

Modelos de Concurrencia

Máster
Métodos Formales en Ingeniería Informática

Carlos Gregorio Rodríguez

Departamento de
Sistemas Informáticos y Computación

Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2018

—Índice general

1. Presentación	3
1.1. Informática	5
1.2. Teoría de la Concurrencia	6
1.3. Teoría de Procesos	8
1.4. Semánticas de Procesos	12
2. Introducción	17
2.1. Definición de los Procesos	19
2.2. Relaciones entre Procesos	32
2.2.1. Bisimulaciones y Simulaciones	35
2.2.2. Trazas, Fallos y otras Denotaciones	40
2.2.3. Testing	44
2.3. Estudios Comparativos de Semánticas	46
2.3.1. Bisimulación como Testing	48
2.3.2. Testing como Bisimulación	49
2.3.3. Una Clasificación	51
3. Epílogo	57

Presentación

Forget for a while about computers and computer programming and think instead about objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behaviour.

Communicating Sequential Processes
Tony Hoare

En este capítulo se traza una breve historia de la evolución de las ideas y conceptos previos que sirven de base a la teoría de procesos concurrentes. El objetivo esencial es introducir un su contexto —genérico y lo más alejado posible de tecnicismos— la evolución de las líneas fundamentales de trabajo que han servido de guía para el desarrollo de la disciplina investigadora.

1.1. Informática

It is possible to invent a single machine which can be used to compute any computable sequence. If this machine I is supplied with a tape on the beginning of which is written the S.D [standar description] of some computing machine M, then I will compute the same sequence as M.

Esta frase, que aparece en el artículo de Allan Turing publicado en *Proceedings of the London Mathematical Society* en 1936 [Tur36], puede ser considerada como la que marca el momento del nacimiento de la Informática, tras siglos de gestación en la matriz de la cultura y el conocimiento humano. En dicho artículo, titulado *On computable numbers, with an application to the Entscheidungsproblem*, Turing demostraba formalmente la posibilidad lógica de la existencia de una máquina de calcular de propósito general, la máquina *I*, *the universal computing machine*, una modelización teórica y formal de lo que hoy llamamos, en castellano, un *ordenador*.

En las poco más de treinta páginas de su artículo, Turing formalizaba el concepto de algoritmo mediante un modelo abstracto, que él denominaba la *máquina automática*, a la postre conocido como *máquina de Turing*; demostraba la existencia de una máquina automática de propósito general que era capaz de producir el mismo resultado que cualquier otra máquina; y, finalmente, probaba la existencia de problemas que no podían decidirse en su modelo, dando así una respuesta, complementaria a la de Kurt Gödel [Gö31], a la tercera de las cuestiones que el gran matemático David Hilbert propuso en 1928, conocida como *Entscheidungsproblem*, o problema de decisión, en castellano.

Pero la materialización física de la máquina de propósito general abstracta propuesta por Turing llevó aún casi una década de trabajos y descubrimientos. En 1938, Claude E. Shannon presenta su tesis doctoral titulada *A symbolic analysis of relay and switching circuits* [Sha38, SW93], en la que muestra cómo se podían diseñar circuitos electrónicos capaces de implementar los conceptos del álgebra booleana. El cálculo y la lógica, las dos grandes líneas de conocimiento sobre las que se asienta la Informática, quedaban así unidas.

En los años 40 del pasado siglo XX, los ingenieros se lanzaron a la construcción de los primeros protoordenadores. Entre 1939 y 1942, un profesor de la Universidad de Iowa, John Vincent Atanasoff, junto con su estudiante Clifford Berry, construyeron el primer ordenador electrónico digital, con circuito lógico sumador-restador, y un ingenioso sistema de refresco de memoria que consistía en un rodillo giratorio [Ata84,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTScheidungsproblem

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

[Extracted from the Proceedings of the London Mathematical Society, Ser. 2, Vol. 42, 1937.]

The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbersome technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.



Figura 1.1: Alan Turing 1912–1954

Mac96]. El ingeniero alemán Konrad Zuse terminó en 1941 la Z3, una calculadora binaria programable con ciclos pero sin saltos condicionales, con memoria y una unidad de cálculo basada en relés telefónicos. En 1943, con la participación de Allan Turing, se construye en Inglaterra el Colossus, y en 1946, en la Universidad de Pennsylvania, se finalizó el *Electronic Numerical Integrator and Automatic Calculator* (ENIAC), si no el primer computador electrónico de propósito general, sin duda sí el más conocido.

Pero son los sucesores del ENIAC, el *Electronic Delay Storage Automatic Computer* (EDSAC), completado en 1949 y el *Electronic Discrete Variable Computer* (ED-VAC), completado en 1952, junto con el *Ferranti Mark 1* de 1951 de la Universidad de Manchester —que presume de ser el primer ordenador comercializado— los verdaderos arquetipos de los ordenadores actuales. La colaboración, directa o indirecta, de John von Neumann [vN93] les confirió lo que a la postre se ha llamado *arquitectura von Neumann*, que consistía en tener una única memoria que almacenaba programas y datos y una unidad aritmético-lógica sencilla pero rápida y barata.

1.2. Teoría de la Concurrencia

Tanto en el formalismo de la máquina de Turing, como en la arquitectura von Neumann, en cada momento sólo una única acción puede ocurrir: cálculos, operaciones y decisiones son tomadas de forma consecutiva, una tras otra. Una posible explicación de porqué este *castigo secuencial* fue asumido es dada por Robin Milner de forma especulativa en su ensayo titulado *Turing, Computing and Communication*:

By 1937 there was already a rich repertoire of computational procedures.

Typically, they involved a hand calculating machine and a schematic use of paper in solving, say, a type of differential equation following a specific algorithm. [...] The familiar calculation procedures, which computers were designed to relieve us of, were all inherently sequential; not at all like cooking recipes which ask you to conduct several processes at once—for example, to slice the beans while the water is coming to the boil. This in turn may be because our conscious thought process is sequential; we have so little short term memory that we can't easily think of more than one thing at once. [Mil06]

Durante la década de los 50 se mejoraron los componentes físicos de los ordenadores y nacieron los primeros compiladores para lenguajes de alto nivel (Fortran, Algol, Cobol, Lisp...) pero en estos años el desarrollo de la Informática se mantuvo esencialmente dentro de los límites de la secuencialidad. Habría que esperar hasta la década de los 60 para que se fraguaran las condiciones que permitieran vislumbrar los límites que imponía este modelo primigenio de computación secuencial.

Por un lado, los componentes electrónicos alcanzaron un gran nivel de desarrollo al aparecer los transistores y las memorias de núcleo magnético. Surgió así la necesidad de optimizar el trabajo de los ordenadores que, gracias a estos avances, habían llegado a ser mucho más rápidos que los periféricos que tenían que controlar o que los usuarios que los tenían que manejar. El concepto de sistema operativo nació precisamente de la necesidad de encontrar soluciones a estos objetivos: la gestión y coordinación de actividades o procesos, la compartición de los recursos propios como la memoria y la unidad central de proceso, y el uso eficiente de los periféricos que el ordenador controlaba. En 1961, Kilburn, Payne y Howarth fueron los primeros en utilizar interrupciones para simular la ejecución concurrente de varios programas, y para solapar las operaciones de entrada y salida, en el ordenador *Atlas* de la Universidad de Manchester [KPH61].

Por otro lado, el desarrollo de los lenguajes de programación llevó a los diseñadores a elevar cada vez más el nivel de abstracción. En 1965, Charles Antony Richard Hoare —que acababa de asombrar al mundo, siendo aún un veinteañero, con el *quicksort*, el mejor algoritmo de ordenación basado en comparaciones— publicaba el artículo titulado *Record Handling* [Hoa65], en el que proponía métodos para la representación y manipulación de objetos estructurados complejos, así como de las relaciones entre ellos. Kristen Nygaard y Ole-Johan Dahl, que habían desarrollado unos años antes el lenguaje de simulación Simula1, reconocieron inmediatamente el potencial de las ideas presentadas en el artículo de Hoare para poder abordar uno de los problemas fundamentales en la programación de simulación: la representación del mundo que se desea simular. Su trabajo, junto con las aportaciones de Hoare [DH72], condujo al desarrollo de Simula67 [DN67, ND78], que fue sin duda uno de los eventos más importantes en el desarrollo de la programación orientada a objetos. Como destaca Robin Milner, uno de los efectos más importantes de la idea de *objeto* es que aporta una nueva metáfora a la forma de entender la resolución de problemas

mediante programas: la idea de una comunidad de agentes interactuando entre sí, cada uno de ellos persistiendo en el tiempo pero evolucionando simultáneamente en su estado.

Pero tanto en los sistemas operativos como en los lenguajes de alto nivel, para poder expresar, formalizar, pensar y razonar con propiedad sobre las nuevas actividades e interacciones concurrentes que se planteaban, con una naturaleza tan diferente a las clásicas actividades secuenciales, se necesitaban nuevas metáforas, nuevos conceptos y nuevas teorías. Es fundamentalmente a partir de la segunda mitad de la década de los 60 y sobre todo en los 70, cuando se presentan los principios básicos de los modelos teóricos basados en interacciones de sistemas concurrentes.

1.3. Teoría de Procesos

El trabajo de Edsger Wybe Dijkstra, *Cooperating Sequential Processes*, publicado en 1965, es uno de los documentos fundamentales de la teoría de procesos y uno de los mejores artículos en la historia de la programación. Ese artículo fue expresamente citado por el correspondiente jurado como uno de los méritos destacables para justificar la concesión a su autor del premio Turing en el año 1972.



Figura 1.2: Edsger Wybe Dijkstra 1930–2002

En dicho trabajo aparecía una formalización de los programas secuenciales, enfatizando el autor el hecho de que el adjetivo *secuencial* no es redundante al hablar de programación, pues existe una programación mucho más extensa, la programación

concurrente, que se encarga de la sincronización, cooperación, intercomunicación y gestión de un número arbitrario de procesos o programas secuenciales.

The technical term for what we have called “rules of behaviour” is an algorithm or program. (It is not customary to call it “a sequential program” although this name would be fully correct.) Equipment able to follow such rules, “to execute such a program” is called “a general purpose sequential computer” or “computer” for short; what happens during such a program execution is called a “sequential process”. [Dij65]

Sobre estos procesos secuenciales Dijkstra definía muchos de los fundamentos conceptuales de la programación concurrente: como primitivas de sincronización inventaba los *semáforos* y con ellos implementaba un mecanismo de paso de mensajes entre procesos; identificaba problemas inherentes a la programación concurrente, como la competencia por el acceso a las secciones críticas o el problema que él llamaba *deadly embrace*, y que posteriormente se ha pasado a denominar *deadlock*, generalmente traducido como interbloqueo; también proponía un ingenioso algoritmo para la prevención del *deadly embrace*, denominado *Banker's algorithm*; investigaba los sistemas reactivos que necesitan interactuar con el mundo exterior; y establecía uno de los principios con más trascendencia en el desarrollo de los modelos abstractos: la asunción de la independencia de las velocidades precisas de los procesos, *speed independence*.

When two or more of such processes has to cooperate with each other, they must be connected, i.e. they must be able to communicate with each other to exchange information. As we shall see below, the properties of these means of intercommunication play a vital role.

Furthermore, we have stipulated that the processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. In particular we disallow any assumption about the relative speeds of the different processes. This independence of speed ratios is strict accordance with our appreciation of the single sequential process: its only essential feature is, that its elementary steps are performed in sequence. [Dij65]

Por su parte, C.A.R. Hoare publicaba en 1971 *Towards a Theory of Parallel Programming*. Este artículo analizaba los principios de diseño que había que considerar para extender los lenguajes de programación de alto nivel con características de programación paralela.

The design of high-level programming languages which simultaneously satisfy these four objectives [Security from error, efficiency, conceptual simplicity and breath of application] is one of the major challenges to the

invention, imagination and intellect of Computer Scientists of the present day. The solutions proposed in this paper cannot claim to be final, but it is believed that they form a sound basis for further advance. [Hoa71]



Figura 1.3: Charles Antony Richard Hoare

Pero estas ideas aún tendrían que madurar en los trabajos de Per Brinch Hansen [Han73] y del propio Hoare [Hoa74] para dar lugar al concepto de *Monitor*, una primitiva de sincronización de alto nivel que auna el acceso en exclusión mutua a los recursos compartidos y mecanismos de espera de los procesos en competencia. A pesar del paso del tiempo, esta primitiva sigue siendo la más utilizada en los lenguajes concurrentes de alto nivel.

En la primera mitad de los años 70 varios trabajos de Dijkstra persiguieron un claro objetivo: el diseño de un lenguaje de programación cuya implementación permitiera potencialmente un alto grado de concurrencia, aunque sin imponerla. Es entonces cuando Dijkstra incorpora a los modelos de procesos dos propiedades fundamentales: el *no determinismo*, la posibilidad de especificar ejecuciones no deterministas, es decir, elecciones potencialmente no deterministas entre distintas componentes del programa [Dij75a]—sin duda una forma semántica de plasmar la asunción de independencia de las velocidades de los procesos; y la *sincronización* entre procesos mediante el uso de canales, imponiendo una notación que aún hoy es ampliamente utilizada: “!” para enviar y “?” para recibir datos a través de un canal.

From the past [...] it was the purpose of our programs to instruct our machines: now it is the purpose of the machines to execute our programs. Whether the machine does so sequentially, one thing at a time, or with

a considerable amount of concurrency, is a matter of implementation and should not be regarded as a property of the programming language. In the years behind us we have carried out this program of non-operational definition of semantics for a simple programming language that admits (trivially) a sequential implementation; our ultimate goal is a programming language that admits (highly?) concurrent implementations equally trivially. The experiments described in this report are a first step towards that goal. [Dij75b]

Pero algo que el siempre visionario Dijkstra parece apuntar en los trabajos mencionados anteriormente y en la cita extraída de ellos, es que el clásico concepto de *computación sencuencial*, operaciones ejecutándose de forma consecutiva, puede ser generalizado por el de *interacción concurrente*, procesos que existen a la vez y que intercambian mensajes. Ésta es precisamente la tesis que defiende Robin Milner:

To handle concurrency, we should not merely add extra material to the languages and theories of sequential computing [...] we should limit ourselves to constructions which are essential for concurrency in its own terms; then indeed we can see sequential computing as a higher, and more specific, level of explanation. [Mil93]

De hecho, Milner sugiere algo aún más trascendente, que los modelos interactivos son un paradigma que no sólo recoge mejor la naturaleza de los sistemas informáticos, sino en general, la naturaleza del mundo físico.

Pero volviendo a los años 70 del pasado siglo, sería Hoare quien al final de la década recogiese y puliese las ideas de Dijkstra para publicar en *Communications of the ACM* el artículo *Communicating Sequential Processes*, en el que se expone claramente que un modelo de computación basado en procesos concurrentes y mecanismos de comunicación entre ellos permite expresar soluciones elegantes y sencillas a una gran cantidad de problemas interesantes de programación.

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises. [Hoa78]

Este artículo marca definitivamente la madurez de los conceptos que se asumen esenciales en los modelos de procesos concurrente: procesos secuenciales que se ejecutan en paralelo de forma independiente, posibilidad de no determinismo, tanto en el comportamiento global del sistema como en los propios procesos, y comunicación e interacción entre procesos. Sin embargo, aún quedaba mucho trabajo por hacer para poder encontrar los marcos matemáticos formales en los que poder expresar dichas



Figura 1.4: Robin Milner

ideas y que permitieran un estudio profundo de los procesos, de sus relaciones y de los problemas intrínsecos que surgían de la interacción, competencia o cooperación entre ellos.

1.4. Semánticas de Procesos

El paradigma de programación secuencial contaba desde comienzos de los años 70 con una semántica denotacional perfectamente definida dada por Dana Scott y Christopher Strachey [SS71]. En dicha semántica los programas eran modelados como funciones que transformaban la entrada recibida produciendo la correspondiente salida. Sin embargo, este modelo no servía para poder dar semánticas formales a los procesos concurrentes.

There are two paradigm shifts that need to be made, before a theory of parallel programs in terms of process algebra can be developed. First of all, the idea of a behaviour as an input/output function needed to be abandoned. A program could still be modelled by an automaton, but the notion of language equivalence is no longer appropriate. This is because

the interaction a process has between input and output influences the outcome, disrupting functional behaviour. Secondly, the notion of global variables needs to be overcome. [Bae05]

Surge entonces la gran cuestión: si no podemos utilizar funciones para descubrir el funcionamiento de los programas concurrentes, ¿entonces qué debemos utilizar? Como veremos, a lo largo de los años 80 se producirían notables avances en el desarrollo de las semánticas formales para procesos. Pese a ello, la búsqueda de respuestas a dicha pregunta continúa siendo el objetivo último de los trabajos de investigación más importantes en este campo.

