Análisis de sistemas de Máquinas de Estados Finitos en comunicación

Analysis of Communicating Finite State Machines

Pablo Hidalgo Palencia

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Grado de Ingeniería Informática

Curso 2019/2020

Directores:

Ismael Rodríguez Laguna
Fernando Rosa Velardo

Resumen

En este trabajo presentamos un modelo de cómputo formado por varias máquinas de estados finitos que se pueden comunicar entre sí a través de canales FIFO. Estudiamos cuál es su expresividad y la complejidad de resolver algunos problemas en este modelo, que yace entre lo decidible y lo indecidible por aunar la simplicidad de las máquinas de estados finitos con la complejidad que aportan comunicaciones no deterministas. Estudiamos además diversas variaciones en la definición y las implicaciones que tienen estas modificaciones sobre la expresividad y complejidad del modelo.

Palabras clave

Máquinas de estados finitos, máquinas de estados finitos en comunicación, sistemas de mensajes perdidos, redes de Petri, autómata 110.

Abstract

In this work we present a computation model in which several Finite State Machines communicate via FIFO channels. We study its expressivity and the complexity of solving some decision problems regarding this model, lying on the edge between decidability and undecidability because of bringing together the simplicity of Finite State Machines and the complexity of non-deterministic communications. We also study some variations of the main model and the changes they generate in terms of expressivity and complexity.

Keywords

Finite State Machines, Communicating Finite State Machines, Lossy Channel Systems, Petri nets, rule 110.

Índice general

Capítulo 1

Introducción

En la literatura científica, en particular en la de la Informática, es frecuente encontrarse situaciones en las cuales los investigadores tienen un problema que resolver que no se ajusta perfectamente a ningún modelo preestablecido conocido, si bien puede ser solventado de forma sencilla creando un modelo *ad hoc* para la situación a partir de otros ya estudiados en profundidad. Por ejemplo, en numerosas ocasiones, al enfrentarse a ciertos problemas, los autores deciden dividir el problema en varias secciones sencillas e interconectarlas entre sí. Notemos que esto también se hace en la vida real: la mayoría de los trabajos complejos se suelen realizar dividiéndolos en partes más pequeñas y sencillas (que podrán serán realizadas por diferentes personas) y después juntándolas.

Este es el caso de nuestro estudio: es frecuente que al tener que modelizar protocolos concurrentes en la Informática, por ejemplo, se utilice la misma táctica que comentábamos. En concreto, el protocolo se realiza por medio de instancias pequeñas más sencillas, como pueden ser las que realizan las máquinas de estados finitos, elementos bastante sencillos y estudiados; estableciendo una comunicación entre las distintas máquinas para que puedan simular el protocolo completo.

No obstante, estas construcciones complejas hechas a partir de máquinas de estados sencillas se suelen hacer *ad hoc* para la situación en la que se esté interesado, y esto hace que en general no se puedan reutilizar ciertos resultados ya existentes en ejemplos similares en la literatura.

En este trabajo pretendemos hacer un estudio más general sobre estos sistemas formados por máquinas de estados en comunicación, de forma que pueda establecer unas bases para saber qué propiedades podría tener un sistema concreto que utilicemos en otros trabajos con un fin más determinado. Por ejemplo, si para una investigación posterior usamos un modelo similar a los que vamos a explicar en este estudio, ¿resultarán propiedades de decisión básicas como terminación o alcanzabilidad decidibles o tratables? Los resultados que veremos en los siguientes capítulos ayudarán a responder este tipo de preguntas.

Además, dada la variabilidad y versatilidad de este tipo de sistemas, resultará interesante investigar sus propiedades en función de los diferentes tipos de definición que podamos hacer de ellos, por lo cual trataremos varios tipos de sistemas para poder establecer comparativas entre sus propiedades, lo que creemos que podría ser de utilidad a la hora de vernos en la necesidad de usar alguno de estos sistemas.

El texto irá organizado por capítulos. En el capítulo 2 daremos una visión amplia sobre literatura relacionada con nuestro trabajo. En el capítulo 3 veremos una

definición de un modelo simplificado de los sistemas de máquinas de estados en comunicación, que nos ayudará a comprender mejor los conceptos antes de entrar en el caso general. En el capítulo 4 veremos que esta simplificación de los sistemas es un modelo Turing completo. Con esto, en el capítulo 5 llegamos ya a la definición de los sistemas generales que trataremos a partir de ahí. En el capítulo 6 estudiaremos más a fondo el caso de los sistemas donde los outputs no proliferan, que resultan ser, como reconocedores de lenguajes, un modelo completo dentro del de los dependientes del contexto. En el siguiente capítulo, el 7, explicamos que los sistemas donde los buffers están acotados no pueden reconocer más allá de los lenguajes regulares. En el capítulo 8 estudiamos qué ocurre en el caso de que se pierda el orden de los mensajes intercambiados en nuestros sistemas, ya que obtenemos modelos comparables a diferentes tipos de redes de Petri, los modelos más representativos cuando no hay orden en los sistemas. Más tarde, en el capítulo 9, estudiamos qué ocurre cuando otra propiedad básica de los sistemas cambia, concretamente qué ocurre cuando los canales de comunicación pueden fallar, y su relación con los conocidos Sistemas de Mensajes Perdidos. Tras todo esto, llegamos al capítulo 10, donde estudiamos la complejidad de resolver diferentes problemas de decisión sobre las cualidades de nuestros sistemas, centrándonos en el problema de la alcanzabilidad finita y de regreso al estado inicial. Por último, presentamos en el capítulo 11 un sistema bastante sencillo que es Turing universal, por medio de una simulación del autómata 110.

Introduction

As we can see in the existing scientific literature, it is quite usual that a researcher faces a problem which does not fit exactly in an existing model, but can be easily solved creating an *ad hoc* model for this situation based on others which have been studied in depth. For instance we can find the situation where a researcher splits a problem into smaller and simpler sections and then connects all them properly. Notice that this is not a unique feature from research, we do this in our daily life: we carry out the vast majority of the difficult works using a strategy that first divides the whole task into smaller subtasks (which are commonly done by different people) and then join everything.

This will be our case: for instance it is common that the strategy explained above is used when modelling concurrent protocols in Computer Science. More concretely, the protocol is carried out by means of smaller systems, such as Finite State Machines -which are really simple and have been studied broadly- which can join their efforts if we establish some kind of communication among them.

Although this is a common situation, these complex constructions are often in practice really *ad hoc*, in the sense that they are not directly reusable in other situations than the one they were thought for. This makes it more difficult to reuse not only the definitions, but also the results and properties that were proved for those models.

In this work we aim to study in a broad generality this kind of systems where some Finite State Machines can communicate among them, with the purpose of establishing some foundation in the topic which can be later used to know which kind of properties are to expect from a concrete similar model that anyone could need in a research. If it were the case that we used a similar model in another work, would some basic properties of that model such as termination or reachability be decidable or tractable? The results we are about to develop in the next chapters will help to answer this sort of questions.

Furthermore, given the variability and flexibility of this kind of systems, it will be of great interest to investigate their properties in terms of the different definitions we can use, in order to state a comparative among some of the possible models that can be created from the same abstraction. This can turn out to be really useful when deciding which model best fits a concrete situation we are interested in.

The text will be divided into chapters. In Chapter 2 we will take a look at the existing literature related to our work. In Chapter 3 we will introduce the topic by studying a simplified version of our final model, which only comprises a maximum of two Finite State Machines, and will help us understand better the key concepts. In Chapter 4 we prove that this simplified model is Turing complete. In Chapter 5 we arrive to the definition of the main model we are interested in. In Chapter 6 we

study a submodel where the amount of information used to communicate is limited (the Finite State Machines can communicate with only one machine at a time), which results to be complete among the context-sensitive language recognizers. We study next, in Chapter 7, the case where the size of the buffers used to communicate is bounded, which results in a model that can only recognize regular languages. In our path studying variations of the main model we get to the Chapter 8, where the order of the messages exchanged between machines is in some sense lost, and the resulting model is comparable to different kinds of Petri nets. In Chapter 9 we study the case where another crucial property of the communications is changed: if the channels used for communications are faulty, the formalism becomes really similar to Lossy Channel Systems, and we will study the connections more in detail. All this been studied we arrive to Chapter 10, where we look into some decision problems on our models and their complexity, focusing on some finitary versions of reachability and home-state problems. Finally in Chapter 11 we present a really simple system which is (Turing-)universal, based on Rule 110.

Capítulo 2

Estado del arte

Como decíamos, en la literatura ya hay numerosos ejemplos de situaciones en las cuales una agrupación de máquinas de estados finitos (de diferentes tipos) se pueden comunicar a través de una serie de canales (con diferentes propiedades). Esto en general se debe a que, al estar constituidas por máquinas de estados finitos, son fáciles de programar y entender su funcionamiento, a la par que son muy versátiles (podemos añadir propiedades a las máquinas o a los canales dependiendo de nuestra situación) y pueden llegar a tener una expresividad muy grande.

Por esta versatilidad de la que hablamos, en general no suele haber una definición que concuerde en todos los casos. En numerosas ocasiones la definición de estos sistemas de máquinas de estados en comunicación coincide con el que usaremos nosotros en el caso general: canales FIFO ideales y máquinas de estados finitos deterministas. Ejemplos de artículos que usen esta definición son numerosos: [7, 10, 13, 15, 16, 23, 25].

No obstante, hay algunos casos en los que difieren algunos detalles. Por ejemplo, en [7], que fue uno de los primeros artículos que introdujo este tipo de sistemas ya en 1983, la comunicación entre n máquinas se produce con n^2 canales: cada máquina tiene un canal de comunicación directo con cualquier otra máquina, de forma que no se mezclan los mensajes. En otros casos, como [13, 23] se restringen al estudio de únicamente 2 máquinas, que ya veremos más adelante (capítulo 4) que es suficientemente interesante. Hay también situaciones en los que se introducen capacidades especiales a las máquinas, como las de hacer test de vacío de los canales que leen, como en [25] (veremos una situación similar en el apartado 8.2).

Como hemos explicado, nuestro objetivo es inferir propiedades de este tipo de sistemas de forma teórica. Sobre todo porque este tipo de trabajo no se ha hecho en profundidad con estos sistemas: muchos autores utilizan estas construcciones para modelizar distintos protocolos de comunicación por la versatilidad de las comunicaciones (en artículos similares a [6, 7, 10, 23]). Pero no muchos artículos versan sobre averiguar las propiedades generales de estos sistemas. Quizá en esta dirección podríamos citar [13, 15, 16, 23, 25], pero en su mayoría no investigan sistemas generales, sino únicamente subclases que les interesan para sus propósitos determinados. En general, la propiedad más estudiada es la acotación de los canales a lo largo de las ejecuciones de los sistemas, por lo que en nuestro caso no la vamos a tratar tanto.

También es amplio el espectro de ejemplos en los cuales los canales no son de tipo FIFO e ideales. Por ejemplo, hay una gran teoría desarrollada sobre Sistemas de Mensajes Perdidos, en los cuales los canales pueden perder mensajes de manera

no determinista. Ejemplos de artículos que los analizan son [2, 26], cuyos autores han escrito bastantes más artículos sobre este tema. Nosotros los trataremos en el capítulo 9 por ser uno de los sistemas más importantes que se pueden modelizar con máquinas de estados en comunicación.

También hay ocasiones en los que hay ciertas nociones de prioridad entre los canales, como en [14]. O incluso situaciones que llegan a un nivel mayor de abstracción y subsumen a varias de las que hemos comentado, como en el caso de [5, 6].

Viendo la literatura existente, de la que hemos intentado resumir aquí sus principales vertientes (aunque cada una de ellas es mucho más amplia), concluimos que los sistemas de máquinas de estados en comunicación son ampliamente utilizados con diferentes propósitos y definiciones, pero no se han estudiado sus propiedades salvo en casos concretos que vinieran bien para un trabajo. Por tanto, vemos conveniente intentar al menos hacer un estudio de varias de las propiedades de este tipo de sistemas en su versión más usual. Y al haber muchas variaciones en la definición de los sistemas, exploraremos cómo van variando estas propiedades al variar las definiciones.

Capítulo 3

Definición de los sistemas con 2 máquinas

Antes de ir al caso general de nuestro estudio, vamos a estudiar un subcaso sencillo que tiene interés por sí mismo, y que allanará el camino para entender mejor los siguientes razonamientos. En este apartado limitaremos nuestros sistemas a tener únicamente dos máquinas de estados, con lo que será más fácil comprender su funcionamiento.

Supongamos que tenemos F_1, F_2 dos máquinas de estados finitos (FSM por sus siglas en inglés), de tipo Mealy: $F_1 = (Q_1, I_1, O_1, q_1^0, \delta_1)$, $F_2 = (Q_2, I_2, O_2, q_2^0, \delta_2)$, donde Q_i es el conjunto (finito) de estados de la máquina F_i, I_i el alfabeto de sus inputs, O_i el alfabeto de sus outputs, $q_i^0 \in Q_i$ es su estado inicial y $\delta_i : Q_i \times I_i^? \longrightarrow Q_i \times O_i^?$ es una función parcial, llamada función de transición, siendo $X^? := X \cup \{\epsilon\}$. $\delta_i(q, \alpha) = (q', \beta)$ significa que si la máquina F_i está en el estado q y le llega el input $\alpha \neq \epsilon$, puede transitar hacia el estado q' produciendo el output β. Y si $\alpha = \epsilon$, significa que dicha transición se puede tomar sin que llegue ningún literal, como es habitual. Notemos que al ser δ_i una función parcial, no tiene por qué estar definida para todos los posibles pares $(q, \alpha) \in Q_i \times I_i^?$: habrá valores que tengan imagen, pero no necesariamente todos.

Gráficamente representaremos los diferentes estados de una FSM con círculos, y las transiciones mediante flechas: si una flecha va de q a q' y tiene la anotación X/Y, estará representando la transición $\delta(q, X) = (q', Y)$.

Es importante darse cuenta de que las máquinas F_1, F_2 pueden efectuar transiciones sin necesidad de consumir literales de su input, pues $\epsilon \in I_i^?$. Esto significa que pueden generar más outputs que inputs reciben, fenómeno al cual llamaremos **proliferación de outputs**. Notemos que esta forma de que los outputs proliferen es equivalente a otras, como por ejemplo la que se generaría si $\delta_i : Q_i \times I_i \longrightarrow Q_i \times O_i^*$. Esta equivalencia se puede ver de la siguiente forma: si tenemos una función de transición del tipo $\delta : Q \times I \longrightarrow Q \times O^*$, simplemente podemos crear nuevos estados allá donde se produzca más de un output (si se producen n outputs, generamos $n-1$ estados auxiliares), como sugiere la figura 3.1, y que más adelante formalizaremos más en detalle. En el otro sentido, tendríamos que eliminar estos estados auxiliares, en los cuales se generan outputs sin consumir inputs, es decir, que tiene transiciones consumiendo ϵ. Esto se puede hacer siguiendo un procedimiento similar al conocido método que nos permite reducir autómatas finitos no deterministas con transiciones ϵ a autómatas finitos no deterministas (sin transiciones ϵ), tomando las denominadas ϵ-clausuras.

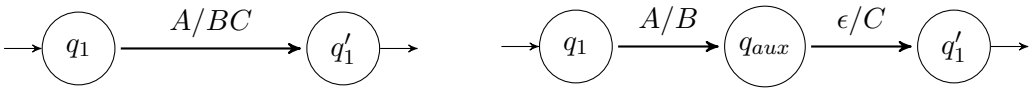


Figura 3.1: Ejemplo de transformación de una transición en la que se pueden producir varios outputs en otra en la cual solo se permite producir un output en cada transición.

Como vamos a establecer sistemas para que las máquinas se comuniquen, notemos que los alfabetos con los que se comunican F_1, F_2 tienen que ser coherentes entre sí. En concreto, para que la máquina F_1 comprenda los literales que F_2 le manda, se debe cumplir $O_2 \subseteq I_1$. Por simplicidad podemos suponer en realidad $O_2 = I_1$ sin más que pensar que F_2 no tiene por qué generar outputs de todos los literales de su alfabeto, como es natural. Análogamente, podemos suponer $O_1 = I_2$. Además, por simplicidad también asumiremos en ocasiones que estos 4 alfabetos están dentro de otro más grande que denotaremos Σ, que vistas las anteriores relaciones, podemos definir simplemente como $\Sigma := I_1 \cup I_2$, y sirve para hacer razonamientos con las máquinas F_1, F_2 tan bien como I_1, I_2, O_1, O_2.

Para que realmente exista comunicación entre las máquinas F_1 y F_2 debemos disponer de un medio. En este caso tendremos dos buffers B_1, B_2, siendo B_1 el buffer en el cual escribe la máquina F_2 y lee F_1, y B_2 el análogo en el sentido contrario, como se puede ver en la figura 3.2. Viendo estos buffers como estructura de datos, son colas de elementos de nuestro alfabeto Σ: B_1 será una cola en la cual F_2 inserta elementos por un lado, mientras que F_1 los retira desde el otro lado. Numeraremos las posiciones del buffer empezando en 0 (el extremo cercano a F_1), y crecientes a medida que nos acercamos al extremo por el que F_2 inserta. De manera análoga podríamos definir B_2, intercambiando los papeles de F_1 y F_2.

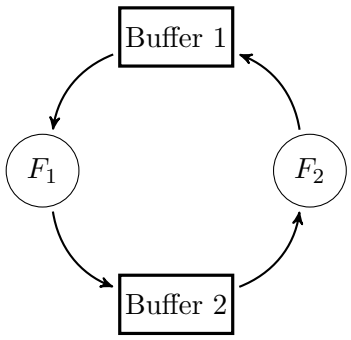


Figura 3.2: Visualización de la forma en que las 2 máquinas se comunican.

Ahora nos damos cuenta de que el conjunto de configuraciones de un buffer posibles es simplemente Σ^*, el conjunto de sucesiones finitas de letras de nuestro alfabeto. Asimismo, definimos la operación $+$ como la concatenación de elementos de Σ^*. De esta forma, si en un buffer cuyo contenido es α encolamos la palabra β, su contenido final lo denotaremos simplemente $\alpha + \beta$.

En estas condiciones, podemos pasar a definir realmente los sistemas que vamos a utilizar:

Definición 3.1. *Dadas dos FSMs F_1, F_2, decimos que $\mathcal{S} = (F_1, F_2)$ es un **sistema de dos máquinas de estados en comunicación** asociado a F_1, F_2. Las configuraciones de $\mathcal{S}$ son tuplas de la forma $(q_1, B_1, q_2, B_2) \in Q_1 \times \Sigma^* \times Q_2 \times \Sigma^*$. La función de transición de $\mathcal{S}$ es $\delta : Q_1 \times \Sigma^* \times Q_2 \times \Sigma^* \longrightarrow Q_1 \times \Sigma^* \times Q_2 \times \Sigma^*$, construida de la siguiente forma:*

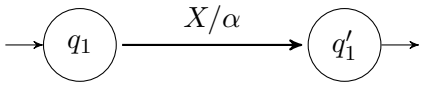
1. $B_1 = X + B_1' \wedge \delta_1(q_1, X) = (q_1', \alpha) \implies (q_1', B_1', q_2, B_2 + \alpha) \in \delta(q_1, B_1, q_2, B_2)$
2. $B_2 = Y + B_2' \wedge \delta_2(q_2, Y) = (q_2', \beta) \implies (q_1, B_1 + \beta, q_2', B_2') \in \delta(q_1, B_1, q_2, B_2)$

Primero notar que las configuraciones (q_1, B_1, q_2, B_2) se pueden interpretar como si estuviéramos indicando el estado de cada FSM mediante q_1 y q_2 y los contenidos de los buffers que en los párrafos anteriores comentábamos fueran B_1 y B_2, respectivamente.

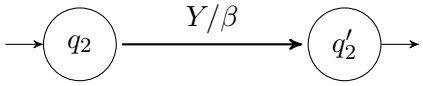
Las transiciones expuestas en la definición simplemente significan que, en cada configuración (q_1, B_1, q_2, B_2) de $\mathcal{S}$, se puede avanzar de dos maneras: dando F_1 un paso según su definición, o dándolo F_2. En este sentido, si es la máquina F_1 la que avanza, ha de usar el primer literal de su buffer, digamos X (y B_1' es el resto del buffer), para saber qué transición toma, siendo esta $\delta_1(q_1, X)$. Si esta entrada de la función δ_1 está definida, es decir, $\delta_1(q_1, X) = (q_1', \alpha)$ para ciertos $q_1' \in Q_1, \alpha \in \Sigma^?$, entonces se podrá encolar α en B_2 para llegar a la nueva configuración de $\mathcal{S}$: $(q_1', B_1', q_2, B_2 + \alpha)$. De manera análoga podemos razonar qué sucede en $\mathcal{S}$ si es F_2 quien avanza un paso.

Teniendo estas transiciones en nuestras FSMs:

En F_1:



En F_2:



Obtendríamos un comportamiento similar al siguiente:

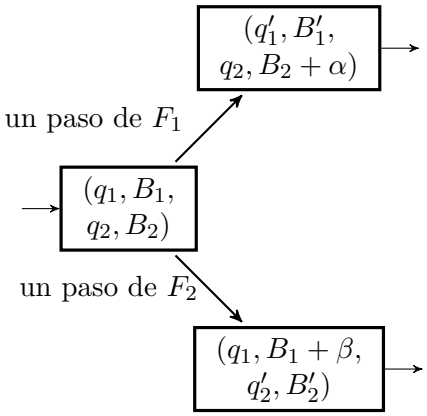


Figura 3.3: Representación de las diferentes transiciones de $\mathcal{S}$ en función de las transiciones que pueden tomar F_1 y F_2 estando en los estados q_1 y q_2, respectivamente, y siendo los contenidos de los buffers $B_1 = X + B_1'$ y $B_2 = Y + B_2'$, respectivamente.

Los sistemas de máquinas de estados en comunicación así construidos son inherentemente no deterministas, a pesar de que las máquinas de que están compuestos pudieran ser deterministas: hay configuraciones del sistema $\mathcal{S}$ en las cuales podrían avanzar un paso tanto la máquina F_1 como la máquina F_2 (cuando el primer literal del contenido sus buffers no bloquea su avance). Así, el sistema puede avanzar en diferentes direcciones en una misma situación, lo cual resulta en un comportamiento no determinista. Básicamente esto se produce porque las comunicaciones entre F_1 y F_2 dentro de $\mathcal{S}$ se producen de forma asíncrona: no hay ningún mecanismo que regule quién puede mandar información a la otra máquina, y esto genera que ambas

máquinas puedan avanzar pasos en su ejecución de forma casi independiente, lo cual da mucha libertad al sistema $\mathcal{S}$.

Además, en este tipo de sistemas existe también el concepto de **configuración de aceptación**. Consideraremos que $\mathcal{S}$ tiene un conjunto de estados finales $Q_F \subseteq Q_1 \cup Q_2$, y que una configuración (q_1, B_1, q_2, B_2) será de aceptación si $q_1 \in Q_F$ o $q_2 \in Q_F$. Es decir, los sistemas de FSMs en comunicación aceptan cuando una de sus máquinas acepta, independientemente de cuáles sean los contenidos de los buffers.

Como es habitual, gráficamente los distintos estados de aceptación de las FSMs los representaremos con círculos dobles, para diferenciarlos de los círculos simples que representan el resto de estados.

Naturalmente habría más maneras de definir el concepto de aceptación en este tipo de sistemas, como aceptar por *buffers vacíos*, con unos ciertos contenidos de los buffers, generando literales especiales, cuando *cada* máquina estuviera en un estado de aceptación... No obstante, consideramos que la forma que hemos definido y usaremos es bastante útil y versátil en la práctica. De hecho, incluso podría servir para simular otras de las formas que acabamos de comentar.

De esta forma, los sistemas de FSMs en comunicación tienen una forma de reconocer lenguajes: diremos que un sistema $\mathcal{S} = (F_1, F_2)$ acepta una palabra $\omega \in \Sigma^*$ si puede llegar a una configuración de aceptación desde la configuración inicial $(q_1^0, \omega, q_2^0, \epsilon)$, es decir, la configuración de $\mathcal{S}$ en la cual cada FSM está en su estado inicial y la palabra a evaluar está íntegramente en el buffer de la primera máquina (sin pérdida de generalidad, pues podríamos haberlas renombrado). Notemos que esta forma de aceptar se parece bastante a la de las máquinas de Turing usuales, pues en ellas también la configuración inicial parte de tener la palabra por completo en la cinta, con el cabezal apuntando a su principio.

3.1. Sistemas donde los outputs no proliferan

Como comentábamos al inicio, en este estudio pretendemos hacer comparativas entre diferentes tipos de definiciones de los sistemas. En este sentido, vamos a definir un primer subtipo de nuestros sistemas generales.

Como hemos visto en las páginas anteriores, gran parte del potencial de los sistemas de FSMs en comunicación parece provenir del fenómeno de proliferación de los outputs. Para confirmar esta intuición, posteriormente haremos una comparativa con el comportamiento de sistemas en los cuales los outputs no proliferan. Para ello, pasamos a definirlos brevemente.

Definición 3.2. *Decimos que $\mathcal{S} = (F_1, F_2)$ es un **sistema de FSMs en comunicación en el cual los outputs no proliferan** si se cumple que las funciones de transición de las máquinas F_1 y F_2 no dejan proliferar los outputs, es decir, si $\delta_1 : Q_1 \times I_1 \longrightarrow Q_1 \times O_1^?$ y $\delta_2 : Q_2 \times I_2 \longrightarrow Q_2 \times O_2^?$.*

De esta forma, en el sistema $\mathcal{S}$ la cantidad de literales que hay en B_1 y B_2 va a permanecer controlada en cada momento: como tanto F_1 como F_2 solo producen

como mucho tantos outputs como inputs consumen, deducimos que la cantidad de literales en $\mathcal{S}$, es decir, $|B_1| + |B_2|$ nunca puede crecer a lo largo de cualquier ejecución.

Capítulo 4

Expresividad de los sistemas

Una vez claras las definiciones, vamos a estudiar primeramente cuál es la expresividad de los sistemas que acabamos de definir. Va a resultar interesante comprobar que, a pesar de la simplicidad de las definiciones, los sistemas de FSMs en comunicación son Turing completos. Para la demostración de este hecho nos basaremos en una comparativa con los autómatas con cola, que presentamos aquí brevemente.

