1>^<1,...,l>^<y> < n then Cola2.S<1,...,l>^<y>
    else ( Cola2.right!1 → Cola2.S<1,...,l>^<y> )
  )
  □
  (Cola2.right!1 → Cola2.S<1,...,l>)
  )
  )
  //
  ( ( Cola1.left!1 → Cola2.left!2 → Timer.left!1 → Cj,l ∇ Termina )
    | |
    ( Timer.left?i → ticki )
  )
  )
) \ { √, tick, Timer.left, Timer.right }
```

<aplicamos la ley A.2.2. | |.4 sobre el operador | |, la ley A.3. | | |.1 sobre el operador ||| y la ley A.10.comm.1 en las comunicaciones, en cada uno de los eventos subrayados>

$$\begin{aligned}
&= (\mathbf{Inicializa} ; \\
&\quad (Cola1.right.1 \rightarrow Cola2.right.1 \rightarrow Cola1.left.1 \rightarrow Cola2.left.2 \rightarrow Timer.left.1 \\
&\quad\quad (Cola1.S_{<2,3,\dots,m,1>} ||| Cola2.S_{<1,\dots,1,2>}) \\
&\quad\quad // \\
&\quad\quad ((C_{1,1} \nabla Termina) || tick_1) \\
&\quad) \\
&) \setminus \{ \checkmark, tick, Timer.left, Timer.right \}
\end{aligned}$$

<substituimos los cinco eventos localizados después de *Inicializa*, por el proceso *Sched(1,1)*, en donde *Sched* está expresado por:

$$\mathbf{Sched}(i,j) = Cola1.right.i \rightarrow Cola2.right.j \rightarrow Cola1.left.i \rightarrow Cola2.left.j+1 \rightarrow Timer.left.1 \rightarrow SKIP$$

y substituyendo $C_{1,1}$ por su definición, tenemos: >

$$\begin{aligned}
&= (\mathbf{Inicializa} ; \mathbf{Sched}(1,1) ; \\
&\quad ((Cola1.S_{<2,3,\dots,m,1>} ||| Cola2.S_{<1,\dots,1,2>}) \\
&\quad\quad // \\
&\quad\quad ((Corrutina5_{1,1} \nabla Termina) || tick_1) \\
&\quad) \\
&) \setminus \{ \checkmark, tick, Timer.left, Timer.right \}
\end{aligned}$$

<substituyendo *Corrutina5_{1,1}*, *Termina* y *tick₁*, tenemos: >

$$\begin{aligned}
&= (\mathbf{Inicializa} ; \mathbf{Sched}(1,1) ; \\
&\quad (\\
&\quad\quad (Cola1.S_{<2,3,\dots,m,1>} ||| Cola2.S_{<1,\dots,1,2>}) \\
&\quad\quad // \\
&\quad\quad ((\underline{m}_{1,1} \rightarrow tick \rightarrow tick \rightarrow Corrutina5_{1,2}) \\
&\quad\quad\quad \nabla \\
&\quad\quad\quad (Timer.right?x \rightarrow Transfer5) \\
&\quad\quad) \\
&\quad\quad || \\
&\quad\quad (if I>0 then (tick \rightarrow tick_0) \\
&\quad\quad\quad else Timer.right!0 \rightarrow Timer \\
&\quad\quad) \\
&\quad) \\
&) \setminus \{ \checkmark, tick, Timer.left, Timer.right \}
\end{aligned}$$

<en este paso sólo se puede ejecutar $m_{1,1}$, dado que *tick* pertenece al alfabeto del *Timer* y al alfabeto de la corrutina, entonces debe ejecutarse de manera sincrona, por otro lado los

eventos de los demás procesos no están disponibles. Aplicando la ley A.2.2. | |.3.a, tenemos>

```
= ( Inicializa ; Sched(1,1) ;
  ( m1,1 →
    ( (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
      //
      ( ( ( tick → tick →Corrutina51,2)
          ∇
          (Timer.right?x → Transfer5)
        )
        ||
        ( tick → tick0 )
      )
    )
  ) \ { √, tick, Timer.left, Timer.right }
```

<sustituimos $m_{1,1}$ por el proceso $Corrut_{1,1}$ el cual es dado por la siguiente expresión

$Corrut_{i,j} = m_{i,j} \rightarrow SKIP$

y aplicamos la ley A.2.2. | |.2 sobre el evento $tick$ >

```
= ( Inicializa ; Sched(1,1) ; Corrut1,1 ;
  (
    (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
    //
    ( tick →( ( ( tick →Corrutina51,2)
                ∇
                (Timer.right?x → Transfer5)
              )
              ||
              ( tick0 )
            )
    )
  ) \ { √, tick, Timer.left, Timer.right }
```

<un nuevo evento $tick$ no se puede dar, ya que no está disponible en el proceso $Timer$, por lo que se aplica la ley A.2.2. | |.3.b sobre el evento $tick$ y se sustituye $tick_0$ por su definición >

```

= ( Inicializa ; Sched(1,1) ; Corrut1,1 ;
  ( tick →
    ( (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
      //
      ( ( tick →Corrutina51,2)
        ▽
        (Timer.right?x → Transfer5 )
      )
      ||
      ( if 0>0 then (tick → tick1)
        else Timer.right!0 → Timer
      )
    )
  )
) \ {√, tick, Timer.left, Timer.right }

```

<en este proceso los únicos eventos que pueden ofrecerse son los que están subrayados, justamente el evento Timer.right!0 es el que interrumpe la rutina de usuario. Aplicamos la ley A.10.comm.1 y la definición de interrupción >

```

= ( Inicializa ; Sched(1,1) ; Corrut1,1 ;
  ( tick →
    ( (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
      //
      (Timer.right!0 → (Transfer5 || Timer) )
    )
  )
) \ {√, tick, Timer.left, Timer.right }

```

<aplicando la ley A.2.2. | .3.b nuevamente, tenemos: >

```

= ( Inicializa ; Sched(1,1) ; Corrut1,1 ;
  ( tick → Timer.right!0 →
    ( (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
      //
      ( (Transfer5 || Timer) )
    )
  )
) \ {√, tick, Timer.left, Timer.right }

```

< sustituimos la secuencia de eventos *tick* → *Timer.right!0*, por el proceso ***Interrup***, el cual esta dado por:

Interrup = *tick* → *Timer.right!0* → *SKIP*

con lo cual tenemos: >

```
= (Inicializa ; Sched(1,1) ; Corrut1,1 ; Interrup ;
  ( (Cola1.S<2,3,...,m,1> ||| Cola2.S<1,...,1,2>)
    //
    ( (Transfer5 || Timer) )
  )
) \ { √, tick, Timer.left, Timer.right }
```

< Aplicando de manera similar a esta expresión, los pasos aplicados a la expresión (1) hasta este punto, tenemos: >