Sin duda uno de los hitos más destacados en la historia de las semánticas de procesos es la publicación en 1980 del libro titulado *A Calculus of Communicating Systems* [Mil80] (CCS) a cargo de Robin Milner y prologado por Hoare. Este trabajo es uno de los que el jurado destacó en la carrera de Milner cuando en 1991 se le adjudicó el premio Turing.

En el texto de Milner de 1980 se presenta por primera vez un álgebra de procesos —o cálculo de procesos, como prefiere denominarlo el propio Milner—, completamente articulada con capacidad para especificar y razonar sobre sistemas concurrentes. Los procesos son definidos sintácticamente por medio de operadores algebraicos; se les dota de una semántica operacional estructurada —siguiendo el formalismo introducido por Gordon Plotkin que a la postre se ha convertido en un estándar [Plo81, Plo04]— utilizando sistemas de transiciones etiquetadas; se definen diversas nociones de equivalencias a partir de las definiciones operacionales de los procesos, justificando el interés de cada una de ellas; y se determina un conjunto de axiomas que permiten realizar manipulaciones algebraicas de los términos sin alterar su significado: se muestra una teoría ecuacional sobre términos sintácticos que es consistente y completa con respecto a la equivalencia definida sobre los términos operacionales.

Our calculus is founded in two central ideas. The first is observation; [...] two systems are indistinguishable if we cannot tell them apart without pulling them apart. We therefore give a formal definition of observation equivalence and investigate its properties. [...] Every interesting concurrent systems is built from independent agents which communicate, and synchronized communication is our second central idea. [Mil80]

Como destaca Milner en la cita anterior, la noción de equivalencia entre procesos es una de las componentes esenciales para definir una semántica e imprime gran parte del carácter de un determinado modelo de procesos. La formalización que utilizó Milner para decidir cuando dos procesos son equivalentes se basa en un juego de imitación mutua en el que cada proceso tiene que ser capaz de hacer lo que el otro hace. Sin saberlo, Milner estaba utilizando una noción muy especial de equivalencia, la bisimulación, que de forma independiente había sido formulada por otro investigador, David Park [Par81]. La elegante construcción matemática con la que Park define la equivalencia de bisimulación es el punto clave, pues proporciona una nueva y

atractiva forma de entender la equivalencia, facilitando además una técnica sencilla para demostrar dicha equivalencia entre procesos. Milner incorporaría la bisimulación en la versión mejorada y actualizada de su libro [Mil89] corrigiendo así, además, sutiles errores de formalización en su definición primigenia.

A lo largo de los años 80 aparecerían muchas otras álgebras de procesos. En 1985, tras una exitosa carrera en la que incluso contaba con el galardón del premio Turing obtenido en 1980, Hoare publicó su primer libro *Communicating Sequential Processes* [Hoa85] (CSP), con título homónimo a su famoso artículo de 1978 y prologado por Dijkstra. En principio, la aproximación de Hoare podría parecer bastante diferente a la de Milner, si bien en el fondo son claramente complementarias. Hoare partía también de una sintaxis algebraica para los procesos, pero no proporcionaba una descripción operacional, sino una semántica denotacional para los mismos. Al igual que Milner, definía un conjunto de axiomas que podían utilizarse para realizar cálculos algebraicos para probar la equivalencia entre procesos. También centrales en el trabajo de Hoare son las nociones de equivalencia utilizadas que estaban inducidas por las denotaciones de los procesos. Partiendo de la equivalencia de trazas, es decir, de las secuencias de acciones que un proceso puede ejecutar, se llega a la equivalencia de fallos, la menor equivalencia que preserva la distinción del deadlock, es decir, no puede igualar a un proceso que se bloquea con otro que no lo hace.

Posteriormente vendrían otras contribuciones, más generales en cierto sentido, pues no se limitaban a estudiar un álgebra concreta, sino que presentaban distintos modelos de formalizaciones de equivalencias entre procesos. La más notable de estas contribuciones es sin duda la de Matthew Hennessy en 1988, presentada por medio de su libro *Algebraic Theory of Processes* [Hen88]. En dicho trabajo se introducía la metodología conocida como *testing*, utilizando pruebas para definir las semánticas de los procesos. El testing consiste en observar las posibles reacciones de los procesos frente a una serie de pruebas, o tests; dos procesos son iguales si obtienen los mismos resultados al pasárselos todas las pruebas de una determinada familia de pruebas. También de obligada referencia es el libro de Jos Baeten y Peter Weijland *Algebra of Communicating Processes* [BW90], a partir de los trabajos de Jan Bergstra y Jan Willem Klop [BK84], donde se presentan diversos modelos puramente axiomáticos que consiguen caracterizar la equivalencia de bisimulación.

Todas las propuestas enumeradas definen lenguajes algebraicos de descripción de procesos. Las diferencias entre ellos estriban sobre todo en sutilezas y detalles acerca del manejo del no determinismo y del paralelismo. Estos aspectos se definen mediante distintos operadores que en cada lenguaje concreto pueden ofrecer pequeñas diferencias.

Concentrándonos en las equivalencias, es muy interesante comprobar cómo los trabajos pioneros del área establecieron dos nociones fundamentales, bisimulación y trazas/fallos, que delimitan por arriba y por abajo el marco en el que pueden establecerse el resto de las equivalencias entre procesos. Hoare —con la claridad que le caracteriza— lo expresa muy bien en el siguiente párrafo.

CCS makes many distinctions between processes which would be regarded as identical in this book. The reason for this is that CCS is intended to serve as a framework for a family of models, each of which may make more identifications than CCS but cannot make less. To avoid restricting the range of models, CCS makes only those identifications which seems absolutely essential. In the mathematical model of this book [CSP] we have pursued exactly the opposite goal —we have made as many identifications as possible, preserving only the most essential distinctions. [Hoa85]

Entre estas dos nociones esenciales de equivalencia —bisimulación y trazas— fueron apareciendo, en las dos últimas décadas del siglo XX, una gran variedad de nuevas equivalencias provenientes de nuevos cálculos y álgebras de procesos que pretendían explorar diversos aspectos de potencia distintiva y expresividad necesitados en múltiples aplicaciones. El trabajo más importante de catalogación y clasificación de semánticas de procesos fue llevado a cabo por Rob van Glabbeek como parte de su tesis doctoral [vG90a]. En dos artículo titulados *Linear time-branching time spectrum* [vG90b, vG93] recopiló las principales de estas equivalencias estableciendo una taxonomía basada en el poder de distinción de las mismas.

CAPÍTULO 2

Introducción

Dictum: All animals are equal, but some animals are more equal than others.

*Animal Farm
George Orwell*

El presente capítulo introduce las ideas y conceptos básicos utilizados para modelar el comportamiento de los procesos, sección 2.1. Sobre las descripciones de los comportamientos se construyen las relaciones semánticas. Las principales formas en las que se han definido las semánticas para los procesos se describen en la sección 2.2. Por otro lado, es obligado comentar los trabajos relacionados en el área de estudio comparativo de semánticas y que más directamente han influenciando y servido de inspiración para afrontar nuestra investigación, sección 2.3.

Este capítulo, como su nombre indica, pretende ser una introducción rigurosa y lo suficientemente completa pero en modo alguno pretende ser enciclopédico o exhaustivo. El lector interesado puede consultar alguno de los numerosos trabajos en los que se presenta una evolución histórica y detallada de las publicaciones o contribuciones más importantes del área, véanse por ejemplo [HDH02, Bae05, Ace03, vG97, San07]. De igual forma, los textos clásicos [Hoa85, Mil89, Hen88] incluyen, de una u otra forma, notas históricas sobre el desarrollo de las teorías, así como las relaciones con otras investigaciones.

2.1. Definición de los Procesos

La Teoría de Procesos surgió en la segunda mitad de la década de los 60 y, sobre todo en la década de los 70, a partir de los trabajos germinales de Edsger Wybe Dijkstra [Dij65, Dij75a, Dij75b] y Charles Antony Richard Hoare [Hoa78].

Los modelos de procesos nacen de la necesidad de estudiar los comportamientos de los sistemas concurrentes, en los que varias entidades evolucionan a la vez comunicándose entre sí.

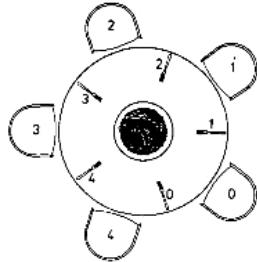
Process algebra is the branch of Computer Science which studies mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artefacts, embodied perhaps in computer hardware or software systems. [Hoa05]

Como subraya la anterior cita de Hoare, uno de los aspectos más atractivos de esta teoría es que plantea un modelo en el que poder expresar y estudiar multitud de problemas provenientes de muy distintos ámbitos: sistemas operativos, máquinas interactivas, todo tipo de protocolos, reacciones químicas, biología celular... En un último término, los procesos constituyen un modelo para toda actividad en nuestro mundo, un mundo en el que el trabajo concurrente y la comunicación son sin duda elementos centrales.

Como defiende Robin Milner, frente al modelo primitivo Turing-Newman —un modelo ideado para el cálculo— los procesos plantean un modelo para la informática entendida en un sentido mucho más amplio, es decir, para los sistemas interactivos.

5.3 Dining Philosophers (Problem due to E.W. Dijkstra)
Problem: Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks (see Figure 1). On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. The room should keep a count of the number of philosophers in it.

Fig. 1.



Solution: The behavior of the i th philosopher may be described as follows:

```
PHIL = *|... during ith lifetime ... →
    THINK;
    roomenter();
    fork(i)!pickup(); fork((i + 1) mod 5)!pickup();
    EAT;
    fork(i)!putdown(); fork((i + 1) mod 5)!putdown();
    roomexit()
]
```

Communications
of
the ACM

August 1978
Volume 21
Number 8

The fate of the i th fork is to be picked up and put down by a philosopher sitting on either side of it

```
FORK =
  •phil(i)?pickup() → phil(i)?putdown()
  •phil((i - 1) mod 5)?pickup() → phil((i - 1) mod 5)?putdown()
]
```

The story of the room may be simply told:

```
ROOM = occupancy:integer; occupancy = 0;
  •(i:0..4)phil(i)?enter() → occupancy = occupancy + 1
  •(i:0..4)phil(i)?exit() → occupancy = occupancy - 1
]
```

All these components operate in parallel:

```
[room:ROOM||fork(r:0..4):FORK||phil(r:0..4):PHIL].
```

Notes: (1) The solution given above does not prevent all five philosophers from entering the room, each picking up his left fork, and starving to death because he cannot pick up his right fork. (2) Exercise: Adapt the above program to avert this sad possibility. Hint: Prevent more than four philosophers from entering the room. (Solution due to E. W. Dijkstra).

Figura 2.1: Ejemplo de formalización utilizando la notación de CSP de 1978.

La idea de proceso, como descripción de un comportamiento que puede ser descompuesto en subprocesos que operan concurrentemente e interactúan entre sí y con el entorno, es relativamente sencilla y natural. Las nociones esenciales sobre las que se asientan las definiciones de comportamientos concurrentes, como el no determinismo o la comunicación entre procesos, fueron ya identificadas y utilizadas en los trabajos fundacionales —aún intuitivos o informales, en cierta manera, al carecer de una semántica formal— aparecidos a lo largo de los años 70 de la mano de Dijkstra y Hoare, véase la sección 1.3, en la página 8.

En la figura 2.1 —página 20— podemos ver la sintaxis utilizada por Hoare, en su artículo *Communicating Sequential Processes* de 1978, para describir el comportamiento de los procesos involucrados en el famoso problema *The dining philosophers* (la cena de los filósofos), propuesto por Dijkstra [Dij71]. Puede apreciarse la cercanía de notación con respecto a los lenguajes de programación imperativa —como Algol y Pascal— en cuyo diseño y desarrollo tanto había colaborado Hoare.

En los desarrollos posteriores de álgebras y cálculos de procesos, las definiciones de los procesos maduraron y se hicieron más completas y detalladas. Se incorporaron operadores que permitían expresar todo tipo de características: tiempo real, probabilidades, prioridades, interrupciones, etc. Pero al mismo tiempo, las definiciones se hicieron más abstractas y formales. Se definieron todo tipo de semánticas formales para dar significado a los procesos sintácticos. Diferentes autores construyeron diversas semánticas siguiendo distintas aproximaciones, véase una introducción en la sección 1.4, en la página 12. Todo esto, junto con la la versatilidad de los procesos para especificar y modelar diversas aplicaciones, ha provocado que el desarrollo de la teoría de procesos no haya seguido una línea única de investigación, sino que hayan surgido diversas alternativas que hacen hincapié en distintos detalles aún compartiendo indudablemente un núcleo común.

El primer modelo completo de procesos fue CCS, abreviatura para *Calculus of Communicating Systems*, que es el título del libro publicado por Robin Milner en 1980 [Mil80] y posteriormente mejorado —como comentaremos más adelante en la sección 2.2.1— en el texto de 1989 [Mil89]. Los objetivos que se propone este trabajo son claramente expuestos en su introducción:

A useful calculus, of computing systems as of anything else, must have a high level of articulacy in a full sense of the word implying not only richness in expression but also flexibility in manipulation. It should be possible to describe existing systems, to specify and program new systems, and to argue mathematically about them, all without leaving the notational framework of the calculus. [Mil80]

En dicho cálculo, los procesos son definidos sintácticamente por medio de operadores algebraicos. La siguiente definición utiliza la notación —más depurada— que aparece en [Mil89].

Definición 1 (Sintaxis básica de CCS) *Dado un conjunto de nombres de nombres*

$A = \{a, b, c, \dots\}$, que induce el conjunto de conombres $\bar{A} = \{\bar{a}, \bar{b}, \bar{c}, \dots\}$, las acciones de los procesos son elementos del conjunto $Act = A \cup \bar{A} \cup \{\tau\}$, donde τ es una acción especial, la acción interna —acción silenciosa o perfecta, según el propio Milner. La definición sintáctica de los procesos en CCS utiliza, además del proceso inactivo 0 , cinco operadores básicos:

- Prefijo de una acción y un proceso, $\alpha.P$
- Elección entre procesos, $P + Q$
- Composición de procesos, $P | Q$
- Restricción de acciones en un proceso, $P \backslash \{l_1, \dots, l_n\}$
- Reetiquetado de acciones, $P[l'_1/l_1, \dots, l'_n/l_n]$

La figura 2.2 ilustra el uso de la sintaxis de CCS para describir un sistema interactivo elemental y ampliamente utilizado como ejemplo: una máquina expendedora. Puede apreciarse cómo la sintaxis de los procesos se aleja de la estética convencional de los lenguajes de programación para ocupar más el terreno de un lenguaje descriptivo. De hecho, una de las principales virtudes de las álgebras de procesos es que sirven tanto para especificar como para implementar sistemas.

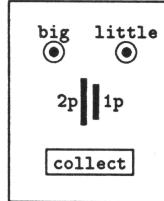
Uno de los principales objetivos de Milner al diseñar CCS era conseguir un conjunto reducido y básico de operadores primitivos capaz de describir por medio de su combinación algebraica todos los comportamientos posibles, así como todos los fenómenos característicos que surgen al estudiar las interacciones de procesos concurrentes. Esta reducción de las construcciones sintácticas a las meramente imprescindibles es un objetivo fundamental en muchos estudios teóricos, pues no sólo simplifica el trabajo en las demostraciones sino que facilita la comprensión de los sistemas sin afectar a la expresividad del modelo. Milner demostró, por ejemplo, cómo la noción de comunicación, que involucra el paso de valores entre procesos, podía ser modelada con el concepto más sencillo de sincronización, en la que no se intercambian valores de manera explícita.

... it turns out in our calculus that synchronization and summation, working together, give the power to express the communication of values of any kind! [Mil89]

Pero además de la descripción sintáctica, para tener una clara representación del comportamiento de los procesos, Milner consideraba esencial la descripción operacional de los mismos, pues sobre esta definición definiría su noción de equivalencia, y con ello la semántica definitiva del modelo.