Definición 4.1. *Un **autómata con cola** $\mathcal{M}$ es una tupla $\mathcal{M} = (Q, \Sigma, \Gamma, \$, q_0, \delta)$, donde Q es un conjunto (finito) de estados, $\Sigma \subset \Gamma$ es el alfabeto usado por los inputs, Γ es el alfabeto que usa la cola, $\$ \in \Gamma \setminus \Sigma$ es el símbolo inicial de la cola, $q_0 \in Q$ es el estado inicial de $\mathcal{M}$ y $\delta : Q \times \Gamma \to Q \times \Gamma^*$ es la función de transición.*

La función de transición actúa de forma similar al caso de los autómatas con pila, que también utilizan cierta memoria adicional, pero en esta ocasión utilizando una estrategia LIFO: las transiciones toman un literal del principio de la cola y devuelven varios literales que se meten por el final de la cola. Cabe mencionar también que los autómatas con cola aceptan por cola vacía.

Lo más interesante para nosotros en este momento es que los autómatas con cola forman un sistema Turing completo, pues pueden simular cualquier máquina de Turing, y de hecho en tiempo polinómico (como se puede consultar en [19], ejercicio 99, por ejemplo). De esto resulta que una demostración sencilla de la Turing completitud de los sistemas de FSMs en comunicación se base en reducir un autómata con cola a uno de estos sistemas. Además, aunque no hace falta para probar la Turing completitud, esta reducción se puede hacer polinómica, lo cual será de utilidad posterior para demostrar otras propiedades. En el siguiente resultado vamos a desarrollar esta idea en profundidad.

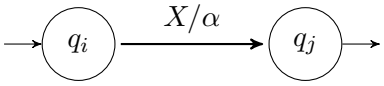
Teorema 4.2. *El conjunto de sistemas de FSMs en comunicación en los cuales los outputs proliferan (según la definición 3.1) es Turing completo.*

Demostración. Sea $\mathcal{M} = (Q, \Sigma, \Gamma, \$, q_0, \delta)$ un autómata con cola, según la definición anterior 4.1. Entonces podemos crear un sistema $\mathcal{S} = (F_1, F_2)$, con buffers B_1, B_2, de dos máquinas en comunicación que simulen el comportamiento de $\mathcal{M}$ de forma que F_1 simule las transiciones de $\mathcal{M}$ (a través de la función de transición δ_1), mientras que F_2 simplemente haga que todos los inputs que le llegan desde B_2 acaben en B_1 (a través de δ_2). De esta forma, al ser los buffers colas, se comportarán como la cola de nuestro autómata $\mathcal{M}$.

No obstante, hemos de poner atención a la hora de definir F_1, pues la forma que tiene $\mathcal{M}$ de hacer que los outputs proliferen es generando varios literales con una

misma transición (recordemos $\delta : Q \times \Gamma \to Q \times \Gamma^*$), mientras que F_1 lo hace un tanto diferente, a través de su función de transición $\delta_1 : Q_i \times \Sigma^? \longrightarrow Q_1 \times \Sigma^?$. En el primer capítulo ya vimos la intuición de que realmente estas dos formas de hacer proliferar los outputs eran equivalentes, y aquí vamos a ver la construcción que necesitamos más en detalle.

Para cada estado $q_i \in Q$, F_1 tendrá asimismo un estado $q_i^1 \in Q_1$. Veamos ahora qué construcción hacemos para simular una transición arbitraria $\delta(q_i, X) = (q_j, \alpha)$ de $\mathcal{M}$. Si el tamaño de α es $n \in \mathbb{N}$, podemos decir que $\alpha = Y_1 + Y_2 + \ldots + Y_n$, donde $Y_k \in \Sigma$ son literales de nuestro alfabeto. En el caso de que $n < 2$ podemos simular fácilmente la transición de δ incluyendo $\delta_1(q_i^1, X) = (q_j^1, \alpha)$. Y en el caso de que $n \geq 2$, simplemente tenemos que añadir $n-1$ estados auxiliares $q_{i,X,1}^1, q_{i,X,2}^1, \cdots, q_{i,X,n-1}^1$, junto con las transiciones $\delta_1\left(q_i^1, X\right) = \left(q_{i,X,1}^1, Y_1\right)$,$\delta_1\left(q_{i,X,1}^1, \epsilon\right) = \left(q_{i,X,2}^1, Y_2\right)$, ..., $\delta_1\left(q_{i,X,n-2}^1, \epsilon\right) = \left(q_{i,X,n-1}^1, Y_{n-1}\right)$. La construcción la podemos ver gráficamente como en la siguiente figura:



se convierte en:

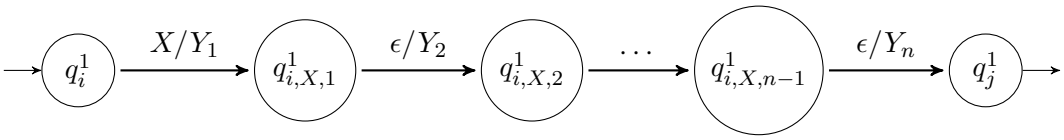


Figura 4.1: Visualización de la transformación para que las FSMs solo produzcan un output con cada transición.

Tenemos que tener en cuenta ahora que hay que simular la aceptación de $\mathcal{M}$, que como dijimos es por cola vacía. Al igual que se hace usualmente con los autómatas con pila, en los cuales se introduce un símbolo especial que denota el fin de la pila, aquí introduciremos el símbolo # con el mismo fin. Lo primero que hará F_1 será generar dicho literal desde su estado inicial $q_\#^1$, que tendrá una única transición: $\delta_1(q_\#^1, \epsilon) = (q_0^1, \#)$, donde q_0^1 es el estado asociado al estado inicial q_0 de $\mathcal{M}$. A partir de ahí, F_1 no tendrá en cuenta el literal #, pues no influye en la simulación de $\mathcal{M}$. Por ello, cada estado $q \in Q_1$ obviará las # mediante la transición $\delta_1(q, \#) = (q, \#)$.

Visto esto, la máquina F_2 sería realmente simple, como explicábamos en la discusión anterior: podría simplemente contar con un estado q_0^2 (que sería el estado inicial de F_2), y con las transiciones $\delta_2(q_0^2, A) = (q_0^2, A)$ para todos los $A \in \Gamma$, de forma que lo único que hace es devolver lo que lee. Y tiene que tener también un mecanismo de detectar la aceptación. Como tenemos que simular la aceptación de $\mathcal{M}$, que es por cola vacía, $\mathcal{S}$ debería aceptar cuando sus buffers están vacíos. Pero para ello hemos introducido el símbolo #: cuando $\mathcal{S}$ llegue a simular a $\mathcal{M}$ con la cola vacía, el único símbolo que habrá en los buffers B_1, B_2 sería #. Por tanto, F_2

puede aceptar en cuanto vea # dos veces seguidas. Una representación gráfica se puede ver en la figura 4.2.

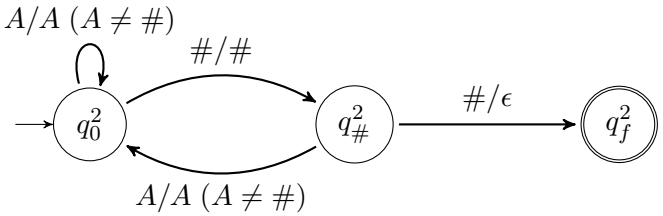


Figura 4.2: La FSM F_2 de nuestro sistema.

De esta forma, el sistema que conforman F_1, F_2 simula el autómata $\mathcal{M}$, usando el alfabeto $\Gamma \cup \{\#\}$, con estados iniciales $(q_\#^1, q_0^2) \in Q_1 \times Q_2$ y estado final $q_f^2 \in Q_2$. □

Notemos además que por cada output que genera $\mathcal{M}$, para simular dicha producción de un output nuestro sistema $\mathcal{S}$ produce únicamente 2 outputs (uno F_1 y otro F_2) Esto quiere decir que los tiempos de ejecución de $\mathcal{M}$ y $\mathcal{S}$ se relacionan también con únicamente ese factor constante 2. Por tanto, la simulación se hace en tiempo polinómico respecto a lo que tarda $\mathcal{M}$ (por cada paso de $\mathcal{M}$, $\mathcal{S}$ da una cantidad $\mathcal{O}(1)$ de pasos). En cuanto a espacio también tenemos una transformación polinómica, lo cual podemos ver en la cantidad de estados de F_1 y F_2. Y como la reducción de máquinas de Turing a autómatas con cola se puede hacer también con únicamente un aumento polinómico de tiempo y espacio, concluimos que la reducción de máquinas de Turing a sistemas de FSMs en comunicación es polinómica.

Capítulo 5

Definición general de los sistemas

Visto ya el interés que tienen los sistemas de FSMs en comunicación a través de su gran expresividad a pesar de su aparente sencillez, como acabamos de comprobar al ver que conforman un sistema Turing completo, pasamos a estudiar una generalización natural de los sistemas que habíamos considerado hasta ahora, introduciendo un mayor número de FSMs.

Sean $F_i = (Q_i, I_i, O_i, q_i^0, \delta_i)$, con $1 \leq i \leq n$, un conjunto de n FSMs. Definimos $\mathcal{S} = (F_1, F_2, \ldots, F_n)$ como el sistema de FSMs en comunicación asociado a $F_1, \ldots, F_n$. Estas máquinas se comunican entre sí a través de ciertos buffers $B_1, B_2, \ldots, B_n$, de forma que F_i consume literales de B_i (desde uno de los extremos del buffer), mientras que cualquier máquina puede escribir literales en B_i (por el extremo contrario al que F_i lee). La configuración de $\mathcal{S}$ dependerá en cada instante del estado en que está cada máquina F_i y del contenido de cada buffer B_i.

Cabe destacar que en este caso vamos a interpretar los outputs de cada máquina F_i de una manera que nos va a resultar más cómoda. Podemos suponer que cada función de transición ha cambiado su rango: $\delta_i : Q_i \times I_i^? \longrightarrow Q_i \times \left(O_i^?\right)^n$ (donde $Q_i \times \left(O_i^?\right)^n \subseteq Q_i \times I_1^? \times I_2^? \times \cdots \times I_n^?$ para que la comunicación tenga sentido), ya que ahora cada transición de F_i va a generar (potencialmente) un output para cada una de las máquinas de $\mathcal{S}$. Notamos además que, al igual que en el caso más sencillo en el cual solo teníamos 2 máquinas, los correspondientes alfabetos de las máquinas tienen que ser coherentes entre sí para que se pueda establecer una comunicación entre ellas. En una situación general, deberíamos tener $O_i \subseteq I_j$ para cada par $1 \leq i, j \leq n$, de forma que F_j entiende los literales que le manda F_i. En los casos donde no sea necesario (y de hecho, resulte engorroso) lidiar con esta suerte de detalles, denotaremos $\Sigma := I_1 \cup I_2 \cup \cdots \cup I_n \cup O_1 \cup O_2 \cup \cdots \cup O_n$ como el alfabeto de todas las máquinas de $\mathcal{S}$ por simplicidad, ya que generaliza al resto de alfabetos.

La función de transición global entre las distintas transiciones de $\mathcal{S}$ toma ahora la forma $\delta : Q_1 \times \Sigma^* \times Q_2 \times \Sigma^* \times \cdots \times Q_n \times \Sigma^* \longrightarrow Q_1 \times \Sigma^* \times Q_2 \times \Sigma^* \times \cdots \times Q_n \times \Sigma^*$ entre las diferentes configuraciones de $\mathcal{S}$ que nos ayuda a saber cómo evoluciona este sistema. Esencialmente, simula ejecuciones de cada máquina por separado, igual que en el caso de únicamente dos máquinas, de forma que en cada transición de $\mathcal{S}$ hay una máquina F_i que da un paso en su ejecución: lee de su buffer B_i, avanza en la dirección que le indique ese literal leído y envía al resto de máquinas los literales que la transición le indique, dejándolos en sus respectivos buffers. Vista la intuición, escribámoslo de manera más precisa.

Definición 5.1. *Sea un conjunto de n FSMs, $F_1, F_2, \ldots, F_n$, donde cada máquina es $F_i = (Q_i, I_i, O_i, q_i^0, \delta_i)$, con función de transición $\delta_i : Q_i \times \Sigma^? \longrightarrow Q_i \times \left(\Sigma^?\right)^n$, como antes hemos descrito. Definimos entonces el* ***sistema de FSMs en comunicación*** *asociado a estas máquinas como $\mathcal{S} = (F_1, F_2, \ldots, F_n)$. Una configuración de $\mathcal{S}$ es una tupla de $Q_1 \times \Sigma^* \times Q_2 \times \Sigma^* \times \cdots \times Q_n \times \Sigma^*$.*

Además, la función de transición de $\mathcal{S}$, δ, tiene transiciones que simulan pasos de las distintas FSMs F_i, con $1 \leq i \leq n$, pues está definida de la siguiente forma:

$$B_i = X + B_i' \wedge \delta_i(q_i, X) = (q_i', \alpha_1, \alpha_2, \ldots, \alpha_n) \implies$$
$$(q_1, B_1 + \alpha_1, \ldots, q_{i-1}, B_{i-1} + \alpha_{i-1}, q_i', B_i' + \alpha_i, q_{i+1}, B_{i+1} + \alpha_{i+1}, \ldots, q_n, B_n + \alpha_n)$$
$$\in \delta(q_1, B_1, \ldots, q_i, B_i, \ldots, q_n, B_n)$$

De nuevo, una configuración $(q_1, B_1, \ldots, q_n, B_n)$ de $\mathcal{S}$ se puede entender como el conjunto de estados $q_1, \ldots, q_n$ en que están las distintas FSMs y el contenido de los buffers $B_1, \ldots, B_n$ en ese determinado instante. Desde una de sus configuraciones, el sistema puede transitar a otras configuraciones con literales distintos de ϵ (como mucho habrá n de este tipo), aquellas que resultan de avanzar un paso en la ejecución de cada una de las máquinas que no tengan su buffer de lectura vacío (y que su función de transición les permita avanzar), y de modificar los buffers adecuadamente con respecto a lo que dicta esta transición. Pero también hay otro tipo de transiciones: las asociadas a pasos en los cuales cada FSM *consume* ϵ de su buffer correspondiente, es decir, avanza sin consumir ningún literal.

Además, recuperamos el concepto de configuración de aceptación que ya teníamos en el caso de sistemas con únicamente 2 máquinas: nuestro sistema $\mathcal{S}$ tiene un conjunto de estados finales $Q_F \subseteq Q_1 \cup Q_2 \cup \ldots \cup Q_n$ de forma que una configuración $(q_1, B_1, q_2, B_2, \ldots, q_n, B_n)$ es de aceptación $q_i \in Q_F$ para algún índice $i \in \{1, \ldots, n\}$.

De igual manera, la forma en que estos sistemas reconocen palabras es la misma que la que usaban sus análogos con únicamente dos máquinas y que explicamos en 3: consideraremos que un sistema acepta una palabra si una configuración de aceptación es alcanzable tras comenzar en la configuración inicial que surge de tener a cada FSM en su estado inicial y todos los buffers vacíos, excepto el de la (sin pérdida de generalidad) primera máquina, que contiene dicha palabra.

En estos sistemas el fenómeno de proliferación de los outputs es todavía más claro: además de lo que ocurría en el caso con únicamente dos máquinas, que aquí también se da (pues seguimos pudiendo generar outputs sin consumir inputs), por cada transición de una máquina se está consumiendo un literal (a lo sumo), mientras que potencialmente se podrían generar n literales como output, lo cual hace que se puedan generar muchos más outputs que los inputs que se consumen.

De hecho, podemos tener una situación en la cual haya máquinas que solo retransmiten lo que les llega a otra máquina. Por ejemplo, si F_1 quisiera mandarle dos literales, AB, a F_2, podría utilizar una máquina intermediaria F_3: F_1 mandaría A a F_2 y B a F_3, y después F_3 retransmitiría esa B para que le llegara a F_2. De esta forma, estaríamos consiguiendo mandar mensajes con más de un literal de F_1 a F_2 sin que nunca añadamos más de un literal a cada buffer en cada transición, lo cual nos indica que el fenómeno de proliferación de los outputs es completamente

inherente a la definición de estos sistemas, pues se puede conseguir incluso con la limitación de que no se pueden generar outputs sin consumir inputs.

De esta forma, al ser la proliferación de los outputs un fenómeno tan inherente a este tipo de sistemas, no nos debería molestar el hecho de que una FSM se pueda mandar mensajes también a sí misma (que en principio podría no parecer la mejor opción), pues siempre podríamos introducir una máquina intermediaria con el mismo propósito (como comentamos en el párrafo anterior). Por tanto, no tendría sentido limitar que una máquina no se pueda enviar mensajes a sí misma.

A la vista de lo anterior, queda claro que la única forma de evitar la proliferación de los outputs en este tipo de sistemas es que cada transición genere a lo sumo un output cada vez que consigue un input. Esto se daría si $\delta_i : Q_i \times I_i \longrightarrow Q_i \times \left(O_i^?\right)^n$ funciona de forma que para cada transición $\delta_i(q_i, \alpha) = (q_i', \alpha_1, \alpha_2, \ldots, \alpha_n)$ a lo sumo una de las producciones $\alpha_1, \ldots, \alpha_n$ es distinto de ϵ. La única manera de conseguir evitar el fenómeno de proliferación de los outputs sería por tanto hacer esta modificación de nuestra definición.

Por ello, al ser inherente a estos sistemas el fenómeno de proliferación de los outputs, podemos modificar ligeramente la definición de las FSMs que usamos, de forma que sean más cómodas de utilizar. En vez de considerar que la función de transición de la máquina F_i es $\delta_i : Q_i \times \Sigma^? \longrightarrow Q_i \times \left(\Sigma^?\right)^n$, podemos considerar equivalentemente $\delta_i : Q_i \times \Sigma^? \longrightarrow Q_i \times (\Sigma^*)^n$, lo cual nos simplificará algunos razonamientos en capítulos venideros. En realidad esto es simplemente una extensión de los argumentos que comentábamos a la hora de definir sistemas con solo dos FSMs, y que demostramos con mayor formalidad en el teorema 4.2, y por tanto no creemos que sea necesario incidir más en ello.

Notemos asimismo que este tipo de sistemas son Turing completos, pues ya hemos probado en el capítulo anterior que una subclase suya, en la que restringimos el número de máquinas a únicamente 2, ya es Turing completa (como vimos en el teorema 4.2).

5.1. Sistemas con buffers acotados

Para poder seguir estableciendo comparativas que nos ayuden a comprender qué características de los sistemas de FSMs en comunicación son los que realmente le dan su gran expresividad, vamos a estudiar otra modificación, en la que esta vez hay una limitación en cuanto a la memoria que el sistema puede usar.

Definición 5.2. *Decimos que un sistema de FSMs en comunicación $\mathcal{S} = (F_1, F_2, \ldots, F_n)$ es un **sistema con buffers acotados** si existe una constante M de manera que en cada una de las configuraciones $(q_1, B_1, q_2, B_2, \ldots, q_n, B_n)$ alcanzables de $\mathcal{S}$, el tamaño de cada buffer es a lo sumo M, es decir, $|B_i| \leq M \; \forall \, 1 \leq i \leq n$.*

Esta limitación podría parecer excesiva una vez que nos damos cuenta de que en realidad, dada la capacidad finita de los buffers, en realidad este tipo de sistemas solo pueden recibir inputs de una longitud acotada por M (o Mn en caso de que

distribuyéramos de alguna forma dichas instancias sobre los cuales el sistema se ejecuta en los diferentes buffers). Esto haría que los únicos lenguajes que se pudieran reconocer con este tipo de sistemas fueran una subclase de los lenguajes finitos, un estadio muy pobre en la jerarquía de Chomsky. Para evitar esta situación un tanto indeseable, pues el hecho de que las máquinas estén en comunicación no añadiría nada a su capacidad expresiva, vamos a hacer una pequeña modificación. Podemos considerar que hay una máquina F_0 que es la que recibe los inputs del sistema, y por tanto su buffer no está acotado, es una FSM *normal*. Pero si dejamos que esta máquina participe en la comunicación con el resto de máquinas, realmente la hipótesis de que los buffers están acotados no tendría ya mucho sentido, pues se podría usar el de F_0, que no lo es. Por tanto, debemos mantener a F_0 al margen de estas comunicaciones. Podemos imponer que lo único que haga F_0 sea enviar inputs poco a poco a (sin pérdida de generalidad) F_1, siempre que B_1 no esté lleno, y que F_0 no interactúe de ninguna otra manera dentro del sistema. En concreto, no puede recibir mensajes de ninguna máquina. De esta forma, queda claro que estamos respetando la restricción de los buffers acotados: todos los cómputos que hace $\mathcal{S}$ utilizan solo los buffers acotados; mientras que el *truco* de añadir F_0 no mejora las características de $\mathcal{S}$, sino que solo permite que además se procesen palabras de longitud arbitraria como input, lo cual es bastante más razonable para un modelo de cómputo.

Capítulo 6

Sistemas donde los outputs no proliferan

Si reducimos la capacidad de comunicación de nuestros sistemas de FSMs no permitiendo que proliferen los outputs como en las situaciones anteriores, tiene sentido pensar que la expresividad de estos sistemas caerá, y puede que no estén como en el caso anterior a la misma altura que las máquinas de Turing en la jerarquía de Chomsky. Pero tampoco pierden demasiada expresividad, pues pueden reconocer lenguajes que no son incontextuales, como el de las palabras del tipo $w\#w$, con $w \in \Sigma^*$ para algún alfabeto no trivial, como por ejemplo $\Sigma = \{0, 1\}$ (lo cual veremos en más detalle a continuación). Esto nos sugiere buscar un estadio intermedio entre los autómatas con pila y las máquinas de Turing, como los autómatas linealmente acotados (LBA por sus siglas en inglés). En este caso, esta intuición va a ser correcta, pues va a existir una equivalencia de expresividad entre los LBA y nuestros sistemas de FSMs donde los outputs no proliferan.

Recordamos brevemente la definición de un LBA:

Definición 6.1. *Se dice que $F = (Q, \Gamma, \flat, q_0, Q_F, \delta)$ es un **LBA** si es una máquina de Turing (con estados Q, alfabeto Γ, símbolo de blanco $\flat$, estado inicial q_0, estados de aceptación Q_F y función de transición $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$) de forma que a lo largo de toda la ejecución procesar un input solo utiliza una cantidad de posiciones de su cinta que es lineal en el tamaño de dicho input.*

Es decir, ante un input de tamaño n, un LBA solo puede utilizar una cantidad $\mathcal{O}(n)$ de posiciones de su cinta; y por lo demás se comporta como una máquina de Turing usual. En concreto, se suelen usar símbolos delimitadores como $\vdash, \dashv$ en ambos extremos de la cinta para que quede claro que la cantidad de cinta que se puede usar está acotada desde el principio. Se puede consultar más sobre ellos en, por ejemplo, el capítulo 9.3 de [17].

6.1. Ejemplo de funcionamiento

Vista la anterior intuición sobre en qué nivel de la jerarquía de Chomsky podrían situarse los sistemas de FSMs en los cuales los outputs no proliferan, consideramos útil desarrollar más el ejemplo anterior para ver más en detalle un ejemplo en el cual la comunicación que se establece entre dos FSMs sencillas es suficiente para reconocer un lenguaje un tanto complejo, que no es incontextual.

Proposición 6.2. *El lenguaje* $L = \{w\#w : w \in \{0,1\}^*\}$ *no es incontextual, pero puede ser reconocido por un sistema de FSMs en comunicación donde los outputs no proliferan.*

Demostración. Demostramos los dos asertos por separado:

- L no es incontextual. Lo demostramos con una aplicación del lema del bombeo. Suponiendo que L fuera incontextual, sea n es la constante del lema del bombeo de L. Considerando la palabra $0^n1^n\#0^n1^n \in L$, tendría que admitir una descomposición $0^n1^n\#0^n1^n = uvwxy$, con $|vwx| \leq n$,$|vx| \geq 1$. Demostraremos que $uv^2wx^2y \notin L$ para cualquiera de estas descomposiciones (llegando por tanto a una contradicción). Como # no se puede duplicar sin que la palabra salga de L, tenemos 3 opciones:

  1. $\# \in w$. En ese caso, como $|vwx| \leq n$, tenemos que $v = 1^a$, $x = 0^b$, donde al menos uno de entre a y b es estrictamente positivo por ser $|vx| \geq 1$. Si $a > 0$, en uv^2wx^2y se descompensarán los unos en la palabra de la izquierda; y si $b > 0$, los ceros en la de la derecha. De ambos modos, $uv^2wx^2y \notin L$.
  2. # está en u o más a su izquierda. Entonces en uv^2wx^2y será forzosamente más larga la palabra de la derecha, pues solo se bombea en ella. Y necesariamente $uv^2wx^2y \notin L$.
  3. # está en y o más a su derecha. Es una situación análoga al punto anterior.

- Pero, de hecho, podemos encontrar un sistema de FSMs en el cual los outputs no proliferan y que reconoce este lenguaje. Nuestro sistema será $\mathcal{S} = (F_1, F_2)$. F_1 irá reconociendo las letras de la primera palabra, una a una, e irá confirmando que están en el mismo orden en la segunda palabra con la ayuda de F_2, que irá resaltando las letras con las que deberían coincidir. Un ejemplo de máquinas que podrían hacer esta tarea serían las siguientes:

1. Máquina F_1: lo primero que hace es insertar ♭ para simbolizar el fin de las palabras. Su funcionamiento es el siguiente: hace un barrido del buffer hasta que reconoce la primera letra (la de después de ♭), y entra en un estado especial donde la recuerda, y la quita del buffer. Espera hasta que F_2 le manda una letra con barra arriba (que serán nuevos literales del alfabeto en los cuales nos apoyaremos en la construcción), que es la confirmación de que después de # ha venido esa letra. Si coincide con lo que recordaba, vuelve a empezar el ciclo, hasta que los buffers se *vacían*, y en ese momento reconocería ♭#♭ y aceptaría. Una versión gráfica de la implementación la podemos encontrar en 6.1.