```
= (Inicializa ; Sched(1,1) ; Corrut1,1 ; Interrup ;
      Sched(2,1) ; Corrut2,1 ; Interrup ;
  ( (Cola1.S<3,...,m,1,2> ||| Cola2.S<1,...,1,2,2>)
    //
    ( (Transfer5 || Timer) )
  )
) \ { √, tick, Timer.left, Timer.right }
```

En este punto nos detenemos, para observar y analizar el conjunto de procesos que ejecuta al inicio el núcleo de nuestro sistema operativo. Si seguimos aplicando leyes y definiciones a nuestro sistema para continuar “expandiéndolo”, los procesos comenzaran a repetirse nuevamente. En este punto vemos que a excepción del proceso *Inicializa*, todos los demás procesos volvieron a aparecer, claro sólo cambiaron los argumentos de aquellos procesos que los requieren. Por lo cual no desarrollaremos más el sistema.

Definamos al último proceso obtenido de *Núcleo5* como el proceso *Implantación*. Para asegurarnos de que el proceso *Implantación* es un refinamiento correcto de *Núcleo5*, o en otras palabras, para asegurarnos de que el conjunto de leyes aplicadas a *Núcleo5* fueron aplicadas correctamente, probemos mediante FDR que el proceso *Implantación* refina al proceso *Núcleo5*. Para ello generamos el siguiente script FDR y probamos su corrección.

Resumen

En este trabajo presentamos, a través de un caso de estudio, el desarrollo de sistemas mediante métodos formales. El caso de estudio es el núcleo básico de un sistema operativo multitareas y el método formal utilizado es la teoría de Procesos Secuenciales Comunicantes (CSP).

Comenzamos este trabajo dando una muy breve explicación de lo que son los métodos formales en general, para después presentar de manera particular el método formal de CSP, así como un demostrador automático de refinamientos CSP, denominado FDR (Refinamientos Falla-Divergencia). Posteriormente desarrollamos el sistema seleccionado mediante CSP.

Comenzamos el desarrollo de nuestro caso de estudio con la especificación inicial del núcleo básico del sistema operativo, en donde sólo expresamos lo que queremos realizar sin dar detalles de implantación. Los detalles de implantación los damos de manera gradual en lo se denominan *pasos de refinamiento*, en cada uno de ellos refinamos una propiedad del sistema. Cada refinamiento debe conservar las propiedades esenciales de la especificación inicial, para asegurarnos de ello realizamos pruebas formales, por un lado usando el álgebra de procesos CSP y por otro lado usando FDR. Una vez que obtuvimos un refinamiento con los suficientes detalles de implantación derivamos un programa en pseudocódigo.

Terminamos la exposición de este trabajo con nuestras conclusiones y un par de apéndices. En el primer apéndice mostramos las leyes CSP utilizadas en el texto, en el segundo apéndice damos un listado de la especificación inicial y de los refinamientos del núcleo básico del sistemas operativo.

Palabras clave:

Método formales, CSP (Procesos Secuenciales Comunicantes), FDR (Refinamientos Falla-Divergencia), pasos de refinamiento, lenguaje de especificación formal, sistema de inferencia, proceso, evento, paralelismo, concurrencia, determinismo, no determinismo, sistemas críticos, corrección.

channel Timer_Right

-- Limitamos el número de procesos a 5.

Inicializa = c1left!1 -> c2left!1 -> c1left!2 -> c2left!1 -> c1left!3 -> c2left!1 ->
 c1left!4 -> c2left!1 -> c1left!5 -> c2left!1 -> SKIP

Sched = c1right?i -> c2right?j -> c1left!i -> c2left!suma1(j) ->SKIP

Corrut(i,j) = m.i.j -> SKIP

Interrup = tick -> Timer_Right -> SKIP

Implantacion = ((Cola1(<>) || Cola2(<>))
 [{} c1left, c2left, c1right, c2right {}])
 (Inicializa ; Sched ; Corrut(1,1); Interrup ;
 Sched ; Corrut(2,1); Interrup ;
 Sched ; Corrut(3,1); Interrup ;
 Sched ; Corrut(4,1); Interrup ;
 Sched ; Corrut(5,1); Interrup ;
 Sched ; Corrut(1,2); Interrup ;
 Sched ; Corrut(2,2); Interrup ;
 Sched ; Corrut(3,2); Interrup ;
 Sched ; Corrut(4,2); Interrup ;
 Sched ; Corrut(5,2); Interrup ;
 Sched ; Corrut(1,3); Interrup ;
 Sched ; Corrut(2,3); Interrup ;
 Sched ; Corrut(3,3); Interrup ;
 Sched ; Corrut(4,3); Interrup ;
 Sched ; Corrut(5,3); Interrup ;
) \ {} tick, c1left, c2left, c1right, c2right, Timer_Right, m.3, m.4, m.5 {}

4.4.1 Subprocesos del núcleo básico del sistema operativo.

Cada subproceso del núcleo es un proceso totalmente determinista y secuencial, y sólo utiliza el operador de prefijo, por lo cual su implantación es relativamente sencilla. Antes de pasar a la implantación del núcleo, veamos que es lo que ejecuta cada uno de los subprocesos del proceso núcleo.

Inicializa : inicializa la *Cola1* con los identificadores de los programas de usuario a ejecutar y la *Cola2* con sus contadores de programa.

Sched(i,j) : este proceso tiene la función de un *---scheduler---*, esto es, conmuta los procesos de usuario a ejecutarse dentro del procesador. Extrae del frente de las listas asociadas a los procesos *Cola1* y *Cola2*, el identificador de la rutina a ejecutar y su contador de programa respectivamente, posteriormente inserta al final de las mismas listas el identificador y contador de programa, este último incrementado en uno. Además inicializa el reloj del sistema, el cual se ejecutará paralelamente con el proceso de usuario, después de cierto tiempo el reloj del sistema activa la interrupción encargada de invocar nuevamente a este proceso.

Corrut_{i,j} : ejecuta la acción *j* del proceso *i* de usuario, estas acciones pueden ser vistas a cualquier nivel de abstracción.

Interrup : ejecuta la acción de revisar si la rebanada de tiempo se ha terminado (*tick*), si es así, “interrumpe” la rutina de usuario para que se realice la conmutación de procesos dentro del procesador.

4.4.2 Comportamiento del núcleo básico del sistema operativo.

Después de haber visto lo que hace cada uno de los subprocesos del proceso *Núcleo5* refinado, veamos ahora su comportamiento completo.

1. Comienza por inicializar las colas con los programas de usuario, al final de este proceso las colas quedan: $Cola1 = \langle 1, 2, \dots, m \rangle$ y $Cola2 = \langle 1, 1, \dots, 1 \rangle$
2. Extrae el identificador del programa de usuario del frente de la *Cola1*, y su contador de programa del frente de la *Cola2*, inserta nuevamente al final de la *Cola1* el identificador de programa y al final de la *Cola2* el contador de programa modificado. Inicializa el *Reloj* del sistema e invoca al programa de usuario cuyo identificador y contador de programa obtuvo anteriormente.
3. Ejecuta el programa de usuario hasta que una interrupción ocurre.
4. Al ocurrir la interrupción se ejecuta nuevamente el paso 2.

Dado que la implantación del sistema está dado por el proceso

Implantación = *Inicializa* ; *Sched(1,1)* ; *Corrut_{1,1}* ; *Interrup* ;
Sched(2,1) ; *Corrut_{2,1}* ; *Interrup* ;

describiremos como implantar en pseudocódigo cada uno de sus subprocesos, esto lo realizaremos de manera semiformal, mapeando cada evento a instrucciones en pseudocódigo.

4.4.3 Mapeo de los subprocesos a pseudocódigo.

Para el mapeo de eventos CSP a instrucciones en pseudocódigo asumiremos de que disponemos de las operaciones del Tipo de Dato Abstracto Colas (TDA Colas), lo cual no resulta grave, pues estas operaciones son sencillas y fáciles de implantar. A continuación describimos las operaciones del TDA colas.

HazNula(Cola) : procedimiento que vacía la cola.

ValorFrente(Cola) : función que devuelve el valor del frente de la Cola sin alterar ésta.

Encola(elem, Cola) : procedimiento que inserta al final de la Cola el elemento elem.

Desencola(Cola) : función que retira el elemento del frente de la Cola, regresándolo a quien invocó la función.

Vacia(Cola) : función que devuelve cierto si la Cola está vacía, falso en caso contrario.

Utilizando las definiciones anteriores realizamos el mapeo de cada uno de los subprocesos del proceso *Implantación*.

Inicializa

Comencemos con el mapeo del proceso *Inicializa*, el cual esta definido por:

Inicializa = *Cola1.left.1* → *Cola2.left.1* → *Cola1.left.2* → *Cola2.left.1* → . . . →
Cola1.left.m → *Cola2.left.1* → *SKIP*

este proceso es mapeado a las siguientes instrucciones en pseudocódigo

Encola(1,Cola1)
Encola(1,Cola2)
Encola(2,Cola1)
Encola(1,Cola2)

Encola(m,Cola1)
Encola(1,Cola2)

en donde Encola es una función del TDA colas.

Sched(i,j)

Este proceso esta dado por la siguiente expresión:

$$Sched(i,j) = Cola1.right.i \rightarrow Cola2.right.j \rightarrow Cola1.left.i \rightarrow Cola2.left.j+1 \rightarrow \\ Timer.left.1 \rightarrow SKIP$$

el cual es mapeado a la siguiente serie de instrucciones en pseudocódigo.

$i := Desencola(Cola1)$
 $j := Desencola(Cola2)$
Encola(Cola1,i)
Encola(Cola2,j+1)
InicializaReloj(1)

En donde la instrucción InicializaReloj se encargará de activar el reloj del sistema, el cual viene en cualquier tipo de computador, para que después de cierto tiempo interrumpa la ejecución del procesador y ejecute una rutina especial, la cual es la de suspender el programa de usuario que se ejecuta y realizar el cambio de contexto en las pilas.

Corrut_{i,j}

Este proceso modela, a cualquier nivel de abstracción, las instrucciones de algún programa de usuario, por lo que sus instrucciones dependerán del programa en particular a ejecutar. Con fines ilustrativos sustituiremos las instrucciones de usuario por mensajes escritos a pantalla. Dado que $Corrut_{i,j}$ fue definido por:

$$Corrut_{i,j} = m_{i,j} \rightarrow SKIP$$

implantaremos el proceso por

Escribe("Corrutina i mensaje j ")

en donde i y j dependerán de la corrutina que se ejecute, así como el número de instrucción que ejecute.

Interrup

Una vez activada la rutina de atención a la interrupción se ejecuta el proceso **Sched(i,j)**

Una vez que hemos mapeado los eventos CSP a pseudocódigo de cada uno de los subprocesos, veamos el pseudocódigo completo del sistema