El formalismo que hoy se conoce como semántica operacional estructurada fue introducido por Gordon D. Plotkin y publicado en 1981 en unas notas tituladas *A Structural Approach to Operational Semantics*. La idea detrás de toda semántica operacional es proporcionar un formalismo en el que describir el comportamiento detallado de los sistemas a estudiar.

Now let us consider a different kind of system: a vending machine. Here (with thanks to Tony Hoare) is a picture of a machine for selling chocolates:



We suppose that a big chocolate costs 2p, a little one costs 1p, and only these coins can be used. One natural way to define the vending machine, V , is in terms of its interaction with the environment at its five ports ($2p$, $1p$, big , little and collect), as follows:

$$V \stackrel{\text{def}}{=} 2p.\text{big.collect}.V + 1p.\text{little.collect}.V$$

This means, for example, that to buy a big chocolate you must put in a 2p coin, press the button marked ‘big’, and collect your chocolate from the tray. Note already some interesting points:

- There are no parameters involved in any of these actions.
- The machine’s behaviour is quite restrictive; it will not let you pay for a big chocolate with two 1p coins, or put in more money before you’ve collected your purchase.

Figura 2.2: Una máquina expendedora como aparece en [Mil89].

Clearly systems have some behaviour and it is that which we wish to describe. In an operational semantics one focuses on the operations the system can perform —whether internally or interactively with some supersystem or the outside world. For in our discrete (digital) computer system behaviour consists of elementary steps which are occurrences of operations. Such elementary steps are called here, (and also in many other situations in Computer Science) transitions (= moves). [Plo81, Plo04]

Los sistemas de transiciones abstraen la distinción entre programas y datos utilizando la noción de transición para indicar cualquier cambio en el estado global del sistema. Este tipo de notación había sido ya utilizada en teoría de autómatas y lenguajes formales, pero la gran aportación de Plotkin consistió en definir las transiciones por medio de reglas estructuradas en las que la combinación de operandos por medio de los operadores sintácticos del lenguaje queda descrita a partir de la combinación de los comportamientos de los operandos. De esta forma, es posible describir con total precisión y economía de notación el funcionamiento de los sistemas.

Aunque originalmente las ideas de Plotkin no se presentaron en el contexto de la teoría de procesos, esta manera de proporcionar un significado a los términos sintácticos se adaptaba perfectamente a la definición algebraica utilizada para describir procesos: cada término es construido de forma estructurada a partir de las constantes y utilizando los operadores de la signatura que define el lenguaje. Por otro lado, los sistemas de transiciones etiquetados proporcionan un modelo que refleja inmediatamente muchas de las propiedades importantes de los procesos: las posibilidades de evolución y los cambios de estado que éstas llevan, las comunicaciones que un proceso es capaz de ofrecer en un determinado momento, el no determinismo...

Milner tuvo muy claro que proporcionar una semántica operacional para sus procesos era al mismo tiempo sencillo y práctico. La semántica operacional que utiliza para describir su CCS básico apenas ocupa media página y es recogida en la siguiente definición.

Definición 2 (Semántica operacional de CCS) *La figura 2.3 recoge las reglas de la semántica operacional de CCS para los operadores de prefijo (Act), elección (Sum_j), composición (Com₁, Com₂ y Com₃), restricción (Res) y reetiquetado (Rel) que aparecen en la descripción sintáctica de los procesos de CCS—definición 1—, junto con la regla de definición de constantes (Con), para definir el comportamiento de un proceso por medio de ecuaciones.*

$$\begin{array}{ll}
 \textbf{Act} \quad \frac{}{\alpha.E \xrightarrow{\alpha} E} & \textbf{Sum}_j \quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j} \quad (j \in I) \\
 \\
 \textbf{Com}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} & \textbf{Com}_2 \quad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'} \\
 \\
 \textbf{Com}_3 \quad \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\bar{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'} \\
 \\
 \textbf{Res} \quad \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L} \quad (\alpha, \bar{\alpha} \notin L) & \textbf{Rel} \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \\
 \\
 \textbf{Con} \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{\text{def}}{=} P)
 \end{array}$$

Figura 2.3: Reglas de la semántica operacional de CCS como aparecen en [Mil89].

La semántica operacional permite construir, a partir de la definición sintáctica de un proceso, véase la definición 1, un árbol de transiciones que agrupa los posibles

comportamiento de dicho proceso. La figura 2.4 —página 25— muestra un ejemplo de árbol de derivaciones obtenido de [Mil89].

We are now in a position to state all the transitions which may occur in the system $(A|B)\backslash c$, and it is convenient to arrange them in a tree – an infinite tree in this case – which we shall call a *transition tree* or *derivation tree*:

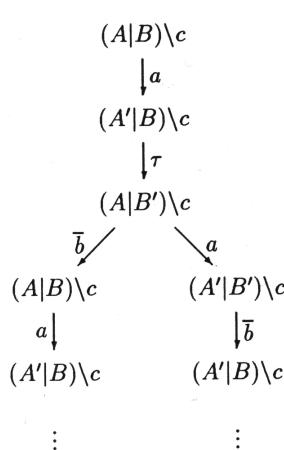


Figura 2.4: Ejemplo de árbol de derivaciones obtenido de [Mil89].

Los árboles de transiciones muestran explícitamente los posibles comportamientos de un proceso: las flechas indican los posibles cambios de estado y las etiquetas sobre ellas la acción o comunicación que se llevaría a cabo. Las ramificaciones indican diferentes evoluciones del comportamiento dependiendo de los estímulos recibidos desde el exterior, con la posibilidad de que aparezcan comportamientos no deterministas.

Una vez que los comportamientos de un proceso están definidos de esta manera Milner puede definir el punto clave de su teoría, su noción de equivalencia. Para ello intentará formalizar la idea natural de que dos procesos son distinguibles si la distinción puede ser detectada por un agente externo interactuando con ellos, continuaremos profundizando en esta definición de equivalencia en la siguiente sección 2.2 —página 32— y comprenderemos cómo la definición operacional de los procesos juega un papel fundamental en el trabajo de Milner.

Unos años después de la aparición de CCS, sería Hoare el que publicaría su libro titulado *Communicating Sequential Processes* [Hoa85], en el que definía un álgebra de procesos habitualmente abreviada por CSP. Si bien la motivación y la metodología que aparecen en el trabajo de Hoare siguen caminos que en apariencia difieren a

la propuesta de Milner, las diferencias no dan lugar a confrontación, sino más bien complementan el estudio sobre el mismo tema. El propio Hoare comenta las virtudes de CCS al comparar su trabajo con el de Milner:

It was an objective of CCS to achieve the maximum expressive power with as few distinct operators as possible. This is the source of the elegance and power of CCS, and greatly simplifies the investigation of families of models defined by different equivalence relations. [Hoa85]

Al igual que Milner, Hoare parte de una sintaxis algebraica para los procesos pero la descripción formal de los comportamientos no es operacional —por medio de sistemas etiquetados de transiciones— sino denotacional. En la descripción denotacional a cada proceso sintáctico se le asigna un significado en un determinado dominio, que suele tener elegantes propiedades matemáticas, entre ellas una noción de orden o equivalencia que induce la correspondiente relación en los procesos y que posibilita la definición de la semántica de los procesos recursivos utilizando la técnica de menor punto fijo.

Contrariamente al minimalismo sintáctico del CSS de Milner, Hoare en su CSP presentaba un elenco más amplio de operadores, que podían resultar útiles a la hora de especificar de manera compacta sistemas relativamente complejos, independientemente de cuales de ellos podrían considerarse como primitivos y cuáles derivados. Hay más de una docena de operadores sintácticos en las versiones estandard de CSP y la motivación para su inclusión tiene dos vertientes: por un lado, la riqueza del lenguaje surge para cubrir las necesidades expresivas de diseñadores e implementadores de sistemas complejos, Hoare mostraba así las posibilidades de las álgebras de procesos para ser utilizadas como un lenguaje tanto de especificación como de implementación; por otro lado, se podían aislar así las distintas características esenciales de los sistemas concurrentes para poder estudiarlas con mayor independencia las unas de las otras.

Simplicity is sought through design of a single simple model, in terms of which it is easy to define as many operators as seen appropriate to investigate a range of distinct concepts. For example, the nondeterministic choice \sqcap introduces nondeterminism in its purest form, and is quite independent of environmental choice represented by $(x : B \rightarrow P(x))$. Similarly, \sqcup introduces concurrency and synchronization, quite independent of nondeterminism or hiding, each of which is represented by a distinct operator. [Hoa85]

Para ilustrar la variedad de los operadores sintácticos de CSP que aparecen propuestos en [Hoa85], se ha incluido la figura 2.5, en la que aparecen algunos de estos operadores junto con una breve descripción informal del significado de cada uno de ellos.

La figura 2.6 —página 28— muestra nuevamente una solución al problema de los filósofos, véase la figura 2.1, esta vez utilizando la notación de CSP del año 1985.

Notación	Significado
$a \rightarrow P$	a entonces P
$(a \rightarrow P) b \rightarrow Q)$	a entonces P en elección con b entonces Q
$(x : A \rightarrow P(x))$	elección de x en A entonces $P(x)$
P / s	P tras ejecutar la traza s
$P \parallel Q$	P en paralelo con Q
$P \sqcap Q$	P o Q no determinista
$P \sqcup Q$	P en elección con Q
$P \setminus C$	P sin C , ocultamiento
$P \parallel\!\!\parallel Q$	P en interleaving con Q
$P \gg Q$	P encadenado a Q
$P // Q$	P subordinado a Q
$P; Q$	P seguido de Q
$P \nless b \ntriangleright Q$	P si b en otro caso Q
$*P$	repite P
$b * P$	mientras b repite P
$l : P$	P con nombre l

Figura 2.5: Algunos operadores sintácticos de CSP.

Hoare no define expresamente una semántica operacional para sus procesos sintácticos, pues según él, la representación pictórica —árboles de transición— plantea algunos problemas —como la multiplicidad de representaciones y la posible extensión de los dibujos— que su aproximación denotacional no tiene. Sin embargo, a lo largo de su libro, de forma puntual, sí que utiliza el equivalente a lo que Milner definía como árboles de derivación para ilustrar el comportamiento de algunos procesos. Incluso, no puede evitar el mostrar la estrecha relación existente entre su modelo denotacional más sencillo, las trazas, y la representación arbórea de los procesos.

There is a close relationship between the traces of a process and the pictures of its behaviour drawn as a tree. For any node on the tree, the trace of the behaviour of a process up to the time when it reaches that node is just the sequence of labels encountered on the path leading from the root of the tree to that node. [Hoa85]

El que Hoare no proponga una semántica operacional estructurada para su lenguaje de procesos es una cuestión estética y metodológica, pero dicha semántica puede perfectamente describirse. De hecho, el padre de las semánticas operacionales estructuradas, Gordon Plotkin, publicó en el año 1983 un artículo [Plo83] —considerado todo un clásico— en el que proponía una semántica operacional para el CSP que Hoare describía en su artículo de 1978 [Hoa78]. De igual forma, poco después de la aparición del libro de Hoare de 1985, la publicación [RBW86], encabezada por William Roscoe —uno de los doctorandos de Hoare y que había colaborado en la elaboración

Apart from thinking and eating which we have chosen to ignore, the life of each philosopher is described as the repetition of a cycle of six events

$$\begin{aligned} PHIL_i = & (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow i.picks\ up\ fork.(i+1) \rightarrow \\ & i.puts\ down\ fork.i \rightarrow i.puts\ down\ fork.(i+1) \rightarrow \\ & i.gets\ up \rightarrow PHIL_i) \end{aligned}$$

The rôle of a fork is a simple one; it is repeatedly picked up and put down by one of its adjacent philosophers (the same one on both occasions)

$$\begin{aligned} FORK_i = & (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i \\ & |(i\ominus 1).picks\ up\ fork.i \rightarrow (i\ominus 1).puts\ down\ fork.i \rightarrow FORK_i) \end{aligned}$$

The behaviour of the whole College is the concurrent combination of the behaviour of each of these components

$$\begin{aligned} PHILOS &= (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4) \\ FORKS &= (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4) \\ COLLEGE &= (PHILOS \parallel FORKS) \end{aligned}$$

An interesting variation of this story allows the philosophers to pick up their two forks in either order, or put them down in either order. Consider the behaviour of each philosopher's hand separately. Each hand is capable of picking up the relevant fork, but both hands are needed for sitting down and getting up

$$\begin{aligned} \alpha LEFT_i &= \{i.picks\ up\ fork.i, i.puts\ down\ fork.i, \\ &\quad i.sits\ down, i.gets\ up\} \\ \alpha RIGHT_i &= \{i.picks\ up\ fork.(i+1), i.puts\ down\ fork.(i+1), \\ &\quad i.sits\ down, i.gets\ up\} \\ LEFT_i &= (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow \\ &\quad i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i) \\ RIGHT_i &= (i.sits\ down \rightarrow i.picks\ up\ fork.(i+1) \rightarrow \\ &\quad i.puts\ down\ fork.(i+1) \rightarrow i.gets\ up \rightarrow RIGHT_i) \\ PHIL_i &= LEFT_i \parallel RIGHT_i \end{aligned}$$

Synchronization of sitting down and getting up by both $LEFT_i$ and $RIGHT_i$ ensures that no fork can be raised except when the relevant philosopher is seated. Apart from this, operations on the two forks are arbitrarily interleaved.

Figura 2.6: Formalización del problema de los filósofos que aparece en [Hoa85].

de CSP [BHR84]— proponía una semántica operacional sobre sistemas etiquetados de transiciones para el lenguaje descrito en el libro. De esta forma se mostraba que cualquier proceso sintáctico de CSP puede expresarse como un árbol de derivación —al estilo de Milner— en el que tenemos estados y transiciones etiquetadas que surgen de dichos estados y van hacia otros estados. Además, en dicho modelo operacional es posible definir órdenes y equivalencias que inducen las mismas clases de procesos que generan las relaciones definidas a través del modelo denotacional que caracteriza a CSP.

A finales de la década de los 80 aparecería un tercer texto que se ha convertido en una referencia obligada dentro de la teoría de procesos. Matthew Hennessy publicaba en 1988 el libro titulado *Algebraic Theory of Processes* [Hen88]. En este trabajo, el autor se beneficia de la perspectiva que ofrecen los numerosos trabajos de investigación publicados en el área durante la década, a los que él mismo contribuyó con aportaciones muy destacadas [DH83, HM85, HS85]. Una de las grandes virtudes del libro de Hennessy es que identifica y unifica metodologías, conceptos y técnicas que en ese tiempo habían alcanzado su madurez.

Su propuesta puede calificarse de integradora teniendo un vínculo entre las aproximaciones de Milner y Hoare. Hennessy utiliza semántica operacional como en CCS, semántica denotacional como en CSP y, por supuesto, la semántica axiomática que ambos trabajos previos consideraban esencial.

Para Hennessy la definición sintáctica concreta de los procesos no es tan importante, buena parte de la teoría de procesos puede desarrollarse de forma genérica utilizando álgebra abstracta: la sintaxis de un lenguaje corresponde con un álgebra de términos para una determinada firma que define los operadores del lenguaje.

La formalización de semánticas operacionales se había convertido en un estándar a la hora de describir el comportamiento de los procesos a finales de los 80. Al igual que Milner, Hennessy consideraba que estas semánticas son un punto de partida esencial, pues ofrecen una descripción detallada del comportamiento y, por tanto, de la manera en la que los procesos pueden ser utilizados.

Mucho más importante que el lenguaje sintáctico concreto utilizado en la descripción de un modelo de procesos es la noción de equivalencia semántica que se empleará para trabajar con dichos procesos.

Of course any particular model decides between identifying and distinguishing these pairs [of processes]. However the justification lies hidden in the definition of the model or in the mind of the designer of the model. We argue that this justification which underlies a model is its most important feature and should always be elucidated. Rather than starting with a model of processes, we develop a rationale for distinguishing of identifying them. [Hen88]

Con su *Algebraic Theory of Processes* Hennessy pretende presentar toda una metodología más que un modelo concreto. El punto clave y más novedoso de su propuesta es la manera de decidir cuándo dos procesos son iguales: siempre que pasen

las mismas pruebas, lo que se conoce como *testing*. Esta definición —que veremos en más profundidad en la sección 2.2.3— se basa en la descripción operacional de los procesos.