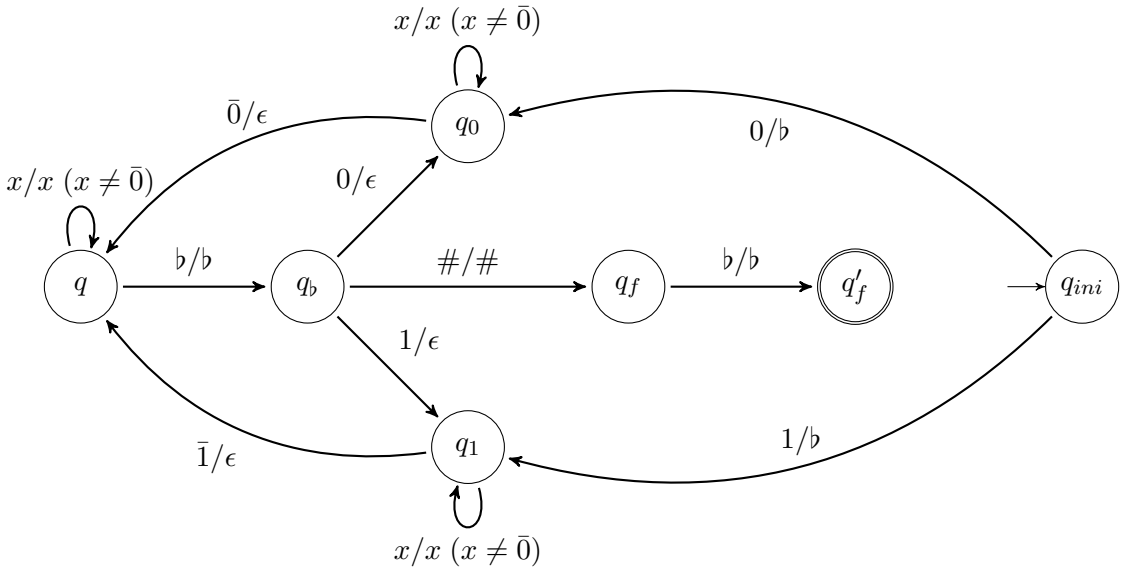


Figura 6.1: Representación gráfica de la máquina F_1.

2. Máquina F_2: su funcionamiento es el siguiente: hace un primer barrido hasta que llega a #, y marca el siguiente literal con una barra para que la máquina 1 lo reconozca. Después recorre la longitud de un buffer completo esperando a que la máquina 1 reconozca el literal que acaba de marcar. Tras esto, repite el ciclo que explicábamos, y vuelve a marcar lo que aparezca después de #. Gráficamente podemos pensar la implementación de esta FSM como en 6.2.

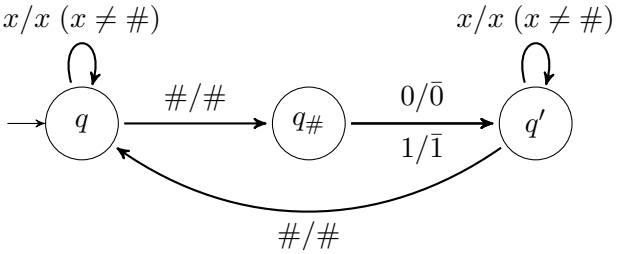


Figura 6.2: Representación gráfica de la máquina F_2.

□

6.2. Expresividad de los sistemas donde los outputs no proliferan

Veamos ahora que, de hecho, los sistemas de FSMs en comunicación en los cuales no se permite el fenómeno de proliferación de outputs solo pueden reconocer lenguajes dependientes del contexto, pues pueden ser simulados por LBAs.

La idea detrás de la demostración es que los autómatas linealmente acotados se ajustan perfectamente a esta situación, pues en ellos la información que hay en la cinta tiene siempre la misma longitud, y esa es exactamente la situación que nos encontramos en nuestro sistema de FSMs. En concreto, guardaremos la información de todos los buffers dentro de la cinta de nuestro LBA.

Para hacer el razonamiento más claro y comprensible, vamos a considerar primero el caso de que tenemos un sistema de dos FSMs en comunicación. Después quedará claro que el razonamiento se puede hacer extensible al caso general.

Proposición 6.3. *Los sistemas de 2 FSMs en comunicación en los cuales los outputs no proliferan pueden ser simulados por LBAs.*

Demostración. Sea $\mathcal{S} = (F_1, F_2)$ nuestro sistema de FSMs, con buffers B_1, B_2. Vamos a construir un LBA $F = (Q, \Gamma, \flat, q_0, Q_F, \delta)$ que simule el comportamiento de este sistema. La cinta de F tendrá siempre la información de B_1 y B_2: será una concatenación de ambos buffers, con un símbolo especial separador entre los dos (denotado aquí por $\# \in \Gamma \setminus \Sigma$), y con otros dos símbolos especiales que denotarán el inicio y el final de la franja usable de la cinta. Recordemos que en un LBA solo podemos usar una cantidad lineal en el número de posiciones de la cinta que ocupe el input inicial, así que lo rodearemos con estos símbolos: $\vdash \in \Gamma \setminus \Sigma$ marcará el inicio y $\dashv \in \Gamma \setminus \Sigma$, el final. La forma de la cinta sería entonces la siguiente: $\vdash B_1 \# B_2 \dashv$. Para ser coherentes con la representación indexada que definimos de los buffers, B_1 y B_2 están mapeados de izquierda a derecha según crecen sus índices.

Podemos suponer igualmente que la situación inicial de la cinta de F es exactamente $\vdash B_1 \# B_2 \dashv$, donde B_1, B_2 son los buffers al inicio de la ejecución, pues esto no supone una complejidad de cálculo alguna.

Pasemos ahora a ver más en detalle cómo se construye F. Como comentábamos, $\Gamma := \Sigma \cup \{\vdash, \dashv, \#, \flat\}$. Para cada par de estados $(q_i, q_j) \in Q_1 \times Q_2$, F tendrá un estado $q_{i,j}$ que lo simula. Para simular cada transición de $\mathcal{S}$ necesitaremos hacer recorridos por la cinta de F para modificarla de manera acorde con los contenidos de los buffers B_1, B_2. Para que estas modificaciones de la cinta sean correctas, usaremos una serie de estados auxiliares: al simular una transición de F_1 que transita de q_i a $q_{i'}$, mientras que F_2 está en el estado q_j, tendremos un conjunto de estados auxiliares del tipo $q_{i \to i',j}$ (que variarán dependiendo del tipo de transición, como veremos en unas líneas). De manera análoga se generarán estados del tipo $q_{i,j \to j'}$ para simular transiciones de F_2.

Vamos a ver ahora de forma más detallada cómo son las construcciones concretas que hacen que F pueda simular el sistema $\mathcal{S}$:

- Por cada transición $\delta_1(q_i, a) = (q_{i'}, b)$ de F_1, con $b \neq \epsilon$, tendremos que transformar, cuando sea posible (es decir, cuando haya una a al inicio de la zona donde representamos B_1 en la cinta), el estado de la cinta de $\vdash a\alpha\#\beta \dashv$ a $\vdash \alpha\#\beta b \dashv$. Esto es un sencillo ejercicio de programación en autómatas, que se puede resolver de la siguiente forma ayudándonos de ciertos estados auxiliares: simplemente desplazamos el contenido de toda la cinta una posición a la izquierda, siguiendo el bucle de los estados $q^1_{i \to i',j}$ y los estados del tipo

$q^{x,0}_{i\to i',j}, q^{x,1}_{i\to i',j}$, con x recorriendo Γ; y hecho todo este recorrido, ponemos una b al final de la cinta, en la posición correspondiente al final de B_2. Este diseño es como ilustra la figura 6.3.

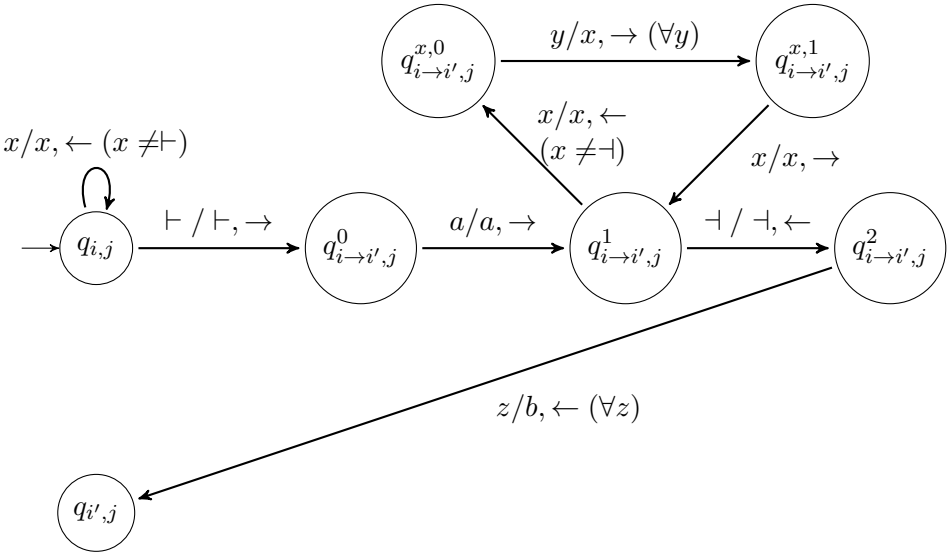


Figura 6.3: Representación gráfica de la implementación de una transición del tipo $\delta_1(q_i, a) = (q_{i'}, b)$.

- En el caso de que tengamos una transición de F_1 que no genere un literal como output, es decir, $\delta_1(q_i, a) = (q_{i'}, \epsilon)$, la situación es muy similar: basta repetir el bucle de la construcción precedente, y terminar desplazando el símbolo $\dashv$ también una posición a la izquierda, como ilustra la figura 6.4.

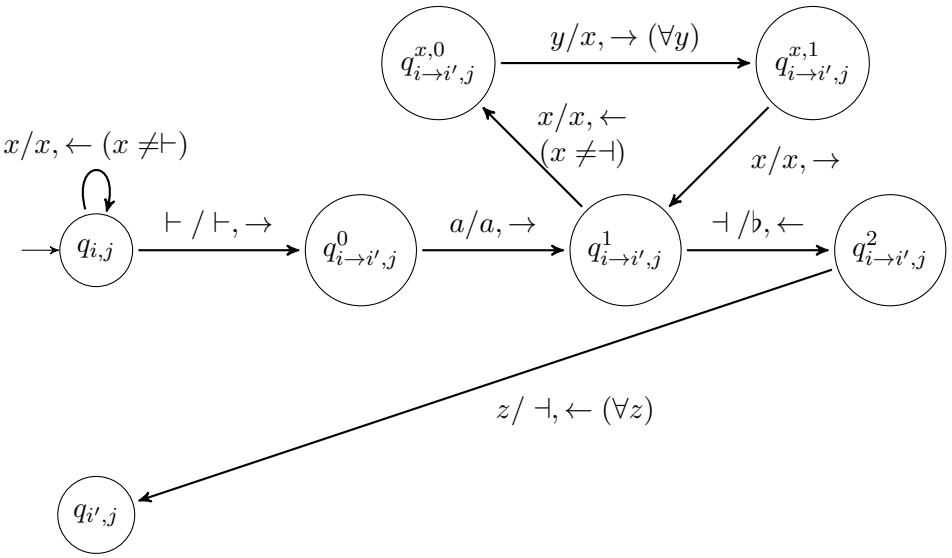


Figura 6.4: Representación gráfica de la implementación de una transición del tipo $\delta_1(q_i, a) = (q_{i'}, \epsilon)$.

- Por cada transición $\delta_2(q_j, a) = (q_{j'}, b)$ de F_2, con $b \neq \epsilon$ tenemos igualmente que transformar (cuando la configuración de la cinta sea adecuada, es decir, cuando la región donde está B_2 no esté vacía y tenga una a al principio) la cinta de ser $\vdash \alpha \# a \beta \dashv$ a $\vdash \alpha b \# \beta \dashv$. Esto es un ejercicio más sencillo aún que el anterior, pues básicamente se trata de encontrar el delimitador central # y hacer un par de cambios a su lado, como se puede comprobar en el diagrama 6.5.

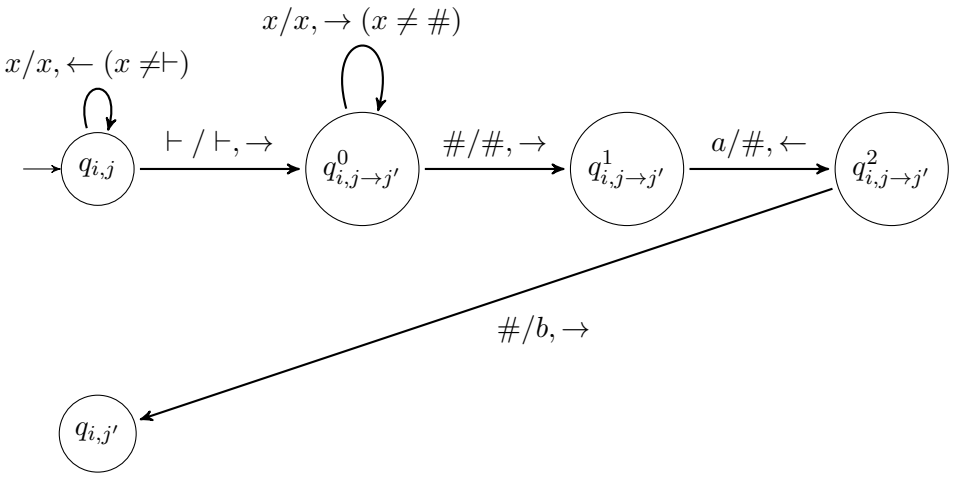


Figura 6.5: Representación gráfica de la implementación de una transición del tipo $\delta_2(q_j, a) = (q_{j'}, b)$.

- Y por último, tratamos el caso de que tengamos una transición de F_2 que no produzca outputs: si tenemos una transición $\delta_2(q_j, a) = (q_{j'}, \epsilon)$, tenemos que trasladar la región de la cinta asociada a B_2 hacia la izquierda, para borrar el literal que estaba en su primera posición. Esto se puede hacer con un esquema muy similar al bucle que nos encontrábamos al simular los movimientos de F_1, como podemos comprobar gráficamente en la figura 6.6.

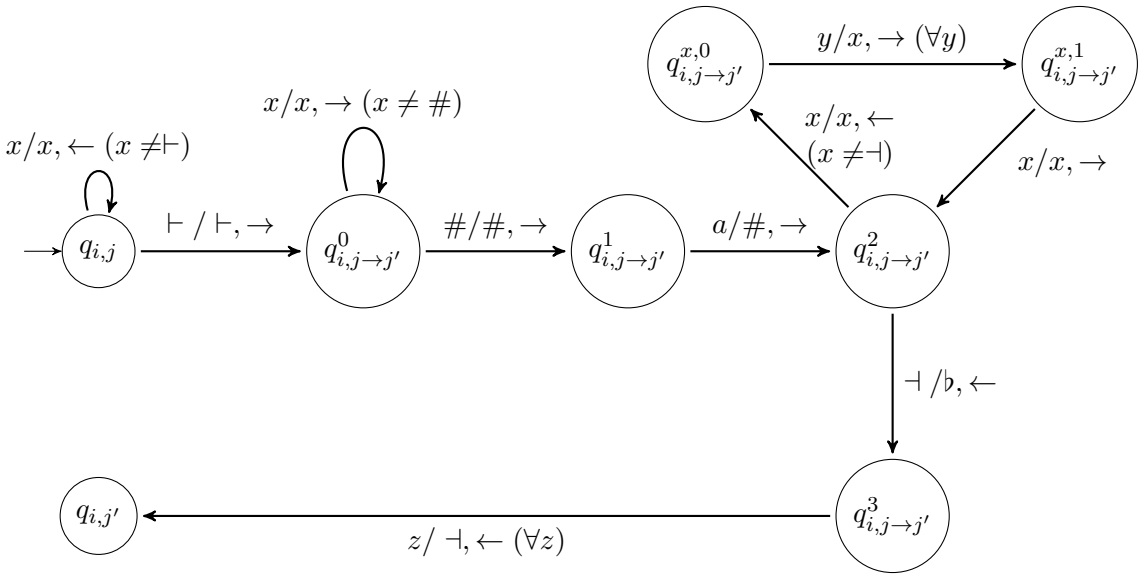


Figura 6.6: Representación gráfica de la implementación de una transición del tipo $\delta_2(q_j, a) = (q_{j'}, \epsilon)$.

Por último hemos de comentar que los estados finales de nuestro LBA F serán simplemente aquellos estados $q_{i,j}$ que simulen una configuración de estados $(q_i, q_j) \in Q_1 \times Q_2$ que sea de aceptación en $\mathcal{S}$.

De esta forma, es claro que, dada la construcción que hemos hecho, hay una ejecución de F de forma que su cinta va reflejando exactamente el contenido de los buffers B_1 y B_2 a lo largo de cualquier ejecución del sistema $\mathcal{S}$. Además, la cinta de F solo utiliza una cantidad limitada de cinta: como podemos ver en los diferentes casos que hemos ido desgranando, la distancia máxima entre los símbolos $\vdash$ y $\dashv$ en la cinta de F, que reflejan la cantidad de cinta que F usa, se da al inicio, pues esa distancia nunca aumenta (y de hecho puede disminuir si nos encontramos con alguna transición que no genera outputs en $\mathcal{S}$). Por tanto, el autómata F que hemos definido es realmente un LBA, como queríamos comprobar. Por tanto, hemos conseguido simular un sistema de FSMs mediante un LBA, y de ahí deducimos que estos sistemas de FSMs en los cuales los outputs no proliferan no pueden reconocer más que los lenguajes dependientes del contexto. □

Además, la reducción que hemos hecho tiene buenas propiedades: la cantidad de estados que tiene F es polinómica con respecto a la que tienen F_1 y F_2 juntas: por cada par $(q_i, q_j) \in Q_1 \times Q_2$ tenemos un estado en F. Y además, por cada una de las transiciones que pueden salir de este estado (hay a lo sumo $2|\Sigma|$ posibles), tenemos una cantidad acotada de estados auxiliares, a lo sumo $4 + 2|\Gamma|$. Por tanto, la cantidad de estados que hay en F la podemos acotar de la siguiente forma:

$$|Q| \leq |Q_1| \cdot |Q_2| \cdot (2|\Sigma| \, (4 + 2|\Gamma|)) \in \mathcal{O}(|Q_1| \, |Q_2|) \subset \mathcal{O}((|Q_1| + |Q_2|)^2)$$

que es una cantidad polinómica en la cantidad de estados de $\mathcal{S}$, $|Q_1| + |Q_2|$.

Además, el tiempo que tarda F en simular una ejecución de $\mathcal{S}$ es también polinómico. Notemos que para simular cualquier transición de $\mathcal{S}$, F pasa por cada

elemento de la cinta a lo sumo 3 veces (que recordemos que tiene un tamaño acotado por su tamaño inicial). Por tanto, visita una cantidad acotada de estados por cada transición que toma $\mathcal{S}$, y esto resulta en que la reducción es también polinómica en cuanto a tiempos de ejecución.

Con lo anterior quedaría completa la prueba para el caso de que el sistema de FSMs en comunicación tuviera únicamente 2 máquinas. Pero el razonamiento en el caso de que hubiera n máquinas sería bastante similar, como vamos a comprobar.

Teorema 6.4. *Cualquier sistema de FSMs en comunicación en el cual los outputs no proliferan puede ser simulado por un LBA.*

Demostración. Teniendo en mente la demostración del caso en el que teníamos únicamente 2 FSMs, vamos a esbozar las modificaciones que permiten demostrar el caso general. En este caso, en lugar de que F tenga estados formados por pares, tendrá estados que ser n-tuplas $q_{i_1,i_2,\ldots,i_n}$, que se encargaría de la simulación de la configuración del sistema cuando F_1 está en el estado q_{i_1}, F_2 en el q_{i_2}, y así sucesivamente. Asimismo, dada una transición en la máquina F_k del estado q_{i_k} al estado $q_{i'_k}$, usaríamos estados auxiliares $q_{i_1,\ldots,i_{k-1},i_k \to i'_k,i_{k+1},\ldots,i_n}$. Estos estados auxiliares ayudarían a modificar la cinta, que contendría el contenido de los n buffers uno detrás de otro: $\vdash B_1 \# B_2 \# \cdots \# B_n \dashv$.

Las transiciones del tipo $\delta_i(q_i, a) = (q_{i'}, \epsilon, \epsilon, \ldots, \overset{pos.\ j}{b}, \ldots, \epsilon)$, que hacen que la máquina F_i ponga un literal b en el buffer de F_j, se simularían del siguiente modo. Supongamos primero que $i > j$. En este caso, primero tendríamos que encontrar la posición del i-ésimo buffer en la cinta, y para eso hacemos un bucle en el cual contamos $i-1$ vistas de # según avanzamos hacia la derecha, y con un paso más llegamos a la a. Hecho esto, nos desplazamos a la izquierda hasta que veamos $i-j-1$ veces el literal #, y vamos copiando cada posición a su derecha, para simular el borrado de la a. Una vez lleguemos a la posición donde estaba la $i-j-1$-ésima #, ponemos la b, y ya quedaría modificado convenientemente el buffer. En el caso de que $i \leq j$, tendríamos que avanzar hacia la derecha copiando literales en su posición de la izquierda, pero el espíritu del proceso sería el mismo.

Es sencillo darse cuenta de que esta construcción, a pesar de ser un poco más compleja en cuanto a notación, funciona por el mismo motivo que lo hacía la anterior, pues el modo de razonar es exactamente el mismo. Igualmente el tamaño de la cinta de F nunca crece porque la cantidad de literales que hay en un sistema de FSMs donde los outputs no proliferan tampoco crece; y por tanto F es un LBA. □

Además, si consideramos n (el número de FSMs de nuestro sistema) prefijado, todas las construcciones que hemos comentado siguen siendo de tipo polinómico en cuanto al número de estados se refiere (aunque esta vez polinomios de grado n en vez de cuadráticos, claro), y siguen siendo también del mismo orden de complejidad en cuanto a tiempo de ejecución (este no depende de n), así que las generalizaciones naturales de la primera construcción heredan de alguna forma estas buenas propiedades.

Sabiendo ya que estos sistemas pueden reconocer únicamente lenguajes dependientes del contexto, vamos a afinar más. Ahora vamos a comprobar que este tipo de

sistemas no pertenece a ningún nivel inferior de la jerarquía de Chomsky pues, como vamos a demostrar, pueden simular cualquier LBA. Por tanto, son completos dentro de los autómatas dependientes del contexto. Veamos ahora esta demostración.

Cabe destacar que en este caso utilizaremos una definición un tanto distinta de los LBA, pues hay una definición equivalente que restringe la cantidad de cinta que puede usar un LBA a únicamente la que ocupa su input inicial. Esta versión, en la cual no permitimos una cantidad lineal de cinta extra, es equivalente, en términos de capacidad de cómputo, a la que propusimos anteriormente al definir los LBAs como consecuencia del Teorema del *linear speedup* en máquinas de Turing (se puede consultar en [17], capítulos 9.3, 12.2).

Teorema 6.5. *Cualquier LBA puede ser simulado por un sistema de FSMs en comunicación donde los outputs no proliferan.*

Demostración. Sea $F = (Q, \Sigma, \flat, q_0, Q_F, \delta_F)$ un LBA. Vamos a definir un sistema $\mathcal{S} = (F_1, F_2)$ de FSMs donde los outputs no proliferan que simule F. F_2 será una máquina muy sencilla, que simplemente devuelve todo literal que le llega. Podríamos representarla del siguiente modo que nos muestra la siguiente figura:

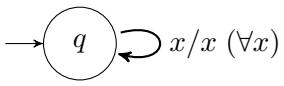


Figura 6.7: Representación gráfica de F_2.

La máquina F_1 llevará toda la complejidad de simular F. Como F es esencialmente una máquina de Turing usual, tiene una cinta, que intentaremos mantener en B_1. Pero F tiene además un cabezal que puede modificar literales de cualquier posición de la cinta, mientras que F_1 solo puede tomar literales desde el principio de su buffer B_1. Esto genera una dificultad, pero que se solventa de manera sencilla, como usualmente en estas situaciones: iterando sobre todo el buffer B_1 cada vez que F dé un paso. De esta manera, aunque de una forma un poco costosa, podemos simular cada movimiento de F con la generación de un nuevo buffer B_1, que será la cinta de F en el siguiente instante de tiempo.

Además, a la hora de simular esta cinta, aparecen ciertas dificultades. La primera es la de simular el cabezal de F. Lo solventaremos simplemente introduciendo más literales: por cada $\gamma \in \Sigma$, el nuevo literal $\bar{\gamma} \in \Gamma$ del alfabeto de $\mathcal{S}$ significará que en el buffer hay una γ y además en el cabezal está justo en esa posición.

Se nos presenta también la dificultad de que el cabezal de F puede avanzar hacia la izquierda y a la derecha en la cinta, mientras que nuestra máquina F_1 solo puede leer su buffer en una dirección (digamos que de izquierda a derecha). La idea para solventar esto es simplemente retrasar un ciclo todas las decisiones de F_1, pues así nunca tomaremos decisiones incorrectas del siguiente tipo: si teníamos $\alpha\bar{\beta}$ y tenemos que simular un movimiento a la izquierda del cabezal en una transición del tipo $\delta(q, \beta) = (q', \gamma, \leftarrow)$, estos literales tendrían que pasar a ser $\bar{\alpha}\gamma$; pero solo nos damos cuenta de que α tendrá el cabezal encima una vez que hemos visitado

β y hemos visto que ella sí tiene el cabezal encima. Un esquema visual de estas producciones atrasadas sería el siguiente (a_i^t denota el i-ésimo carácter de la cinta en el instante t, y las flechas discontinuas denotan con la lectura de qué literal se generan otros literales):

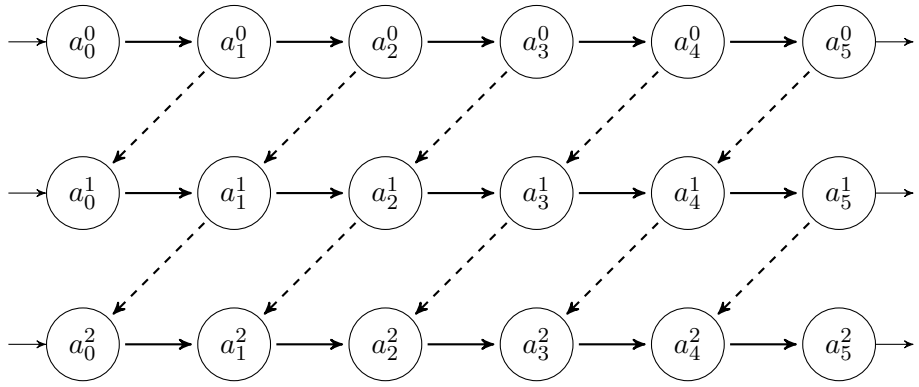


Figura 6.8: Ejemplo explicativo de en qué momento se lleva a cabo cada producción.