1.	Encola(1,Cola1)	}	Se inician los n programas de usuario a ejecutarse
2.	Encola(1,Cola2)		
3.	Encola(2,Cola1)		
4.	Encola(1,Cola2)		
5.	Encola(m ,Cola1)	}	
6.	Encola(1,Cola2)		
7.	$i :=$ Desencola(Cola1)	}	Se despacha el proceso que está al frente de la cola1, realizándose el cambio de contexto dentro del procesador.
8.	$j :=$ Desencola(Cola2)		
9.	Encola(Cola1, i)		
10.	Encola(Cola2, $j+1$)		
11.	Escribe("Corrutina i , mensaje j ")	}	Ejecución del programa de usuario
12.	Interrup	}	Ocurre una interrupción
16.	Ejecuta nuevamente a partir de 7		

En este pseudocódigo vemos claramente como n programas de usuario se ejecutan de manera concurrente dentro de un solo procesador. Este mapeo incluso se puede realizar de manera formal a código fuente en un lenguaje de programación como Ada, haciendo uso de la

teoría de procesos receptivos (el cual es una extensión a CSP) [Hinchey93 y Hinchey94], pero por fines de tiempo hemos decidido presentar el trabajo hasta este punto.

Cabe hacer mención que las instrucciones de los programas de usuario se ilustran con una sola impresión de mensaje, pero esta puede ser desde una instrucción hasta la ejecución de un conjunto de instrucciones de algún lenguaje fuente de alto nivel o una subrutina.

Una vez mapeado nuestro último refinamiento a pseudocódigo, pasemos a las conclusiones del presente trabajo.

Conclusiones

En este punto haremos un recuento de lo presentado. Hemos descrito de manera somera las bases de los métodos formales y de manera detallada el método formal CSP y la herramienta automatizada FDR2, la cual nos ayudó a probar los refinamientos de nuestro sistema. Describimos la especificación inicial del núcleo básico de nuestro sistema operativo. Mediante pasos de refinamiento llevamos esta especificación a una en donde tuviéramos detalles suficientes para su implantación. En cada paso de refinamiento motivamos el refinamiento, mostramos su especificación, probamos de manera manual el refinamiento (sólo los primeros tres refinamientos) y después corroboramos nuestro resultado con el probador de refinamientos de CSP, FDR2.

Una vez que llegamos al quinto refinamiento, el cual no probamos de manera manual debido a su alto grado de complejidad, procedimos a aplicarle a este refinamiento leyes CSP, de tal forma que fuéramos obteniendo el conjunto de eventos a ejecutar sin más operadores que el de prefijo. Una vez que obtuvimos el proceso en base a subprocesos cuyo único operador es el de prefijo, procedimos a su implantación en pseudocódigo. La implantación de cualquier otro sistema puede no seguir estos últimos pasos (entiéndase estos últimos pasos como aquellos que realizamos después del quinto refinamiento). Los pasos que se pueden seguir para cualquier otro sistema dependerán del sistema a implantar en sí y de la plataforma sobre la cual será implantado, existen trabajos que nos muestran como implantar un sistema bajo lenguajes de programación como *Ada* [Hinchey93 y Hinchey94] o al lenguaje de programación de Transputers *ocamm* [Hinchey94].

El objetivo de la presente tesis es el estudiar y aplicar métodos formales. Una vez que hemos desarrollado el núcleo básico de un sistema operativo multitarea, desde su especificación inicial hasta una especificación en donde tuvimos suficientes detalles de implantación, todo esto de manera formal, y después de haber mapeado este último refinamiento a pseudocódigo de manera semiformal, creemos haber logrado nuestro objetivo. Esperamos que este documento sirva para motivar el uso de los métodos formales en el desarrollo de sistemas, sobre todo en el desarrollo de sistemas críticos.

El sistema aquí desarrollado, es un ejemplo de un programa que continuamente interactúa con su medio ambiente, de manera similar a los programas de control y comunicaciones en aplicaciones industriales. Estos programas han sido objeto de una gran

investigación para asegurar la ausencia de errores, y así evitar los enormes costos que éstos ocasionan. El resultado expuesto en esta tesis, es sólo una muestra de la ayuda que prestan los métodos formales en la construcción razonada y metódica de programas.

Un inconveniente de CSP y de los métodos formales en general, es su costo cuando se aplican por primera vez [Hinchey95 b]. Hay una curva de aprendizaje muy costosa, pues su uso efectivo sólo se lleva a cabo después de un monto considerable de: tiempo de entrenamiento, investigación bibliográfica y del uso de las herramientas de soporte disponibles. Justamente la parte de investigación del método formal y su herramienta de soporte fue lo que consumió la mayor parte del tiempo de lo dedicado al presente trabajo.

Otro inconveniente de los métodos formales es su costo cuando se usa de manera intensiva. Para sistemas demasiado grandes, es conveniente desarrollar parte del sistema, la parte crítica, aplicando métodos formales, y la otra parte mediante métodos estructurados, por ejemplo. Es deseable integrar los métodos formales en el proceso de desarrollo de una manera efectiva en costo. Una forma de realizarlo es investigando cómo un método formal puede ser combinado efectivamente con un método estructurado ya usado en la industria, un ejemplo de esto es reportado en [Mander95]. Claro, los métodos formales y los estructurados tienen sus fortalezas y debilidades e idealmente combinados ofrecen los beneficios de ambos.

Otro punto a considerar es que un método formal es un medio para obtener algo, y no un fin mismo. El usar métodos formales por puro capricho puede no ser provechoso en algunos casos. Por lo cual debe de hacerse un estudio de cuál método formal es el más adecuado para el sistema a desarrollar, se debe investigar si existen herramientas de soporte para este método, si se tiene el personal preparado en esta área, se debe de estimar el tiempo de desarrollo para ver si coincide con la fecha de entrega del sistema, etc.

En el desarrollo del presente trabajo observamos que los métodos formales no son una panacea, pues son un conjunto de técnicas que cuando son aplicadas correctamente generan resultados de la más alta confiabilidad.

Por otro lado, observamos que hay evidencias considerables de que los proyectos que utilizan métodos formales pueden ser tan económicos, en el costo de desarrollo (o más), como los proyectos que usan métodos convencionales de desarrollo, como los estructurados [Hinchey95 b]. Algunos proyectos industriales cuyo costo de desarrollo fue menor usando métodos formales son por ejemplo: el desarrollo de la unidad de punto flotante de el transputer T800 fabricado por Inmos, cuyo microcódigo fue formalmente desarrollado usando CSP. Otro ejemplo es el 9% de ahorro en el costo de desarrollo del sistema de procesamiento de transacciones de un procesador CICS de IBM usando Z, entre otros. La reducción en el tiempo y costo de los proyectos que usan métodos formales se debe a que una vez aplicado un método formal, la siguiente vez que se aplique ya no se invierte tiempo en aprenderlo, si no que se pasa directamente al desarrollo del sistema.

Es por ello que el autor de la presente tesis se siente muy motivado por continuar con la investigación y uso de métodos formales. Ya se utilizó el método formal CSP, para el sistema

que se presentó, ahora el interés se centra en utilizarlo en un sistema grande de aplicación real. Se pretende explorar el desarrollo de alguna parte de un sistema operativo distribuido mediante CSP.

En el desarrollo de la presente tesis nos encontramos con algunos artículos sobre la teoría y uso de CSP, pero ninguno contenía pruebas de refinamientos, lo más que logramos obtener fueron descripciones someras de como realizar las pruebas, fue así como empezamos a probar nuestros refinamientos. Una de las principales contribuciones de la presente tesis es que mostramos como probar refinamientos utilizando sólo la teoría CSP, por un lado, y por otro mostramos como probar refinamientos mediante la herramienta automatizada FDR2. Lo anterior es lo que hace del presente documento una fuente invaluable para aquellas personas que deseen desarrollar sistemas usando CSP, y por otro lado se convierte en una fuente pionera en el desarrollo de sistemas usando CSP, al mostrar todos los detalles de las pruebas de refinamiento.

Al desarrollar el núcleo del sistema operativo observamos que a partir del último refinamiento, se puede derivar de manera formal un programa escrito en los lenguajes de programación *Ada* u *ocamm*. Pero si nosotros queremos derivar el programa en algún otro lenguaje de programación, esta derivación formal no existe, por lo que tendremos que hacerla de manera semiformal. Es aquí donde creemos que deben centrarse las investigaciones sobre CSP para hacer un método formal más completo, y pueda llevar de manera formal el desarrollo de un sistema desde su especificación hasta su implantación. Esta es una de las posibles líneas de investigación que se pueden abrir después del desarrollo de la presente tesis.

Apéndice I. Leyes CSP.

A1. Introducción

En este apéndice listamos las principales leyes que pueden ser usadas para el razonamiento de sistemas de procesos CSP. El apéndice intenta actuar como referencia para los lectores de este documento mas que para ser leído de manera completa. Las leyes listadas pueden ser usadas para probar la equivalencia semántica de procesos, para eliminar operadores de expresiones de procesos y para sistemáticamente reducir el número de procesos.

A.2. Composición Paralela

A.2.1 Alfabetos iguales

Operador: $||$ con alfabetos iguales.

Descripción: Denota procesos ejecutándose en paralelo con alfabetos iguales, obligándolos a sincronizarse en todos los eventos.

Leyes:

$||$.1 Deadlock

$$(e_1 \rightarrow P) || (e_2 \rightarrow Q) = \text{STOP}$$