Pero central en el trabajo de Hennessy es tender los vínculos entre las tres formas clásicas de dotar de significado a los procesos, a partir de una definición de equivalencia basada en comportamientos, se pasa a un modelo denotacional *fully abstract* —equivalente al primero— y de ahí a una axiomatización de la relación expresada sobre los procesos sintácticos.

Su método de trabajo, bastante independiente del lenguaje, como hemos comentado, es puesto en práctica considerando diversos lenguajes de procesos que van adquiriendo, de forma gradual, mayor sofisticación en sus construcciones sintácticas: procesos no deterministas, recursivos, comunicantes... Desde un punto de vista sintáctico, su propuesta no ofrece ninguna innovación, utilizándose operadores que de una u otra forma ya habían aparecido en CCS y CSP: prefijo, elección, paralelismo, elección no determinista, restricción, etc. Desde el punto de vista de la representación de los comportamientos, Hennessy destaca el hecho de que el no determinismo resulta ser el verdadero centro de estudio de la teoría de procesos concurrentes al encargarse de modelar la simultaneidad.

...parallelism or concurrency is explained in terms of nondeterminism.

A parallel process is equivalent to a nondeterministic one obtained by interleaving the actions of its constituent subprocesses. This is also true of the semantics theories presented in Milner 1980, Hoare 1985, and related work. [Hen88]

Concluyendo, la descripción de más bajo nivel del comportamiento de los procesos puede realizarse utilizando sistemas de transiciones etiquetadas —árboles de derivaciones— que reflejan las posibilidades de interacción que un proceso tiene en un determinado momento con el entorno y la evolución que experimentará si realiza una u otra acción, véase la figura 2.7. En este formalismo el no determinismo potencial se expresa de forma natural: dos o más ramas que sale de un nodo etiquetadas del mismo modo; de igual forma, la asociatividad y la comutatividad de las posibilidades de evolución se reflejan en la carencia de estructura del conjunto de transiciones que parten de cada nodo.

Pero podemos dar también una definición formal de estas estructuras arbóreas. Dado un conjunto de etiquetas *Act* que representan todas las posibles acciones que pueden ejecutar los procesos, existe una firma muy sencilla $\Sigma = \{\mathbf{0}, a \in Act, +\}$ y un conjunto de ecuaciones $E = \{x + y = y + x, x + \mathbf{0} = x, x + (y + z) = (x + y) + z\}$ —leyes del monoide comunitativo— cuya álgebra inicial coincide precisamente con las descripciones de todos los comportamientos posibles etiquetados con las acciones en *Act*, véase por ejemplo [Mil80]. De esta forma, podemos definir un álgebra de procesos que es capaz de describir de forma sintáctica todos los árboles de comportamiento finitos sobre un determinado alfabeto.

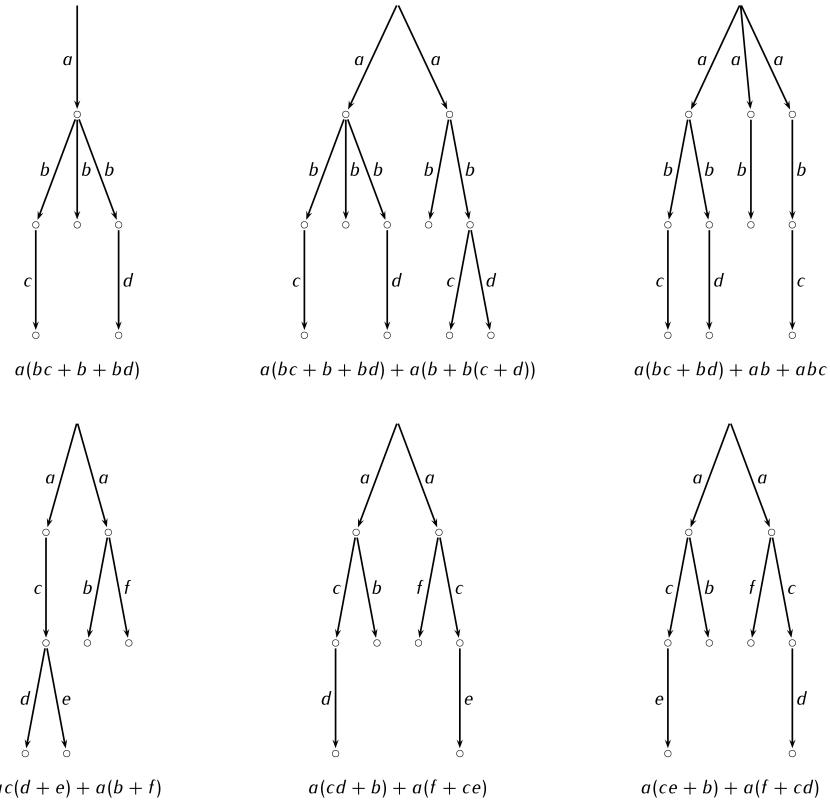


Figura 2.7: Descripción operacional y con sintaxis BCCSP de algunos procesos.

Definición 3 (Sintaxis de BCCSP) Dado un conjunto de acciones Act , el conjunto de los procesos $BCCSP$ se define de acuerdo con la siguiente gramática expresada en forma de Bakus-Naur.

$$p ::= \mathbf{0} \mid ap \mid p + q$$

donde $a \in Act$. $\mathbf{0}$ representa el proceso que no ejecuta ninguna acción; para cada acción en Act existe un operador de prefijo; $+$ es el operador de elección.

La semántica operacional de los términos $BCCSP$ es muy natural y se basa en tres principios:

- el proceso $\mathbf{0}$ no ejecuta ninguna acción;
- ap representa un proceso que puede ejecutar la acción a y transformarse, o evolucionar, en el proceso representado por p ;
- $p + q$ representa un proceso que puede comportarse como el proceso que representa p o como el que representa q .

Definición 4 (Reglas de la semántica operacional de BCCSP)

$$ap \xrightarrow{a} p \quad \frac{}{p + q \xrightarrow{a} p'} \quad \frac{p \xrightarrow{a} p' \quad q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'}$$

Con esta sencilla semántica se puede demostrar que el operador de elección $+$ es asociativo y conmutativo y tiene al $\mathbf{0}$ como elemento neutro. De esta forma, todos los árboles de comportamiento admiten una descripción sintáctica sencilla, véanse los ejemplos que aparecen en la figura 2.7.

$BCCSP$ ha sido ampliamente utilizada —en ocasiones con otros nombres, como BPA — en la literatura sobre teoría de procesos, véase por ejemplo [Hen88, vG01, dFG05, AFI07]. Estos trabajos están especialmente interesados en el estudio de las propiedades más fundamentales de las semánticas, en estos casos, los comportamientos de los procesos son habitualmente los objetos esenciales de estudio y el disponer de una representación sintáctica absolutamente *isomorfa* a la semántica perseguida permite mantener alejadas las interferencias que un modelo sintáctico no inicial introduciría.

2.2. Relaciones entre Procesos

En la sección anterior nos hemos centrado en introducir la descripción de los objetos de estudio en los que estamos interesados: los procesos concurrentes. Los procesos ofrecen una abstracción del comportamiento de los sistemas concurrentes, pero para estudiar estos sistemas, para intentar indagar en sus propiedades, en definitiva, para desarrollar una *teoría* sobre los mismos, se necesita algo más, una noción de significado más profunda que la mera enumeración de los comportamientos posibles.

Definir una relación de equivalencia, una noción de igualdad, entre los objetos de estudio es una de las formas más naturales y sencillas a nivel teórico de introducir una noción de significado: una equivalencia indica cuándo dos objetos pueden ser considerados como diferentes caras, diferentes representaciones, de una misma entidad conceptual.

Partiendo de una noción de equivalencia tendremos varias maneras de trabajar en la práctica con los procesos. En un marco compositivo como el que estamos considerando, en el que el comportamiento de un proceso es definido a partir del comportamiento de sus subprocesos, una equivalencia nos permite poder cambiar iguales por iguales. Si en un determinado sistema concurrente —por ejemplo un sistema de control de tráfico aéreo— compuesto por multitud de subsistemas —aeronaves, torre de control, radares, sistemas de enrutamiento...— deseamos actualizar, mejorar o cambiar uno de los componentes, podemos hacerlo de forma segura, es decir, manteniendo el mismo comportamiento global, siempre que el nuevo subsistema sea equivalente al anterior. Si esto ocurre, el comportamiento global se mantendrá inalterado. Esto ocurre exactamente igual en programación secuencial pues es una propiedad de los sistemas definidos estructuradamente.

Otra aplicación práctica de la equivalencia entre procesos se tiene cuando las descripciones que estamos considerando pueden ser interpretadas como especificaciones e implementaciones al mismo tiempo. La descripción de un proceso puede considerarse una implementación posiblemente por ser más detallada o estar formada por un cierto número de subsistemas trabajando concurrentemente. La especificación, en cambio, puede ser más sencilla, estableciendo las condiciones esenciales que rigen el funcionamiento del interfaz del proceso. En este caso, demostrar la equivalencia de comportamiento entre la supuesta implementación y la especificación refleja la corrección de la primera con respecto a la segunda. Combinando este uso con el anteriormente descrito, podríamos *verificar* un sistema utilizando las especificaciones de los subsistemas y posteriormente sustituir cada especificación por una implementación adecuada, manteniendo las mismas propiedades que la relación de equivalencia es capaz de discernir.

Todo lo dicho anteriormente es igualmente válido si definimos en nuestros procesos una relación de orden, en lugar de una equivalencia. En primer lugar, toda relación de orden induce una única relación de equivalencia, pero además, la relación de orden entre los procesos puede tener múltiples interpretaciones útiles a la hora de trabajar con ellos, reflejando propiedades tales como *ser una implementación de*, *ser más rápido que*, *ser más seguro que...*

En el marco de la programación secuencial —en cierta forma antecesora de la teoría de los procesos— encontramos un buen ejemplo de definición de significado. A comienzo de los años 70, Dana Scott y Christopher Strachey [SS71] propusieron un modelo en el que los programas eran representados como funciones que trasformaban una entrada produciendo una salida. A partir de esta idea, resulta muy natural indicar que dos programas son iguales si computan la misma función. Con estas premisas, su propuesta definía una semántica denotacional estructurada —compositivo— en la

que incluso las características más sofisticadas de los lenguajes secuenciales podían describirse con elegancia dentro del modelo matemático.

Pero desafortunadamente, para los procesos concurrentes, en los que no solo la *computación*, sino la *interacción* tienen importancia, la equivalencia de los programas secuenciales no tiene ni siquiera sentido.

Through the seventies, I became convinced that a theory of concurrency and interaction requires a new conceptual framework, not just a refinement of what we find natural for sequential computing. [Mil93]

La cita de Milner recoge el sentimiento general de las personas que investigaban los fundamentos de la teoría de procesos. Los años 80 abrirían una época dedicada sobre todo a la búsqueda de nuevos marcos que facilitaran la definición del significado de este tipo de objetos. Como veremos en las siguientes secciones, una noción de equivalencia —o de orden en algunos casos— es la clave a partir de la cual definir el significado, la semántica formal, de los procesos.

Pero, ¿qué alternativas tenemos a la hora de precisar cuando dos comportamientos son equivalentes? Consideremos la descripción de los procesos que propone la definición 4. Como se comentó en la sección anterior, esta descripción sintáctica puede equipararse a la representación arbórea de los procesos, figura 2.7. Seguramente no hay ningún contexto en el que resulte interesante decir que $a0$ y $aa0$ son procesos equivalentes, mientras que el primero puede ejecutar la acción a y evolucionar al proceso 0 , incapaz de toda actividad, el segundo puede ejecutar la acción a y llegar a un estado en el que es capaz de volver a ejecutar dicha acción. Si la acción a es una abstracción de $x = x + 1$ o de *recibe una moneda*, el significado de los dos procesos es sustancialmente diferente. Sin embargo, si consideramos los procesos $a0$ y $a0 + a0$, podemos plantearnos que no hay tanta diferencias entre ellos: los dos hacen exactamente lo mismo, pueden hacer la acción a y nada más. Este sencillo ejemplo ilustra cómo pueden surgir descripciones de comportamientos que queramos identificar. Debido al carácter composicional de los procesos, si consideramos que $a0$ y $a0 + a0$ son iguales, también los procesos $ba0$ y $b(a0 + a0)$ tendrían que ser iguales, y lo mismo ocurriría con $a0 + b0 + b0$ y $a0 + a0 + b0$. Existe una forma de expresar con precisión estas equivalencias que se seguirán a partir de otras de forma muy elegante: la lógica ecuacional.

Definición 5 (Lógica ecuacional) *Si se cuenta con una descripción sintáctica, como las que se comentaron en la sección 2.1 —CCS, CSP o BCCSP— y un conjunto E de ecuaciones —axiomas— que igualan términos, la equivalencia inducida por E es aquella en la que dos procesos son iguales si se puede deducir su igualdad utilizando las reglas que aparecen en la figura 2.8 y que corresponden a las propiedades de reflexividad, simetría y transitividad —propiedades obvias de la equivalencia—, las ecuaciones que pertenecen al conjunto E y la sustitución en cualquier contexto posible, en el caso de BCCSP los inducidos por los operadores de prefijo y elección.*

La notación habitual que se utiliza para indicar que dos procesos p y q son iguales a partir del conjunto de axiomas E es $E \vdash p = q$. Desde otro punto de vista, si una determinada relación de equivalencia coincide con la inducida por un conjunto de axiomas E_0 , se dice que los axiomas de E_0 *caracterizan* dicha equivalencia.

$$\begin{array}{lll}
 \text{Ref} & \frac{}{p = p} & \text{Sim} \quad \frac{p = q}{q = p} \\
 \text{Tran} & \frac{p = q \quad q = r}{p = r} & \text{Ecu} \quad \frac{p = q \in E}{p = q} \\
 \text{Sus}_1 & \frac{p = q}{ap = aq} & \text{Sus}_2 \quad \frac{p = q}{p + r = q + r}
 \end{array}$$

Figura 2.8: Reglas de la lógica ecuacional para BCCSP.

Si en lugar de una equivalencia quisiéramos utilizar este tipo de construcción para capturar el orden inducido por una serie de ecuaciones orientadas —o inecuaciones—, también podríamos hacerlo. En este caso, las reglas que se utilizaría serían básicamente las mismas que aparecen en la figura 2.8, pero adaptadas a las inecuaciones, con la excepción de la regla de simetría que tenemos que eliminar, pues no se verifica en las relaciones de orden. Esa nueva lógica se suele denominar *lógica inecuacional*.

Los axiomas son sin duda una buena forma de expresar de manera muy *condensada* una relación, ya sea de orden o de equivalencia. No es de extrañar por tanto que para casi todos los modelos de procesos se hayan intentado encontrar los axiomas que definen la equivalencia deseada. Sin embargo, el punto clave sigue siendo cuándo decidimos que dos procesos, a pesar de tener comportamientos o expresiones sintácticas diferentes, son equivalentes, en qué razones o motivaciones apoyamos nuestras decisiones. De eso tratan precisamente las siguientes secciones, no sólo se encargan de mostrar las definiciones de las equivalencias —u órdenes— más importantes definidas hasta la fecha entre procesos, sino que también se busca el analizar y destacar cuales han sido las metodologías de trabajo utilizadas, las ontologías de procesos que diferentes autores han considerado para derivar unas relaciones u otras.

2.2.1. Bisimulaciones y Simulaciones

En la sección anterior ya se han comentado muchas de las características del lenguaje CCS de Milner. En la definición 1 —página 21— aparece la descripción sintáctica de los procesos de CCS. Pero para Milner, son los sistemas de transiciones los que dan la verdadera información acerca del comportamiento de los procesos —véase la definición 2, en la página 24—, y es sobre esta descripción operacional del

comportamiento sobre la que desea definir su noción de equivalencia.

Es destacable que Plotkin en su trabajo del año 1981, en el que introducía las semánticas operacionales estructurales, se planteaba la cuestión de que un sistema de transiciones podía dar lugar a diversas *interpretaciones*, dependiendo de lo que decidísemos *observar* en él.

A point to watch is to make a distinction between internal and external behaviour. Internally a system's behaviour is nothing but the sum of its transitions. However externally many of the transitions produce no detectable effect. It is a matter of experience to choose the right definition of external behaviour. [Plo81]

Esta misma cuestión se la planteó Milner dentro del contexto de la teoría de procesos. Una vez definidos los comportamientos de los procesos concurrentes como sistemas de transiciones etiquetadas no deterministas, es necesario decidir cuándo dos comportamientos corresponden esencialmente al mismo proceso. Como la propiedad más característica de los procesos es su capacidad de interacción o comunicación con el exterior, la noción de equivalencia debería basarse en dicha capacidad.