Además tenemos que conseguir este retraso en la ejecución sin que proliferen los outputs, lo cual no es inmediato de conseguir en una primera aproximación al problema. Esto se puede solucionar introduciendo un carácter # que indique el fin de la cinta de F en nuestros buffers (es decir, que separe los buffers que se refieren a diferentes instantes de tiempo), lo cual sigue la idea de los LBA de delimitar la memoria que podemos utilizar.

Con estas ideas en mente, pasemos a la implementación de manera más concreta de la máquina F_1. Lo primero es comentar que el alfabeto que usará el sistema $\mathcal{S}$ será, como antes se indicaba: $\Gamma := \Sigma \cup \{\bar{\gamma} : \gamma \in \Sigma\} \cup \{\#\}$.

Para cada estado $q_i \in Q$ de F tendremos un cluster de estados en F_1 que simularán un barrido de la cinta de F mientras está en el estado q_i, cuyos estados denotaremos q_i (además de más subíndices para indicar la función de esos estados dentro del cluster). En cada uno de estos clusters tenemos que implementar la idea anterior de retrasar la producción de outputs un ciclo, así que tendremos que tener un estado diferente para cada literal de Γ para recordar cuál es el literal que acabamos de ver. Además, tenemos que conseguir simular las transiciones de F. Por ejemplo, si en F hay una transición de q_i a q_j cuando el cabezal ve una A, nuestra máquina F_1, en su cluster relativo a q_i, tendrá una transición diferente cuando vea $\bar{A}$, pues transitará hacia una región diferente del cluster en la cual recordaremos que el siguiente estado al que va a ir F es q_j. De esta forma, cuando le llegue la siguiente #, F_1 transitará hacia el cluster relativo a q_j, y comenzará de nuevo a barrer la cinta de F leyendo de su buffer.

Para poder cumplir esto, cada cluster q_i tendrá un conjunto de estados q_i^γ, con $\gamma \in \Sigma \cup \{\#\}$, de forma que ejecutan la creación del nuevo buffer antes de que se haya visitado la posición donde está el cabezal lector (es decir, van devolviendo exactamente lo que leen, pero un ciclo más tarde). Posteriormente, al ver el cabezal, hemos de distinguir si las diversas transiciones de F en q_i mueven el cabezal a la

izquierda o a la derecha, pues se nos generará una estructura diferente de estados en cada caso.

Por ejemplo, si en F tenemos una transición que mueve el cabezal lector hacia la izquierda, del tipo que nos muestra la siguiente figura:

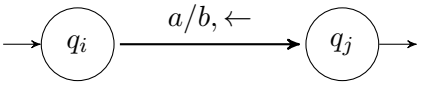


Figura 6.9: Representación gráfica de un ejemplo de transición que mueve el cabezal a la izquierda.

Tenemos por tanto que tener una estructura de estados dentro del cluster asociado a q_i que pueda solventar esta situación. Tendremos para ello un estado $q^0_{i\to j}$ que visitaremos justo cuando hayamos visto el cabezal con una A, y que nos ayudará a transitar hacia una zona del cluster donde recordamos que el siguiente estado que visitará F es q_j. Una vez visitado $q^0_{i\to j}$, nos encontramos con otra región del cluster donde lo único que hacemos es *copiar* el buffer para la siguiente iteración, al igual que hacíamos con los estados del tipo q^γ_i. Por eso llamamos a estos estados $q^\gamma_{i\to j}$, pero esta vez únicamente con $\gamma \in \Sigma$, pues la llegada de una # nos hará transitar hacia el cluster asociado a q_j. Suponiendo, por simplicidad para la representación gráfica, que $\Sigma = \{a, b\}$ (una vez visto el diagrama quedará claro que es sencillo generalizarlo a un alfabeto con más literales), podríamos plasmar estas ideas sobre el cluster de q_i de la forma que nos indica la figura 6.10.

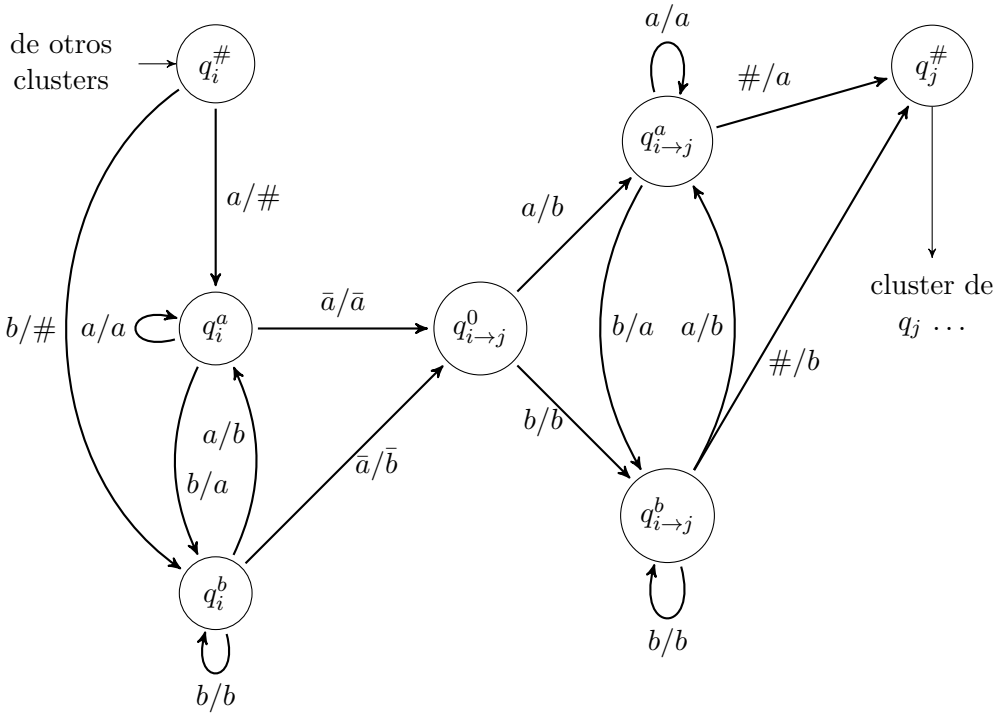


Figura 6.10: Representación gráfica de la implementación de un cluster simulando transiciones que mueven el cabezal hacia la izquierda.

Y, en cambio, si la transición de F mueve el cabezal a la derecha, tenemos que introducir algunos estados más para recordar los literales que vemos durante este movimiento. Supongamos entonces que tenemos en F tenemos la transición entre $q_i, q_j \in Q$, del tipo de la figura siguiente:

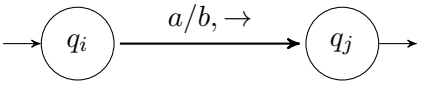


Figura 6.11: Representación gráfica de un ejemplo de transición que mueve el cabezal a la derecha.

Entonces en F tendremos que tener una estructura del estilo de la mostrada en la figura 6.12 dentro del cluster relativo a q_1, que es bastante similar a la anterior, pero donde introducimos algunos estados más $q_{i\to j}^{1,\gamma}$, con $\gamma \in \Sigma$ para recordar los literales que vemos justamente al llegar al cabezal (de nuevo supondremos $\Sigma = \{a, b\}$ por simplicidad del diagrama):

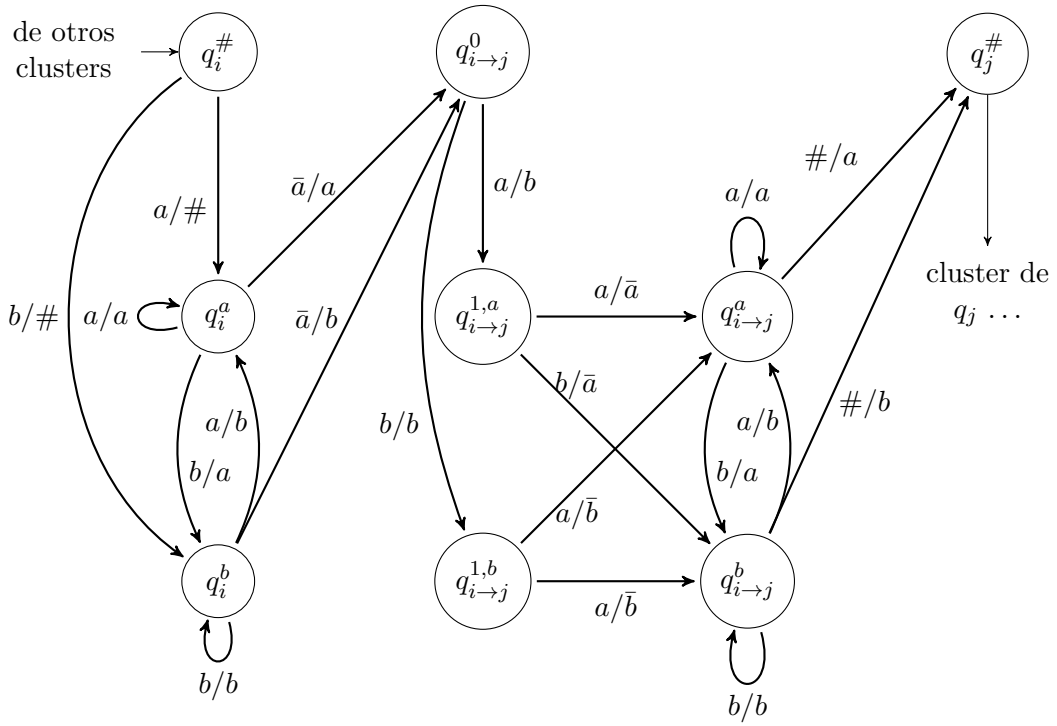


Figura 6.12: Representación gráfica de la implementación de un cluster simulando transiciones que mueven el cabezal hacia la derecha.

Naturalmente, desde un estado q_i de F podría darse el caso de que hubiera transiciones a más de un estado del tipo q_j en los ejemplos anteriores. Esto haría

que nuestro cluster asociado a q_i fuera más grande, ya que para cada transición de q_i a q_j, el cluster debería contener una estructura de estados $q_{i\to j}$. Es decir, dentro de un mismo cluster, tendremos que tener tantas estructuras del tipo $q_{i\to j}$ como transiciones tenga F en q_i.

De esta forma, en el cluster del estado q_i de F tenemos una columna de estados en los cuales no hemos visto todavía la posición del cabezal (los q_i^{γ}, después una serie de estados (un conjunto de estados por cada transición saliente de q_i en F) que manejan el tratamiento del literal con el cabezal y la transición de F (los $q_{i\to j}^{0}$, $q_{i\to j}^{1,\gamma}$), y una serie de columnas de estados (también una por cada transición de q_i) que recuerdan a qué estado transita F y, por tanto, cuál es el siguiente cluster al que irá F_1 (los estados del tipo $q_{i\to j}^{\gamma}$).

Así podemos ir simulando transiciones de F, que modifican un par de posiciones de la cinta, con un recorrido dentro de uno de los clusters de F_1, que genera un nuevo buffer simulando la cinta de F, y se la envía a F_2. F_2, por su parte, reenvía el buffer a F_1 para que pueda efectuar la siguiente transición.

Es importante hacer hincapié en el inicio de todo este procedimiento: al principio, el input de F se pone en el buffer B_1 (sin ninguna modificación), y el estado inicial de F_1 es $q_0^{\#}$ (recordemos que el estado inicial de F era q_0). Así, lo primero que hará F_1 será generar una #, lo cual le ayudará a distinguir ese buffer del que se corresponderá al siguiente estado de la cinta de F. Después seguirá tratando los literales de B_1 con total normalidad, empezando en el cluster de q_0, claro.

Y simular el criterio de aceptación de F también es sencillo: simplemente tenemos que hacer que $(q_i, q) \in Q_1 \times Q_2$ sea de aceptación siempre que q_i represente en F_1 un estado final de F, es decir, $q_i \in Q_F \subseteq Q$.

De esta forma, hemos hecho una reducción del LBA F a un sistema de FSMs en el cual los outputs no proliferan, como hemos comprobado a la hora de construirlos. □

Además, en el sistema de FSMs que hemos construido, el número de estados es polinómico con respecto a los de F:

$$\begin{aligned} |Q_{F_1}| &= \#clusters \cdot \#estados\ por\ cluster \\ &\leq |Q_F| \cdot \big((|Q_F|+1)\cdot(|\Sigma_F|+1)+1+|\Sigma_F|+1\big) \in \mathcal{O}\left(|Q_F|^2\right) \end{aligned}$$

Esto demuestra que la creación del sistema de FSMs en función de F se puede hacer con una cantidad de estados polinómica respecto al tamaño de F. Además, la reducción, en cuanto a tiempos de ejecución, también es polinómica: si tenemos un input de longitud n y, para procesar dicho input, F da k pasos; entonces nuestro sistema $\mathcal{S}$, por cada uno de estos k pasos recorrerá la cinta entera de F dos veces (una será por un recorrido de F_1 y otra por el de F_2). Pero como F es un LBA, el tamaño de la cinta siempre está acotado por n. Es decir, el sistema $\mathcal{S}$ dará $\mathcal{O}(nk)$ pasos en su ejecución. Por tanto, podemos afirmar que la reducción es polinómica y no se añade por tanto un sobrecoste reseñable.

Capítulo 7

Sistemas con buffers acotados

Otro caso que podemos estudiar es el de un sistema de FSMs en comunicación en el cual los buffers, por alguna razón, tienen tamaño acotado desde el principio, como definimos en el apartado 5.1. Esto supone una limitación similar a la estudiada en el caso de que los outputs no proliferaran en el sentido de que estamos limitando la expresividad de nuestro sistema. En el caso de que los outputs no proliferaran ya estudiamos que el sistema dejaba de ser Turing completo, y se quedaba en un estadio un poco anterior de la jerarquía de Chomsky, el de los lenguajes dependientes del contexto. En este caso, vamos a ver sin mucha dificultad que podemos comprobar que la limitación de la expresividad es mucho más potente, y este tipo de sistemas se van a quedar en lo más bajo de la jerarquía, en los lenguajes regulares. Esto se debe a que la información que potencialmente puede tener el sistema es en cualquier caso finita, pues las FSMs involucradas lo son, y los buffers también. Esto contrasta con el caso anterior por una sutileza: cuando no proliferaban los outputs, la información estaba acotada dependiendo del tamaño inicial del input, pero como el tamaño inicial del input no está acotado de antemano, la cantidad de información que tenían esos sistemas es mucho mayor que la de los sistemas que pasamos a estudiar ahora.

Vamos a proceder a simular un sistema de FSMs con buffers acotados mediante un autómata finito. Esto demostraría que nuestro sistema de FSMs como mucho puede reconocer lenguajes regulares. Pero como, en concreto, el sistema está formado por FSMs, que son autómatas finitos, también el sistema puede simular cualquier autómata finito, y esto demuestra la equivalencia entre ambos modelos. La conclusión clara es que añadir buffers acotados a unas FSMs para que se comuniquen entre ellas no añade en ningún caso expresividad.

Teorema 7.1. *Cualquier sistema de FSMs en comunicación con buffers acotados puede ser simulado por un autómata finito.*

Demostración. Sea $\mathcal{S} = (F_0, F_1, F_2, \ldots, F_n)$ un sistema de FSMs en comunicación con buffers acotados, con buffers $B_0, B_1, B_2, \ldots, B_n$. Recordemos que esto significa que existe un M prefijado de forma que la longitud de todos los buffers (excepto B_0, que corresponde a la máquina especial F_0 según definimos en la sección 5.1) está siempre acotada por M. Digamos que el alfabeto común de $\mathcal{S}$ es Σ, y con él definimos $\Sigma^{\leq M} := \{w \in \Sigma^* : |w| \leq M\}$, que nos ayudará a simplificar las expresiones relativas a los buffers. Además, las máquinas serán $F_i = (Q_i, I_i, O_i, q_i^0, \delta_i)$, con $0 \leq i \leq n$, y $I_i, O_i \subseteq \Sigma$, naturalmente.

Definamos ahora el autómata finito $F = (Q, \Sigma, q^0, Q^F, \delta)$ que los simulará. La clave está en hacer $Q = Q_1 \times \Sigma^{\leq M} \times Q_2 \times \Sigma^{\leq M} \times \cdots \times Q_n \times \Sigma^{\leq M}$, que

simulará por completo el estado del sistema $\mathcal{S}$ de la siguiente forma: el estado $(q_1, \alpha_1, q_2, \alpha_2, \ldots, q_n, \alpha_n) \in Q$ simbolizará que F_1 está en el estado q_1, F_2, en q_2, y así sucesivamente; y que el buffer B_1 tiene por contenido la palabra α_1, B_2 tiene la palabra α_2, y así sucesivamente. Notemos que algunas de las posiciones de los buffers pueden estar vacías, pues no necesariamente $|\alpha_i| = M$, sino que en general tendremos $|\alpha_i| \leq M$ por ser palabras de $\Sigma^{\leq M}$.

Como vemos, el alfabeto será Σ, el mismo que tenía $\mathcal{S}$, lo cual es necesario para que F pueda aceptar los mismos símbolos. Para cada símbolo $\gamma \in \Sigma$ que marque una transición de F, querremos que signifique que la máquina F_0 introduce en ese instante el siguiente literal del input inicial, que esta vez será γ, dejándolo en el final del buffer B_1, claro. Pero en cada estado de F habrá más transiciones: aquellas que pueda dar el sistema $\mathcal{S}$ sin consumir literales del input (y por eso F consumirá únicamente ϵ), y que reflejan que una máquina F_i, con $i \geq 1$, ha dado un paso en su ejecución.

Nos falta definir la función de transición δ de F, lo cual es sencillo a partir de las ideas anteriores. Tenemos que entender cómo reacciona ante los diferentes literales de Σ, así que pasamos a definir estas transiciones.

Supongamos que estamos en un estado arbitrario $(q_1, \alpha_1, q_2, \alpha_2, \ldots, q_n, \alpha_n) \in Q$. Veamos cuáles son las diferentes transiciones que tiene este estado en F.

- Para los literales $\gamma \in \Sigma$, tenemos que añadir γ a B_1 en caso de que este buffer no esté lleno.

  $$|\alpha_1| < M \implies \delta((q_1, \alpha_1, q_2, \alpha_2, \ldots, q_n, \alpha_n), \gamma) = (q_1, \alpha_1 + \gamma, q_2, \alpha_2, \ldots, q_n, \alpha_n)$$

- Para las transiciones que se toman con ϵ, tenemos que simular una transición de la máquina F_i, para cada $1 \leq i \leq n$. Esta transición dependerá de $head(\alpha_i)$ (en caso de que el buffer B_i no esté vacío). Digamos que la transición es la siguiente: $\delta_i(q_i, head(\alpha_i)) = (q_i', \beta_1, \beta_2, \ldots, \beta_n)$, donde $q_i' \in Q_i$ es el estado de llegada y $\beta_j \in \Sigma^?$ es lo que se añade a B_j para $1 \leq j \leq n$ (recordemos que F_0 no participa en la comunicación, así que por simplicidad la omitimos). La transición será posible en caso de que ninguno de los buffers exceda su tamaño al añadir estos literales, lo cual podemos expresar de la siguiente forma:

  $$\begin{aligned} &|\alpha_i| > 0 \ \wedge \ \alpha_i = X + \alpha_i' \\ &\wedge \ \delta_i(q_i, X) = (q_i', \beta_1, \beta_2, \ldots, \beta_n) \\ &\wedge \ |\alpha_j + \beta_j| \leq M \ \forall 1 \leq j \leq n \ con \ i \neq j \implies \\ &(q_1, \alpha_1 + \beta_1, q_2, \alpha_2 + \beta_2, \ldots, q_i', \alpha_i' + \beta_i, \ldots, q_n, \alpha_n + \beta_n) \\ &\qquad \in \delta((q_1, \alpha_1, q_2, \alpha_2, \ldots, q_i, \alpha_i, \ldots, q_n, \alpha_n), \epsilon) \end{aligned}$$

Por último, comentar el estado inicial de la máquina F. Simplemente sería el estado $(q_1^0, \epsilon, q_2^0, \epsilon, \ldots, q_n^0, \epsilon)$. Y naturalmente, los estados de aceptación de F serán aquellos en los cuales $(q_1, q_2, \ldots, q_n)$ sea una configuración de aceptación de $\mathcal{S}$.

Con este sencillo procedimiento queda claro que el autómata F (finito) que hemos definido simula el comportamiento de $\mathcal{S}$, pues simula cada uno de los pasos que puede tomar en cada una de sus configuraciones. □

Además la reducción es polinómica conforme crece el tamaño de las máquinas del sistema $\mathcal{S}$: si $\mathcal{S}$ tiene k estados (es decir, $|Q_1| + |Q_2| + \ldots + |Q_n| = k$), entonces F tiene, a lo sumo, con una aproximación naïve, $(kM)^n$ estados, lo cual es una cantidad polinómica en k, supuesto que el número n de máquinas que conforman $\mathcal{S}$ y la constante que acota los buffers M son conocidas de antemano.

Capítulo 8

Sistemas con buffers sin orden

Otra modificación posible de los sistemas de FSMs en comunicación residiría en modificar el funcionamiento de los buffers. Hasta ahora, siempre han estado implementados en forma de cola, de forma que los literales siempre quedaban ordenados conforme a su momento de llegada. Ahora vamos a estudiar la situación en la cual este orden se perdiera, por ejemplo por las características físicas de nuestro sistema, que podría no soportar estructuras de tipo cola o simplemente estructuras ordenadas, o porque el protocolo usado no permite asegurar que se respete el orden de llegada a los buffers. Se nos plantea entonces la siguiente modificación: si el contenido de los buffers puede ser representado por multiconjuntos en lugar de colas, es decir, estos buffers pertenecen a $\Sigma^{\oplus}$ (recordemos que $X^{\oplus}$ es la clase de multiconjuntos finitos con elementos en X) en vez de a Σ^* (donde Σ es el alfabeto del sistema), ¿qué características tienen estos sistemas?

Lo primero que vamos a hacer es definir cómo evolucionan este tipo de sistemas, pues las reglas que definimos en la sección 5 no funcionan. Sin embargo, una ligera modificación de cómo se comportan las funciones de transición que habíamos estudiado anteriormente nos proporciona un marco para este tipo de sistemas.

Definición 8.1. *Decimos que $\mathcal{S} = (F_1, F_2, \ldots, F_n)$ es un **sistema de FSMs en comunicación sin orden** en el caso de que las distintas FSMs generan outputs en $\Sigma^{\oplus}$ (en vez de en Σ^*, como era habitual), es decir, $\delta_i : Q_i \times \Sigma^? \longrightarrow Q_i \times (\Sigma^{\oplus})^n$ para $1 \leq i \leq n$. Una configuración de $\mathcal{S}$ será una tupla de $Q_1 \times \Sigma^{\oplus} \times Q_2 \times \Sigma^{\oplus} \times \cdots \times Q_n \times \Sigma^{\oplus}$.*

La función de transición entre distintas configuraciones de $\mathcal{S}$ es $\delta : Q_1 \times \Sigma^{\oplus} \times Q_2 \times \Sigma^{\oplus} \times \cdots \times Q_n \times \Sigma^{\oplus} \longrightarrow Q_1 \times \Sigma^{\oplus} \times Q_2 \times \Sigma^{\oplus} \times \cdots \times Q_n \times \Sigma^{\oplus}$, que viene definida de la siguiente manera:

$$\begin{gathered} contains(B_i, \gamma) \wedge \delta_i(q_i, \gamma) = (q_i', \alpha_1, \alpha_2, \ldots, \alpha_n) \implies \\ (q_1, B_1 + \alpha_1, \ldots, q_{i-1}, B_{i-1} + \alpha_{i-1}, q_i', erase(B_i, \gamma) + \alpha_i, q_{i+1}, B_{i+1} + \alpha_{i+1}, \ldots, q_n, B_n + \alpha_n) \\ \in \delta(q_1, B_1, \ldots, q_i, B_i, \ldots, q_n, B_n) \end{gathered}$$

En este caso $contains(B, \gamma)$ es una función que dice si hay alguna ocurrencia del literal $\gamma \in \Sigma$ en el multiconjunto B (trivialmente cierto si $\gamma = \epsilon$), $erase(B, \gamma)$ es una función que elimina *una* ocurrencia de γ de B (lo cual podríamos entender como $B \setminus \{\gamma\}$), y $+$ representa la suma usual de multiconjuntos (si un literal γ aparece a veces en el multiconjunto A y b veces en B, entonces aparece $a + b$ veces en $A + B$).

Como hasta ahora, las configuraciones dependen de los estados en que están las diferentes FSMs en cada instante, y los multiconjuntos se pueden interpretar como el contenido de los buffers $B_1, B_2, \ldots, B_n$, que esta vez funcionan como multiconjuntos. La máquina F_i toma inputs de B_i y genera outputs que se añaden al resto de buffers.

Una vez vista la adaptación de la definición de sistemas de FSMs en comunicación a esta situación, vamos a establecer una comparativa con las redes de Petri, pues son los sistemas más conocidos que trabajan con multiconjuntos, es decir, con inputs y outputs sin un orden definido.