$||$.2 Sincronización

$$(e_1 \rightarrow P) || (e_1 \rightarrow Q) = e_1 \rightarrow (P || Q)$$

$||$.3 Alfabetos iguales

$$\alpha (P || Q) = \alpha P = \alpha Q$$

$||$.4 Idempotencia

$$P || P = P$$

$||$.5 Conmutatividad

$$P || Q = Q || P$$

||.6 Asociatividad

$$P \mid (Q \mid R) = (P \mid Q) \mid R$$

||.7 Transitividad

$$(P \mid Q) \wedge (Q \mid R) \Rightarrow (P \mid R)$$

||.8 Elemento identidad

- a) $P \mid \text{RUN}_{\alpha P} = P$
- b) $P \mid \text{SKIP} = P$

||.9 Exactitud

$$P \mid \perp = \perp$$

||.10 Distribución sobre el operador Π

- a) $P \mid (Q \Pi R) = (P \mid Q) \Pi (P \mid R)$
- b) $(P \Pi Q) \mid R = (P \mid R) \Pi (Q \mid R)$

||.11 Distribución sobre el operador \square

- a) $P \mid (Q \square R) = (P \mid Q) \square (P \mid R)$
- b) $(P \square Q) \mid R = (P \mid R) \square (Q \mid R)$

||.12 Renombramiento

$$f(P \mid Q) = f(P) \mid f(Q)$$

||.13 After

$$(P \mid Q) \text{ after } t = (P \text{ after } t) \mid (Q \text{ after } t)$$

||.14 Etiquetamiento

$$l : (P \mid Q) = (l : P) \mid (l : Q)$$

||.15 Ocultamiento

$$(P \mid Q) \setminus C = (P \setminus C) \mid (Q \setminus C) \Leftrightarrow \alpha P \cap C = \alpha Q \cap C = \{ \}$$

||.16 STOP

$$P \mid \text{STOP}_{\alpha P} = \text{STOP}_{\alpha P}$$

||.17 Composición secuencial.

$$(P ; Q) || R \neq (P || R) ; (Q || R)$$

||.18 Trazas

$$\text{Trazas}(P || Q) = \text{Trazas}(P) \cap \text{Trazas}(Q)$$

||.19 Divergencias

$$\begin{aligned} \mathbf{d}(P || Q) = \{ & t_1 \wedge t_2 \mid t_2 \in \mathbf{seq}(\infty P) \\ & \wedge (t_1 \in \mathbf{dP} \wedge t_1 \in \text{Trazas}(Q) \\ & \vee t_1 \in \text{Trazas}(P) \wedge t_1 \in \mathbf{dP}) \} \end{aligned}$$

||.20 Fallas

$$\begin{aligned} \mathbf{f}(P || Q) = \{ & (t_1, A \cup B) \mid t_1 \in \mathbf{seq}(\infty P) \\ & \wedge (t_1, A) \in \mathbf{fP} \\ & \wedge (t_1, B) \in \mathbf{fQ} \} \\ & \cup \{ (t_2, D) \mid t_2 \in \mathbf{d}(P || Q) \} \end{aligned}$$

||.21 Rechazos

$$\mathbf{r}(P || Q) = \{ A \cup B \mid A \in \mathbf{rP} \wedge B \in \mathbf{rQ} \}$$

A.2.2 Alfabetos diferentes

Operador: || con alfabetos diferentes

Descripción: Ejecución paralela de los procesos, en donde los eventos en común a éstos, se ejecutarán de manera síncrona, los demás se ejecutarán en cualquier orden.

Leyes:

$$\begin{aligned} \text{sea: } e_1 & \in (\infty P - \infty Q) \\ e_2 & \in (\infty Q - \infty P) \\ e_3, e_4 & \in (\infty P \cap \infty Q) \end{aligned}$$

||.1 Deadlock

$$(e_3 \rightarrow P) || (e_4 \rightarrow Q) = \text{STOP}$$

||.2 Sincronización

$$(e_3 \rightarrow P) || (e_3 \rightarrow Q) = e_3 \rightarrow (P || Q)$$

||.3 El evento independiente ocurre primero

a. $(e_1 \rightarrow P) || (e_3 \rightarrow Q) = e_1 \rightarrow (P || e_3 \rightarrow Q)$

b. $(e_3 \rightarrow P) || (e_2 \rightarrow Q) = e_2 \rightarrow (e_3 \rightarrow P || Q)$

||.4 Entrelazamiento

$$(e_1 \rightarrow P) || (e_2 \rightarrow Q) = (e_1 \rightarrow (P || e_2 \rightarrow Q) | e_2 \rightarrow (e_1 \rightarrow P || Q))$$

||.5 Unión de alfabetos

$$\alpha(P || Q) = \alpha P \cup \alpha Q$$

||.6 Idempotencia

$$P || P = P$$

||.7 Conmutatividad

$$P || Q = Q || P$$

||.8 Asociatividad

$$P || (Q || R) = (P || Q) || R$$

||.9 Transitividad

$$(P || Q) \wedge (Q || R) \Rightarrow (P || R)$$

||.10 Elemento identidad

a. $P || \text{Run}_{\alpha P} = P$

b. $P || \text{SKIP} = P$

||.11 Exactitud

$$P || \perp = \perp$$

||.12 Distribución sobre el operador Π

a. $P || (Q \Pi R) = (P || Q) \Pi (P || R)$

b. $(P \Pi Q) || R = (P || R) \Pi (Q || R)$

||.13 Distribución sobre el operador \square

- a. $P \parallel (Q \square R) = (P \parallel Q) \square (P \parallel R)$
- b. $(P \square Q) \parallel R = (P \parallel R) \square (Q \parallel R)$

||.14 Renombramiento

$$f(P \parallel Q) = f(P) \parallel f(Q)$$

||.15 Después (**after**)

$$(P \parallel Q) \text{ after } t = (P \text{ after } t \uparrow \alpha P) \parallel (Q \text{ after } t \uparrow \alpha Q)$$

||.16 Etiquetamiento

$$l : (P \parallel Q) = (l : P) \parallel (l : Q)$$

||.17 Ocultamiento

$$\begin{aligned} (P \parallel Q) \setminus C &= (P \setminus C) \parallel (Q \setminus C) \\ &\Leftrightarrow \alpha P \cap \alpha Q \cap C = \{\} \end{aligned}$$

||.18 Alto (Stop)

- a. $P \parallel \text{STOP}_{\alpha P} = \text{STOP}_{\alpha P}$
- b. $P \parallel \text{STOP}_{\alpha R} = P \Leftrightarrow \alpha R \cap \alpha P = \{\}$
 $= \text{STOP}_{\alpha R} \Leftrightarrow \alpha P \subseteq \alpha R$
 $= \text{STOP}_{\alpha P \cap \alpha R} \Leftrightarrow \alpha R \cap \alpha P \neq \{\}$

||.19 Composición secuencial

$$(P ; Q) \parallel R \neq (P \parallel R) ; (Q \parallel R)$$

||.20 Trazas

$$\begin{aligned} \text{Trazas}(P \parallel Q) &= \{ t \mid t \in \text{seq}(\alpha P \cup \alpha Q) \\ &\quad \wedge (t \uparrow \alpha P) \in \text{Trazas}(P) \\ &\quad \wedge (t \uparrow \alpha Q) \in \text{Trazas}(Q) \} \end{aligned}$$

||.21 Divergencias

$$\begin{aligned} \mathbf{d}(P \parallel Q) &= \{ t_1 \wedge t_2 \mid t_2 \in \text{seq}(\alpha P \cup \alpha Q) \\ &\quad \wedge (t_1 \uparrow \alpha P \in \mathbf{d}P \wedge t_1 \uparrow \alpha Q \in \text{Trazas}(Q) \\ &\quad \vee t_1 \uparrow \alpha P \in \text{Trazas}(P) \wedge t_1 \uparrow \alpha Q \in \mathbf{d}Q) \} \end{aligned}$$

$$\begin{aligned} \mathbf{f}(P \parallel Q) = \{ & (t_1, A \cup B) \mid t_1 \in \mathbf{seq}(\alpha P \cup \alpha Q) \\ & \wedge (t_1 \uparrow \alpha P, A) \in \mathbf{f}P \\ & \wedge (t_1 \uparrow \alpha Q, B) \in \mathbf{f}Q \} \\ & \cup \{ (t_2, X) \mid t_2 \in \mathbf{d}(P \parallel Q) \} \end{aligned}$$

$$\mathbf{r}(P \parallel Q) = \{ A \cup B \mid A \in \mathbf{r}P \wedge B \in \mathbf{r}Q \}$$

A.3 Entrelazamiento

Operador: \parallel

Descripción: Ejecuta procesos de manera paralela e independiente sin importar sus eventos.

Leyes:

|||.1 Aplicación general

$$\begin{aligned} (e_1 \rightarrow P) \parallel (e_2 \rightarrow Q) = & (e_1 \rightarrow (P \parallel (e_2 \rightarrow Q)) \\ & \square (e_2 \rightarrow ((e_1 \rightarrow P) \parallel Q)) \end{aligned}$$

|||.2 Unión de alfabetos

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

|||.3 Idempotencia

$$P \parallel P \neq P$$

|||.4 Conmutatividad

$$P \parallel Q = Q \parallel P$$

|||.5 Asociatividad

$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$$

|||.6 Transitividad

$$(P \parallel Q) \wedge (Q \parallel R) \Rightarrow (P \parallel R)$$

|||.7 Elemento identidad

- a. $P \parallel \text{Run}_{\alpha} P = \text{Run}_{\alpha} P \Leftrightarrow \mathbf{dP} = \{\}$
- b. $P \parallel \text{SKIP} = P$

|||.8 Exactitud

$$P \parallel \perp = \perp$$

|||.9 Distribución sobre el operador Π

- a. $P \parallel (Q \Pi R) = (P \parallel Q) \Pi (P \parallel R)$
- b. $(P \Pi Q) \parallel R = (P \parallel R) \Pi (Q \parallel R)$

|||.13 No distribución sobre el operador \square

- a. $P \parallel (Q \square R) \neq (P \parallel Q) \square (P \parallel R)$
- b. $(P \square Q) \parallel R \neq (P \parallel R) \square (Q \parallel R)$

|||.11 Renombramiento

$$\hat{f}(P \parallel Q) = \hat{f}(P) \parallel \hat{f}(Q)$$

|||.12 Después (**after**)

$$(P \parallel Q) \text{ after } t = (P \text{ after } t_1) \parallel (Q \text{ after } t_2) \\ \Leftrightarrow t = \text{interleaving}(t_1, t_2)$$

|||.13 Etiquetamiento

$$l : (P \parallel Q) = (l : P) \parallel (l : Q)$$

|||.14 Ocultamiento

$$(P \parallel Q) \setminus C = (P \setminus C) \parallel (Q \setminus C) \\ \Leftrightarrow \alpha P \cap \alpha Q \cap C = \{\}$$

|||.15 Alto (Stop)

- a. $P \parallel \text{STOP}_{\alpha P} = P$
- b. $P \parallel \text{STOP}_{\alpha R} = P, \quad \alpha P \neq \alpha R$

|||.16 Composición secuencial

$$(P ; Q) \parallel R \neq (P \parallel Q) ; (Q \parallel R)$$

|||.17 Trazas

$$\text{Trazas}(P \parallel Q) = \{ \text{interleaving}(t_1, t_2) \mid (t_1 \in \text{Trazas}(P) \wedge t_2 \in \text{Trazas}(Q)) \}$$

|||.18 Divergencias

$$\mathbf{d}(P \parallel Q) = \{ \text{interleaving}(t_1, t_2) \mid (t_1 \in \mathbf{d}P \wedge t_2 \in \text{Trazas}(Q)) \cup (t_1 \in \text{Trazas}(P) \wedge t_2 \in \mathbf{d}Q) \}$$

|||.19 Fallas

$$\mathbf{f}(P \parallel Q) = \{ (\text{interleaving}(t_1, t_2), X) \mid (t_1, X) \in \mathbf{f}P \wedge (t_2, X) \in \mathbf{f}Q \} \cup \{ (t, X) \mid t \in \mathbf{d}(P \parallel Q) \}$$

|||.20 Rechazos (refusals)

$$\begin{aligned} \mathbf{r}(P \parallel Q) &= \mathbf{r}(P \square Q) \\ &= \mathbf{r}P \cap \mathbf{r}Q \end{aligned}$$

A.4. No Determinismo

A.4.1. Elección Interna

Operador: Π

Descripción: Denota el no determinismo “inesperado”, en el que $P \Pi Q$ puede comportarse como P o Q . No existe manera de saber cual elección se efectuará.

Leyes:

Π .1 Aplicación general

$$(e_1 \rightarrow P) \Pi (e_1 \rightarrow Q) = (e_1 \rightarrow (P \Pi Q))$$

Π .2 Alfabetos iguales

$$\alpha(P \Pi Q) = \alpha P = \alpha Q$$

Π .3 Idempotencia

$$P \Pi P = P$$

Π.4 Conmutatividad

$$P \Pi Q = Q \Pi P$$

Π.5 Asociatividad

$$P \Pi (Q \Pi R) = (P \Pi Q) \Pi R$$

Π.6 Exactitud

$$P \Pi \perp = \perp$$

Π.7 Distribución sobre el operador ||

$$\mathbf{a.} \quad P || (Q \Pi R) = (P || Q) \Pi (P || R)$$

$$\mathbf{b.} \quad (P \Pi Q) || R = (P || R) \Pi (Q || R)$$

Π.8 Distribución sobre el operador □

$$\mathbf{a.} \quad P \Pi (Q \square R) = (P \Pi Q) \square (P \Pi R)$$

$$\mathbf{b.} \quad P \square (Q \Pi R) = (P \square Q) \Pi (P \square R)$$

Π.9 Renombramiento

$$f(P \Pi Q) = f(P) \Pi f(Q)$$

Π.10 Después (**after**)

$$(P \Pi Q) \mathbf{after} t = (P \mathbf{after} t) \Pi (Q \mathbf{after} t)$$

$$\Leftrightarrow t \in (\text{Trazas}(P) \cap \text{Trazas}(Q))$$

$$= (P \mathbf{after} t) \Leftrightarrow t \in (\text{Trazas}(P) - \text{Trazas}(Q))$$

$$= (Q \mathbf{after} t) \Leftrightarrow t \in (\text{Trazas}(Q) - \text{Trazas}(P))$$