We shall set up a notion of equivalence based between agents based intuitively upon the idea that we only wish to distinguish between two agents P and Q if the distinction can be detected by an external agent interacting with each of them. [Mil89]

Para profundizar en la comprensión de la interacción con el exterior, Milner propone una metáfora basada en interpretar los procesos como cajas negras con botones —uno por cada una de las acciones que pueden realizar— que pueden ser presionados o no dependiendo de si el proceso ofrece dicha acción en un determinado momento. Una vez que un botón se presiona el sistema evoluciona al siguiente estado dónde nuevamente algunos de los botones pasarán a estar activos y otros se bloquearán. Después de mostrar algunos ejemplos de procesos que deberían —al ser interpretados de esta forma— ser diferentes o iguales, propone la siguiente definición de equivalencia:

P and Q are equivalent iff, for every action α , every α -derivative of P is equivalent to some α -derivative of Q and conversely. [Mil89]

La primera formalización —texto de 1980— que Milner utilizó para la equivalencia entre sus procesos aparece en la definición de la figura 2.9.

Sin embargo, poco después de la aparición del *Calculus of Communicating Systems* en 1980, David Park publicó un artículo [Par81] en el que introducía la definición de la equivalencia de *bisimulación*. Esta nueva noción se adaptaba perfectamente al trabajo de Milner y definía una relación conceptualmente idéntica a la originalmente propuesta por éste, si bien mejoraba la forma matemática de presentar dicha equivalencia. Milner reconoció inmediatamente las ventajas de la bisimulación incorporándola

Definition (Observation equivalence) $p \approx_0 q$ is always true;

$$p \approx_{k+1} q \text{ iff } \forall s \in \Lambda^* \quad$$

- (i) if $p \xrightarrow{s} p'$ then for some q' , $q \xrightarrow{s} q'$ and $p' \approx_k q'$;
- (ii) if $q \xrightarrow{s} q'$ then for some p' , $p \xrightarrow{s} p'$ and $p' \approx_k q'$;

$$p \approx q \text{ iff } \forall k \geq 0. \quad p \approx_k q \quad (\text{i.e. } \approx = \bigcap_k \approx_k)$$

Figura 2.9: Definición de equivalencia entre procesos CCS en [Mil80].

a su cálculo. Posteriormente, en 1989 publicaría un nuevo libro —bajo el título de *Communication and Concurrency* [Mil89]— en el que presentaba una versión actualizada y mejorada de su CCS y en el que la bisimulación se convertía en una de las piedras angulares de su teoría.

La siguiente definición parafrasea a la que aparece en el texto de Milner de 1989 pero, para hacerla más inteligible, aquí ha sido ligeramente adaptada a la notación sintáctica de BCCSP que se ha presentado en detalle en la sección 2.1.

Definición 6 (Bisimulación) Una relación binaria $S \subseteq BCCSP \times BCCSP$ sobre procesos es una bisimulación si $(p, q) \in S$ implica que para todo $a \in Act$,

- Siempre que $p \xrightarrow{a} p'$ entonces, existe q' , $q \xrightarrow{a} q'$ y $(p', q') \in S$,
- Siempre que $q \xrightarrow{a} q'$ entonces, existe p' , $p \xrightarrow{a} p'$ y $(p', q') \in S$.

A parte de que esta última definición de la equivalencia por bisimulación es matemáticamente más elegante que la que aparece en la figura 2.9, desde un punto de vista práctico, la bisimulación posibilita una nueva técnica de demostración de dicha equivalencia. Mientras que para probar que dos procesos son equivalentes — $p \sim q$ según la notación de [Mil80]— hay que probar que $p \sim_k q$ por inducción sobre k , para probar $p \sim q$ por bisimulación basta con presentar una relación que contenga al par (p, q) y demostrar que es una bisimulación en el sentido de la definición 6, comprobando para ello que cada uno de sus pares cumple las condiciones de dicha definición.

La bisimulación es un ejemplo de definición coinductiva y el método de demostración por bisimulación es una instancia del método de demostración por coinducción. Intuitivamente, un conjunto definido coinductivamente es la mayor solución de una determinada ecuación, el principio de prueba coinductivo dice que cualquier otra solución a dicha ecuación está contenida en la mayor solución. Estas ideas son duales a las mejor conocidas de inducción: un conjunto definido inductivamente es la menor solución de una determinada ecuación, el principio de inducción dice que cualquier otro conjunto que resuelva dicha ecuación contiene a la menor solución.

Como comentamos en la sección 2.2, la descripción por medio de axiomas de una relación permite condensar mucho sus propiedades. Queda reflejado en la definición 5 que la caracterización axiomática de una relación depende de manera crítica de la sintaxis de los procesos que se estén considerando. Con respecto al lenguaje BCCSP, el conjunto de axiomas que aparece en la figura 2.10 caracteriza la equivalencia de bisimulación es decir dos proceso p y q son bisimilares si y solo si podemos derivar su igualdad en el sistema ecuacional inducido por dichos axiomas, $\{B_1, B_2, B_3, B_4\} \vdash p = q$.

$$\begin{array}{ll} (B_1) & x + y = y + x \\ (B_2) & (x + y) + z = x + (y + z) \\ (B_3) & x + \mathbf{0} = x \\ (B_4) & x + x = x \end{array}$$

Figura 2.10: Axiomas que caracterizan la bisimulación para el lenguaje BCCSP.

Aunque el conjunto de axiomas $\{B_1, B_2, B_3, B_4\}$ es el estrictamente necesario a nivel algebraico, el axioma que realmente da la clave para entender el poder de la equivalencia de bisimulación es (B_4) $x + x = x$. El resto de los axiomas $\{B_1, B_2, B_3\}$ caracterizan la equivalencia de árboles no ordenados, que son los sistemas de transiciones que utilizamos para describir el comportamiento de los procesos.

Aparte de esta caracterización axiomática, la bisimulación tiene muchas otras presentaciones alternativas, lo que indudablemente guarda relación con la tremenda importancia que el concepto posee y la variedad de contextos en los que aparece. De forma genérica, podemos decir que la bisimulación es la forma adecuada de describir la igualdad entre objetos definidos por una coalgebra. En los últimos años han aparecido una gran cantidad de trabajos que están explorando el terreno de las semánticas en un marco coalgebraico [HJ04, KS03, Rut03, BG03, Kli04, Jac04]. Un excelente repaso a la historia y a las aplicaciones de la bisimulación y la coinducción puede encontrarse en [San07].

Entre las caracterizaciones de la bisimulación destaca la expresada en términos de una lógica bastante sencilla y natural, conocida como Hennessy–Milner Logic (HML) [HM85]. También es muy útil interpretar la bisimulación como un juego [Sti98, dFG06], lo que proporciona una metáfora muy visual: en el juego el *atacante* trata de probar la no equivalencia en un número finito —pero no acotado a priori— de pasos, el *defensor* trata de evitar que el contrario gane; si el atacante tiene una estrategia para ganar los procesos no son equivalentes y la estrategia muestra un contraejemplo de cómo se contaría las condiciones que definen la bisimulación. Por el contrario, si el defensor tiene una estrategia ganadora el desplegado de la misma constituye de hecho la relación de bisimulación que prueba la equivalencia de los dos procesos.

Por último, para completar con este rápido vistazo a las virtudes de la bisimu-

lación, indicar que existen algoritmos eficientes [PT87, KS90] que han permitido la implementación de herramientas [CPS93] que pueden comprobar la equivalencia de bisimulación sobre sistemas de tamaño bastante grande.

Sin embargo, a pesar de las muchas ventajas que la bisimulación ofrece, también ha sido criticada por ser una equivalencia demasiado fuerte, que sólo identifica procesos que son *muy iguales*. De hecho, muchas otras equivalencias para procesos han sido propuestas casi todas ellas más débiles que la bisimulación. En las secciones siguientes veremos algunas de ellas.

Para terminar esta sección nos centraremos en otras relaciones definidas sobre la descripción operacional de los procesos que aparecen en la literatura. La definición de simulación, parecida a la de la bisimulación, también es frecuente en la literatura y como indica Davide Sangiorgi [San07] ha sido utilizada en diversos trabajos por autores como Milner y Hoare entre otros. Desde un punto de vista matemático, la simulación está emparentada con el concepto algebraico de homomorfismo.

Definición 7 (Simulación) Una relación binaria $S \subseteq BCCSP \times BCCSP$ sobre procesos es una simulación si $(p, q) \in S$ implica que para todo $a \in Act$,

$$\text{Siempre que } p \xrightarrow{a} p' \text{ entonces, existe } q', q \xrightarrow{a} q' \text{ y } (p', q') \in S.$$

En términos simbólicos, la definición de la simulación es *media* bisimulación. Mientras que la bisimulación entre dos procesos se puede explicar informalmente como la capacidad mutua de imitar cada uno el comportamiento del otro, en la simulación hay un proceso que realiza acciones y otro que trata simplemente de imitarlo.

Pero a pesar de la proximidad en la definición, la simulación induce una relación muy alejada a la de la equivalencia de bisimulación. En primer lugar, la simulación induce un orden entre procesos y, contrariamente a lo que se podría pensar a primera vista, el núcleo —la relación resultante de la intersección de una relación y de su inversa— de dicha relación de orden no coincide con la equivalencia de bisimulación, de hecho la equivalencia de simulación está muy por debajo del poder de distinción de la bisimulación y apenas por encima de la equivalencia de trazas. Es destacable el hecho de que no se conoce ningún preorden —no trivial— cuyo núcleo sea la equivalencia de bisimulación.

El preorden de simulación sobre procesos BCCSP coincide con el orden inducido por los el conjunto de axiomas $\{B_1, B_2, B_3, B_4, S\}$. Es decir, los axiomas de la equivalencia de bisimulación y el axioma de orden

$$(S) \quad x \sqsubseteq x + y.$$

Una relación de orden muy parecida a la simulación es la que se conoce con el nombre de *ready simulation*. Esta relación tiene una definición idéntica a la simulación, salvo que añade la obligación de comprobar en cada paso que los procesos relacionados por ella son capaces de realizar las mismas acciones iniciales. Una forma de describir dicha relación es la siguiente:

Definición 8 (Ready simulación) Una relación binaria $S \subseteq BCCSP \times BCCSP$ sobre procesos es una ready simulación si $(p, q) \in S$ implica que para todo $a \in Act$,

- Siempre que $p \xrightarrow{a} p'$ entonces, existe q' , $q \xrightarrow{a} q'$ y $(p', q') \in S$,
- Además, si $q \xrightarrow{a}$ entonces $p \xrightarrow{a}$.

Como hemos comentado antes de la bisimulación y la simulación, también la ready simulación ha aparecido en diversos contextos. Por un lado en el trabajo original de Bard Bloom, Sorin Istrail y Albert Meyer [BIM88] en el que se buscaba una relación de equivalencia que fuera una congruencia con respecto a una amplia familia de operadores sintácticos que extendían a CCS, propiedad que no verificaba la bisimulación. También, de forma independiente, en el trabajo acerca de semánticas probabilísticas de procesos de Kim Larsen y Arne Skou [LS91], con el nombre allí de $\frac{2}{3}$ -bisimulation. La ready simulación ha sido caracterizada con una lógica modal derivada de la Hennessy-Milner Logic [BM92, BIM95] y admite una justificación en un contexto de pruebas mediante cajas negras y botones —como el propuesto inicialmente por Milner para la bisimulación— mucho más plausible que el escenario necesario para justificar la equivalencia de bisimulación. Además, tiene un poder de distinción de procesos por debajo de la bisimulación, pero por encima de gran parte de las semánticas de procesos.

Es posible dar un conjunto de axiomas que definen sobre los procesos BCCSP el preorden de ready simulación, este conjunto es $\{B_1, B_2, B_3, B_4, RS\}$

$$(RS) \quad ax \sqsubseteq ax + ay.$$

La ready simulación ha jugado un papel muy importante en muchos estudios de teóricos de semánticas de procesos [BFN03, AFI07], así como en las investigaciones que se presentan en la tesis [GR09].

2.2.2. Trazas, Fallos y otras Denotaciones

La forma en la que Hoare se enfrenta el problema de dar una semántica formal a los procesos en su libro *Communicating Sequential Processes* es muy distinta a la que hemos comentado en el apartado anterior. Sin embargo, el punto de partida, la idea informal que motiva su investigación, es enunciada de forma muy parecida a la de Milner: tratar de *observar* los procesos desde fuera.

Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs. [Hoa85]

Hoare presenta los operadores sintácticos de su lenguaje CSP de forma gradual, movido por la necesidad de ir añadiendo elementos expresivos para modelar sistemas cada vez más complejos. La noción de equivalencia entre procesos sintácticos es introducida mediante leyes algebraicas —axiomas o ecuaciones— que *intuitivamente*

definen las propiedades de los operadores. Pero en cuanto el conjunto de leyes crece, llega un momento en el que la intuición no basta.

But the questions also arises, are these laws in fact true? Are they even consistent? Should there be more of them? Or are they complete in the sense that they permit all true facts about processes to be proved from them? Could one manage with fewer and simpler laws? These are questions for which an answer must be sought in a deeper mathematical investigation. [Hoa85]

Para poder responder con propiedad a las preguntas anteriores Hoare utiliza una técnica que se había empleado con éxito en el estudio de las semánticas de los lenguajes de programación: asignar a cada proceso sintáctico un objeto en un dominio matemático, lo que se conoce como una denotación.

Por extensión, a esta forma de dar significado a los términos sintácticos se la denomina semántica denotacional. Con esta técnica, dos descripciones sintácticas de procesos son equivalentes si se les asocia el mismo objeto denotacional. El dominio denotacional es un modelo matemático sobre el que el significado de los procesos se determina de manera composicional a partir del significado de los constituyentes. En la medida de lo posible, este modelo tiene que ser elegante y permitir métodos de prueba que pongan a nuestra disposición la potencia de las matemáticas para razonar sobre los procesos. En particular, la semántica denotacional debería ser una buena herramienta para poder probar las leyes o ecuaciones de la semántica axiomática que es la que describe la equivalencia entre procesos mediante la manipulación sintáctica de los términos.

Las denotaciones más elementales que se consideran son las trazas. Una traza de un proceso es simplemente una secuencia finita —lineal— de símbolos que recoge las acciones que el proceso ha llevado a cabo durante alguno de sus posibles cómputos parciales.

Definición 9 (Trazas de un proceso) *La definición de las trazas de un proceso se hace —siguiendo el estilo denotacional clásico— por inducción estructural, definiendo el valor de la denotación para cada uno de los operadores del lenguaje. Las trazas de los procesos BCCSP se definen como sigue:*

$$\begin{aligned}\text{traces}(0) &= \{\langle \rangle\} \\ \text{traces}(ap) &= \{\langle \rangle\} \cup \{\langle a \rangle^{\wedge} t \mid t \in \text{traces}(p)\} \\ \text{traces}(p + q) &= \text{traces}(p) \cup \text{traces}(q)\end{aligned}$$

Dónde $\langle \rangle$ denota la traza vacía y el operador $^{\wedge}$ representa la concatenación de cadenas de símbolos. Esta última notación es poco habitual pero se ha utilizado por ser la que aparece en [Hoa85].

A partir de esta definición, cualquier proceso sintáctico tiene una única denotación que corresponde al conjunto de trazas que somos capaces de observar. Partiendo de estas denotaciones podemos inducir relaciones entre los procesos: dos procesos son iguales si tienen el mismo conjunto de trazas, o un proceso es menor que otro si las trazas del primero están incluidas en las del segundo, esto es $p \sqsubseteq q$ si y solo si $\text{traces}(p) \subseteq \text{traces}(q)$.

Utilizando estas denotaciones es fácil probar que el conjunto formado por las ecuaciones $\{B_1, B_2, B_3, B_4, T_=\}$

$$(T_=) \quad ax + ay = a(x + y)$$

caracteriza la equivalencia de trazas sobre procesos BCCSP. Como es habitual en las descripciones axiomáticas de las semánticas que estamos considerando, este conjunto incorpora los axiomas que caracterizan a la bisimulación y, por tanto, todo par de procesos bisimilares es también equivalente en trazas.