8.1. Expresividad de los sistemas con buffers sin orden

Lo primero que vamos a hacer es comprobar que estos sistemas son en realidad simulables por redes de Petri usuales. Vamos a recordar este concepto:

Definición 8.2. *Una **red de Petri** $\mathcal{N}$ es una tupla (P, T, A, M, W). P es el conjunto de lugares de $\mathcal{N}$, T el conjunto de transiciones y $A \subseteq (P \times T) \cup (T \times P)$ el conjunto de arcos (o relación de flujo entre lugares y transiciones). En cada momento, cada lugar de P tiene una cantidad no negativa de tokens, indicada por el marcaje $M : P \longrightarrow \mathbb{N}$. Y cada transición de T puede consumir o generar una cantidad distinta de tokens en cada uno de los lugares que se relacionan con ella a través de los arcos de A. La cantidad de tokens que se intercambian con cada transición por medio de cada arco vienen determinados por $W : A \longrightarrow \mathbb{N}$.*

A la vista de la definición, veamos el resultado que augurábamos.

Proposición 8.3. *Cualquier sistema de FSMs en comunicación con buffers sin orden puede ser simulado por una red de Petri.*

Demostración. Dado un sistema $\mathcal{S} = (F_1, F_2, \ldots, F_n)$, vamos a construir una red de Petri $\mathcal{N}$ que lo simule. La red de Petri será $\mathcal{N} = (P, T, A, M, W)$, como en la definición anterior. Supongamos que el alfabeto de $\mathcal{S}$ es Σ.

Por cada estado q_i de cada máquina F_j tendremos un lugar en P, que denotaremos q_i^j. Estos lugares tendrán siempre a lo sumo un token, y representarán si la máquina F_j está en el estado q_i a lo largo de la simulación que hace $\mathcal{N}$. Además, por cada literal $\gamma \in \Sigma$ y cada *buffer* B_j tendremos también un lugar, b_γ^j, que tendrá tantos tokens como ocurrencias de γ haya en B_j en un instante determinado. En principio, por las características de los sistemas de FSMs, estos lugares no están acotados.

Por cada transición de F_j, $1 \leq j \leq n$, tendremos una transición en $\mathcal{N}$. Es decir, si numeramos las transiciones de cada máquina F_j desde el 1 hasta r_j, el conjunto de transiciones de $\mathcal{N}$ será $T = \{t_i^j : 1 \leq j \leq n, 1 \leq i \leq r_j\}$, donde t_i^j simulará la i-ésima entrada de δ_j.

Queda por tanto definir las relaciones entre lugares y transiciones, a través de relación de flujo A. Supongamos que tenemos una transición arbitraria de δ_j, digamos

que la i-ésima, que funciona de la siguiente forma: $\delta_j(q_k, \alpha) = (q_{k'}, O_1, O_2, \ldots, O_n)$, donde O_a es el multiconjunto de literales que se mandan a F_a. Entonces la transición t_i^j simplemente tendrá que tomar un token de q_k^j para que la transición se tome desde el estado q_k de F_j, y de b_α^j para simbolizar que se está consumiendo el símbolo α de B_j (ambas con peso 1 en nuestra función W, pues solo se consumen un token cada vez). Y tras esto pondrá un token en $q_{k'}^j$ para simbolizar el cambio de estado de F_j, y en cada lugar b_γ^a tantos tokens como veces aparezca el literal γ en O_a, simbolizando que se envían estos literales al buffer B_a de entrada de la máquina F_a. Esto vendrá reflejado en nuestro caso por medio de pesos de los arcos de $\mathcal{N}$: si γ aparece m veces en O_a, entonces $W(t_i^j, b_\gamma^a) = m$.

A modo de ejemplo de la construcción podríamos tener la siguiente situación. Imaginemos un sistema con $\Sigma = \{\alpha, \beta, \gamma\}$, y dos FSMs con 3 estados cada una. Solo tendremos dos transiciones: $\delta_1(q_1, \alpha) = (q_2, \varnothing, \{\beta, \gamma, \gamma\})$, $\delta_2(q_3, \gamma) = (q_1, \{\gamma\}, \{\beta\})$. La red de Petri que simula este sistema, según la construcción anterior, sería (con ciertos tokens puestos en distintos lugares a modo de ejemplo) la de la siguiente figura (donde usamos el convenio usual para dibujar redes de Petri: círculos para lugares y rectángulos para transiciones):

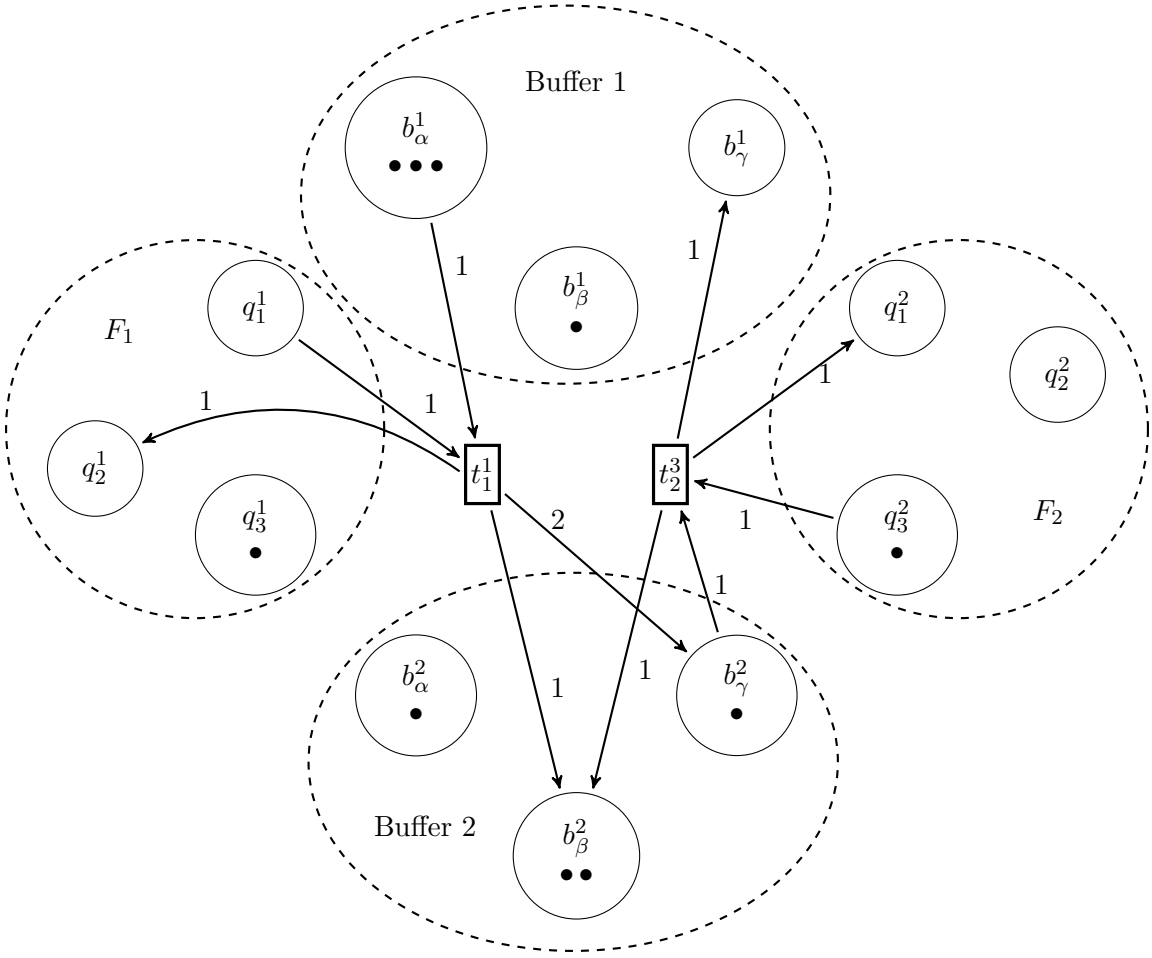


Figura 8.1: Red de Petri que simularía el ejemplo explicado anteriormente según la construcción que proponemos en esta proposición.

De esta forma es claro que la forma de actuar de $\mathcal{N}$ es la de una simulación completa de movimientos de $\mathcal{S}$. Si además ponemos tokens inicialmente en los estados iniciales de cada máquina de $\mathcal{S}$, y en los buffers para que simbolicen el input, conseguimos que sus comportamientos sean el mismo, con lo que quedaría demostrado que podemos simular un sistema de FSMs mediante una red de Petri. □

Hasta ahora hemos visto que las redes de Petri sirven para simular los sistemas de FSMs en comunicación de una manera muy natural, pues basta tener un lugar que guarde cada información presente en el sistema (estados de las FSMs y sus buffers) y una transición de la red de Petri por cada transición de las FSMs. Pero lo más interesante es que también se pueden simular redes de Petri con estos sistemas de FSMs en comunicación, lo que nos arroja la conclusión de que están en el mismo nivel en cuanto a expresividad se refiere. Vamos a hacer la demostración de este hecho.

Proposición 8.4. *Cualquier red de Petri puede ser simulada por un sistema de FSMs en comunicación con buffers sin orden.*

Demostración. Supongamos que tenemos una red de Petri $\mathcal{N} = (P, T, A, W)$. Los lugares serán $P = \{p_1, p_2, \ldots, p_n\}$ y las transiciones $T = \{t_1, t_2, \ldots, t_m\}$. Veamos cómo construir un sistema de FSMs en comunicación que la simule. En este caso, va a ser más sencillo utilizar únicamente una FSM, por lo que tendremos un sistema del tipo $\mathcal{S} = (F_1)$. En realidad esta situación es la misma que si tuviéramos una máquina más que devolviera todo lo que le llega sin modificarlo, como ya hicimos en la demostración de la Turing completitud de los sistemas generales en 4.2. Por tanto, realmente esta no es una situación nueva.

Consideraremos un alfabeto con n literales, tantos como lugares tenga $\mathcal{N}$, digamos $\Sigma := \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$. De esta forma, el número de literales γ_j que tenga nuestro buffer se corresponderá con la cantidad de tokens que tendrá $\mathcal{N}$ en p_j.

Tendremos además un estado inicial q_0, y una serie de estados auxiliares de la familia q_j para simular cada una de las transiciones t_j de $\mathcal{N}$. Todo estado q_j estará conectado con q_0 mediante una transición $(q_j, \varnothing) \in \delta_1(q_0, \epsilon)$. De hecho, si la transición t_j toma tokens de $p_{i_1}, p_{i_2}, \cdots, p_{i_N}$ (donde posiblemente varios de estos lugares estén repetidos, refiriéndose esto a que se toman varios tokens de un mismo lugar según especifica W) y deja tokens en $p_{i'_1}, p_{i'_2}, \ldots, p_{i'_{N'}}$ (donde de nuevo posiblemente haya repetidos), usaremos N estados auxiliares para ir tomando en cada movimiento un token, pues F_1 está limitada a consumir un literal en cada una de sus transiciones. Y al llegar al final, pondremos sin problema todos los tokens en sus lugares correspondientes. Hay que tener en cuenta que el comportamiento de la red de Petri no es *secuencial* como en este caso: no toma los tokens uno a uno para ver si se activa una transición. Por tanto, hemos de tener cuidado con quedarnos bloqueados en mitad de la simulación de una de estas transiciones porque falta uno de los literales que comprobamos al final, por ejemplo. Para evitar esto, simplemente construiremos transiciones que permitan deshacer estas acciones parciales que hemos acometido, sin dejar por ello de simular la red de Petri $\mathcal{N}$.

De manera más precisa, para simular la transición anterior t_j, crearemos estados auxiliares $q_j^1, q_j^2, \ldots, q_j^N$. Además, usaremos las siguientes transiciones para simular cómo t_j coge tokens: $\delta_1(q_j, \gamma_{i_1}) = (q_j^1, \varnothing)$, $\delta_1(q_j^1, \gamma_{i_2}) = (q_j^2, \varnothing), \ldots, \delta_1(q_j^{N-1}, \gamma_{i_N}) = (q_j^N, \varnothing)$. Sin olvidarnos de las transiciones que nos permiten deshacer todos estos pasos hechos individualmente: $\delta_1(q_j, \epsilon) = (q_0, \varnothing)$, $\delta_1(q_j^1, \epsilon) = (q_j, \{\gamma_{i_1}\})$, $\delta_1(q_j^2, \epsilon) = (q_j^1, \{\gamma_{i_2}\}), \ldots, \delta_1(q_j^N, \epsilon) = (q_j^{N-1}, \{\gamma_{i_N}\})$. Y una última transición para poner los tokens donde corresponde: $\delta_1(q_j^N, \epsilon) = (q_0, \{\gamma_{i'_1}, \gamma_{i'_2}, \ldots, \gamma_{i'_{N'}}\})$.

Para que se entienda mejor se puede ver la figura 8.2:

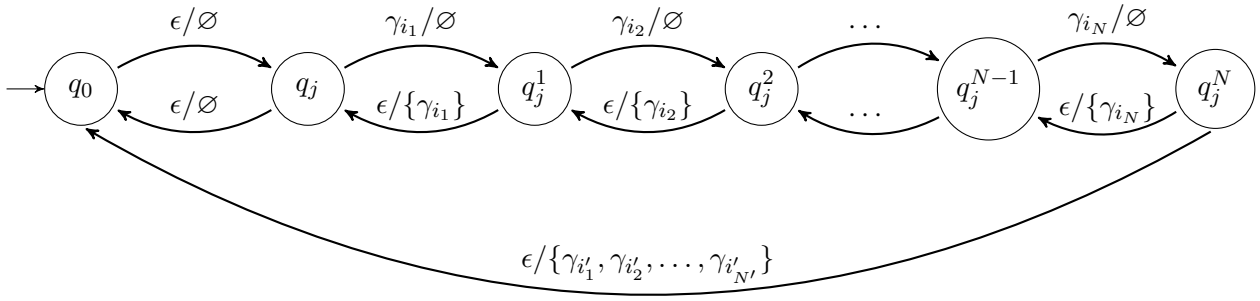


Figura 8.2: Representación gráfica de la los diferentes estados y transiciones usados en la simulación de la transición t_j.

Con esta construcción podemos simular entonces la red de Petri $\mathcal{N}$ a través de un sistema de FSMs, ya que hemos conseguido que las configuraciones del sistema simulen los marcajes de $\mathcal{N}$, con lo que la demostración habría terminado. □

A la vista de la demostración, comentaremos una contraparte de esta construcción: al introducirse transiciones que lo que hacen es deshacer acciones ya acometidas, estamos haciendo que se generen una especie de *livelocks* en nuestro sistema de FSMs, pues el sistema podría avanzar en su ejecución sin realmente progresar.

8.2. Comparativa con otros tipos de redes de Petri

Existen multitud de variantes de las redes de Petri usuales que tienen diferentes características. Es interesante ver que, con ligeras modificaciones de nuestros sistemas de FSMs en comunicación, podemos conseguir dar cabida a algunas de estas variantes. En este caso nos centraremos en dos tipos:

Definición 8.5. *Las redes de Petri con* ***arcos inhibidores*** *(inhibitor arcs): son redes de Petri usuales con un tipo adicional de arcos, denominados inhibidores. Se añade a la red un conjunto $I \subseteq P \times T$ de forma que una transición $t \in T$ solo se puede tomar si no hay ningún token en cada lugar p de forma que $(p, t) \in I$.*

Definición 8.6. *Las redes de Petri con* ***arcos restablecedores*** *(reset arcs): de nuevo son redes de Petri usuales con un tipo adicional de arcos. Estas redes tendrán*

un conjunto $R \subseteq P \times T$ de forma que después de que se ejecute una transición $t \in T$, se quitan todos los tokens que haya en los lugares p de forma que $(p, t) \in R$.

Vistos estos nuevos modelos de redes de Petri, veamos qué modificaciones podemos introducir en nuestros sistemas de FSMs para que se asemejen a ellos. Haremos en este caso un análisis no tan exhaustivo de las construcciones para poder centrarnos realmente en cómo varían las propiedades de los distintos tipos de sistemas de FSMs dependiendo de sus definiciones.

Para poder definir correctamente las transiciones en estos sistemas con buffers sin orden, hasta ahora hemos hecho uso de una función $contains(B, \gamma)$ que, dados un multiconjunto B y un literal γ, nos dice si hay alguna ocurrencia de γ en B, vamos a considerar un nuevo tipo de transición en nuestras FSMs. Dado un alfabeto Σ, definimos $\overline{\Sigma} := \{\overline{\gamma} : \gamma \in \Sigma\}$. Estos nuevos literales significarán lo siguiente: si tenemos una transición $\delta_i(q_i, \overline{\gamma}) = (q_i', A)$ en la FSM F_i de un sistema de FSMs en comunicación (ahora $\delta_i : Q_i \times \left(\Sigma^? \cup \overline{\Sigma}\right) \longrightarrow Q_i \times (\Sigma^{\oplus})^n$), esto significará que la transición se toma si en el *buffer* B_i no hay ningún literal γ, y se transitaría de q a q' produciendo A como output. Es decir, con las notaciones del apartado anterior:

$$\begin{gathered}\neg contains(B_i, \gamma) \wedge \delta_i(q_i, \overline{\gamma}) = (q_i', \alpha_1, \alpha_2, \ldots, \alpha_n) \implies \\ (q_1, B_1 \cup \alpha_1, \ldots, q_{i-1}, B_{i-1} \cup \alpha_{i-1}, q_i', B_i \cup \alpha_i, q_{i+1}, B_{i+1} \cup \alpha_{i+1}, \ldots, q_n, B_n \cup \alpha_n) \\ \in \delta(q_1, B_1, \ldots, q_i, B_i, \ldots, q_n, B_n)\end{gathered}$$

Con este nuevo alfabeto y sus nuevas transiciones asociadas es claro que podemos introducir arcos inhibidores de la misma manera en la que introducíamos flechas usuales en el apartado anterior a la hora de simular una red de Petri. Por tanto, cualquier red de Petri con arcos inhibidores puede ser simulada por este tipo de sistemas de FSMs en comunicación. Y el recíproco también es trivialmente cierto, la misma construcción anterior serviría asimismo.

Esto es interesante por las propiedades que adquieren los sistemas de FSMs con esta pequeña modificación de comportamiento. Por ejemplo, en el caso de las redes de Petri (y por tanto en nuestros sistemas de FSMs en comunicación sin orden, según hemos definido en la definición 8.1) es bien conocido que el problema de recubrimiento (*coverability* o *control state reachability* en inglés, el problema de saber si se puede alcanzar un marcaje mayor que uno prefijado, que es el que más se asemeja al de la alcanzabilidad en sistemas de FSMs) es decidible. Pero pasa a ser indecidible en cuanto usemos transiciones con literales del tipo $\overline{\gamma}$, pues en redes de Petri con arcos inhibidores el problema no es decidible (como podemos ver en [12], teorema 3.2.21).

Al ser los sistemas con arcos inhibidores Turing completos ([3], teorema 3), sería posible simular también arcos restablecedores mediante los sistemas de FSMs en comunicación que acabamos de definir. Básicamente habría que añadir ciertos estados en los que las FSMs quedarían en bucle vaciando de un tipo de literal sus buffers de lectura, utilizando para terminar dicho bucle los literales $\overline{\gamma}$, que señalizarán que ya se ha vaciado el buffer de dicho literal.

Pero esto no resulta tan interesante como definir una modificación de los sistemas que tenga exactamente la misma expresividad que las redes de Petri con arcos restablecedores, en un estadio anterior a la Turing completitud. Esto lo podemos comprobar, por ejemplo, porque en redes de Petri con arcos restablecedores el problema de la terminación es decidible [11], mientras que en las máquinas de Turing no lo es (es el Problema de Parada, resuelto por Turing).

Definamos por tanto otro tipo de sistemas de FSMs, de la siguiente forma. Dado un sistema usual $\mathcal{S}$ de FSMs en comunicación en el que los buffers no tienen orden, con alfabeto Σ, llamaremos $\hat{\Sigma} := \{\hat{\gamma} : \gamma \in \Sigma\}$. Estos nuevos literales servirán para simular los arcos restablecedores, de forma que al tomarse una transición con $\hat{\gamma}$, se eliminan todos los literales γ del buffer en cuestión. Más formalmente, las transiciones asociadas a estos nuevos literales son:

$$\delta_i(q_i, \hat{\gamma}) = (q_i', \alpha_1, \alpha_2, \ldots, \alpha_n) \implies$$
$$(q_1, B_1 + \alpha_1, \ldots, q_{i-1}, B_{i-1} + \alpha_{i-1}, q_i', erase\_all(B_i, \gamma) + \alpha_i, q_{i+1}, B_{i+1} + \alpha_{i+1}, \ldots, q_n, B_n + \alpha_n)$$
$$\in \delta(q_1, B_1, \ldots, q_i, B_i, \ldots, q_n, B_n)$$

donde $erase\_all(B, \gamma)$ es una función sobre multiconjuntos que elimina todas las ocurrencias del literal γ en B.

Notamos que básicamente la interpretación de estas transiciones es que, tras tomar la transición, vacían el buffer del literal γ. Además, se toma la transición con literal $\hat{\gamma}$ sin necesidad de que haya literales γ en el buffer, como sugiere la definición de arcos restablecedores (no obstante esto se podría cambiar fácilmente). Por ello, es claro que usando la construcción descrita en la proposición 8.4 servirá para simular redes de Petri con arcos restablecedores si usamos FSMs de este nuevo tipo que hemos construido. Y es fácil darse cuenta de que el recíproco es también cierto: si introducimos arcos restablecedores en la construcción de la proposición 8.3, podemos simular cualquiera de estos nuevos sistemas de FSMs en comunicación.

En este caso resulta de nuevo interesante la variación en cuanto a propiedades que experimentan los sistemas de FSMs con esta modificación, que podría en principio no parecer muy grande. En este caso, el problema de recubrimiento sigue siendo decidible (porque lo es para redes de Petri con arcos restablecedores, como podemos ver en el capítulo 4 de [11]), pero el problema de la alcanzabilidad (llegar a un marcaje concreto en vez de a un marcaje mayor que uno concreto) se convierte en indecidible (pues lo es para las redes de Petri con arcos restablecedores, como podemos comprobar en [4]), al igual que cuando introducíamos arcos inhibidores. Esto nos indica que realmente los problemas interesantes relacionados con las configuraciones alcanzables están en este caso realmente en el borde entre lo decidible y lo no decidible. Al final estos resultados se apoyan en una noción relacionada, la denominada *backwards reachability*, que sí continúa siendo decidible ([12], 3.2.20) cuando hay arcos restablecedores, y marca realmente la diferencia con la introducción de arcos inhibidores.

En cuanto a complejidad de estos problemas de decisión, resulta que todos ellos son $EXSPACE$-duros. Esto en realidad es consecuencia de que los problemas de decisión más usuales (como el de la alcanzabilidad que hemos comentado, la terminación, la acotación...) en redes de Petri usuales lo son, y además hay una cota

inferior de $2^{\mathcal{O}(\sqrt{n})}$ en el espacio que requieren para ser resueltos (como se puede ver, por ejemplo, en [12], pág. 57). Esto nos dice bastante sobre la dificultad que tiene en el caso general resolver estos problemas.

Capítulo 9

Sistemas de Mensajes Perdidos

En nuestro camino explorando diferentes versiones de nuestros sistemas de FSMs podemos visitar uno de los modelos más conocidos y estudiados de este tipo. Si suponemos que los canales de comunicación que usamos (los buffers que ayudan a la comunicación en nuestro caso) no son ideales y, por tanto, son susceptibles a fallos de forma que algunos literales puedan desaparecer de ellos en cualquier momento sin ningún mecanismo que controle estas pérdidas, llegamos a un concepto bastante similar al de los Sistemas de Mensajes Perdidos (LCS por su nombre en inglés: *Lossy Channel Systems*).

Los LCS usuales que podemos encontrar en la literatura son un tipo de sistemas de FSMs en comunicación en los cuales hay un tipo de transición adicional: en cualquier punto de la ejecución, pueden desaparecer un conjunto de literales de cualquier buffer del sistema. Este comportamiento es puramente no determinista: no hay manera de controlarlo, puede surgir en cualquier momento y afectar a cualquier grupo de literales. Dado este no determinismo inherente a los LCS, en general se suele permitir también que las FSMs utilizadas sean no deterministas (y recordemos que no toda FSM no determinista se puede convertir en una determinista, pues ante una misma secuencia de inputs las FSMs no deterministas pueden generar potencialmente más cadenas de outputs diferentes que las deterministas), lo cual los diferencia un poco más de nuestros sistemas de FSMs.

En general, un LCS puede, en cierto sentido, *simular* las ejecuciones de un sistema de FSMs, pues una de sus posibles ejecuciones es aquella en la que las transiciones que pierden literales nunca se activan, y en concreto este sería el comportamiento que tienen nuestros sistemas. No obstante, las propiedades de ambos tipos de sistemas no son las mismas porque en los LCS hay muchas más posibles ejecuciones en las que se pierden algunos literales de los buffers. En cambio, una *simulación* en sentido contrario no es posible a menos que introduzcamos cambios en la definición de nuestros sistemas (no es posible simular esas pérdidas de literales no deterministas con nuestros sistemas de FSMs usuales), pues hay propiedades en las que ambos modelos difieren: por ejemplo, el problema de la terminación (que el sistema no tenga ramas de ejecución infinitas dado un input) es decidible en el caso de los LCS e indecidible en el de los sistemas de FSMs en comunicación (Problema de Parada).

Una forma de hacer que nuestros sistemas de FSMs pudieran tener la capacidad de perder mensajes de forma no determinista, como ocurre en los LCS, sería introducir un cierto grado de no determinismo en las FSMs. Por ejemplo, si para cada transición $\delta(q, \alpha) = (q', \beta)$ de una de las FSMs de nuestro sistema permitimos añadir también la transición $\delta(q, \alpha) = (q', \epsilon)$, estaríamos pudiendo simular este

fenómeno ya que, de manera no determinista, podrían no llegar los mensajes a su buffer destino. Además, esto es introducir un no determinismo muy controlado, así que tampoco es que el cambio haya sido excesivo. Lo interesante va a ser la diferencia entre propiedades que generan estos cambios.