Π.11 Etiquetamiento

$$l : (P \Pi Q) = (l : P) \Pi (l : Q)$$

Π.12 Ocultamiento

$$(P \Pi Q) \setminus C = (P \setminus C) \Pi (Q \setminus C)$$

Π.13 Alto (Stop)

$$P \Pi \text{STOP}_{\alpha P} \neq \text{STOP}_{\alpha P}$$

Π.14 Composición secuencial

$$(P ; Q) \Pi R \neq (P \Pi R) ; (Q \Pi R)$$

Π.15 Trazas

$$\text{Trazas}(P \Pi Q) = \text{Trazas}(P) \cup \text{Trazas}(Q)$$

Π.16 Divergencias

$$d(P \Pi Q) = dP \cup dQ$$

Π.17 Fallas

$$f(P \Pi Q) = fP \cup fQ$$

Π.18 Rechazos (refusals)

$$r(P \Pi Q) = rP \cup rQ$$

A.4.2 Elección Externa (general).

Operador: \square

Descripción: Denota no determinismo “manejable”, en el que $P \square Q$ puede comportarse como P o Q , el ambiente será el que determine la elección.

Leyes:

\square .1 Aplicación general

- a. $(e_1 \rightarrow P) \square (e_2 \rightarrow Q) = (e_1 \rightarrow P \mid e_2 \rightarrow Q)$
- b. $(e_1 \rightarrow P) \square (e_1 \rightarrow Q) = (e_1 \rightarrow P) \Pi (e_1 \rightarrow Q)$
 $(e_1 \rightarrow (P \Pi Q))$

\square .2 Alfabetos iguales

$$\alpha(P \square Q) = \alpha P = \alpha Q$$

□.3 Idempotencia

$$P \square P = P$$

□.4 Conmutatividad

$$P \square Q = Q \square P$$

□.5 Asociatividad

$$P \square (Q \square R) = (P \square Q) \square R$$

□.6 Exactitud

$$P \square \perp = \perp$$

□.7 Distribución sobre el operador \parallel

a. $P \parallel (Q \square R) = (P \parallel Q) \square (P \parallel R)$

b. $(P \square Q) \parallel R = (P \parallel R) \square (Q \parallel R)$

□.8 Distribución sobre el operador Π

a. $P \Pi (Q \square R) = (P \Pi Q) \square (P \Pi R)$

b. $P \square (Q \Pi R) = (P \square Q) \Pi (Q \square R)$

□.9 Renombramiento

$$f(P \square Q) = f(P) \square f(Q)$$

□.10 Después (**after**)

$$(P \square Q) \text{ after } t = (P \text{ after } t) \square (Q \text{ after } t)$$

$$\Leftrightarrow t \in (\text{Trazas}(P) \cap \text{Trazas}(Q))$$

$$= (P \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(P) - \text{Trazas}(Q))$$

$$= (Q \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(Q) - \text{Trazas}(P))$$

□.11 Etiquetamiento

$$l : (P \square Q) = (l : P) \square (l : Q)$$

□.12 Ocultamiento

$$(P \square Q) \setminus C = (P \setminus C) \square (Q \setminus C)$$

□.13 Alto (Stop)

$$P \square \text{STOP}_{\alpha P} = P$$

□.14 Composición secuencial

$$(P ; Q) \square R \neq (P \square R) ; (Q \square R)$$

□.15 Trazas

$$\text{Trazas}(P \square Q) = \text{Trazas}(P) \cup \text{Trazas}(Q)$$

□.16 Divergencias

$$d(P \square Q) = dP \cup dQ$$

□.17 Fallas

$$\begin{aligned} f(P \square Q) = & \{ (\langle \diamond, X \rangle \mid \langle \diamond, X \rangle \in (fP \cap fQ)) \\ & \cup \{ (t, X) \mid t \neq \diamond \wedge (t, X) \in (fP \cup fQ) \} \\ & \cup \{ (t, X) \mid t \in d(P \square Q) \} \end{aligned}$$

□.18 Rechazos (refusals)

$$r(P \square Q) = rP \cap rQ$$

A.5 Ocultamiento

Operador: \backslash

Descripción: Es un mecanismo el cual incrementa los niveles de abstracción al “ocultar” eventos. Usualmente se utiliza para ocultar canales comunes y dar la ilusión de que un número de procesos ejecutándose y sincronizándose en paralelo, es un simple proceso.

Leyes:

$\backslash.1$ Aplicación general

$$\begin{aligned} (e_1 \rightarrow P) \backslash C &= P \backslash C \Leftrightarrow e_1 \in C \\ &= (e_1 \rightarrow (P \backslash C)) \Leftrightarrow e_1 \notin C \end{aligned}$$

\.2 Reducción del alfabeto

$$\alpha(P \setminus C) = (\alpha P) - C$$

\.3 Unión de alfabetos

$$(P \setminus C) \setminus D = P \setminus (C \cup D)$$

\.4 Identidad

$$P \setminus \{\} = P$$

\.5 Exactitud

$$\perp \setminus C = \perp$$

\.6 Renombramiento

$$f(P \setminus C) = f(P) \setminus f(C)$$

\.7 Etiquetamiento

$$i : (P \setminus C) = (i : P) \setminus g(C) \quad \text{donde } g(e_i) = i.e_i$$

\.8 Después (**after**)

$$(P \text{ after } t) \setminus C = (P \setminus C) \text{ after } t \uparrow (\text{elementos } t - C)$$

\.9 Distribución sobre el operador ||

$$(P || Q) \setminus C = (P \setminus C) || (Q \setminus C) \Leftrightarrow \alpha P \cap \alpha Q \cap \alpha C = \{\}$$

\.10 Distribución sobre el operador |||

$$(P ||| Q) \setminus C = (P \setminus C) ||| (Q \setminus C) \Leftrightarrow \alpha P \cap \alpha Q \cap \alpha C = \{\}$$

\.11 Distribución sobre el operador Π

$$(P \Pi Q) \setminus C = (P \setminus C) \Pi (Q \setminus C)$$

\.12 No distribución sobre el operador \square

$$(P \square Q) \setminus C \neq (P \setminus C) \square (Q \setminus C)$$

\.13 Alto (Stop)

$$STOP_{\alpha P} \setminus C = STOP_{(\alpha P - C)}$$

\.14 Composición secuencial

$$(P ; Q) \setminus C \neq (P \setminus C) ; (Q \setminus C)$$

\.15 Trazas

$$\begin{aligned} \text{Trazas}(P \setminus C) &= \{ t \uparrow (\infty P - C) \mid t \in \text{Trazas}(P) \} \\ &\Leftrightarrow \neg \forall n \in \mathbf{N} \bullet \exists t_1 \in (\text{Trazas}(P) \cap \text{seq } C) \bullet \text{len } t_1 > n \end{aligned}$$

\.16 Divergencias

$$\begin{aligned} \mathbf{d}(P \setminus C) &= \{ (t_1 \uparrow (\infty P - C)) \wedge t_2 \mid t_1 \in \mathbf{d}P \\ &\quad \vee n \in \mathbf{N} \bullet \\ &\quad \exists t_3 \in \text{seq } C \mid \text{len } t_3 > n \bullet \\ &\quad t_1 \wedge t_3 \in \text{Trazas}(P) \} \\ &\quad \wedge t_2 \in \text{seq } (\infty P - C) \} \end{aligned}$$

\.17 Fallas

$$\begin{aligned} \mathbf{f}(P \setminus C) &= \{ (t_1, X) \mid t_1 \in \mathbf{d}(P \setminus C) \} \\ &\quad \cup \{ (t_2 \uparrow (\infty P - C), X) \mid (t_2, X \cup C) \in \mathbf{f}P \} \end{aligned}$$

\.18 Rechazos (refusals)

$$\mathbf{r}(P \setminus C) = \{ X - C \mid X \in \mathbf{r}P \}$$

A.6 Después (After)

Operador: after

Descripción: P after t , describe el comportamiento del proceso P “después” de haberse comprometido o ejecutado la traza t , la cual esta compuesta de eventos dentro del alfabeto de P

Leyes:

after.1 Aplicación general

$$(e_1 \rightarrow P) \text{ after } \langle e_1 \rangle = P$$

after.2 Alfabeto

$$\infty(P \text{ after } t) = \infty P$$

after.3 Concatenación

$$(P \text{ after } t_2) \text{ after } t_1 = P \text{ after } (t_1 \wedge t_2)$$

after.4 Identidad

$$P \text{ after } \diamond = P$$

after.5 Exactitud

$$\perp \text{ after } t = \perp$$

after.6 Ocultamiento

$$(P \setminus C) \text{ after } t = (P \text{ after } t) \setminus C$$

after.7 Etiquetamiento

$$i : (P \text{ after } t) = (i \ P) \text{ after } \text{map } g \ t$$

donde $g(e_i) = i.e_i$

after.8 Renombramiento

$$f(P \text{ after } t) = f(P) \text{ after } \text{map } f \ t$$

after.9 Distribución sobre el operador ||

$$(P \ || \ Q) \text{ after } t = (P \text{ after } t \uparrow \infty P) \ || \ (Q \text{ after } t \uparrow \infty Q)$$

after.10 Distribución sobre el operador |||

$$(P \ ||| \ Q) \text{ after } t = (P \text{ after } t_1) \ ||| \ (Q \text{ after } t_2)$$

$\Leftrightarrow t = \text{interleaving}(t_1, t_2)$

after.11 Distribución sobre el operador Π

$$(P \ \Pi \ Q) \text{ after } t = (P \text{ after } t) \ \Pi \ (Q \text{ after } t)$$

$\Leftrightarrow t \in (\text{Trazas}(P) \cap \text{Trazas}(Q))$
 $= (P \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(P) - \text{Trazas}(Q))$
 $= (Q \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(Q) - \text{Trazas}(P))$

after.12 Distribución sobre el operador \square

$$(P \ \square \ Q) \text{ after } t = (P \text{ after } t) \ \square \ (Q \text{ after } t)$$

$\Leftrightarrow t \in (\text{Trazas}(P) \cap \text{Trazas}(Q))$
 $= (P \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(P) - \text{Trazas}(Q))$

$$= (Q \text{ after } t) \Leftrightarrow t \in (\text{Trazas}(Q) - \text{Trazas}(P))$$

after.13 Alto (Stop)

$$\text{STOP}_{\alpha P} \text{ after } t = \text{STOP}_{\alpha P}$$

after.14 Composición secuencial

$$\begin{aligned} \text{a. } (P ; Q) \text{ after } t &= (P \text{ after } t) ; Q \Leftrightarrow t \in \text{Trazas}(P) \\ &= Q \Leftrightarrow t \wedge \langle \sqrt{\rangle} \in \text{Trazas}(P) \\ &= (Q \text{ after } t_2) \Leftrightarrow t_1 \wedge \langle \sqrt{\rangle} \in \text{Trazas}(P) \\ &\quad \wedge t_1 \wedge \langle \sqrt{\rangle} \leq t \\ &\quad \wedge t \wedge t_2 \in \text{Trazas}(P ; Q) \\ \text{b. } P \text{ after } t \text{ [SKIP} &\Rightarrow t \wedge \langle \sqrt{\rangle} \in \text{Trazas}(P) \end{aligned}$$

after.15 Trazas

$$\text{Trazas}(P \text{ after } t) = \{ t_1 \mid (t \wedge t_1) \in \text{Trazas}(P) \}$$

after.16 Divergencias

$$d(P \text{ after } t) = \{ t_1 \mid (t \wedge t_1) \in dP \}$$

after.17 Fallas

$$f(P \text{ after } t) = \{ (t_1, x) \mid (t \wedge t_1, x) \in fP \}$$

after.18 Rechazos (refusals)

$$r(P \text{ after } t) = \{ X - \text{elementos } t \mid X \in rP \}$$

A.7 Renombramiento

Operador: f

Descripción: Es una función, $f: \alpha P \rightarrow A$, que mapea eventos del alfabeto del proceso P a eventos del conjunto A, por lo cual facilita el rehuso de especificaciones. No es necesario que αP y A sean conjuntos disjuntos, por lo que los nombres de los eventos no necesariamente tienen que cambiar.

Leyes:

f.1 Aplicación general

$$f(e_1 \rightarrow P) = f(e_1 \rightarrow f(P))$$

f.2 Alfabeto

$$\alpha f(P) = f(\alpha P)$$

f.3 Elementos mínimos

a. $f(\text{SKIP}_{\alpha P}) = \text{SKIP}_{f(\alpha P)}$

b. $f(\text{RUN}_{\alpha P}) = \text{RUN}_{f(\alpha P)}$

f.4 Exactitud

$$f(\perp) = \perp$$

f.5 Composición

$$f(f' (P)) = f \circ f' (P)$$

f.6 Distribución sobre el operador ||

$$f(P || Q) = f(P) || f(Q)$$

f.7 Distribución sobre el operador |||

$$f(P ||| Q) = f(P) ||| f(Q)$$

f.8 Distribución sobre el operador Π

$$f(P \Pi Q) = f(P) \Pi f(Q)$$

f.9 Distribución sobre el operador \square

$$f(P \square Q) = f(P) \square f(Q)$$

f.10 Ocultamiento

$$f(P \setminus C) = f(P) \setminus f(C)$$

f.11 Etiquetamiento

$$f(i : P) = f(i) : f(P)$$