El orden de trazas también admite una axiomatización natural definida por el conjunto de ecuaciones $\{B_1, B_2, B_3, B_4, T_-, T_\sqsubseteq\}$

$$(T_\sqsubseteq) \quad x \sqsubseteq x + y.$$

Sin embargo las observaciones que llevan a cabo las trazas no permiten distinguir procesos que tendrían que serlo. Por ejemplo, hay muchas razones para pensar que el proceso $a0 + ab0$ es distinto del proceso $ab0$, en particular, el primero puede quedarse detenido después de ejecutar la acción a , mientras que el segundo siempre realizará la acción b . Estos dos procesos, sin embargo, definen el mismo conjunto de trazas $\{\langle \rangle, \langle a \rangle, \langle a, b \rangle\}$.

Hoare necesita entonces sofisticar la observación que se realiza de los procesos: cuántos más detalles observemos más capacidad de distinción ganaremos. Por otro lado, esta observación debe ser lo más simple posible pues Hoare está interesando en encontrar una noción de equivalencia entre procesos que haga tantas identificaciones como sea posible conservando únicamente las distinciones más esenciales.

El resultado es que además de observar las trazas, el modelo puede observar las acciones que un proceso no es capaz de realizar en un determinado estado, lo que se denomina *refusals*.

The refusal set seems to be the weakest kind of observation that efficiently represents the possibility of nondeterministic deadlock; and it therefore leads to a much weaker equivalence, and to a more powerful set of algebraic laws than CCS. [Hoa85]

A partir de esta observación de las acciones que un proceso no puede realizar se construye la noción de fallo. Un fallo es un par (s, X) donde s es una traza y X un conjunto de acciones. Diremos que el par (s, X) es un fallo de p , si p puede ejecutar la traza de acciones s y entonces llegar a un estado en el que es posible rechazar —no ser capaz de hacer— cualquiera de las acciones en el conjunto X .

Como ocurría antes para las trazas, los fallos pueden ser definidos por inducción estructural sobre los procesos sintácticos.

Definición 10 (Fallos de un proceso) *Los fallos de los procesos BCCSP se definen como sigue:*

$$\begin{aligned}\text{failures}(0) &= \{(\langle\rangle, X) \mid \forall X \subseteq \text{Act}\} \\ \text{failures}(ap) &= \{(\langle\rangle, X) \mid \forall X \subseteq \text{Act} - \{a\}\} \cup \\ &\quad \{(\langle a \rangle^{\wedge} t, X) \mid (t, X) \in \text{failures}(p)\} \\ \text{failures}(p + q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \text{failures}(p) \cap \text{failures}(q)\} \cup \\ &\quad \{(t, X) \mid t \neq \langle\rangle, (t, X) \in \text{failures}(p) \cup \text{failures}(q)\}\end{aligned}$$

Ahora el conjunto de fallos de un proceso es la denotación del mismo y por tanto se pueden inducir relaciones de equivalencia y orden en los procesos a partir de la igualdad y la inclusión conjuntista, respectivamente.

Estas relaciones también cuentan con una caracterización axiomática. En particular, el conjunto $\{B_1, B_2, B_3, B_4, RS, F_{\sqsubseteq}\}$

$$(F_{\sqsubseteq}) \quad a(x + y) \sqsubseteq ax + a(y + z)$$

define el orden de fallos. Este conjunto incluye a los axiomas que definen el orden de ready simulación, definición 8.

Por su parte, la equivalencia de fallos puede ser axiomatizada considerando el conjunto —nada trivial— de ecuaciones $\{B_1, B_2, B_3, B_4, F_{\sqsubseteq}^1, F_{\sqsubseteq}^2\}$. Como siempre que hablamos de axiomatizaciones, teniendo en cuenta la sintaxis de los procesos BCCSP y las reglas de la definición 5 de la correspondiente lógica ecuacional.

$$\begin{aligned}(F_{\sqsubseteq}^1) \quad a(bx + u) + a(by + v) &= a(bx + by + u) + a(by + v) \\ (F_{\sqsubseteq}^2) \quad ax + a(y + z) &= ax + a(x + y) + a(y + z)\end{aligned}$$

Volviendo a las denotaciones de los procesos, como observa el propio Hoare, en lugar de decidir observar las acciones que un proceso rechaza, podríamos decidir observar aquellas que acepta. La semántica resultante es ligeramente diferente —tiene un poder de distinción mayor— a la de fallos y se conoce con el nombre de *readiness* [OH86].

La metodología propuesta en esta sección ha sido ampliamente utilizada en la búsqueda de semánticas para procesos y se han estudiado todo tipo de denotaciones, en general, asumiendo cada vez más poder de observación. Entre ellas destacamos dos *failure trace* y *ready trace*. Como su nombre permite adivinar, las denotaciones se obtienen de observar los fallos —respectivamente las aceptaciones— de los procesos a lo largo de cada uno de los puntos intermedios en cada una de las trazas que pertenecen a dichos procesos, en lugar de sólo al final.

Las relaciones de inclusión entre las clases de equivalencia que cada una de estas semánticas inducen en los procesos definen un *diamante* —figura 2.11—.

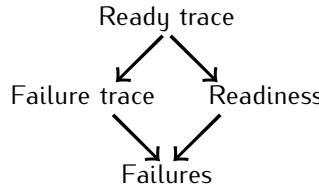


Figura 2.11: Orden de inclusión de las semánticas denotacionales más clásicas.

2.2.3. Testing

En las secciones previas se han expuesto las metodologías que los dos modelos más clásicos de semánticas de procesos —CCS y CSP— siguieron para dar una definición de la equivalencia entre procesos: mientras en CCS se utilizaba una descripción operacional y sobre ella se definía —de forma coinductiva— la relación de equivalencia, en CSP se proponía una descripción denotacional de los procesos y la relación de equivalencia era la inducida en el dominio matemático de las denotaciones.

En esta sección presentamos una tercera metodología empleada por Matthew Hennessy y extensamente expuesta en el libro titulado *Algebraic Theory of Processes* publicado en 1988 [Hen88]. La propuesta de Hennessy es a la vez integradora y novedosa: es integradora pues su modelo tiende un vínculo entre la definición operacional y la denotacional de las semánticas que utiliza; y también es novedosa pues vuelve a reinterpretar —una vez más— la idea esencial de lo que supone observar un proceso para decidir si está relacionado o no con otro.

Si repasamos las justificaciones que Milner y Hoare nos proponen para encontrar una forma razonable de definir la equivalencia entre procesos, encontramos que los dos utilizan palabras muy parecidas. A pesar de las grandes diferencias entre sus propuestas para la equivalencia, los dos toman como punto de partida la *observación* de los procesos.

Por otra parte, también en sus propuestas hay grandes similitudes a la hora de entender lo que es el contexto o entorno de un proceso. Subyacente a ambos modelos está el hecho de que la interacción es uno de los conceptos básicos que la teoría de procesos desean capturar e investigar, y por tanto, un contexto y un proceso *interactuando* tienen que poder ser modelados por la propia teoría que se quiere definir.

For we suppose that the only way to observe a system is to communicate with it, which makes the observer and system together a larger system. [Mil80]

In fact, it is best to forget the distinction between processes, environments and systems; they are all of them just processes whose behaviour may be prescribed, described, recorded and analysed in a simple and homogeneous fashion. [Hoa85]

Hennessy discute en profundidad lo que supone observar un proceso. Puesto que los procesos tienen comportamientos no lineales e incluso no deterministas, una única observación no basta. A una observación concreta la denomina prueba o experimento, un *test*. Un experimento está compuesto por el proceso que se quiere observar y por un *observador* concreto. Pero podríamos considerar múltiples observadores, cada observador, dependiendo de sus habilidades, podría comportarse de una determinada manera. Como los procesos tratan de definir comportamientos, el comportamiento del observador puede ser considerado nuevamente un proceso. De esta forma, si llevamos al extremo la tarea de observar un proceso desde todos los puntos de vista posibles, lo mejor es hacer un experimento con cada uno de los observadores posibles, de decir, con cada uno de los procesos posibles.

Uno de los puntos más delicados de esta metodología es la obtención de un resultado de la prueba realizada. Puesto que los procesos son no deterministas hay varias formas de interpretar cuándo una prueba es pasada por un proceso:

... testing gives rise to three different behavioural relations: one in terms of the tests a process always passes, one in terms of those it sometimes passes; and the third is simply a conjunction of the first two. [Hen88]

Entonces hay tres posibles nociones semánticas, las que se conocen como *must*, *may* y *may-must*, respectivamente con respecto a la cita. Estas interpretaciones no son tan arbitrarias como podría parecer y en realidad corresponden a los tres posibles dominios potencias, *powerdomains* [Plo76], de un retículo de dos puntos, que incluso cuentan con nombres propios: *must* se corresponde con el dominio potencia de Smyth, *may* con el de Hoare y el tercero se corresponde con el dominio de Egli-Milner, también llamado de Plotkin.

Decimos entonces que dos procesos son iguales bajo testing —en cada una de las tres formas— si el resultado de cada prueba coincide en ambos procesos. Es decir, si todos los experimentos posibles no consiguen encontrar ninguna diferencia entre ellos. De igual forma, también pueden definirse de manera natural las correspondientes relaciones de orden, pues un proceso será mejor que otro si éste pasa al menos todas las pruebas que aquél.

Desde el punto de vista técnico, la manera de definir estas ideas en un marco formal no es demasiado complicada, puesto que en todo momento se están utilizando los propios conceptos de la teoría de procesos. Hennessy utiliza, al igual que hacía Milner, una semántica operacional sobre sistemas etiquetados de transiciones. Los observadores, como hemos comentado antes, son también procesos; la única salvedad es que disponen de una acción especial que les permite indicar en qué momento el experimento ha terminado de forma exitosa. En consecuencia, la propia semántica operacional que describe el comportamiento de los procesos describe también lo que es un experimento.

En principio esta definición extensiva no es nada práctica pues el número de pruebas a realizar es potencialmente infinito. Sin embargo, desde el punto de vista

metodológico sí que es muy interesante, pues el concepto de equivalencia emana de la aplicación de los propios principios de interacción que se expresan dentro del modelo. Además, una cosa es la definición *natural* de la equivalencia y otra muy distinta es que tras elaborar una teoría propiamente dicha podamos expresarla de otras formas más sencillas o utilizando otras formulaciones.

De hecho, Hennessy propone otras dos formas alternativas de entender las definiciones de las relaciones que propone. Una de estas formas es mediante una semántica denotacional al estilo del CSP de Hoare y la segunda es una caracterización axiomática. De esta forma obtiene lo que denomina la *trinidad*, tres modelos para los procesos interconectados entre sí. Los procesos pueden verse como términos sintácticos de un álgebra, como descripciones operacionales en sistemas de transiciones y como objetos denotacionales en un modelo matemático. Sobre estas representaciones podemos ver la relación de igualdad entre dos procesos de forma operacional, mediante el paso de pruebas; de forma denotacional, en el dominio de los objetos que Hennessy llama árboles de aceptación; y definida mediante ecuaciones en un sistema de demostración sobre la forma sintáctica de los procesos.

Como ya comentamos en la sección 2.1, el modelo de testing hace más énfasis en la metodología que en el lenguaje concreto a utilizar.

The general idea of testing processes can be formalized in many different ways and can lead to a variety of interesting equivalences. [Hen88]

Las definiciones de equivalencias y órdenes entre procesos varían mucho dependiendo del lenguaje que consideremos pues la potencia expresiva del mismo afecta tanto a los procesos como a las pruebas, de hecho, encontrar un conjunto *minimal* de tests que tengan la misma potencia diferenciadora que el conjunto total suele ser un paso esencial intermedio para entender y desarrollar completamente —conseguir la trinidad— un determinado modelo de pruebas. En concreto, para los procesos que se presentan en [Hen88] la semántica must, quizás la más representativa del trabajo de Hennessy, coincide con la semántica de fallos de CSP.

Una versión de testing ligeramente diferente, pues puede observar no sólo las acciones que se pueden hacer, sino también las que se pueden rechazar, es presentada en [Phit87] con el nombre de *refusal equivalence*. Como se comenta en el trabajo [vG01] —que se tratará en detalle en la sección 2.3.3— la semántica inducida por esta versión de testing coincide con la de semántica de *failure trace* comentada al final de la sección anterior. Este tipo de testing sirvió también de base para dar semántica a un álgebra con probabilidades en [GN99].

2.3. Estudios Comparativos de Semánticas

El trabajo de Hennessy que hemos comentado en la sección anterior implica ya un cierto estudio comparativo, pues expone una aproximación en la que es posible integrar las descripciones operacional y denotacional que habían servido de base para

el desarrollo de los cálculos de procesos CCS y CSP, respectivamente, mostrando así una reciprocidad entre las semánticas, ya sean estas presentadas de forma operacional, denotacional o axiomática.

Desde un punto de vista más explícito, en [Hen88] el autor incluye unas notas históricas en las que se traza una cronología de los trabajos y las aportaciones que de forma más decisiva han contribuido al desarrollo de la teoría de procesos, mostrando otras aproximaciones alternativas, relaciones entre los trabajos, parecidos, diferencias, problemas abiertos, desarrollos a realizar...

Los otros dos textos esenciales que hemos comentado a lo largo de estas páginas también, de una u otra forma, introducen comentarios detallados en los que se comparan los diferentes modelos. En [Hoa85], Hoare comenta las diferencias y similitudes entre los lenguajes CCS y CSP y el porqué las equivalencias que define cada modelo están tan alejadas una de otra. Fragmentos de esta comparación ya han aparecido anteriormente en algunas de las citas que se han incluido de dicho libro.

Por su parte, cuando Milner publicó [Mil80] tenía aún poco trabajo relacionado que comentar. Sin embargo, la revisión y actualización de su modelo CCS que presentó en [Mil89] incluye ya múltiples referencias a CSP y dedica el último capítulo a comentar publicaciones, ideas y aportaciones que han influido en su cálculo, que proponen alternativas a sus propuestas o que profundizan en los desarrollos expuestos en su libro.

Dentro de la bibliografía de la teoría de procesos también se encuentran trabajos específicos que se centran en la comparación y en el análisis de los diferentes modelos y propuestas. El artículo de Rob van Glabbeek [vG97] ofrece una magnífica comparativa de las diferentes decisiones de diseño que subyacen en CCS y CSP: comunicación, deadlock, no determinismo, divergencia... En la misma línea, pero mucho menos técnico y más retrospectivo está el artículo de Hoare [Hoa05] que pertenece al libro *Algebraic Process Calculi: The First Twenty Five Years and Beyond* [AG05] que celebra los primeros veinticinco años de investigación en el área de la teoría (algebraica) de procesos reflexionando sobre los logros y mostrando líneas de trabajo futuro. En dicho libro, junto a este artículo de Hoare arriba comentado, se encuentran otras muchas contribuciones realizadas por los grandes nombres —de los cuales muchos ya han aparecido citados de una u otra forma en estas páginas— que han contribuido a establecer y desarrollar este área de investigación: Abramsky, Aceto, Baeten, Bergstra, Brooks, De Nicola, van Glabbeek, Klop, Milner, Larsen, Roscoe, Sangiorgi...

En las siguientes secciones se comentan tres trabajos muy importantes para entender la comparación de semánticas. En los dos primeros artículos presentados en las secciones 2.3.1 y 2.3.2 la *comparación* entre diversos modelos de procesos puede no ser tan explícita como en algunos de los trabajos que hemos comentado más arriba, pero en el fondo sus aportaciones son muy profundas al mostrar cómo un modelo puede *expresarse* en otro.

En el artículo que se comenta en la sección 2.3.3 se busca expresar a las semánticas en un mismo contexto, para de esta forma, poder analizar y entender las diferencias y similitudes entre las diferentes propuestas. Este será uno de nuestros primeros puntos

de partida para tratar de encontrar cierta regularidad en las semánticas que sea capaz de dar las claves para explicar la teoría de procesos desde un punto de vista unificador.

2.3.1. Bisimulación como Testing

En 1987 apareció publicado en la revista *Theoretical Computer Science* el artículo de Samson Abramsky titulado *Observation Equivalence as a Testing Equivalence* [Abr87].