Aunque pueda parecer algo paradójico, numerosos problemas de decisión son más tratables en los LCS que en el caso de los sistemas de FSMs en comunicación y las máquinas de Turing, donde esencialmente los problemas son indecidibles en su mayoría. En concreto, un problema de vital importancia, como es el de la alcanzabilidad, es decidible para los LCS ([2], capítulo 5) . También es decidible el problema de la inevitabilidad (dado un conjunto de estados, ¿es inevitable que cualquier ejecución maximal pase por alguno de dichos estados?), que engloba el problema de la terminación (usando como conjunto de estados aquellos estados en los cuales la ejecución termina), muy importante también en la teoría de autómatas ([21], teorema 8). También son decidibles otras propiedades que comprueban que a lo largo de las ejecuciones maximales se conservan ciertas propiedades, lo cual es de utilidad en este caso debido a que es importante tener ciertos métodos de verificación del funcionamiento de los sistemas que palien en cierto modo su naturaleza, que permite que los mensajes se pierdan.

No obstante, ni mucho menos todas las propiedades de estos sistemas son decidibles: en general, otras como la acotación (¿la cantidad de literales que puede haber en un canal/buffer a lo largo de cualquier ejecución está acotada?) o ciertas propiedades de *justicia*, como saber si es inevitable pasar por un cierto estado de control infinitas veces, son indecidibles ([1], teorema 3.7).

Más aún: a pesar de ser decidibles los problemas antes comentados, como el de la alcanzabilidad, tienen una complejidad extremadamente grande. No están asociados a funciones recursivas primitivas ([27], teoremas 4.2, 4.4). De hecho, se puede ver la gran distancia a la que están de esta clase de funciones con un breve vistazo a la Jerarquía de Crecimiento Rápido (*Fast Growing Hierarchy*): las funciones recursivas primitivas están asociadas a ordinales acotados superiormente por ω (el primer ordinal numerable), mientras que los problemas de la alcanzabilidad e inevitabilidad están asociados al ordinal ω^ω.

9.1. Introduciendo justicia en las ejecuciones

Una vez vistas las propiedades básicas de los LCS en general, vamos a comprobar que introducir a nuestros sistemas de FSMs la capacidad de hacer que los mensajes se puedan perder de forma no determinista hace que consigamos una clase restringida de los LCS generales, y esto nos va a proporcionar alguna ventaja interesante.

Por ejemplo, sería interesante introducir ciertas nociones de justicia en este tipo de sistemas. Esto se debe a que el hecho de que los mensajes se puedan perder puede llegar a degradar bastante la comunicación entre varias máquinas en una situación real, en la cual podría ser que la red fallara mucho y casi ningún mensaje se mandara. Si tuviéramos asegurada cierta justicia en nuestro ambiente, como que ciertos comportamientos se tienen que dar periódicamente sean cuales sean las circunstancias,

podríamos al menos intentar paliar los posibles fallos de la red repitiendo el envío de los mensajes, pues la justicia nos aseguraría que, tarde o temprano, es seguro que los mensajes llegarán.

Dos modelos de justicia interesantes que van a ser de especial interés son los que se estudian en [20]. Para un sistema $\mathcal{S} = (F_1, \ldots, F_n)$:

- Decimos que una ejecución es *débilmente justa* si para cada máquina F_i se cumple que está bloqueada (es decir, que dada la composición de su buffer y su estado actual, no puede avanzar en ese instante) un número infinito de veces a lo largo de la ejecución o que infinitos pasos de la ejecución de $\mathcal{S}$ son pasos de F_i en los que desaparecen literales de su buffer.
- Decimos que una ejecución es *fuertemente justa* si para cada máquina F_i se cumple que a partir de un cierto paso de la ejecución de $\mathcal{S}$ F_i está siempre bloqueada o si infinitos pasos de la ejecución de $\mathcal{S}$ son pasos de F_i en los que desaparecen literales de su buffer.

Ambos conceptos básicamente de lo que nos están informando es de que, mientras no están bloqueadas, las máquinas van alternándose relativamente entre ellas en su ejecución, hay una cierta justicia entre todas las máquinas. En la justicia fuerte además se exige que cada máquina no se bloquee casi nunca o que termine en tiempo finito su ejecución, lo cual da la idea de que, entre aquellas máquinas que no se bloqueen en tiempo finito, tendrá que haber gran grado de alternancia en las ejecuciones. Además, lógicamente, si una ejecución es fuertemente justa, también es débilmente justa.

Veamos ahora qué pasa si imponemos estos sentidos de justicia a la modificación de los sistemas de FSMs en comunicación en los cuales permitimos que se pierdan mensajes. Estudiaremos qué pasa con el problema de la terminación imponiendo estos conceptos de justicia: ¿todas las ramas de ejecución de nuestro sistema acaban en tiempo finito imponiendo justicia fuerte/débil? La respuesta es la contraria a ¿hay alguna ejecución infinita en la que se respete la justicia fuerte/débil?, que también puede servir en ocasiones para diferentes interpretaciones.

Lo que podemos ver es que, por la forma en que hemos diseñado nuestros sistemas de FSMs, cada buffer solo sirve como input de una única máquina (lo cual podemos ver gráficamente en la figura 9.1). Esto hace que tengamos una clase restringida de los LCS, que en [20] llaman sistemas sin canales multiplexados.

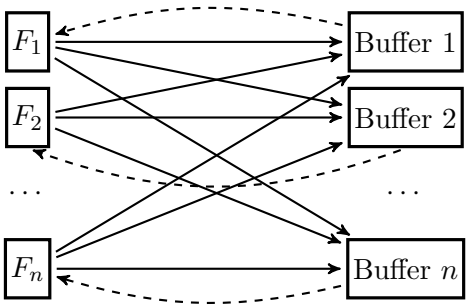


Figura 9.1: Esquema de las comunicaciones en un sistema de FSMs en comunicación.

Esto es interesante porque en LCS generales se tiene el siguiente resultado:

Proposición 9.1 (3.1 en [20])**.** *En LCS sin canales multiplexados el problema de la terminación imponiendo justicia fuerte/débil es decidible.*

Por tanto, en el caso de nuestros sistemas de FSMs modificados, como acabamos de ver en los párrafos anteriores, están contenidos en la subclase de los LCS sin canales multiplexados, podemos aplicarles el teorema 9.1 (sabiendo que, como es sencillo comprobar, ser decidible es una propiedad que las subclases heredan). Esto nos conduce al siguiente resultado:

Corolario 9.2. *En los sistemas de FSMs en comunicación modificados para que se puedan perder mensajes de los buffers de manera no determinista, el problema de la terminación es decidible si imponemos justicia fuerte o débil.*

Esta propiedad es interesante porque nos informa de que podemos introducir conceptos de justicia a bajo coste (sin perder la decibilidad de la propiedad de terminación). Y es interesante introducir la justicia porque así tenemos algunas certezas más que en el caso de que no tengamos absolutamente ningún control sobre la pérdida de mensajes en nuestro sistema.

Capítulo 10

Complejidad de algunas propiedades

A pesar de que, en virtud del teorema de Rice, todos los problemas de decisión no triviales sobre sistemas de FSMs en comunicación son indecidibles (pues lo son para las máquinas de Turing), podemos estudiar la complejidad de resolver alguno de estos problemas en algunas de sus versiones simplificadas que sí queden en el rango de lo decidible.

10.1. El problema de la alcanzabilidad finita

En este apartado vamos a entrar más a fondo en una de las propiedades básicas: la alcanzabilidad. Como el problema de la alcanzabilidad, en su versión general, resulta indecidible, en este caso estudiaremos la alcanzabilidad finita (que aquí denotaremos también N-alcanzabilidad, para que quede claro que la limitación de la finitud viene prefijada por una constante N), que es el siguiente problema: dado un natural N y una posible configuración de nuestro sistema de FSMs en comunicación $\mathcal{S}$, ¿es posible alcanzar dicha configuración dando $\mathcal{S}$ a lo sumo N pasos en su ejecución? De hecho, veremos el caso de que la configuración que buscamos alcanzar sea una cualquiera en la que $\mathcal{S}$ acepta, planteándonos el problema ¿puede aceptar $\mathcal{S}$ en a lo sumo N pasos en su ejecución? Veremos que este problema de decisión no es sencillo de resolver en el caso general, tanto que resulta ser un problema $\mathcal{NP}$-completo. Vamos a proceder a la demostración de este hecho.

Lema 10.1. *El problema de la alcanzabilidad finita en sistemas de FSMs en comunicación es $\mathcal{NP}$.*

Demostración. Si tenemos un sistema de FSMs en comunicación $\mathcal{S} = (F_1, F_2, \ldots, F_n)$, saber si $\mathcal{S}$ alcanzará un determinado estado en menos de N pasos es un problema $\mathcal{NP}$: de manera no determinista se puede, en caso de que haya forma de alcanzar dicho estado, *acertar* cuál es el orden de ejecución, es decir, si el siguiente paso del sistema será ejecutar una transición de F_1, de $F_2 \ldots$ o de F_n. Dado este orden de ejecución, es ya trivial comprobar que funciona: simplemente vamos ejecutando las transiciones en $F_1, F_2, \ldots, F_n$ y modificando los buffers correspondientes, y al final comprobamos si el estado al que hemos llegado es el esperado. Esta comprobación es claramente polinómica, de hecho $\mathcal{O}(N)$ si implementamos convenientemente las operaciones de inserción y extracción de elementos de los buffers. □

Hay que tener en cuenta que para que esto sea realmente un comportamiento polinómico, N tiene que haber sido recibido como input en *base unaria*, es decir, el input N se representa recibiendo N tokens del mismo tipo. Notemos que es importante porque si nos dieran N como input en binario, por ejemplo, $\mathcal{O}(N)$ sería un tiempo exponencial en el tamaño del input. Pero es una hipótesis razonable trabajar en *base unaria* en este tipo de situaciones porque si no, incluso el simple problema de que un autómata dé N pasos tarda un tiempo exponencial en ejecutarse; mientras que considerando la *base unaria*, es como si nuestro autómata fuera consumiendo tokens con cada paso que da y el comportamiento sería polinómico, lo cual resulta bastante más razonable.

Ahora querríamos demostrar la $\mathcal{NP}$-completitud del problema de la N-alcanzabilidad en estos sistemas: para ello, vamos a reducir el problema del 3-SAT (del cual es bien conocida su $\mathcal{NP}$-completitud) al de la N-alcanzabilidad de uno de estos sistemas. Describamos esta reducción en detalle.

Teorema 10.2. *El problema de la alcanzabilidad finita en sistemas de FSMs en comunicación es $\mathcal{NP}$-completo.*

Demostración. Supongamos que tenemos como input una fórmula de 3-SAT ϕ con n variables, y queremos decidir si ϕ es o no satisfactible (es decir, resolver el problema del 3-SAT) usando un sistema de FSMs.

Vamos a crear un sistema $\mathcal{S}$ de FSMs en comunicación con varias máquinas (donde no todas las máquinas se comunicarán con todas las restantes). Dos de ellas serán F_1, F_2: F_1 tiene un único estado en el cual devuelve 0 ante cualquier input (y permanece en ese estado); mientras que F_2 tiene, de nuevo, un único estado (donde siempre permanece) en el cual devuelve esta vez un 1 ante cualquier input. De esta forma, fruto del no determinismo de las comunicaciones de F_1, F_2, podríamos generar cualquier cadena compuesta por ceros y unos, lo cual vamos a utilizar con más máquinas. Los outputs tanto de F_1 como de F_2 irán únicamente a la máquina F_3, que posteriormente definiremos.

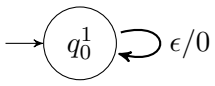


Figura 10.1: Esquema gráfico de F_1.

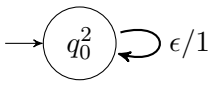


Figura 10.2: Esquema gráfico de F_2.

Usaremos también un checker de fórmulas de 3-SAT, que funcionará de la siguiente forma: como la fórmula ϕ, que este checker recibirá como input, tiene n variables (que podemos suponer ordenadas de algún modo), el checker usará los n primeros inputs que le lleguen (en este caso de F_1 y F_2, como veremos) para asignárselos como valores de las variables de ϕ, en ese mismo orden. Una vez le lleguen esos primeros n inputs, podrá comprobar si esos valores satisfacen ϕ, y generar una respuesta afirmativa o negativa. Este checker se puede construir de manera sencilla como máquina de Turing. Y como vimos en el apartado 4.2, los sistemas de FSMs en comunicación con 2 FSMs son una clase Turing completa, así que este checker puede ser implementado mediante 2 FSMs en comunicación, digamos F_4, F_5.

Además, el checker puede ser implementado sencillamente de forma que se ejecute

en tiempo lineal con respecto a los inputs de las fórmulas que recibe. Y, como vimos en el apartado 4.2, la transformación de máquinas de Turing a sistemas de FSMs en comunicación era suficientemente buena como para poder afirmar que solo añadíamos una cantidad polinómica de sobrecoste. Por tanto, podemos afirmar que el comportamiento del checker implementado con F_4, F_5 será polinómico en el número de literales de la fórmula ϕ que nos den como input.

Para comunicar las máquinas F_1, F_2 con este checker (F_4, F_5), usaremos una sencilla máquina F_3 intermediaria: recibirá los outputs que F_1 y F_2 generen, y los dejará pasar (en un único canal), únicamente, hacia el checker, en el orden que le lleguen. De esta forma conseguimos simplificar la forma en que llegan los ceros y unos generados por F_1, F_2 al checker.

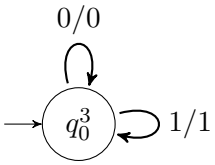


Figura 10.3: Esquema gráfico de F_3.

Y por último tenemos una máquina F_6 que lo único que hace es esperar la respuesta del checker. Cuando F_4 y F_5 hayan procesado suficiente información como para saber si la fórmula ϕ es o no satisfactible, enviarán un mensaje (afirmativo o negativo en cada caso, digamos que mediante los literales $\top, \bot$) a F_6. La recepción de un mensaje afirmativo hará que F_6 transite a un estado diferente q_f^6, que será el único estado de aceptación de $\mathcal{S}$, o sea, que las únicas configuraciones de $\mathcal{S}$ de aceptación son aquellas en las que F_6 está en q_f^6 (reduciendo así el problema a una especie de alcanzabilidad finita de F_6 dentro del sistema $\mathcal{S}$). Y el problema de la N-alcanzabilidad de $\mathcal{S}$ en este caso lo plantearemos en este caso, como explicábamos al inicio, como ¿alcanza $\mathcal{S}$ su configuración de aceptación en menos de N pasos de $\mathcal{S}$?

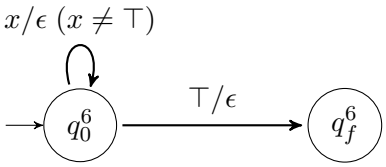


Figura 10.4: Esquema gráfico de F_6.

Con esto ya tendríamos nuestro sistema $\mathcal{S} = (F_1, F_2, F_3, F_4, F_5, F_6)$ construido, de forma que las diferentes máquinas se comunican únicamente con las que muestra el diagrama de la figura 10.5. Veamos ahora por qué funciona realmente esta reducción.

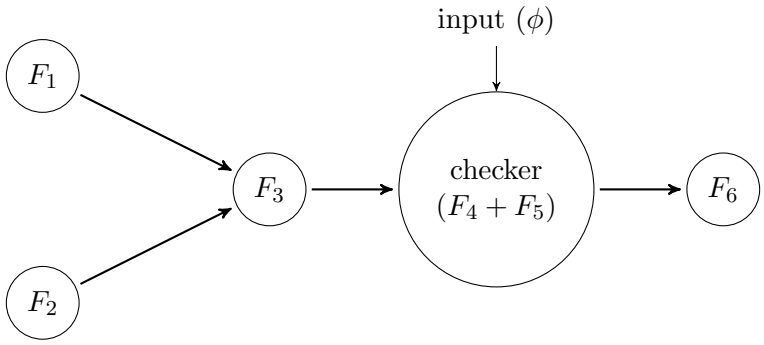


Figura 10.5: Diagrama explicativo de las comunicaciones que se establecen en $\mathcal{S}$.

Dada la construcción anterior, al checker le llegarán ceros o unos en función de qué máquina ejecute el siguiente paso de entre F_1, F_2. Supongamos que el checker formado por F_4, F_5 tiene que ejecutar $p(|\phi|)$ pasos para poder decidir si la fórmula ϕ (denotamos por $|\phi|$ el tamaño de la fórmula ϕ como input, y recordamos que tiene $n \leq |\phi|$ variables) es o no satisfactible. Por la discusión anterior, p es un polinomio.

Entonces, en caso de que ϕ sea satisfactible, definiendo $N := p(|\phi|) + 2n + 1 \leq p(|\phi|) + 2|\phi| + 1$, es claro que en N pasos $\mathcal{S}$ podrá llegar a su *estado de aceptación*, es decir, aquel en el que F_6 está en q_f^6 ($p(|\phi|)$ pasos serán del checker, n de las máquinas F_1, F_2 para generar los literales que hacen ϕ cierta, n de F_3 para dejar pasar estos literales por un mismo canal y 1 final de la respuesta que el checker envía a F_6).

Y, ciertamente, el recíproco también es cierto: si la fórmula ϕ no es satisfactible, en ninguna ejecución de $\mathcal{S}$ de longitud N el sistema podrá llegar a su estado de aceptación. Por tanto, podemos concluir que la reducción del problema de la satisfactibilidad de ϕ, una fórmula cualquier de 3-SAT, ha sido reducida a un problema de N-alcanzabilidad en sistemas de FSMs en comunicación.

Además, la reducción es polinómica porque la construcción de $\mathcal{S}$ es polinómica respecto al tamaño de ϕ (pues la construcción de cada máquina es en verdad independiente de ϕ, así que podemos considerar que la construcción de $\mathcal{S}$ es $\mathcal{O}(1)$ en cuanto a número de estados, y lineal en cuanto a tamaño de los buffers, pues el checker recibe el input completo y ninguna máquina más tiene literales en su buffer inicialmente), y N también lo es.

De esta forma, estamos ya en posición de afirmar que el problema de la N-alcanzabilidad en sistemas de FSMs es en efecto $\mathcal{NP}$-completo, como queríamos demostrar. □

De hecho, que el problema sea $\mathcal{NP}$-completo no debería resultarnos una gran sorpresa en realidad, pues el altísimo grado de no determinismo que tienen los sistemas de FSMs en sus comunicaciones hace prácticamente imposible conseguir que pasen a la frontera de lo polinómico. De hecho, incluso con únicamente 2 máquinas, como hacíamos en la demostración anterior con F_1 y F_2, el intercalado que puede existir entre sus ejecuciones hace que el número de posibles cadenas de ceros y unos que le puedan llegar a la F_3 anterior crezca exponencialmente conforme a los pasos

que ejecutan F_1 y F_2.

Si dejamos que a F_3 le lleguen n literales, habrá 2^n cadenas posibles que le puedan llegar, un crecimiento claramente exponencial. Pero resulta más interesante comprobar que aunque limitemos la capacidad de intercalado de F_1 y F_2, vuelven a generarse una cantidad exponencial de cadenas. Esto lo podemos comprobar imponiendo unas ciertas condiciones de *justicia* en estas comunicaciones.

Por ejemplo, si no dejamos que una máquina ejecute más de $k(\geq 2)$ pasos seguidos, podríamos esperar que disminuyera bastante el número de cadenas posibles que se pueden generar. Pero resulta que sigue siendo exponencial, pues la cantidad de cadenas de longitud n que se podrán generar cumple la recurrencia $c_n = c_{n-1} + c_{n-2} + \ldots + c_{n-k}$, una de las llamadas sucesiones de Fibonacci generalizadas, que tienen carácter exponencial, con bases comprendidas entre $\varphi \approx 1{,}6180\ldots$ y 2 ([28], lema 3.6). Si limitamos que las 2 máquinas no puedan ejecutar k pasos consecutivos, obtenemos un resultado similar.

El caso más fuerte en este sentido sería uno en el cual implementáramos una justicia tal que solo permitiéramos ejecuciones *intercaladas* de F_1 y F_2, de forma que, de alguna forma, pudiéramos dividir cada cadena recibida por F_3 en grupos de dos literales que podrían ser únicamente 01 o 10. Igualmente, en este caso $c_n \approx 2^{n/2} = \sqrt{2}^n$, que seguiría siendo exponencial.

De esta forma vemos que, aún limitando artificialmente la capacidad de comunicación de las distintas máquinas mediante diferentes tipos de justicia, continuamos obteniendo cantidades no polinómicas de posibles ejecuciones, lo cual nos indica que podíamos esperar el resultado que hemos dado en este capítulo.

10.1.1. En sistemas con buffers acotados

A modo de comparativa podemos considerar el problema de la alcanzabilidad finita en el caso de sistemas de FSMs en comunicación con buffers acotados, tal y como definimos en el apartado 5.1. En este caso, y en el mismo espíritu que obteníamos que su expresividad era bastante limitada, pues son tan expresivos como los autómatas finitos (como vimos en 7.1), vamos a conseguir ver que el problema de la N-alcanzabilidad es mucho más sencillo de resolver, pues es polinómico en N.

Lema 10.3. *El problema de la alcanzabilidad finita en sistemas de FSMs en comunicación con buffers acotados es polinómico* ($\mathcal{P}$).

Demostración. Dado un sistema $\mathcal{S}$ de FSMs en comunicación con buffers acotados, podemos conseguir un autómata finito F equivalente, cuyo tamaño no depende de N y es polinómico en el tamaño de $\mathcal{S}$, como vimos al final del capítulo 7. Y la construcción del grafo de alcanzabilidad de F hasta llegar a profundidad N (pues más allá no vamos a obtener respuestas al problema de la N-alcanzabilidad, así que no lo necesitamos) es $\mathcal{O}(N)$: en efecto, a cualquier profundidad d, F solo podrá estar en una cantidad $\mathcal{O}(1)$ de estados, pues son los que resultan de considerar los estados de F (constantes respecto a N por la discusión anterior); y para construir el siguiente nivel del grafo (profundidad $d+1$) necesitamos solo considerar los estados

del nivel d. Por tanto, cada nivel se crea en tiempo constante, y de ahí resulta que llegar hasta el nivel N es $\mathcal{O}(N)$.

Al ser la reducción del problema del problema de la N-alcanzabilidad en $\mathcal{S}$ al de la N-alcanzabilidad en F polinómica en el tamaño de $\mathcal{S}$, y ser también grafo de alcanzabilidad de F de tamaño polinómico en N, es claro que el problema de la N-alcanzabilidad se puede resolver en tiempo polinómico respecto al tamaño de los datos del problema. □

Como vemos, esto establece de nuevo una gran diferencia entre la complejidad de los sistemas de FSMs en comunicación generales y los que tienen la limitación de tener los buffers acotados.

10.2. El problema de regreso al estado inicial

Analizamos ahora otro problema de decisión interesante en el caso de los sistemas de máquinas de estados o, en general, de distintos tipos de autómatas: el problema de saber si desde cualquier configuración alcanzable se puede regresar a la configuración inicial. De nuevo, este problema resulta en su versión general indecidible para nuestros sistemas fruto del Teorema de Rice, pero podemos crear versiones finitistas que sean decidibles e interesantes al mismo tiempo. En este caso la pregunta será: para cualquier secuencia de M pasos de nuestro sistema, ¿existe un camino de longitud k, con $k \leq N$, de forma que tras esos $M + k$ pasos el sistema vuelva a su configuración inicial? Consideraremos M y N prefijados, como datos del problema.

Este problema se puede expresar de forma lógica como (a modo de pseudocódigo):

$$\forall\, paso_1, paso_2, \ldots, paso_M \;\; \exists\, paso'_1, paso'_2, \ldots, paso'_k$$
$$es\_igual(inicial, aplicar(inicial, [paso_1, \ldots, paso_M, paso'_1, \ldots, paso'_k]))$$

Como podemos aplicar acciones sobre nuestros sistemas en tiempo polinómico y la cantidad de pasos es también polinómica en M y N, es claro que este problema es del tipo $\forall^P \exists^P P$, es decir, que pertenece a la clase de complejidad Π_2^P.

Lo interesante va a ser comprobar que de hecho este problema es completo dentro de esta clase, lo cual nos da la idea de que es en realidad difícil de resolver. Para ello, vamos a reducir un problema del tipo del de SAT a nuestro problema. En concreto, será el problema $QSAT_2$ (también llamado QBF_2), que trata de determinar si una fórmula de SAT con ciertos cuantificadores sobre variables (en este caso primero ciertos cuantificadores universales y después otros existenciales, necesariamente en ese orden) es satisfactible. Y es bien sabido que este problema es Π_2^p-completo (al igual que sus análogos $QSAT_k$ en Π_k^p), pues de hecho son los problemas más prototípicos con esta propiedad.

La idea para la simulación de este problema es construir un sistema de FSMs en comunicación que simule exactamente lo que esperamos en el problema $QSAT_2$: habrá una primera fase en la que asignaremos cualquier valor a ciertas variables de nuestra fórmula (y para conseguir potencialmente cualquier asignación, usaremos el

no determinismo de las comunicaciones entre las máquinas) y después habrá otra fase donde el resto de las variables intentarán tomar valores que satisfagan la fórmula de SAT (para lo cual también utilizaremos el no determinismo de las comunicaciones entre máquinas). Tras estas asignaciones, comprobaremos si la fórmula se satisface, y solo en ese caso mandaremos ciertos mensajes a cada máquina para que vuelva a su estado inicial, y que por tanto el sistema en conjunto vuelva a la configuración inicial.

Vemos que, aunque las ideas son similares a las que usábamos en el caso de la alcanzabilidad finita (usar el no determinismo de las comunicaciones para generar una cantidad exponencial de caminos y después mezclarlo con otros comportamientos deterministas que nos permitan comprobar las propiedades requeridas), el hecho de tener que distinguir muy claramente la parte en que simulamos los cuantificadores universales y existenciales va a hacer que la construcción se complique un poco respecto a la de la alcanzabilidad finita, pues debemos tener cuidado con que varias máquinas han de estar bloqueadas en ciertos momentos para no mezclar las dos fases de la simulación. No obstante, la idea principal sigue siendo aprovechar y controlar el no determinismo de las comunicaciones entre máquinas.

Teorema 10.4. *La versión finitista del problema de regreso a la configuración inicial en sistemas de FSMs en comunicación es* Π_2^P*-completo.*

Demostración. Supongamos que tenemos como input una fórmula φ de SAT, con $m + n$ variables que denotaremos $X_1, \ldots, X_{m+n}$. Ahora queremos resolver el problema de $QSAT_2$: ¿para cualquier asignación de valores de verdad a las m primeras variables (porque sin pérdida de generalidad podemos suponer que están nombradas para que sean las primeras) existe una asignación de valores de las otras n variables que satisfaga φ?