f.12 Después (**after**)

$$f(P \text{ after } t) = f(P) \text{ after } \text{map } f \ t$$

f.13 Alto (Stop)

$$f(\text{STOP}_{\alpha P}) = \text{STOP}_{f(\alpha P)} = \text{STOP}_{\alpha f(P)}$$

f.14 Composición secuencial

a. $f(P ; Q) = f(P) ; f(Q)$
b. $f(\sqrt{\quad}) = \sqrt{\quad}$

f.15 Trazas

$$\text{Trazas}(f(P)) = \{ \text{map } f \ t \mid t \in \text{Trazas}(P) \}$$

f.16 Divergencias

$$d(f(P)) = \{ \text{map } f \ t \mid t \in dP \}$$

f.17 Fallas

$$f(f(P)) = \{ (\text{map } f \ t, f(x)) \mid (t, X) \in fP \}$$

f.18 Rechazos (refusals)

$$r(f(P)) = \{ f(X) \mid X \in rP \}$$

A.8 Etiquetamiento

Operador: :

Descripción: Prefija eventos con la etiqueta indicada, lo que nos permite distinguir eventos con el mismo nombre (i.e. $i.e \neq j.e$) y así evitamos los problemas de comunicación y sincronización. Esto nos permite definir múltiples instancias de un mismo proceso, cada una ejecutándose de manera independiente; además de facilitarnos la comunicación sobre múltiples canales.

Leyes:

:.1 Aplicación general

$$i : (e_1 \rightarrow P) = (i.e_1 \rightarrow (i : P))$$

2.2 Alfabeto

$$\alpha(i : P) = (i.e \mid e \in \alpha P)$$

2.3 Exactitud

$$i : \perp = \perp$$

2.4 Composición

$$\begin{aligned}\alpha(i : (j : P)) &= \{ i.e_1 \mid e_1 \in \alpha(j : P) \} \\ &= \{ i.j.e \mid e \in \alpha P \}\end{aligned}$$

2.5 Ocultamiento

$$\begin{aligned}i : (P \setminus C) &= (i : P) \setminus g(C) \\ &\text{donde } g(e_1) = i.e_1\end{aligned}$$

2.6 Renombramiento

$$f(i : P) = f(i) : f(P)$$

2.7 Después (**after**)

$$\begin{aligned}i : (P \text{ after } t) &= (i : P) \text{ after } \mathbf{map} \ g \ t \\ &\text{donde } g(e_1) = i.e_1\end{aligned}$$

2.8 Distribución sobre el operador \parallel

$$i : (P \parallel Q) = (i : P) \parallel (i : Q)$$

2.9 Distribución sobre el operador $\parallel\parallel$

$$i : (P \parallel\parallel Q) = (i : P) \parallel\parallel (i : Q)$$

2.10 Distribución sobre el operador Π

$$i : (P \Pi Q) = (i : P) \Pi (i : Q)$$

2.11 Distribución sobre el operador \square

$$i : (P \square Q) = (i : P) \square (i : Q)$$

2.12 Alto (Stop)

$$i : (\text{STOP}_{\alpha P}) = \text{STOP}_{g(\alpha P)} \quad \text{donde } g(e_1) = i.e_1$$

..13 Composición secuencial

$$i : (P ; Q) = (i : P) ; (i : Q)$$

..14 Trazas

$$\text{Trazas}(i : P) = \{ \mathbf{map} \ g \ t \mid t \in \text{Trazas}(P) \} \\ \text{donde } g(e_1) = i.e_1$$

..15 Divergencias

$$\mathbf{d}(i : P) = \{ \mathbf{map} \ g \ t \mid t \in \mathbf{d}P \} \\ \text{donde } g(e_1) = i.e_1$$

..16 Fallas

$$\mathbf{f}(i : P) = \{ (\mathbf{map} \ g \ t, g(X)) \mid (t, X) \in \mathbf{f}P \} \\ \text{donde } g(e_1) = i.e_1$$

..17 Rechazos (refusals)

$$\mathbf{r}(i : P) = \{ g(X) \mid X \in \mathbf{r}P \} \\ \text{donde } g(e_1) = i.e_1$$

A.9 Composición secuencial

Operador: ;

Descripción: $P ; Q$, denota la ejecución del proceso P , seguido por la ejecución del proceso Q . Q se ejecuta una vez que P halla terminado de manera satisfactoria (i.e que se halla comprometido en el evento \surd). Si P no termina, Q nunca se ejecuta.

Leyes:

;1 Aplicación general

$$(e_1 \rightarrow P) ; Q = (e_1 \rightarrow (P ; Q))$$

;2 Iteración

- a. $P^0 = \text{SKIP}$
- b. $P^n = P ; P^{n-1}$, para $n > 0$

;3 Alfabeto

$$\alpha(P; Q) = \alpha P \cup \alpha Q$$

;4 Antisimetría

$$(P; P) \neq P$$

;5 Anti-conmutatividad

$$(P; Q) \neq (Q; P)$$

;6 Asociatividad

$$(P; Q); R = P; (Q; R)$$

;7 Elementos mínimos

a. $P; \text{SKIP} = P$

b. $\text{SKIP}; P = P$

;8 Exactitud

$$\perp; P = \perp$$

;9 Etiquetamiento

$$i: (P; Q) = (i: P); (i: Q)$$

;10 Renombramiento

$$f(P; Q) = f(P); f(Q)$$

;11 Después (**after**)

a. $(P; Q) \text{ after } t = (P \text{ after } t); Q \Leftrightarrow t \in \text{Trazas}(P)$
 $= Q \Leftrightarrow t \wedge \langle \sqrt{\ } \rangle \in \text{Trazas}(P)$
 $= (Q \text{ after } t_2) \Leftrightarrow t_1 \wedge \langle \sqrt{\ } \rangle \in \text{Trazas}(P)$
 $\quad \wedge t_1 \wedge \langle \sqrt{\ } \rangle \leq t$
 $\quad \wedge t \wedge t_2 \in \text{Trazas}(P; Q)$

b. $P \text{ after } t \ll \text{SKIP} \Rightarrow t \wedge \langle \sqrt{\ } \rangle \in \text{Trazas}(P)$

;12 No-distribución sobre el operador \parallel

a. $P \parallel (Q; R) \neq (P \parallel Q); (P \parallel R)$

b. $(P \parallel Q); R \neq (P; R) \parallel (Q; R)$

;.13 No-distribución sobre el operador \parallel

- a. $P \parallel (Q ; R) \neq (P \parallel Q) ; (P \parallel R)$
- b. $(P \parallel Q) ; R \neq (P ; R) \parallel (Q ; R)$

;.14 Distribución sobre el operador Π

- a. $(P \Pi Q) ; R = (P ; R) \Pi (Q ; R)$
- b. $P ; (Q \Pi R) = (P ; Q) \Pi (P ; R)$

;.15 Distribución sobre el operador \square

- a. $(P \square Q) ; R = (P ; R) \square (Q ; R)$
- b. $P ; (Q \square R) = (P ; Q) \square (P ; R)$

;.16 Alto (Stop)

- a. $STOP ; P = STOP$
- b. $P ; STOP = P$

;.17 Trazas

$$\begin{aligned} \text{Trazas}(P ; Q) = & \{ t_1 \mid t_1 \in \text{Trazas}(P) \wedge \sqrt{\quad} \notin \text{elementos } t_1 \} \\ & \cup \{ t_1 \wedge t_2 \mid t_1 \wedge \langle \sqrt{\quad} \rangle \in \text{Trazas}(P) \wedge t_2 \in \text{Trazas}(Q) \} \end{aligned}$$

;.18 Divergencias

$$\begin{aligned} \mathbf{d}(P ; Q) = & \{ t_1 \mid t_1 \in \mathbf{d}P \wedge \sqrt{\quad} \notin \text{elementos } t_1 \} \\ & \cup \{ t_1 \wedge t_2 \mid t_1 \wedge \langle \sqrt{\quad} \rangle \in \text{Trazas}(P) \\ & \quad \wedge \sqrt{\quad} \notin \text{elementos } t_1 \\ & \quad \wedge t_2 \in \mathbf{d}Q \} \end{aligned}$$

;.19 Fallas

$$\begin{aligned} \mathbf{f}(P ; Q) = & \{ (t, X) \mid (t, X \cup \{\sqrt{\quad}\}) \in \mathbf{f}P \} \\ & \cup \{ (t_1 \wedge t_2, X) \mid t_1 \wedge \langle \sqrt{\quad} \rangle \in \text{Trazas}(P) \wedge (t_2, X) \in \mathbf{f}Q \} \\ & \cup \{ (t, X) \mid t \in \mathbf{d}(P ; Q) \} \end{aligned}$$

;.20 Rechazos (refusals)

$$\begin{aligned} \mathbf{r}(P ; Q) = & \{ A \mid (A \cup \{\sqrt{\quad}\}) \in \mathbf{r}P \} \\ & \cup \{ B \mid \langle \sqrt{\quad} \rangle \in \text{Trazas}(P) \wedge B \in \mathbf{r}Q \} \end{aligned}$$

A.10 Comunicación

Operadores : ? y !

Descripción: La expresión $c?x$ denota la recepción de algún valor x por el canal de nombre c , y la expresión $c!y$ denota el envío del contenido de la variable y a través del canal de nombre c . Los canales utilizados para comunicar procesos son unidireccionales.

Leyes:

comm.1 Sincronización

$$(c_1?b \rightarrow P(b)) \parallel (c_1!a \rightarrow Q) = (c_1!a \rightarrow (P(a) \parallel Q))$$

comm.2 Ocultamiento

$$\begin{aligned} \text{a. } & ((c_1?b \rightarrow P(b)) \parallel (c_1!a \rightarrow Q)) \setminus \{c_1.x \mid x \in \alpha(c_1)\} \\ &= (P(a) \parallel Q) \setminus \{c_1.x \mid x \in \alpha(c_1)\} \\ \text{b. } & ((c_1?b \rightarrow P(b)) \parallel (c_1!a \rightarrow Q)) \setminus \{c_2.x \mid x \in \alpha(c_2)\} \\ &= (c_1!a \rightarrow (P(a) \parallel Q)) \setminus \{c_2.x \mid x \in \alpha(c_2)\} \end{aligned}$$

comm.3 Renombramiento

$$\begin{aligned} \text{a. } & f(c_1?b \rightarrow P(b)) = (f(c_1)?b \rightarrow f(P(b))) \\ \text{b. } & f(c_1!a \rightarrow Q) = (f(c_1)!a \rightarrow f(Q)) \end{aligned}$$

comm.4 Etiquetamiento

$$\begin{aligned} \text{a. } & i : (c_1?b \rightarrow P(b)) = (i.c_1?b \rightarrow (i.P(b))) \\ \text{b. } & i : (c_1!a \rightarrow Q) = (i.c_1!a \rightarrow (i.Q)) \end{aligned}$$

comm.5 Trazas

$$\begin{aligned} \text{a. } & \text{Trazas}(c_1?b \rightarrow P(b)) = \{ \langle c_1.b \rangle^{\wedge} t \mid t \in \text{Trazas}(P(b)) \} \\ \text{b. } & \text{Trazas}(c_1!a \rightarrow Q) = \{ \langle c_1.a \rangle^{\wedge} t \mid t \in \text{Trazas}(Q) \} \end{aligned}$$

comm.6 Divergencias

$$\begin{aligned} \text{a. } & \mathbf{d}(c_1?b \rightarrow P(b)) = \{ \langle c_1.b \rangle^{\wedge} t \mid t \in \mathbf{d}P(b) \} \\ \text{b. } & \mathbf{d}(c_1!a \rightarrow Q) = \{ \langle c_1.a \rangle^{\wedge} t \mid t \in \mathbf{d}Q \} \end{aligned}$$

comm.7 Fallas

$$\begin{aligned} \text{a. } & \mathbf{f}(c_1?b \rightarrow P(b)) = \{ (\langle \rangle, X) \mid X \subseteq (\alpha P(b) - \{c_1.b\}) \} \\ & \cup \{ (\langle c_1.b \rangle^{\wedge} t, X) \mid (t, X) \in \mathbf{f}P(b) \} \end{aligned}$$

$$\mathbf{b.} \quad r(c_1!a \rightarrow Q) = \{ \langle \cdot, Y \rangle \mid Y \subseteq (\infty Q - \{c_1.a\}) \} \\ \cup \{ \langle c_1.a \wedge t, Y \rangle \mid (t, Y) \in fQ \}$$

comm.8 Rechazos

$$\mathbf{a.} \quad r(c_1?b \rightarrow P(b)) = \{ X \mid X \subseteq (\infty P(b) - \{c_1.b\}) \}$$

$$\mathbf{b.} \quad r(c_1!a \rightarrow Q) = \{ Y \mid Y \subseteq (\infty Q - \{c_1.a\}) \}$$

A.11 Subordinación

Operador: //

Descripción: $P // Q$ (P es subordinado a Q). P puede comprometerse en los eventos de su alfabeto solamente cuando Q lo permite. Q puede comprometerse en sus propios eventos de manera independiente de P . El ambiente no puede interrumpir o sincronizarse con el proceso subordinado.

Leyes:

//.1 Alfabetos

$$\mathbf{a.} \quad P // Q \Rightarrow \infty P \subseteq \infty Q$$

$$\mathbf{b.} \quad \infty(P // Q) \Rightarrow \infty Q - \infty P$$

//.2 Definición

$$P // Q = (P \parallel Q) \setminus \infty P$$

//.3 Sincronización

$$\mathbf{a.} \quad i : (c!v \rightarrow P) // (i.c?x \rightarrow Q(x)) = i : P // Q(v)$$

$$\mathbf{b.} \quad i : (c?x \rightarrow P(x)) // (i.c!v \rightarrow Q) = i : P(v) // Q$$

Existen muchísimas leyes más, pero consideramos que el presentar todas las leyes aquí no tiene sentido dada la limitación en espacio existente. El lector interesado puede encontrar en [Hinchey95 a] una lista bastante amplia sobre las demás leyes.

Apéndice II. Script FDR2 de los refinamientos del núcleo básico del sistema operativo.

En este apéndice mostramos la especificación inicial y todos los refinamientos del núcleo de el sistema operativo, en el formato de entrada a FDR2.