Las premisas de este trabajo son muy sencillas: por un lado, la equivalencia de bisimulación —sección 2.2.1— es formulada como una relación sobre la descripción operacional de dos procesos; por otro lado, Hennessy describió una metodología —sección 2.2.3— que también tiene como punto de partida la descripción operacional de los procesos pero en la que la semántica es inducida por los resultados de los experimentos a los que es sometido un determinado proceso. El objetivo que persigue es claro: ¿hay alguna forma de hacer testing de tal manera que la equivalencia obtenida sea la de bisimulación? Se puede concretar más: puesto que es bien sabido que la equivalencia de bisimulación tiene mayor poder de distinción que la equivalencia de testing —clásicamente, igual a lo que tienen los *fallos* de Hoare—, la pregunta natural es ¿con qué tipo de pruebas necesitamos incrementar el testing? ¿qué se necesita distinguir para poder alcanzar la distinción de la bisimulación?

A continuación se incluye el *abstract* de dicho artículo, prototípico por su brevedad y claridad. Como única aclaración al respecto indicaré que *Observational equivalence* es el nombre que utilizó Milner para su equivalencia, véase la figura 2.9 en la página 37.

A notion of testing is developed for transition systems with divergence. The forms of testing include traces, refusals, copying and global testing. Both denotational and operational formulations of testing are given. The equivalence based on this notion of testing is shown to coincide with observation equivalence. [Abr87]

El artículo se desarrolla de una forma igualmente clara, siguiendo un estilo muy conciso, aunque desgraciadamente abunden los errores tipográficos. Para los procesos se considera una descripción operacional sin sintaxis concreta. Se define una familia de tests que cuenta con varias familias de operadores. Para los experimentos se define una semántica operacional y otra denotacional, y se demuestra que ambas coinciden. Puesto que se necesita el mayor poder de distinción de testing se utiliza la semántica may-must de Hennessy —sección 2.2.3— que corresponde con el uso del powerdomain de Plotkin.

Para probar el principal resultado, que indica que la equivalencia de testing introducida en este artículo y la bisimulación coinciden, se utiliza una de las caracterizaciones más interesantes de la bisimulación, la lógica HML [HM85]. El autor demuestra que existe una transformación que permite ver que las fórmulas HML están

incluidas en el conjunto de las pruebas y por tanto, pasar las mismas pruebas, equivalencia de testing, implica satisfacer las mismas fórmulas y por tanto la equivalencia de bisimulación.

Estudiando los operadores que aparecen en las pruebas que introduce Abramsky puede encontrarse la clave que permite alcanzar la distinción de la bisimulación. Desde un punto de vista matemático se puede explicar por la necesidad de contar en las pruebas con operadores monótonos, pero no lineales. Desde un punto de vista más observacional, se puede explicar como la habilidad de las pruebas para seguir cada una de las ramas no deterministas de un proceso y poder encontrar diferencias entre las continuaciones.

Por ejemplo, los procesos $p = a(bc + b)$ y $q = abc + a(bc + b)$ son distintos con respecto a la equivalencia de bisimulación, pero son iguales considerando todas las demás semánticas que han aparecido anteriormente en estas páginas. La prueba de Abramsky que consigue mostrar la diferencia entre ellos es $\forall a \exists b \bar{c}$, que puede ser interpretada como, “*toda acción a es seguida de alguna acción b, que después puede rechazar la acción c*”. El proceso p pasa la prueba con éxito, mientras que el proceso q no la pasa.

Este es precisamente el origen de las críticas por parte de algunos autores a la bisimulación: consideran que como equivalencia *observacional* es demasiado fina, pues diferencia más allá del no determinismo *razonable* que un observador externo podría distinguir.

El artículo de Abramsky no se centra tanto en discutir si la bisimulación es una equivalencia adecuada o no para procesos sino en mostrar lo que se necesita para conseguir dicho poder de distinción mediante pruebas. Por otro lado, otro resultado muy interesante que revela esta investigación —y que demuestra la generalidad del mismo— es que podríamos incorporar a la descripción de las pruebas cualquier otro operador monótono sin incrementar por ello el poder de distinción del testing realizado con dichas pruebas. Este resultado aporta un indicio más al hecho de que la bisimulación puede establecerse como un claro límite superior de las relaciones de equivalencia *naturales* que se pueden esperar entre procesos.

2.3.2. Testing como Bisimulación

El trabajo que presenta esta sección recorre la dirección inversa al expuesto en la anterior: ¿es posible utilizar la bisimulación para establecer otras equivalencias más débiles? Puesto que la bisimulación se define sobre la descripción operacional de los procesos en sistemas de transiciones, la pregunta puede reformularse de la siguiente manera: ¿puede transformarse la descripción operacional de los procesos de tal forma que la equivalencia entre los procesos originales coincida con la bisimulación en los procesos transformados?

Esto es precisamente lo que aborda el artículo titulado *Testing Equivalence as a Bisimulation Equivalence* [CH93], cuyo *abstract* es el que sigue:

In this paper we show how the testing equivalences and preorders on transition systems may be interpreted as instances or generalized bisimulation equivalences and prebisimulation preorders. The characterization relies on defining transformations on the transition system in such a way that the testing relations on the original systems correspond to (pre)bisimulation relations on the altered systems. On the basis of these results, it is possible to use algorithms for determining the (pre)bisimulation relations in the case of finite-state transition systems to compute the testing relations. [CH93]

Los autores para llevar a cabo este trabajo no podían ser más adecuados, por una lado Matthew Hennessy, el padre de la metodología de testing que ya hemos comentado en varias ocasiones —sección 2.2.3— y Rance Cleaveland, que había colaborado junto con Joachim Parrow y Bernhard Steffen, en la génesis de la herramienta denominada *Concurrency Work Bench* [CPS90, CPS93] que permite analizar procesos de estados finitos, en particular, comprobar la equivalencia de bisimulación.

Como adelanta la cita anterior, la técnica fundamental a utilizar es la transformación de los sistemas de transiciones que describen el significado operacional de los procesos. La idea intuitiva para llevar a cabo esta transformación es eliminar el no determinismo de los procesos y etiquetar los estados en el nuevo sistema de transición con información convenientemente añadida. Estos nuevos sistemas de transición se denominan *acceptance graphs* y se parecen bastante a los árboles de aceptación, los objetos denotacionales que Hennessy utilizaba en su libro. La gran diferencia estriba en que mientras los árboles de aceptación se constrúan a partir de la definición sintáctica de los procesos, los grafos de aceptación se construyen a partir de la descripción operacional de los mismos.

Sin embargo, la materialización concreta de la transformación arriba indicada no es en absoluto sencilla desde el punto de vista técnico. El artículo es muy denso, plagado de definiciones y de resultados. La definición de bisimulación tiene que ser ligeramente generalizada y en consecuencia, los algoritmos para decidir la equivalencia de bisimulación necesitan ser adaptados.

Otro punto interesante a tener en cuenta es el coste de la transformación de los sistemas de transiciones. En general, se sabe que el problema de decidir la equivalencia de testing es PSPACE-completo [KS90], mientras que la equivalencia de bisimulación se puede decidir en tiempo polinomial. De ello se sigue que en ocasiones el tamaño del sistema transformado tendrá que ser por fuerza grandísimo. Sin embargo, los autores comentan que con ejemplos reales probados en el ya citado *Concurrency Work Bench* el tamaño de los sistemas resultantes después de aplicar las correspondientes transformaciones es moderadamente grande, con lo que la complejidad total para decidir la equivalencia de testing en estos casos se mantiene aceptable.

2.3.3. Una Clasificación

Las dos secciones anteriores presentan trabajos muy interesantes desde el punto de vista del estudio teórico de las relaciones entre los procesos, pues profundizan en la *equiparación* de dos metodologías diferentes utilizadas para definir dichas relaciones: bisimulación y testing.

El trabajo de Rob van Glabbeek, *The linear time-branching time spectrum* que presentamos en esta sección perseguía también el objetivo de la equiparación de diversas semánticas —de la mayoría de las semánticas conocidas—, pero para ello lo que proponía era presentarlas desde un principio dentro de marcos comunes que posibilitasen un estudio comparativo de sus propiedades. Este es sin duda el trabajo más ambicioso e importante en el área del estudio comparativo de semánticas hasta la fecha.

La primera aparición del famoso *linear time-branching time spectrum* tuvo lugar en 1990 en una ponencia [vG90b] presentada en la primera edición de la *International Conference on Concurrency Theory*, que con el paso de los años ha llegado a ser el congreso más prestigioso del área. Posteriormente, en el año 2001 una versión revisada y ampliada de dicho trabajo [vG01] tenía también el honor de ser el primer capítulo del *Handbook of Process Algebra* [BPS01], editado por Jan Bergstra, Alban Ponse y Scott Smolka.

De nuevo, como hemos hecho en las secciones anteriores, para rendir homenaje y dar acceso a las ideas del autor expresadas de su propia mano, se incluye el *abstract* de la publicación de 2001.

In this paper various semantics in the linear time-branching time spectrum are presented in a uniform, model-independent way. Restricted to the class of finely branching, concrete, sequential processes, only fifteen of them turn out to be different, and most semantics found in the literature that can be defined uniformly in terms of action relations coincide with one of these fifteen. Several testing scenarios, motivating these semantics, are presented, phrased in terms of ‘button pushing experiments’ on generative and reactive machines. Finally twelve of these semantics are applied to a simple language for finite, concrete, sequential, nondeterministic processes, and for each of them a complete axiomatization is provided[vG01]

Uno de los objetivos fundamentales que se plantea el artículo es extraer y abstraer las ideas que se encuentran detrás de las definiciones de las semánticas. Una vez aisladas, estas ideas se podrían utilizar —por combinación de las mismas— para definir nuevas semánticas y sobre todo, para esclarecer las similitudes y diferencias entre las existentes.

Pero uno de los problemas a resolver para poder comparar las semánticas es establecer un marco común para todas ellas. Como hemos visto a lo largo de este capítulo, la presentación de las semánticas puede variar notablemente dependiendo

Definition 5

- The *refusal relations* \xrightarrow{X} for $X \subseteq Act$ are defined by: $p \xrightarrow{X} q$ iff $p = q$ and $I(p) \cap X = \emptyset$.
 $p \xrightarrow{X} q$ means that p can evolve into q , while being idle during a period in which X is the set of actions allowed by the environment.
- The *failure trace relations* $\xrightarrow{\sigma}$ for $\sigma \in (Act \cup \mathcal{P}(Act))^*$ are defined as the reflexive and transitive closure of both the action and the refusal relations. Again the overloading of notation is harmless.
- $\sigma \in (Act \cup \mathcal{P}(Act))^*$ is a *failure trace* of a process p if there is a process q such that $p \xrightarrow{\sigma} q$. Let $FT(p)$ denote the set of failure traces of p . Two processes p and q are *failure trace equivalent*, notation $p =_{FT} q$, if $FT(p) = FT(q)$.

Figura 2.12: Definición de la semántica de fallos como aparece en [vG01].

de la metodología utilizada. Conseguir una presentación unificada es el gran logro de van Glabbeek.

En primer lugar, se fija la definición de los procesos y para ello se utiliza una descripción operacional: los sistemas de transiciones. Sobre esta descripción operacional se definen las distintas semánticas, o mejor dicho, se redefinen las semánticas en este contexto. En el trabajo pueden encontrarse multitud de referencias a los orígenes de cada una de las semánticas que aparecen. También se comenta el hecho de que algunas de estas semánticas han aparecido en diferentes trabajos, e incluso en diferentes formulaciones o contextos y sin embargo coinciden *per se*. También otras semánticas coinciden al aplicarse sobre el marco concreto que propone el autor, en cierta forma simplificado.

Un ejemplo de esta unificación en la formulación de las relaciones entre procesos la podemos ver en la figura 2.12. La presentación denotacional de la semántica de fallos original de [Hoa85] —que ya hemos comentado y que puede encontrarse en la definición 10, página 43— es descrita en [vG01] utilizando la definición operacional de los procesos.

Además de estas definiciones uniformes sobre los sistemas de transiciones, van Glabbeek propone otros marcos en los que mostrar a cada una de las semánticas. Uno de ellos sigue la metáfora introducida por Milner de entender los procesos como cajas negras con botones —y quizás otros añadidos más sofisticados—, que reflejan el tipo de interacción y de observación que sobre dicho proceso puede hacerse. Para cada semántica se propone una explicación intuitiva de un escenario en el que se ponen de manifiesto las propiedades de los procesos que un observador externo puede recoger, así como el tipo de interacción que se le exige a dicho observador.

Más interesante y útil que los escenarios de pruebas son las caracterizaciones de las semánticas en términos de fórmulas lógicas. Para cada semántica, se identifica con precisión el conjunto de fórmulas modales cuya equivalencia de satisfactibilidad identifica los mismos procesos que la semántica correspondiente. La definición composicional de los correspondientes conjuntos de fórmulas hace que de hecho esta caracterización pueda ser considerada una semántica denotacional.

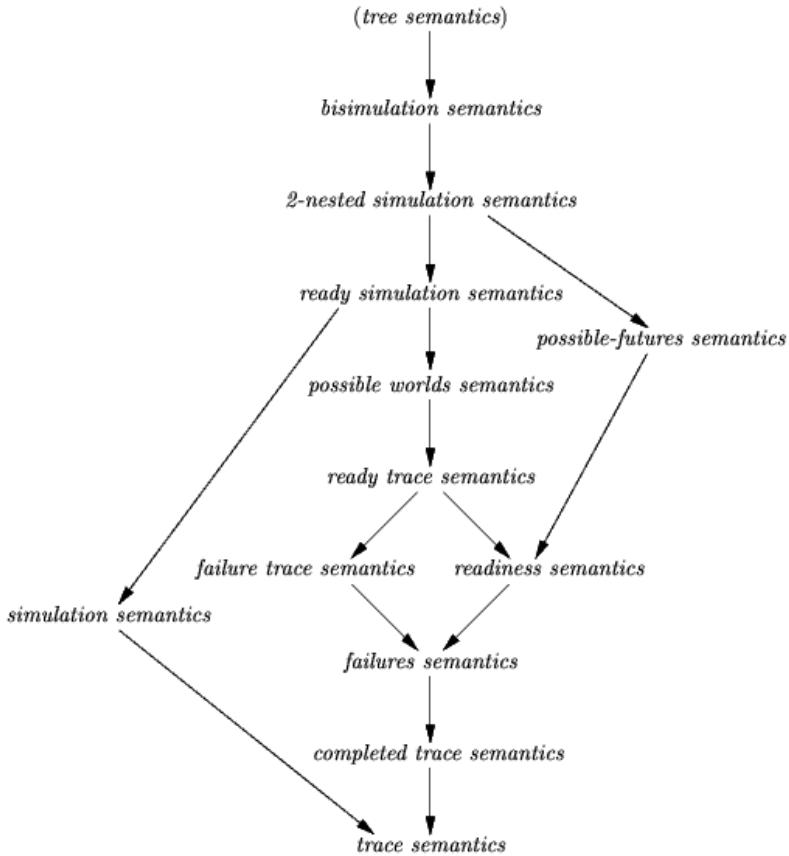


Figura 2.13: Linear time–branching time spectrum como aparece en [vG01].

El principal resultado del trabajo —al menos uno de los más visibles y citados en la literatura al respecto— es la catalogación de las semánticas teniendo en cuenta la capacidad de distinción de procesos. Esta taxonomía puede realizarse pues ahora las semánticas tienen marcos concretos y objetivos sobre los que compararse, como la citada caracterización en términos de fórmulas modales. La figura 2.13 muestra dicha clasificación tal y como aparece en [vG01].

Muchas de las semánticas en la figura 2.13 ya han aparecido a lo largo de estas páginas —trace, failures, ready trace, ready simulation...— y se han comentado en las secciones anteriores. Puede apreciarse como la bisimulación está en lo alto de la clasificación y por tanto es la relación que más procesos diferencia, con la salvedad de la semántica de árboles, en la que los procesos $a + a$ y a serían distintos.

Por su parte, la semántica de trazas —que se corresponde con el lenguaje que acepta un proceso en teoría clásica de autómatas— está en la parte más baja del spectrum, al producir el mayor número de identificaciones razonables entre procesos.