Vamos a construir un sistema $\mathcal{S}$ muy similar al de que ya usamos en la demostración del teorema 10.2, aunque va a tener dos partes bien diferenciadas que nos van a ayudar a distinguir entre las dos partes que tiene el problema que queremos resolver. Tendremos 9 FSMs: dos generadores de ceros (F_1, F_4), dos generadores de unos (F_2, F_5), dos máquinas que intercalan los ceros y unos de los generadores anteriores (F_3, F_6), dos máquinas que simularán un checker de SAT (F_7 y F_8) y una máquina que se encargará de restablecer la configuración inicial del sistema en caso de que la fórmula se satisfaga (F_9). El esquema de comunicación entre ellas sería el que ilustra la siguiente figura:

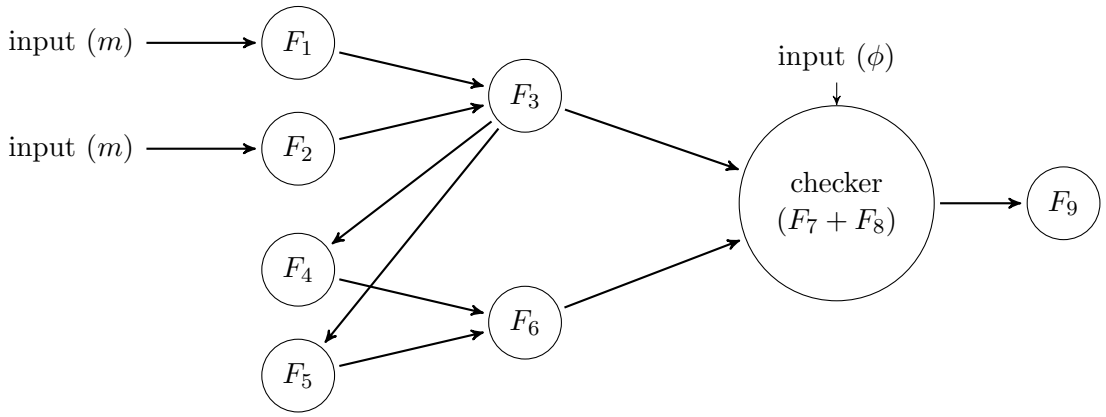


Figura 10.6: Diagrama explicativo de las comunicaciones que se establecen en $\mathcal{S}$.

En el diagrama observamos que esencialmente todas las comunicaciones se hacen de forma que crece el índice de las máquinas, excepto los últimos mensajes que manda F_9 a todas las máquinas para que vuelvan a su estado inicial en caso de que el checker haya determinado que la fórmula se ha satisfecho (que no dibujamos por no hacer el diagrama más farragoso).

La clave de la construcción es aprovechar el no determinismo de las comunicaciones entre los diferentes generadores de ceros y unos, pero con cierto cuidado ya que tenemos que separar bien esa primera parte en la cual estamos simulando cuantificadores universales y la segunda en la cual simulamos los cuantificadores existenciales. Con este propósito hemos introducido dos grupos distintos de generadores: cada uno será el encargado de generar los valores de verdad correspondientes en una fase, pero no funcionarán nunca a la vez.

En concreto, el funcionamiento que buscamos es el siguiente. Las máquinas F_1, F_2 y F_3 se encargarán de generar todas las posibles asignaciones de valores de verdad a las m primeras variables de φ a través del no determinismo de sus comunicaciones, y de transmitírselas al checker. Tras esto, será el turno de F_4, F_5 y F_6, que generarán valores de verdad (de nuevo, aprovechando el no determinismo inherente a sus comunicaciones) para las n últimas variables de φ. Con esto, el checker podrá evaluar si φ se satisface. En caso de que fuera así, le mandaría el mensaje $\top$ a F_9, y F_9 mandaría un mensaje FIN a todas las máquinas para que vuelvan a su estado inicial.

Conforme a esta explicación, podemos hacer una implementación bastante sencilla de las máquinas F_1, F_2 y F_3, en la cual F_1 solo genere m ceros (pues no se van a poder usar más), F_2 también genere m unos y F_3 solo deje pasar hacia el checker los m primeros literales que le lleguen. Tras procesar los $2m$ bits (ceros o unos) que recibe, F_3 les mandará la señal ON a F_4 y F_5 para que empiecen a funcionar, pues ya habría acabado la primera fase del algoritmo en la cual damos valores a las primeras m variables. Las implementaciones las podemos ver gráficamente en las figuras 10.7, 10.8 y 10.9.

Las implementaciones de F_4, F_5 y F_6 serían muy similares a la de F_1, F_2 y F_3, pero

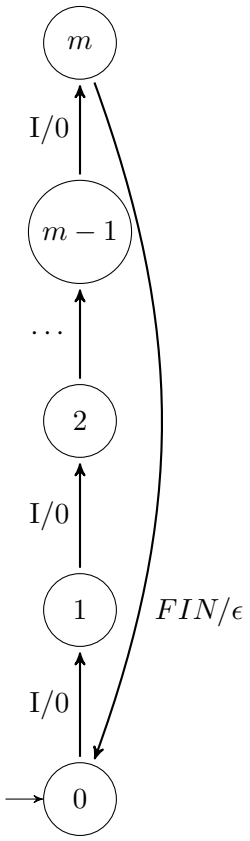


Figura 10.7: Generador de ceros (F_1).

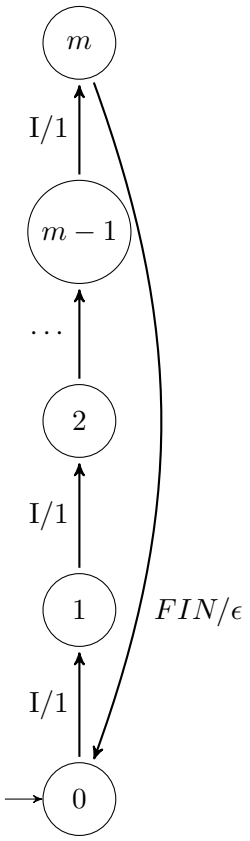


Figura 10.8: Generador de unos (F_2).

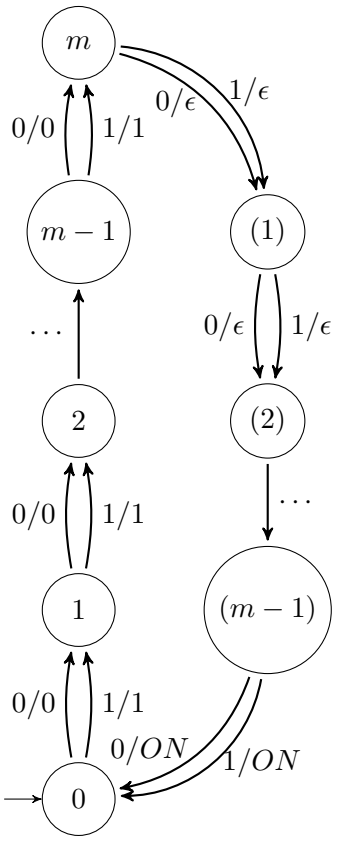


Figura 10.9: Intercalador de ceros y unos (F_3).

generando solo n literales, y solo cuando reciben ON de parte de F_3. Gráficamente podemos ver una implementación de esta idea en las figuras 10.10, 10.11 y 10.12.

Por su parte, F_8 y F_9 funcionan como en la demostración del teorema anterior: toman primero la fórmula φ como input, y después recibirán m ceros o unos de parte de F_3 y n de F_6, que habrán de interpretar como los valores de verdad que se les asignan a las variables $X_1, X_2, \ldots, X_{m+n}$ de φ, en ese orden. En caso de que la fórmula resulte satisfecha, envían un mensaje $\top$ a F_9. Y F_9 solo espera la recepción de $\top$, y de ser así, envía a todas las máquinas el mensaje FIN. Su implementación es muy sencilla, similar a la que teníamos anteriormente, y que podemos ver gráficamente en la figura 10.13.

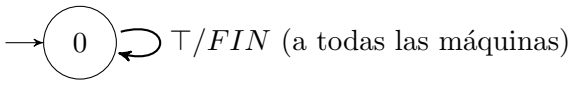


Figura 10.13: Esquema gráfico de F_6.

Con todo esto ya tenemos el sistema creado, ahora tenemos que ver cómo hacemos la reducción. Lo primero es notar que como el checker funciona polinómicamente

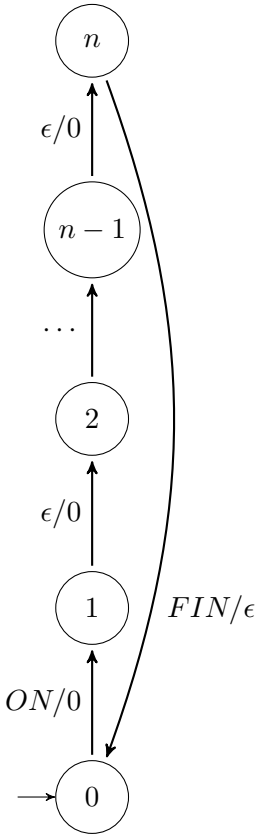


Figura 10.10: Generador de ceros (F_4).

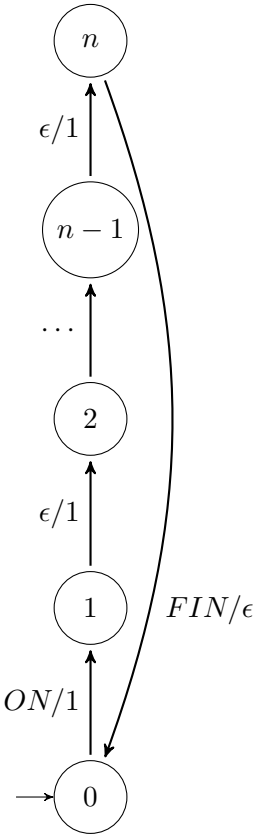


Figura 10.11: Generador de unos (F_5).

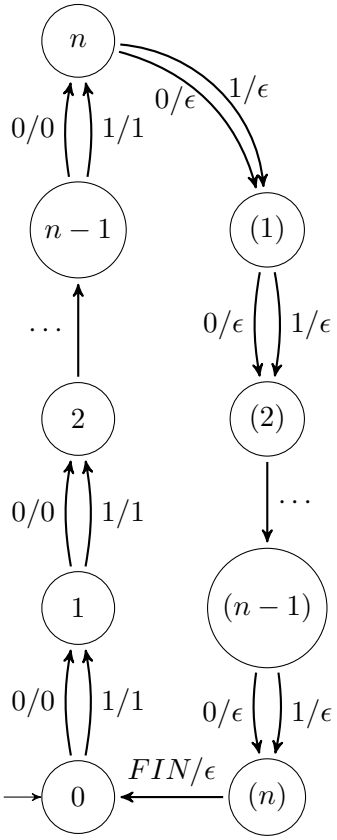


Figura 10.12: Intercalador de ceros y unos (F_6).

en el tamaño de φ (que denotamos $|\varphi|$), podemos decir que, dados el input φ y los $m+n$ bits que F_3 manda al checker, este es capaz de decidir si φ se satisface con estos valores de verdad en a lo sumo $p(|\varphi|)$ pasos, siendo p un polinomio.

Así, podemos reducir el problema:

¿Para cualquier asignación de valores de verdad a $X_1, \ldots, X_m$ existe una asignación a las variables $X_{m+1}, \ldots, X_{m+n}$ de forma que se satisfaga φ?

al problema:

¿Para cualesquiera $3m$ primeros pasos de $\mathcal{S}$, es capaz de volver a su configuración inicial en, a lo sumo, $3n + p(|\varphi|) + 9$ pasos?

Estas cifras son sencillas de entender: $3m$ son los pasos que se necesitan para que F_1, F_2 y F_3 generen suficientes literales como para conseguir asignaciones de valores de verdad para $X_1, \ldots, X_m$; $3n$ son los pasos usados para generar el resto de valores de verdad y pasárselos al checker (dados por F_4, F_5 y F_6); $p(|\varphi|)$ es el número de pasos en que el checker puede decidir si la fórmula se satisface; y 9 son los

pasos necesarios para que cada máquina vuelva a su estado inicial si la fórmula es satisfecha (1 paso para que F_9 mande FIN, y uno más por cada máquina al recibir FIN).

Veamos por qué esta reducción funciona. Lo que tenemos que comprobar primero es que realmente hay una equivalencia entre ambos problemas. Lo veremos viendo las implicaciones en los dos sentidos.

Supongamos primero que para cualesquiera $3m$ primeros pasos de $\mathcal{S}$ hay una secuencia de, a lo sumo, $3n + p(|\varphi|) + 9$ pasos para volver a la configuración inicial. Entonces, en concreto, para las ejecuciones en las cuales los $3m$ primeros pasos de $\mathcal{S}$ se los distribuyen equitativamente F_1, F_2 y F_3 (m pasos cada máquina) para conseguir mandar al checker m ceros o unos, se puede regresar a la configuración inicial en $3n+p(|\varphi|)+9$ pasos. Y este tipo de ejecuciones, debido al no determinismo de las comunicaciones entre los generadores de ceros y unos y las máquinas que los intercalan, modelizan perfectamente el problema de asignar m valores de verdad cualesquiera a $X_1, \ldots, X_m$ y comprobar si hay valores para $X_{m+1}, \ldots, X_{m+n}$ que satisfagan φ. Esto demuestra la primera de las implicaciones.

Comprobemos ahora la otra implicación. Supongamos que para cualquier asignación de valores de verdad a $X_1, \ldots, X_m$ existe una asignación de valores a $X_{m+1}, \ldots, X_{m+n}$ que satisfaga φ. Entonces para cualquier ejecución en la cual los $3m$ primeros pasos los dan F_1, F_2 y F_3 equitativamente habrá una secuencia de a lo sumo $3n + p(|\varphi|) + 9$ para volver a la configuración inicial (esto es claro por lo que vimos en el párrafo anterior). Pero podría ser que los primeros $3m$ pasos de $\mathcal{S}$ no los dieran únicamente F_1, F_2 y F_3. La única forma de que esto ocurriera sería que el checker diera algunos de esos pasos, pues las máquinas F_4, F_5 y F_6 están bloqueadas hasta que les llegue el mensaje ON, que no puede suceder en los primeros $3m$ pasos. Supongamos entonces que, de los $3m$ primeros pasos, el checker da una serie de pasos de forma que solo acaba recibiendo de F_3 el valor de verdad de $k < m$ variables. Como $k < m$, podemos aplicar nuestra hipótesis para afirmar que existen valores de verdad para las variables $X_{k+1}, \ldots, X_{m+n}$ de forma que φ se satisface (en realidad hemos cambiado $m-k$ cuantificadores universales de la hipótesis por cuantificadores existenciales, y esto simplifica el problema). Sabiendo esto, podemos claramente encontrar una secuencia de $3n+p(|\varphi|)+9$ pasos de $\mathcal{S}$ en las cuales F_1, F_2 y F_3 asignan estos valores a $X_{k+1}, \ldots, X_{m+n}$ y la ejecución continúa hasta comprobar que φ se satisface con dichos valores de verdad y se regresa a la configuración inicial.

Formalmente, el argumento se ha basado en la modificación de cuantificadores en la cual se intercambian cuantificadores universales por existenciales, que lógicamente es correcta, como podemos comprobar mediante la siguiente implicación (si $k < m$):

$$\begin{aligned} &\forall\, X_1, \ldots, X_m \,\exists\, X_{m+1}, \ldots, X_{m+n} \text{ tales que } \varphi(X_1, \ldots, X_{m+n}) \\ &\quad \Longrightarrow \ \forall\, X_1, \ldots, X_k \,\exists\, X_{k+1}, \ldots, X_{m+n} \text{ tales que } \varphi(X_1, \ldots, X_{m+n}) \end{aligned}$$

Notemos aquí que ha sido de vital importancia que F_4, F_5 y F_6 estén bloqueadas durante los primeros $3m$ pasos de $\mathcal{S}$, pues de otra forma podríamos haber fijado más de m valores de variables de φ, y a partir de ahí la fórmula podría no ser satisfactible.

Con esto ya hemos visto que los problemas son en realidad equivalentes: ambos dan siempre la misma respuesta ante los mismos datos iniciales. Ahora tenemos que comprobar que la reducción es polinómica. Esto es sencillo de ver por todo lo que hemos ido explicando: en cuanto a tiempos de ejecución es claramente polinómica en el tamaño del input, pues hemos visto que todo se puede resolver en $3(m+n)+p(|\varphi|)+9$ pasos. Además, la construcción del sistema $\mathcal{S}$ no depende del tamaño de la fórmula φ, y el input inicial del sistema es de tamaño $|\varphi|+2m$, que es claramente polinómico en el tamaño del input φ.

Visto todo lo anterior, dado que el problema $QSAT_2$ es Π_2^P-completo y lo hemos conseguido reducir polinómicamente a nuestra versión finitista del problema de regreso a la configuración inicial, este último problema resulta asimismo Π_2^P-completo. □

10.3. Otros problemas de complejidad mayor

En nuestro camino hemos encontrado ya varios problemas de decisión que, al restringirlos a una versión finitista, se vuelven decidibles en el caso de los sistemas de FSMs en comunicación. Si bien, aunque difícilmente tratables en la práctica (pues resolver un problema Π_2^P-completo puede resultar muy costoso), hay algunos problemas cuya complejidad está más arriba aún en la jerarquía. Aquí vamos a ver algunos problemas que son $PSPACE$-completos, es decir, los más difíciles de resolver de la categoría $PSPACE$ de problemas que pueden ser resueltos por una máquina de Turing utilizando una cantidad polinómica de memoria.

Uno de los problemas más conocidos que es $PSPACE$-completo es decidir si un LBA acepta un input dado. De esta forma, dado que las simulaciones que hacíamos en el apartado 6 mediante sistemas de FSMs en comunicación donde los outputs no proliferan solo añadían un sobrecoste polinómico en cuanto a tiempos y mantenían las propiedades de uso de espacio, podemos concluir simplemente que el problema de que un sistema de FSMs en comunicación donde los outputs no proliferan llegue a una configuración de aceptación al recibir un input es $PSPACE$-completo.

También es $PSPACE$-completo el problema de, en una red de Petri, saber si alguna posición alcanzable tiene al menos k tokens en alguno de sus lugares, dados k y un marcaje inicial (como se puede ver en [18]). Esta sería la versión finitista más clara del problema de la acotación, que es tan importante en sistemas en los cuales hay comunicación. De esta forma, con la simulación que hemos explicado en el apartado 8, deducimos también que el mismo problema sería $PSPACE$-completo en el caso de los sistemas de FSMs en comunicación modificados que usábamos en 8 para simular redes de Petri.

Ya vemos que no es difícil llegar a problemas $PSPACE$-completos inspeccionando algunos de los problemas más básicos que querríamos poder resolver de nuestros sistemas. Esto no debería ser en realidad muy sorprendente: ya en otros formalismos similares, como el de las redes de Petri, que tienen un nivel de expresividad similar y han sido ampliamente estudiadas, hay una importante cantidad de problemas interesantes que son $PSPACE$-completos, por ejemplo. En nuestro caso, vamos a

presentar un resultado que no se deduce directamente de las propiedades de otro tipo de máquinas, y puede resultar interesante.

Con lo discutido en los anteriores párrafos, resulta razonable definir el problema de la acotación finita de la siguiente forma: dado un número natural k y una configuración inicial de un sistema de FSMs en comunicación, ¿hay alguna configuración alcanzable en la cual alguno de los buffers tenga al menos k literales?

Proposición 10.5. *El problema de la acotación finita para sistemas de FSMs en comunicación es $PSPACE$-completo.*

Demostración. Es sencillo comprobar que el problema es $PSPACE$: si el sistema tiene n máquinas y k es la constante de acotación del input (que, como anteriormente, suponemos que viene dada en *base unaria*), como el sistema de FSMs no va a tener nunca más de nk literales (en caso de que se cumpla el problema), será sencillo simularlo con nk posiciones en la memoria de una máquina de Turing. Además, codificando los buffers de cada FSM en una cinta diferente de la máquina de Turing, es fácil darse cuenta de que podemos simular el sistema con una máquina de Turing de tamaño polinómico respecto al tamaño de nuestro sistema de FSMs (donde este tamaño viene representado por el número de estados que tienen las distintas máquinas). Como podemos ver, esta construcción usa una cantidad polinómica de espacio respecto al tamaño del input y simula perfectamente el sistema de FSMs si sus buffers están acotados; así que el problema es en efecto $PSPACE$.

Y que sea $PSPACE$-duro lo podemos deducir del resultado que hemos comentado antes sobre la aceptación en LBAs. Por tanto, vamos a hacer una reducción de este problema sobre aceptación en LBAs a nuestro problema de sistemas de FSMs en comunicación, que sea además polinómica.

Para cada LBA F y palabra ω, podemos construir un sistema $\mathcal{S}$ de FSMs en comunicación que lo simule, que tendrá k literales al inicio de su ejecución entre todos los buffers. En concreto, lo haremos de forma que los outputs no proliferen. Esto nos asegura que podemos conseguir simular todo el proceso de aceptación (o no aceptación) de ω sin que en $\mathcal{S}$ aumente el número de literales entre todos los buffers (propiedad de los sistemas donde los outputs no proliferan). Añadimos al final una modificación a $\mathcal{S}$ para resolver el problema: en caso de que F acepte ω, $\mathcal{S}$ llegará a una de sus configuraciones de aceptación. Y de cada una de estas configuraciones de aceptación de $\mathcal{S}$ simplemente tenemos que añadir algún estado que entre en bucle a producir literales sin parar.

De esta forma $\mathcal{S}$ será un sistema donde los outputs pueden proliferar, y donde de hecho, solo habrá algún buffer que tenga al menos $k+1$ literales en el caso de que se haya llegado a estos estados en los cuales se producen literales *descontroladamente*, lo cual solo puede ocurrir si F acepta ω. De esta forma deducimos que $\mathcal{S}$ no es $k+1$ acotado si y solo si F acepta ω, y la transformación ha sido polinómica en el tamaño de F y ω. Al ser el problema de si F acepta ω $PSPACE$-completo, también lo ha de ser el de la acotación finita de sistemas de FSMs. □

Con esto hemos comprobado que en sistemas de FSMs en comunicación, dada su gran expresividad, hay problemas de gran variedad de complejidades, y tenemos

también una idea de cuál es esta complejidad para algunos de los problemas más interesantes de los modelos concurrentes que se aplican a nuestros sistemas. Este es por tanto un buen punto de partida al intentar usar este tipo de sistemas.

Capítulo 11

Un sistema Turing universal

En la teoría de computabilidad ha sido siempre un problema interesante conseguir máquinas de Turing universales lo más pequeñas o sencillas posible. Se han llegado a construir máquinas muy pequeñas: ya en 1962 M. Minsky encontró una con únicamente 7 estados y un alfabeto de 4 símbolos, y esto ha ido mejorándose hasta llegar, por ejemplo, a una máquina con solo 2 estados (aunque 18 símbolos en el alfabeto) u otra con 3 estados y 9 literales en su alfabeto (ambas propuestas por Y. Rogozhin en 1996 [24]). Relajando la noción de Turing completitud a máquinas de Turing que usan una cinta inicial un tanto modificada, que no tenga a ambos lados infinitos símbolos de blanco (como vamos a ver en la teoría que desarrollaremos en las siguientes páginas), se han llegado a conseguir máquinas de Turing universales con solo 2 estados y 4 símbolos en el alfabeto (por T. Neary y D. Woods en 2007 [22]); resultados que ya son bastante difíciles de superar, e incluso podría ser que en algunos casos fueran óptimos.

En nuestro caso, vamos a construir un sistema de FSMs en comunicación con 36 estados en total que sea universal, y que tenga un alfabeto con solo 3 literales. No supone ni mucho menos un número tan bajo como los récords que comentábamos antes, pero supone un resultado interesante en términos de simplicidad: esencialmente el funcionamiento del sistema está concentrado en únicamente 4 estados, y la mayoría del resto estarán casi repetidos, pues tendrán una estructura muy bien definida que vamos a iterar. Consideramos por tanto interesante este resultado porque resulta mucho más intuitivo de entender que muchas otras de las máquinas que en su momento constituyeron un récord, que son más artificiales en cuanto a sus métodos de construcción y se entiende, por tanto, peor la forma en que funcionan.

La construcción que haremos está basada en el autómata 110, un autómata celular cuyo comportamiento aparentemente sencillo contrasta con la propiedad básica por la cual es conocido: es Turing completo.

Recordemos que los autómatas celulares son aquellos que tienen una serie de posiciones (normalmente con formas regulares, como en forma de cuadrícula o alineados), pudiendo estar cada posición en una cantidad finita de estados en cada instante. Y estos autómatas van evolucionando con el tiempo de forma que el estado de cada posición en el tiempo $t + 1$ depende únicamente de del estado de sus *posiciones vecinas* en el instante t (siendo esta relación de vecindad y la forma de cambiar de cada posición diferente en cada autómata).

En este sentido, el autómata 110 simplemente opera en paralelo sobre un vector infinito de celdas (que podemos ver como la cinta de una máquina de Turing), de

forma que en cada movimiento genera una nueva cinta completa, en la cual cada posición depende únicamente de su valor en la cinta en el instante anterior y la de sus dos posiciones inmediatamente adyacentes. El alfabeto de cinta se supone binario (aquí trabajaremos con 0 y 1). Las transiciones se pueden ver en la siguiente tabla, que muestra cuál es el valor de una celda en el instante $t+1$ sabiendo el valor que tenían sus 3 celdas vecinas en el instante t (la de su izquierda, ella misma y la de su derecha, en ese orden):

Cinta en t	111	110	101	100	011	010	001	000
Cinta en $t+1$	0	1	1	0	1	1	1	0

Si interpretamos estas transiciones como un número en decimal, resulta ser el 110 (de un total de 256 autómatas diferentes que podrían definirse del mismo modo), de ahí el nombre del autómata.