```
-- =====  
-- Archivo que contiene la especificación inicial y los refinamientos del  
-- núcleo básico de un sistema operativo, el cual forma parte de la  
-- tesis de Maestría titulada: "Especificación y diseño del núcleo básico de un sistema  
-- operativo mediante CSP"  
--  
-- Lenguaje de especificacion formal utilizado: CSP  
--  
-- Dir. de Tesis: Dr. Hector Ruiz Barradas  
--  
-- Tesista : Manuel Aguilar C.  
--  
-- =====  
--                               Especificacion Inicial  
-- =====
```

```
channel m: Int.Int
```

```
Nucleo = Corrutinas(1,1)
```

```
Corrutinas(i,j) = if i<4 then ( m.1.i -> m.2.j -> Corrutinas(i+1,j+1))  
                  else (SKIP)
```

```
-- =====  
--                               Primer Refinamiento  
-- =====
```

```
Nucleo1 = Corrutina1_1(1)
```

```
Corrutina1_1(i) = if i<4 then ( m.1.i -> Corrutina1_2(i))  
                  else (SKIP)
```

```
Corrutina1_2(i) = m.2.i -> Corrutina1_1(i+1)
```

Segundo Refinamiento

```
Nucleo2 = Corrutina2_1(1)

Corrutina2_1(i) = if i<4 then ( m.1.i -> Transfer2(2,i) )
                else (SKIP)

Corrutina2_2(i) = m.2.i -> Transfer2(1,i)

Transfer2(j,i) = if j==1 then Corrutina2_1(i+1)
                else Corrutina2_2(i)
```

Tercer Refinamiento

```
channel c1left, c1right, c2left, c2right :{0,1,2,3,4}

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
-- como el paralelismo entre procesos con sus eventos sincrónicos y ocultos.

Nucleo3 = ( ( Cola1(<>)|| Cola2(<> )
            [{| c1left,c2left,c1right, c2right |}]
            (Init(<1,2>); Transfer3 )
            ) \ {| c1left, c1right, c2left, c2right |}

Init(s) = if ( null(s) ) then (SKIP)
         else ( c1left!head(s) -> c2left!1 -> Init(tail(s)) )

-- El tamaño de las colas lo limitamos a 6 para que la prueba entre procesos fuera rápida.

Cola1(s) = if (s == <>) then ( c1left?x -> Cola1(<x>) )
         else
         (
           (
             (c1left?y -> if ((#s)+1 < 6) then Cola1(s^<y>)
                       else (c1right!(head(s)) -> Cola1(tail(s)^<y>))
             )
           []
           (c1right!(head(s)) -> Cola1(tail(s)) )
           )
         []
         SKIP
         )

-- Definimos Cola2 en función de Cola1 renombrando sus eventos.
```

```
Cola2(s) = Cola1(s)[[ c1left.n <- c2left.n, c1right.n <- c2right.n | n<-(0..6)]]
```

-- Limitamos el número de eventos en los cuales se comprometen los programas de usuario

```
Corrutina3_1(i) = if i<4 then ( m.1.i -> Transfer3 )  
                else (SKIP)
```

```
Corrutina3_2(i) = m.2.i -> Transfer3
```

```
Suma1(x) = (x + 1) % 5
```

```
Transfer3 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i)->  
           ( if j==1 then Corrutina3_1(i)  
             else Corrutina3_2(i)  
           )
```

```
-- =====  
--                      Cuarto Refinamiento  
-- =====
```

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
-- como el paralelismo entre procesos con sus eventos síncronos y ocultos.