	<i>B</i>	<i>RS</i>	<i>PW</i>	<i>RT</i>	<i>FT</i>	<i>R</i>	<i>F</i>	<i>CS</i>	<i>CT</i>	<i>S</i>	<i>T</i>
$(x + y) + z = x + (y + z)$	+	+	+	+	+	+	+	+	+	+	+
$x + y = y + x$	+	+	+	+	+	+	+	+	+	+	+
$x + 0 = x$	+	+	+	+	+	+	+	+	+	+	+
$x + x = x$	+	+	+	+	+	+	+	+	+	+	+
$ax \sqsubseteq ax + ay$		+	+	+	+	+	+	v	v	v	v
$a(bx + by + z) = a(bx + z) + a(by + z)$			+	v	v	v	v		v		v
$I(x) = I(y) \Rightarrow ax + ay = a(x + y)$				+	v	v	v		v		v
$ax + ay \sqsupseteq a(x + y)$					+	v			v		v
$a(bx + u) + a(by + v) \sqsupseteq a(bx + by + u)$						+	v		v		v
$ax + a(y + z) \sqsupseteq a(x + y)$							+		v		v
$ax \sqsubseteq ax + y$								+	+	v	v
$a(bx + u) + a(cy + v) = a(bx + cy + u + v)$								+	+	v	v
$x \sqsubseteq x + y$									+	+	
$ax + ay = a(x + y)$											+
$I(0) = 0$	+	+	+	+	+	+	+	+	+	+	+
$I(ax) = a0$	+	+	+	+	+	+	+	+	+	+	+
$I(x + y) = I(x) + I(y)$	+	+	+	+	+	+	+	+	+	+	+

Figura 2.14: Axiomatización de los preórdenes semánticos como aparece en [vG01].

Por último, la investigación de van Glabbeek también consideraba la componente axiomática de las semánticas. Para poder comparar los axiomas de las diferentes relaciones entre procesos necesitaba un lenguaje sintáctico para la definición de los comportamientos común a todas ellas. El lenguaje elegido fue BCCSP, véase la definición 3. Sobre los términos de este lenguaje sintáctico se definen conjuntos de axiomas que caracterizan a doce de las semánticas del spectrum. Para la mayoría de ellas —exceptuando únicamente a la bisimulación, que no tiene orden— la caracterización es en realidad doble, por una lado la relación de orden natural que define cada semántica —figura 2.14— y por otro la equivalencia inducida —figura 2.15. Muchas de estas caracterizaciones ya existían previamente en la literatura, pero nuevamente, la presentación uniforme de todas ellas es uno de los méritos principales del trabajo que estamos comentando.

Algunas de estas caracterizaciones axiomáticas ya se han comentado en las secciones anteriores en las que hemos descrito diferentes semánticas de procesos. El estudio profundo —individual y comparativo— de estas axiomatizaciones y la búsqueda de respuestas a nuevos interrogantes que surgen de dicho estudio fueron los objetivos que guiaron la tesis ??, que consiguió una nueva forma de representar el spectrum que permite una mejor comprensión de las semánticas, que clarifica sus posiciones relativas en el mismo, y que muestra la existencia de *huecos* que se corresponden con

	<i>U</i>	<i>B</i>	<i>RS</i>	<i>PW</i>	<i>RT</i>	<i>FT</i>	<i>R</i>	<i>F</i>	<i>CS</i>	<i>CT</i>	<i>S</i>	<i>T</i>
$(x + y) + z = x + (y + z)$	+	+	+	+	+	+	+	+	+	+	+	+
$x + y = y + x$	+	+	+	+	+	+	+	+	+	+	+	+
$x + 0 = x$	+	+	+	+	+	+	+	+	+	+	+	+
$x + x = x$	+	+	+	+	+	+	+	+	+	+	+	+
$I(x) = I(y) \Rightarrow a(x + y) = a(x + y) + ay$		+	v	v	v	v	v	v	v	v	v	v
$a(bx + by + z) = a(bx + z) + a(by + z)$			+	v	v	v	v	v	v	v	v	v
$I(x) = I(y) \Rightarrow ax + ay = a(x + y)$				+	v	v	v	v	v	v	v	v
$ax + ay = ax + ay + a(x + y)$					+	v	v	v	v	v	v	v
$a(bx + u) + a(by + v) = a(bx + by + u) + a(by + v)$						+	+	+	v	v	v	v
$ax + a(y + z) = ax + a(x + y) + a(y + z)$							+	+	ω	v	v	v
$a(x + by + z) = a(x + by + z) + a(by + z)$								+	v	v	v	v
$a(bx + u) + a(cy + v) = a(bx + cy + u + v)$									+	v	v	v
$a(x + y) = a(x + y) + ay$										+	v	v
$ax + ay = a(x + y)$											+	+
$I(0) = 0$	+	+	+	+	+	+	+	+	+	+	+	+
$I(ax) = a0$	+	+	+	+	+	+	+	+	+	+	+	+
$I(x + y) = I(x) + I(y)$	+	+	+	+	+	+	+	+	+	+	+	+

Figura 2.15: Axiomatización de las equivalencias semánticas como aparece en [vG01].

semánticas que podrían ser —de acuerdo con los criterios que se han identificado— y cuya adicción al gráfico genera una deseable regularidad que lo hace más claro.

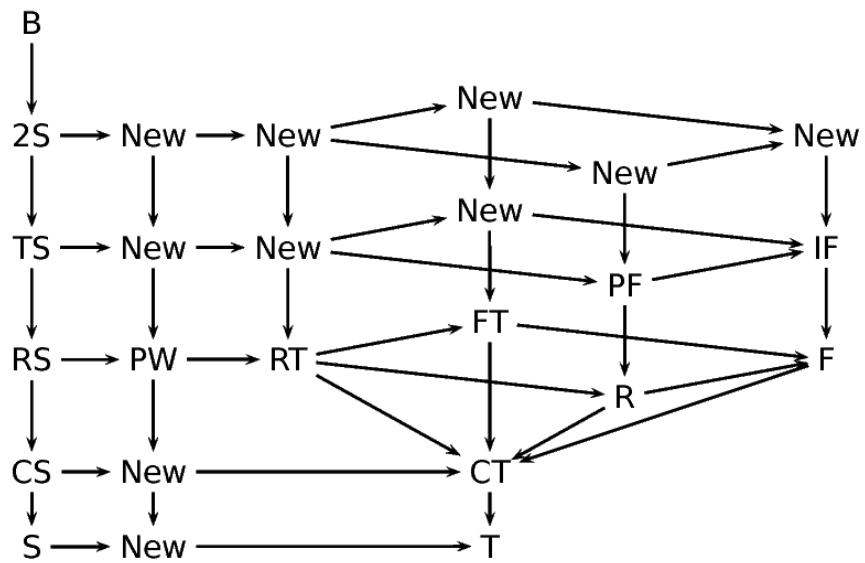


Figura 2.16: Nueva representación del spectrum como aparece en [dFGP09].

Epílogo

El siguiente texto, que el siempre incisivo y brillante Edsger Dijkstra escribió en las postrimerías del siglo que vio nacer a la informática como disciplina, hace hincapié en la imperiosa necesidad de una investigación teórica que no por *intangible* en sus objetivos e *incierta* en su resultados, deja de ser imprescindible para profundizar en el conocimiento, para encontrar respuestas, en definitiva, para saber *what we should be talking about*.

The end of Computing Science?

*"The price of reliability is the pursuit of the utmost simplicity.
It is a price which the very rich find most hard to pay."*

Sir Antony Hoare, 1980

In academia, in industry, and in the commercial world, there is a widespread belief that computing science as such has been all but completed and that, consequently, computing has "matured" from a theoretical topic for the scientists to a practical issue for the engineers, the managers and the entrepreneurs, i.e. mostly people —and there are many of those!— who can accept the application of science for the obvious benefits, but feel rather uncomfortable with its creation because they don't understand what the doing of research, with its intangible goals and its uncertain rewards, entails. This widespread belief, however, is only correct if we identify the goals of computing science with what has been accomplished and forget those goals that we have failed to reach, even if they are too important to be ignored.

I would therefore like to posit that computing's central challenge, viz. "How not to make a mess of it", has not been met. On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence. The average customer of the computing industry has been served so poorly that he

expects his system to crash all the time, and we witness a massive world-wide distribution of bug-ridden software for which we should be deeply ashamed.

For us scientists it is very tempting to blame the lack of education of the average engineer, the short-sightedness of the managers and the malice of the entrepreneurs for this sorry state of affairs, but that won't do. You see, while we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces. We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy. We do not know yet whether trade-offs will be possible. We do not know yet whether we can invent for intricacy a meaningful concept about which we can prove theorems that help. To put it bluntly, we simply do not know yet what we should be talking about, but that should not worry us, for it just illustrates what was meant by "intangible goals and uncertain rewards".

And this was only an example. The moral is that whether Computing Science is finished will primarily depend on our courage and our imagination.

Austin, 19 November 2000

Edwger W. Dijkstra

—Bibliografía

- [Abr87] Samson Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3):225–241, 1987.
- [Ace03] Luca Aceto. Some of my favourite results in classic process algebra. In *In Bulletin of the EATCS*, pages 89–108, 2003.
- [AFI07] Luca Aceto, Wan Fokkink, and Anna Ingólfssdóttir. Ready to preorder: get your BCCSP axiomatization for free! In *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007*, volume 4624 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2007.
- [AG05] Luca Aceto and Andrew D. Gordon, editors. *Algebraic Process Calculi: The first 25 years and beyond*. BRICS NS-05-3, 2005.
- [Ata84] John Vincent Atanasoff. Advent of electronic digital computing. *Annals of the History of Computing*, 6(4):229–282, 1984.
- [Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.
- [BFN03] Stefan Blom, Wan Fokkink, and Sumit Nain. On the axiomatizability of ready traces, ready simulation, and failure traces. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, pages 109–118, 2003.
- [BG03] Michele Boreale and Fabio Gadducci. Denotational testing semantics in coinductive form. In Branislav Rovan and Peter Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003*, volume 2747 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2003.
- [BHR84] Stephen D. Brookes, C.A.R. Hoare, and A. William Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

- [BIM88] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 229–239. ACM Press, 1988.
- [BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.
- [BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
- [BM92] Bard Bloom and Albert R. Meyer. Experimenting with process equivalence. *Theoretical Computer Science*, 101(2):223–237, 1992.
- [BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
- [BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Computer Science. Cambridge University Press, 1990.
- [CH93] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 3:1–21, 1993.
- [CPS90] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. A semantics based verification tool for finite state systems. In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX*, pages 287–302. North-Holland, 1990.
- [CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
- [dFG05] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. Bisimulations up-to for the linear time-branching time spectrum. In *CONCUR 2005 – Concurrency Theory, 16th International Conference*, volume 3653 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2005.
- [dFG06] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. Process equivalences as global bisimulations. *Journal of Universal Computer Science*, 12(11):1521–1550, 2006.
- [dFGP09] David de Frutos-Escrig, Carlos Gregorio-Rodríguez, and Miguel Palomino. On the unification of process semantics: Observational semantics. In *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 2009.

-
- [DH72] Ole-Johan Dahl and C. A. R. Hoare. *Structured programming*, chapter Hierarchical program structures, pages 175–220. Academic Press Ltd., London, UK, UK, 1972.
- [DH83] Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. In *ICALP'83 – 10th International Colloquium on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560. Springer, 1983.
- [Dij65] Edsger W. Dijkstra. Cooperating sequential processes. Published as [Dij68], 1965.
- [Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968. More easily available in [HDH02].
- [Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971. More easily available in [HDH02].
- [Dij75a] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
- [Dij75b] Edsger W. Dijkstra. A synthesis emerging? published as [Dij82], August 1975.
- [Dij82] Edsger W. Dijkstra. A synthesis emerging? In *Selected Writings on Computing: A Personal Perspective*, pages 147–160. Springer, 1982. More easily available in [HDH02].
- [DN67] Ole-Johan Dahl and Kristen Nygaard. Class and subclass declaration. In J. N. Buxton, editor, *IFIP Working Conference on Simulation Programming Languages*, pages 158–174, Lysebu, Oslo, 1967. North Holland.
- [GN99] Carlos Gregorio-Rodríguez and Manuel Núñez. Denotational semantics for probabilistic refusal testing. *Electronic Notes in Theoretical Computer Science*, 22:111–137, 1999.
- [GR09] Carlos Gregorio-Rodríguez. *Understanding Process Semantics*. PhD thesis, Universidad Complutense, Madrid, 2009.
- [Gö31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [Han73] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

- [HDH02] Per Brinch Hansen, Edsger W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
- [Hen88] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
- [HJ04] Jesse Hughes and Bart Jacobs. Simulations in coalgebra. *Theoretical Computer Science*, 327(1–2):71–108, 2004.
- [HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
- [Hoa65] C. A. R. Hoare. Record handling. *ALGOL Bull.*, 21:39–69, 1965.
- [Hoa71] Charles Antony Richard Hoare. *Operating Systems Techniques*, chapter Towards a Theory of Parallel Programming. Academic Press, 1971. More easily available in [HDH02].
- [Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. Also in [HDH02].
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Hoa05] C.A.R. Hoare. Why ever csp? In *Algebraic Process Calculi: The first 25 years and beyond*, BRICS NS-05-3, pages 229–233, 2005.
- [HS85] Matthew Hennessy and Colin Stirling. The power of the future perfect in program logics. *Information and Control*, 67(1–3):23–52, 1985.
- [Jac04] Bart Jacobs. Trace semantics for coalgebras. In *CMCS'04: 7th International Workshop on Coalgebraic Methods in Computer Science*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 167–184. Elsevier, 2004.
- [Kli04] Bartek Klin. A coalgebraic approach to process equivalence and a coinductive principle for traces. In *CMCS'04: 7th International Workshop on Coalgebraic Methods in Computer Science*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 201–218. Elsevier, 2004.
- [KPH61] T. Kilburn, R. B. Payne, and D. J. Howarth. The atlas supervisor. In *AFIPS Computer Conference*, volume 20, pages 279–294, 1961.
- [KS90] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

-
- [KS03] Bartek Klin and Paweł Sobociński. Syntactic formats for free. In *CONCUR 2003 – Concurrency Theory, 14th International Conference*, volume 2761 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2003.
 - [LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.
 - [Mac96] Allan R. Mackintosh. El computador del Dr. Atanasoff. *Investigación y Ciencia. Temas*, 4:34–41, 1996.
 - [Mil80] Robin Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer, 1980.
 - [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
 - [Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
 - [Mil06] Robin Milner. *Interactive Computation, the new paradigm*, chapter Turing, Computing and Communication. Springer, 2006.
 - [ND78] Kristen Nygaard and Ole-Johan Dahl. The development of the simula languages. *SIGPLAN Not.*, 13(8):245–272, 1978.
 - [OH86] Ernst R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23(1):9–66, 1986.
 - [Par81] David M.R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.
 - [Phi87] Iain Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241–284, 1987.
 - [Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal of Computation*, 5:452–487, 1976.
 - [Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
 - [Plo83] Gordon D. Plotkin. An operational semantics for csp. In *Logic of Programs and Their Applications*, volume 148 of *Lecture Notes in Computer Science*, pages 250–252. Springer, 1983.
 - [Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

- [PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.
- [RBW86] A. W. Roscoe, S. D. Brookes, and D. J. Walker. An operational semantics for CSP. Technical report, Oxford University Computing Laboratory, 1986.
- [Rut03] Jan J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1-3):1–53, 2003.
- [San07] Davide Sangiorgi. On the origins of bisimulation and coinduction. Technical Report 24, Department of Computer Science, University of Bologna, 2007.
- [Sha38] Claude Elwood Shannon. *A symbolic analysis of relay and switching circuits*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1938.
- [SS71] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.
- [Sti98] Colin Stirling. The joys of bisimulation. In *Mathematical Foundations of Computer Science, 23rd International Symposium, MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 142–151. Springer, 1998.
- [SW93] N. J. A. Sloane and A. D. Wyner. *Claude Elwood Shannon: Collected Papers*. IEEE Press, Piscataway, NJ, 1993.
- [Tur36] Allan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, ser. 2(42):230–265, 1936.
- [vG90a] Rob J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University, Amsterdam, 1990. Second edition available as *CWI tract 109*, CWI, Amsterdam 1996.
- [vG90b] Rob J. van Glabbeek. The linear time-branching time spectrum. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, number 458 in Lecture Notes in Computer Science, pages 278–297. Springer-Verlag, 1990.
- [vG93] Rob J. van Glabbeek. The linear time - branching time spectrum II. In *CONCUR '93 – Concurrency Theory, 5th International Conference*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.
- [vG97] Rob J. van Glabbeek. Notes on the methodology of ccs and csp. *Theoretical Computer Science*, 177(2):329–349, 1997.

- [vG01] Rob J. van Glabbeek. *Handbook of Process Algebra*, chapter The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes, pages 3–99. Elsevier, 2001.
- [vN93] John von Neumann. First draft of a report on the EDVAC. *IEEE annals of the history of computing*, 15(4):27–75, 1993.