```
Nucleo4 = ( ( Cola1(<>) ||| Cola2(<>) )  
           [({| c1left, c2left, c1right, c2right |})]  
           (Init(<1,2,3,4,5>) ; Transfer4)  
           ) \ { | c1left, c2left, c1right, c2right, m.3, m.4, m.5 | }
```

-- Dado que el etiquetamiento no existe en FDR2 y el remobramiento cuando se utilizó agotó
-- la memoria virtual, tuvimos que buscar un camino alternativo para probar nuestro cuarto
-- refinamiento. El que utilizamos fue invocar una corrutina, la cual fue definida de manera
-- explícita.

```
Transfer4 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i) ->  
           (if j==1 then Corrutina4_1(i)  
            else (if j==2 then Corrutina4_2(i)  
                  else ( if j==3 then Corrutina4_3(i)  
                          else ( if j==4 then Corrutina4_4(i)  
                                  else Corrutina4_5(i)  
                                )  
                            )  
                )  
           )  
           )  
           )
```

-- Debido a que el tiempo de prueba del refinamiento de FDR2 era muy grande limitamos el

-- número de programas de usuario a 5, aunque se observa de manera clara que este número
 -- puede ser ampliado sin ningún problema.

```

Corrutina4_1(i) = if i<4 then ( m.1.i -> Transfer4 )
                  else (SKIP)
Corrutina4_2(i) = m.2.i -> Transfer4
Corrutina4_3(i) = m.3.i -> Transfer4
Corrutina4_4(i) = m.4.i -> Transfer4
Corrutina4_5(i) = m.5.i -> Transfer4

```

 -- Quinto refinamiento

```

channel tick
channel TimerLeft: {0,1,2}
channel TimerRight : {0}

```

-- Dado que el operador de subordinación no existe en FDR2, la subordinación fue implantada
 -- como el paralelismo entre procesos con sus eventos sincronos y ocultos.

```

Nucleo5 = ( ( Cola1(<>) ||| Cola2(<>) )
            [{}| c1left, c2left, c1right, c2right |}]
            ( ( Init(<1,2,3,4,5>); Transfer5 )
              [{}| tick, TimerLeft, TimerRight |}]
              Timer
            )
          ) \ {c1left, c2left, c1right, c2right, TimerLeft, TimerRight, tick, m.3, m.4, m.5 }

```

-- Proceso Transfer5, invoca al proceso que le corresponde ejecutarse, no si antes inicializar el
 -- timer, Cuando se le termina su rebanada de tiempo la interrupción Termina realiza el cambio
 -- de contexto e invoca nuevamente a Transfer5

```

Transfer5 = c1right?j -> c2right?i -> c1left!j -> c2left!Suma1(i) ->TimerLeft!1 ->
            ( if j==1 then (Corrutina5_1(i) ^ Termina)
              else (if j==2 then (Corrutina5_2(i) ^ Termina)
                    else (if j==3 then (Corrutina5_3(i) ^ Termina)
                          else (if j==4 then (Corrutina5_4(i) ^ Termina)
                                else Corrutina5_5(i) ^ Termina)
                          )
                    )
              )
            )

```

-- Debido a que el tiempo de prueba del refinamiento de FDR2 era muy grande limitamos el

-- número de programas de usuario a 5, aunque se observa de manera clara que este número
-- puede ser ampliado sin ningún problema.

```
Corrutina5_1(i) = if i<4 then ( m.1.i -> tick -> tick -> Corrutina5_1(Suma1(i)) )  
                else (SKIP)  
Corrutina5_2(i) = m.2.i -> tick -> tick -> Corrutina5_2(Suma1(i))  
Corrutina5_3(i) = m.3.i -> tick -> tick -> Corrutina5_3(Suma1(i))  
Corrutina5_4(i) = m.4.i -> tick -> tick -> Corrutina5_4(Suma1(i))  
Corrutina5_5(i) = m.5.i -> tick -> tick -> Corrutina5_5(Suma1(i))
```

-- Interrupción que refleja el cambio de contexto en las colas

Termina = TimerRight?x -> Transfer5

Timer = (TimerLeft?i -> Tick(i))

Resta1(x) = if x>0 then (x-1)
 else (0)

```
Tick(i) = ( (if i>0 then (tick -> Tick(Resta1(i)))  
            else (TimerRight!0 -> Timer)  
            )  
            []  
            SKIP  
            )
```

```
-- =====  
--                               Implantacion  
-- =====
```

channel Timer_Right

-- Limitamos el número de procesos a 5.

Inicializa = c1left!1 -> c2left!1 -> c1left!2 -> c2left!1 -> c1left!3 -> c2left!1 ->
 c1left!4 -> c2left!1 -> c1left!5 -> c2left!1 -> SKIP

Sched = c1right?i -> c2right?j -> c1left!i -> c2left!suma1(j) ->SKIP

Corrut(i,j) = m.i.j -> SKIP

Interrup = tick -> Timer_Right -> SKIP

```

Implantacion = ( (Cola1(<>) ||| Cola2(<>))
  [({| c1left, c2left, c1right, c2right |})]
  (Inicializa ; Sched ; Corrut(1,1); Interrup ;
    Sched ; Corrut(2,1); Interrup ;
    Sched ; Corrut(3,1); Interrup ;
    Sched ; Corrut(4,1); Interrup ;
    Sched ; Corrut(5,1); Interrup ;
    Sched ; Corrut(1,2); Interrup ;
    Sched ; Corrut(2,2); Interrup ;
    Sched ; Corrut(3,2); Interrup ;
    Sched ; Corrut(4,2); Interrup ;
    Sched ; Corrut(5,2); Interrup ;
    Sched ; Corrut(1,3); Interrup ;
    Sched ; Corrut(2,3); Interrup ;
    Sched ; Corrut(3,3); Interrup ;
    Sched ; Corrut(4,3); Interrup ;
    Sched ; Corrut(5,3); Interrup ;
  ) \ {| tick, c1left, c2left, c1right, c2right, Timer_Right, m.3, m.4, m.5 |}

```

Bibliografia

- [Dromey89] Dromey R. Geoff (1989), "The development of programs from specifications", International Computer Series, Addison Wesley.
- [FORMAL95] Formal Systems (1995), "Failures Divergence Refinement, FDR2 Preliminary Manual", Formal Systems (Europe) Ltd.
- [Hinchey93] Hinchey M.G. (1993), "A formal design method for real-time Ada software", in L. Collingbourne (ed.), Ada: Towards Maturity, pp 123-137, IOS Press, Amsterdam.
- [Hinchey94] Hinchey M.G. (1994), "Formally specifying real-time Ada software", in Proc. IEEE RTWA'94, 2nd IEEE Workshop on Real-Time Applications, 21-22 July 1994, pp.39-44 Washington DC, IEEE Computer Society Press.
- [Hinchey95 a] Hinchey M.G. (1995), "Concurrent systems: formal development in CSP", The McGraw-Hill International Series in Software Engineering.
- [Hinchey95 b] Hinchey M.G. & Bowen J.P. (1995), "Applications of formal systems", C.A.R. Hoare Series Editor, Prentice Hall International Series in Computer Science.
- [Hoare 85] Hoare C.A.R. (1985), "Communicating Sequential Processes", C.A.R. Hoare Series Editor, Prentice-Hall International Series in Computer Science.
- [Hoare 91] Hoare C.A.R. (1991), "The transputer and *occam*: a personal story", Concurrency: Practice and Experience, 3(4):249-264, August.
- [Inmos 84] Inmos Limited (1984), "*occam* programming manual", Prentice-Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
- [Inmos 88] Inmos Limited (1988), "*occam2* programming manual", Prentice-Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
- [Jeannette90] Jeannette M.W. (1990), "A specifier's introduction to formal methods", Computer IEEE, September 1990, pp. 8-24.
- [Kenneth93] Kenneth J. Turner (1993), "Using formal description techniques: an introduction to Stelle, Lotos and SDL", Edited by Kenneth J. Turner, John Wiley & Sons.
- [Mander95] Mander K.C. and Polak, F.A.C. (1995), "Rigorous specification using structured systems analysis and Z", Information and Software Technology, 37(5), May.

[Roscoe88] Roscoe A.W. and Hoare C.A.R. (1988), "Laws of *occam* programming", *Theoretical Computer Science*, 60:177-229.

[Roscoe94] Roscoe A.W. (1994), "Model-Checking CSP", in A.W. Roscoe (ed.), *A Classical Mind: Essays in Honor of CAR Hoare*, pp 353-378, Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.

[Swartout82] Swartout, W. and Balzer, R. (1982), "On the inevitable intertwining of specification and implementation", *Communications of the ACM*, 25(7): 438-440, July.

Reseña Crítica

La tesis es una contribución a la investigación y desarrollo de software a través de métodos formales. El énfasis se hace en la aplicación de la teoría CSP, en el diseño y construcción del núcleo básico de un sistema operativo multitareas.

En el trabajo se da una visión sobre la teoría CSP, la cual es un ejemplo del enfoque algebraico de procesos de los métodos formales. Se introduce el funcionamiento de la herramienta FDR, la cual permite, de manera automática, verificar la corrección de los refinamientos. Por último, se muestra cómo a través del enunciado del comportamiento simplificado del núcleo básico de un sistema operativo multitarea, es posible, a través de refinamientos de la especificación, obtener el código de dicho núcleo, expresado en pseudoinstrucciones de un lenguaje imperativo.

Es necesario marcar el rigor con el que se llevan a cabo los refinamientos en CSP, y que es respetado en este trabajo. Además de mostrar manualmente los refinamientos, mediante la aplicación de las leyes CSP, se muestra la enorme utilidad de la herramienta FDR, en dicha tarea. Por último, aunque el caso de estudio abordado en la tesis es relativamente modesto, la importancia del trabajo está en mostrar como aplicar correctamente la metodología CSP en el desarrollo de programas de cualquier tamaño.

Héctor Ruiz B.
Elizabeth Pérez C.
Ju Shiguang

Los abajo firmantes, integrantes de jurado para el examen de grado que sustentará el
Lic. Manuel Aguilar Cornejo, declaramos que hemos revisado la tesis titulada:

“Especificación y diseño del núcleo básico de un sistema operativo mediante CSP”,
consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con
especialidad en Ingeniería Eléctrica.

Atentamente

Dr. Héctor Ruíz Barradas

Hector Ruiz Barradas

Dra. Elizabeth Pérez Cortés



Dr. Shiguang Ju



CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro
antes del vencimiento de préstamo señalado
por el último sello.

- 2 ABR. 1998

DEVOLUCION