Para simular el comportamiento del autómata 110, usaremos un enfoque similar al del apartado 6.2, en el cual nuestro sistema barría de izquierda a derecha la cinta de la máquina de Turing, generando así una nueva cinta tras cada barrido. En este caso se aplica también la idea de retrasar los outputs un ciclo respecto a los inputs: antes era porque el cabezal podía moverse a la izquierda, y ahora es debido a que cada letra puede afectar a la que está a su izquierda cuando el autómata genere un output para la siguiente posición de la cinta. Visto de otro modo, a la hora de ir recorriendo de izquierda a derecha la cinta, hemos de recordar los dos valores que acabamos de ver, y solo a la hora de ver el tercero tomaremos la decisión de sacar un output, que estará un ciclo retrasado conforme a la idea que propone el autómata 110 (podríamos decir que el nuevo valor de una celda se genera cuando visitamos el valor antiguo de su vecina derecha), si bien no supone un problema.

A modo de ejemplo tenemos la figura 11.1: si a_i^t representa el contenido de la posición i de la cinta en el instante t, entonces las flechas diagonales lisas indican cuándo se genera cada literal de la cinta de $t = 1$, mientras que las flechas discontinuas nos indican qué otras posiciones hemos tenido que tener en cuenta para generar dicha posición, y que por lo tanto, de alguna manera nuestro sistema ha de recordar.

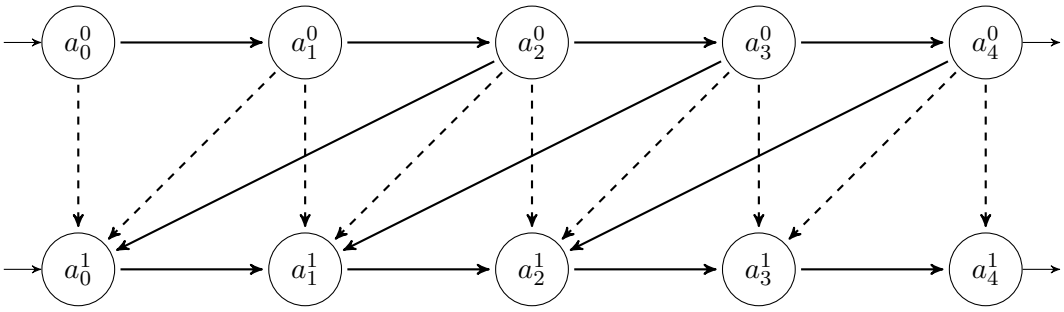


Figura 11.1: Representación gráfica de cómo se retrasan las producciones.

Resulta necesario pensar también en una forma de delimitar la cinta actual, pues no queremos que el final de la cinta en el instante t influya en los outputs generados

al inicio de la cinta del instante $t+1$. Para ello introduciremos el literal #, como ya hicimos anteriormente, que denotará el fin de una cinta y, a la vez, el comienzo de la siguiente.

Con estas ideas, podríamos crear un autómata bastante sencillo que simule el funcionamiento del autómata 110, como representamos en la siguiente figura:

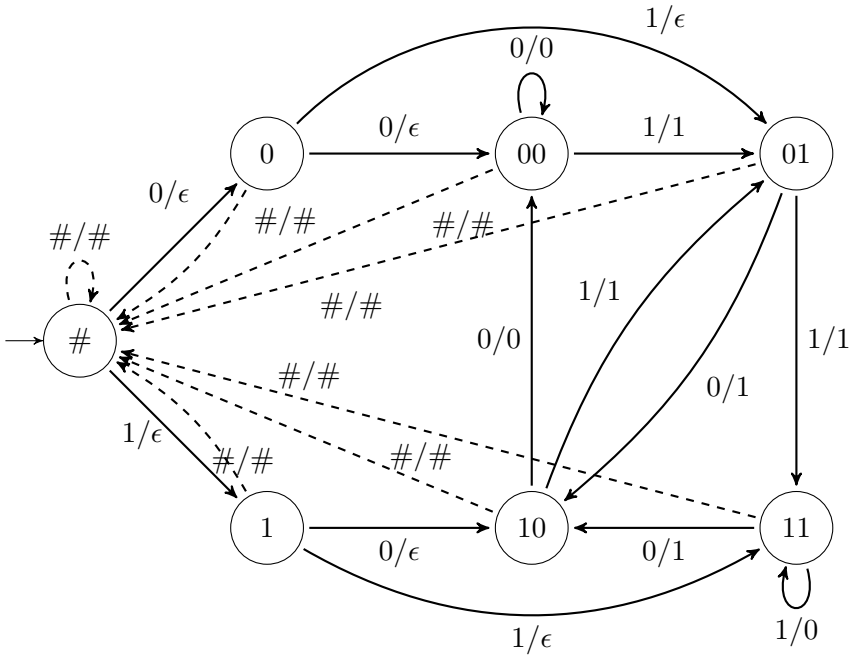


Figura 11.2: Implementación de la FSM que simula las producciones del autómata 110.

Como vemos, esencialmente estos 7 estados ya aglutinan el funcionamiento del autómata 110, y por tanto simulan algo que tiene tanta expresividad como para ser Turing completo. Además, los 4 estados 00, 01, 10, 11 son los que llevan la mayor parte del significado, pues los otros 3 solo reflejan una especie de *estado transitorio* del sistema, en el cual todavía no hemos leído suficientes caracteres como para generar un output.

Pero ahora se nos presenta la mayor dificultad de la construcción: la demostración de que el autómata 110 es Turing completo, llevada a cabo por Matthew Cook en 2004 [8, 9] para dar una contestación positiva a la conjetura de Stephen Wolfram en 1985, presupone que la cinta es un poco diferente a lo que estamos acostumbrados: no está rellena entera de caracteres *blancos* a izquierda y derecha de la zona que estamos tratando, sino que supone que hay un patrón repetitivo que se repite indefinidamente hacia izquierda y derecha. Cook llama a este patrón repetitivo *éter*, y es la cadena 00010011011111, de longitud 14. Al aplicar la regla del autómata 110 sobre el éter, este se desplaza 4 posiciones a la izquierda. De este modo, tras 7 iteraciones, obtenemos de nuevo el éter en su posición inicial, y todo se repite cíclicamente.

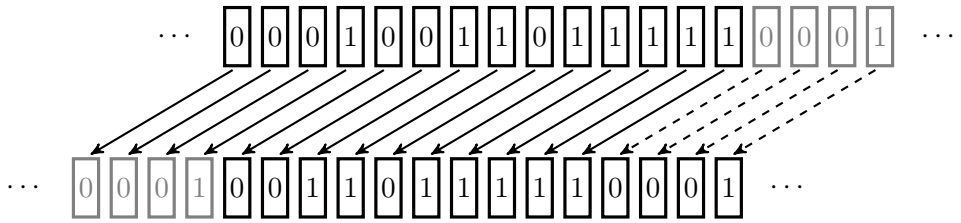


Figura 11.3: Una iteración de la evolución del éter.

Lo que nos falta entonces para conseguir que nuestro sistema simule realmente el autómata 110 de forma que consigamos un sistema Turing universal es, de alguna forma, simular el éter. Para ello introduciremos dos máquinas adicionales: una simulará la creación de éter a la derecha de la zona de la cinta que estamos tratando (F_2), y otra a la izquierda (F_0). La máquina que simulaba el autómata 110 y que hemos explicado en la figura anterior será la máquina F_1. Tendremos por tanto un sistema $\mathcal{S}$ con 3 máquinas que se comunicarán de manera cíclica: F_0 le mandará la cinta completa a F_1, F_1 a F_2 la nueva cinta generada, F_2 a F_0 la cinta habiendo añadido un carácter por la derecha, y cuando le llegue de nuevo a F_1 a través de F_0, esta habrá añadido un carácter más del éter por la izquierda. Como comentamos, tras cada barrido de la cinta, tanto F_0 como F_2 añadirán un carácter nuevo a la cadena que está procesando el sistema $\mathcal{S}$. Esto hace que cada vez haya más literales en $\mathcal{S}$ y que no todos sean especialmente relevantes (pues la mayoría van a continuar siguiendo el patrón del éter, porque las modificaciones no se han extendido tanto). Sin embargo, es una manera fácil de resolver la cuestión de la simulación del éter, porque si no, tendríamos que crear un método de detección de qué zonas de la cinta han cambiado o no en cada momento, lo cual resultaría mucho más costoso.

Sin más dilación, pasamos a definir F_0. Como forzosamente el sistema tiene que recordar en cuál de los 14 elementos del periodo del éter está (por la izquierda en este caso), F_0 tiene que tener al menos 14 estados. Veamos que basta con 14: definimos un estado q_i (para cada $i \in \{0, \ldots, 13\}$) de forma que si F_0 está en el estado q_i, esto significa que el siguiente literal a la izquierda de la región de la cinta que la máquina F_1 ha considerado hasta ahora es el i-ésimo del patrón del éter, que recordemos que es 00010011011111. Como sabemos que, tras una pasada de las máquinas, el éter se desplaza 4 posiciones a la izquierda. Eso nos quiere decir que, si en el tiempo t en una posición de la cinta estábamos en el carácter i del éter, en el instante $t+1$ en esa posición estará el carácter $i+4$ (módulo 14, claro). Pero como ese carácter lo habrá introducido F_0 en el sistema en tiempo t, y ahora estamos interesados en el carácter justo a su izquierda, es decir, en el $i+3$ del éter. Por tanto, de q_i transitaremos a $q_{(i+3) \mod 14}$. Con esto se nos crea una estructura muy regular y sencilla que nos permite generar éter paso a paso a la izquierda de la zona que está tratando nuestro sistema hasta el momento.

Una vez tenemos esto claro, solo tenemos que darnos cuenta de cuál es el funcionamiento que esperamos de F_0: debería reenviar todo lo que lee, y únicamente añadir una letra en la siguiente cinta, es decir, que solo añade la letra correspondiente del éter cuando le llega el carácter # (lo cual hacemos, por simplicidad, generando 2 outputs al consumir el input #, lo cual vimos ya tras la definición 5.1 que era una

forma razonable de interpretar el fenómeno subyacente de proliferación de los outputs). Todo este pensamiento quedaría plasmado en un autómata de la forma que mostramos en la figura 11.4.

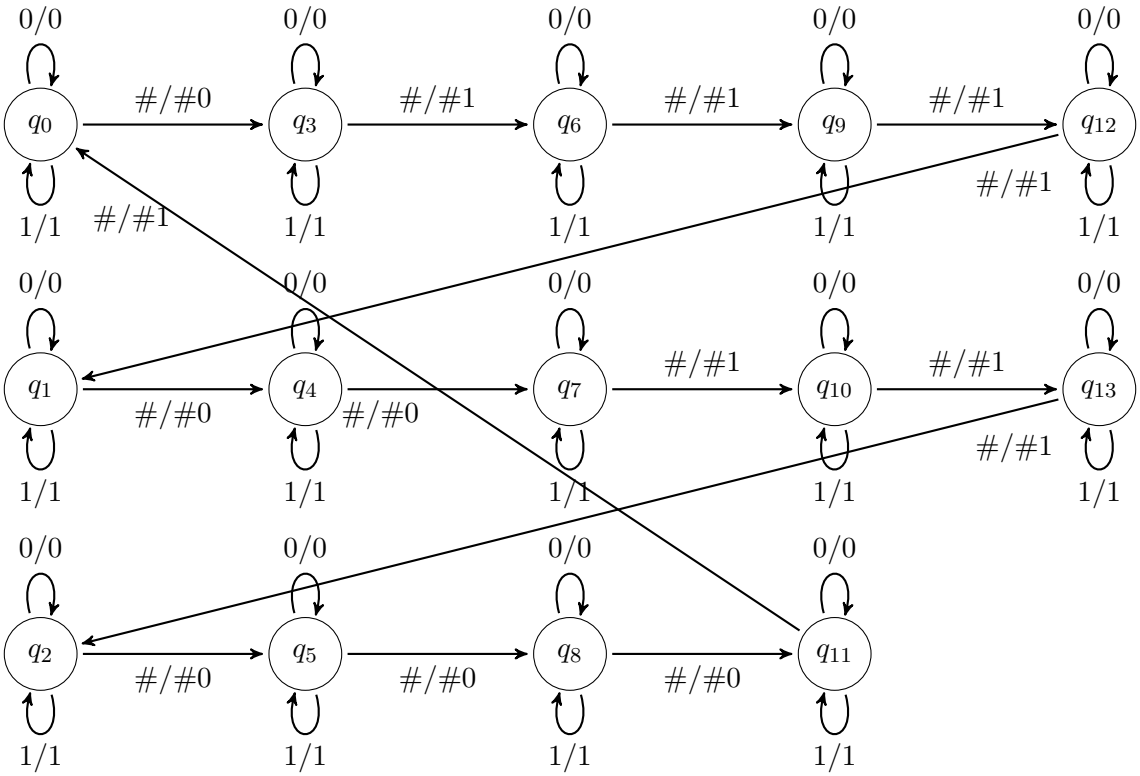


Figura 11.4: Implementación de F_0 en la construcción.

Una vez entendida la construcción del autómata F_0, la construcción de F_2 resulta sencilla por analogía. Las transiciones cambiarán poco, pues desde el estado q_i transitaremos al $q_{(i+5) \mod 14}$ porque estamos interesados en la posición a la derecha de la $i+4$, no en la izquierda como en F_0. Y de nuevo, F_2 solo añade literales al sistema una vez sabe que le ha llegado el fin de la cinta, es decir, cuando lee #. Esto nos daría el autómata de la figura 11.5.

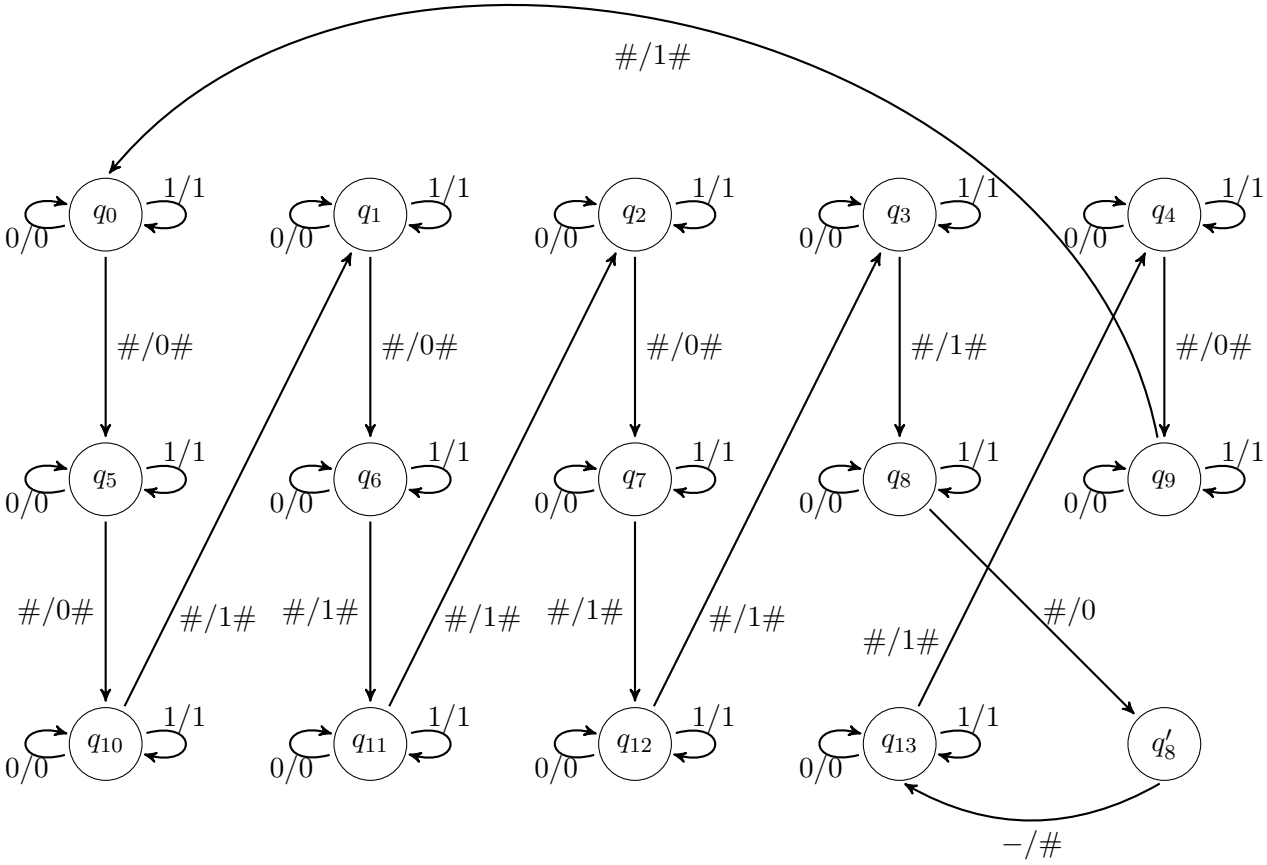


Figura 11.5: Implementación de F_2 en la construcción.

Por último, comentamos cómo se inicia todo el procedimiento de simulación: en F_1 el estado inicial es #; en F_0 es q_0; y en F_2 es q'_8 (que hemos separado de q_8 para que al inicio solo se produzca la # que necesitamos). Además, el input se pondrá en el buffer del que lee F_2. Así, habrá una primera vuelta en la cual F_2 añade # y F_0 y F_2 la reenvían. Por tanto, el buffer de F_2 ahora tendrá el input inicial seguido de #. Y con eso ya empieza el reconocimiento usual, y F_2 empieza a añadir caracteres correctamente porque está en q_{13} (y el éter va hacia la izquierda), y F_0 estaba añadiendo partes correctas del éter desde el principio.

Capítulo 12

Conclusiones

A la vista de todo el trabajo expuesto hasta ahora, ya podemos extraer varias conclusiones interesantes sobre los sistemas de máquinas de estados finitos en comunciación. Lo primero que hemos comprobado es que dentro del mismo marco teórico tienen cabida diferentes definiciones, y cada una de ellas implica diferentes propiedades de los sistemas. En concreto, las versiones más usuales tienen una expresividad bastante alta y están siempre en la línea entre lo decidible y lo indecidible (como hemos visto en los capítulos 4, 6, 8 y 9). El hecho de que estén en esta frontera motiva la variedad de definiciones que tenemos: dependiendo del caso estaremos interesados en una ganancia en expresividad o en comprensión de nuestros sistemas, y elegiremos versiones en uno u otro lado de la frontera dependiendo de estos intereses.

En concreto, sabemos que los sistemas de máquinas de estados generales son Turing completos (capítulo 4) y eso hace que los problemas de decisión asociados a ellos sean indecidibles (Teorema de Rice). Pero además, versiones simplificadas de estos problemas de decisión (que hemos estudiado en el capítulo 10) son también difíciles de resolver en general, como el caso de la alcanzabilidad finita, que hemos comprobado que es $\mathcal{NP}$-completo. Además, por ser Turing completo, podemos construir sistemas que sean universales. En este caso ha sido interesante la construcción basada en el autómata 110 (capítulo 11) por ser simple y sencilla de entender, además de relativamente pequeña.

Respecto a las diferentes modificaciones podemos decir también varias cosas. Sabemos que los sistemas donde los outputs no proliferan son completos dentro de los reconocedores de lenguajes dependientes del contexto, lo que los sitúa cerca de la Turing completitud, pero hace que sean más sencillos de estudiar a pesar de esa gran expresividad. En concreto hemos podido comprobar que algunos problemas de vital importancia, como el de saber si aceptan un input dado, es decidible, aunque tiene una complejidad muy elevada (vimos que era $PSPACE$-completo en el capítulo 10).

El caso de los sistemas donde los buffers están acotados es más sencillo. Comprobamos que son equivalentes a los autómatas finitos en el capítulo 7 y que eso hace que problemas anteriormente estudiados como la alcanzabilidad finita sean polinómicos (en el capítulo 10).

Era más interesante el caso de los sistemas donde los buffers no tienen orden, estudiados en el capítulo 8. En este caso, diferentes adaptaciones de la definición llevaban a la equivalencia con diferentes redes de Petri: las usuales, con arcos inhibidores o con arcos restablecedores. En todos los casos resultaba interesante la

comparativa entre la complejidad de ciertos problemas básicos como la alcanzabilidad o el recubrimiento, que están en la frontera de lo decidible e indecidible en estos tres tipos de redes de Petri, como vimos al final del capítulo.

Por último estudiamos una variación en la cual los canales de comunicación pueden fallar, dando lugar al formalismo (ampliamente estudiado en la literatura) de los Sistemas de Mensajes Perdidos. Hemos recopilado algunos de las propiedades básicas de estos sistemas en el capítulo 9, como que el problema de la terminación es decidible (aunque de una complejidad extremadamente alta).

Con esto hemos establecido una teoría básica bastante amplia sobre sistemas formados por máquinas de estados finitos en comunicación, que puede servir en muchos otros trabajos como referencia de qué propiedades son esperables dependiendo de las características que tengan nuestros canales de comunicación. Además hemos expuesto una colección bastante amplia de variaciones en la definición principal, de forma que todos estos modelos se podrán reutilizar en casos en los que tengamos que implementar protocolos en los cuales máquinas sencillas tienen que comunicarse para realizar una tarea más compleja.

Conclusions

From the work that has been done we can already draw several interesting conclusions about communicating Finite State Machines. The first thing that has become clear through all these pages is that several definitions are possible under this abstract framework, and each of them has different properties. The most usual definitions are on the edge between decidability and undecidability and have very high expressivity (as we have seen in Chapters 4, 6, 8 and 9). The fact of being in that frontier motivates the variability of definitions we have introduced: depending on the situation we will be interested in a more expressive model or in a model where properties are easier to comprehend.

In particular we can affirm that the general model of communicating Finite State Machines are Turing complete (seen in chapter 4), and that makes its decision problems to be undecidable (Rice's Theorem). Moreover, some simplified versions of those decision problems are still quite difficult to solve: for instance we have proved that a finitary version of reachability is $\mathcal{NP}$-complete in chapter 10. Furthermore, due to being Turing complete, we can construct universal systems. We exploit this in chapter 11, getting a fairly simple system (yet small) by simulating Rule 110.

Regarding the different modifications on the definition, we can also say some interesting things. In the case of the systems where outputs are not allowed to spread, we have verified that they are complete among the context-sensitive language recognizers. Thus they are a really expressive model (close to Turing machines) but easier to study in terms of their decidable properties. Concretely we have confirmed that some crucial properties such as the word problem are decidable, though their complexity is really high (in chapter 10 we saw it was in fact $PSPACE$-complete).

The case of systems where the buffers used to communicate are bounded turned out to be easy to study. In chapter 7 we found that they were equivalent to finite automata and that makes some decision problems easy, as in the case of the finitary reachability, which we proved to be polynomial in chapter 10.

The case where the order inside those buffers is suppressed, studied in chapter 8, is more interesting: different definitions led to equivalences with different kinds of Petri nets: the usual ones, with inhibitor arcs or with reset arcs. In all these cases it was interesting to compare how some basic problems -such as reachability or coverability- are on the thin edge between decidable or undecidable depending on the definitions.

Lastly we studied a modification where the communication channels are faulty, leading us to the -well studied in the literature- model of Lossy Channel Systems. We have summed up some of the most important and basic properties of these systems in chapter 9. For instance, the problem of termination becomes -possibly *a priori* counterintuitively- decidable, though having an extremely high complexity.

With all this we have set a basic theory on communicating Finite State Machines which is fairly broad: it can be useful in many other works as a reference to know what to expect of a system depending on the properties of our channels of communications. Furthermore we have provided a wide spectrum of possible variations which can also be reused to implement different protocols where simple machines are supposed to communicate to perform a more difficult task.

Bibliografía

[1] P. Abdulla y B. Jonsson. «Undecidable Verification Problems for Programs with Unreliable Channels». En: *Information and Computation* 130.1 (1996), págs. 71-90. DOI: https://doi.org/10.1006/inco.1996.0083.

[2] P. Abdulla y B. Jonsson. «Verifying Programs with Unreliable Channels». En: *Information and Computation* 127 (1996), págs. 91-101.

[3] T. Agerwala. «A Complete Model for Representing the Coordination of Asynchronous Processes». En: *Hopkins Computer Research Reports* 32 (1974).

[4] T. Araki y T. Kasami. «Some decision problems related to the reachability problem for Petri nets». En: *Theoretical Computer Science* 3.1 (1976), págs. 85-104. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(76)90067-0.

[5] M. Blondin, A. Finkel y P. McKenzie. «Handling infinitely branching well-structured transition systems». En: *Information and Computation* 258 (2017). DOI: 10.1016/j.ic.2017.11.001.

[6] G. Bochmann. «Finite State Description of Communication Protocols». En: *Computer Networks* 2 (1978), págs. 361-372.

[7] D. Brand y P. Zafiropulo. «On Communicating Finite State Machines». En: *Journal of the Asociation for Computing Machinery* 30.2 (1983), págs. 323-342.

[8] M. Cook. «A Concrete View of Rule 110 Computation». En: *The Complexity of Simple Programs* 15 (2009), págs. 31-55. DOI: 10.4204/EPTCS.1.4.

[9] M. Cook. «Universality in Elementary Cellular Automata». En: *Complex Systems* 15 (2004), págs. 1-40.

[10] G. Díaz, J. Mateo, P. Rabanal e I. Rodríguez. «A centralized and a decentralized method to automatically derive choreography-conforming web service systems». En: *J. Log. Algebr. Program.* 81 (2012), págs. 127-159. DOI: 10.1016/j.jlap.2011.10.001.

[11] C. Dufourd, A. Finkel y Ph. Schnoebelen. «Reset nets between decidability and undecidability». En: *Proc. 25th. Int. Coll. Automata, Languages and Programming* 1443 (1998), págs. 103-115.

[12] J. Esparza. *Petri Nets, Lecture Notes*. 2017.

[13] M. Gouda, E. Manning e Y. Yu. «On the progress of communication between two machines». En: (oct. de 1980), págs. 369-389. DOI: 10.1007/3-540-11604-4_62.

[14] M. Gouda y L. Rosier. «Communicating finite state machines with priority channels». En: *Paredaens J. (eds) Automata, Languages and Programming. ICALP 1984. Lecture Notes in Computer Science* 172 (1984).

[15] M. Gouda e Y. Yu. «Deadlock Detection for a Class of Communicating Finite State Machines». En: *IEEE Transactions on Communications* 30.12 (1982), págs. 2514-2518.

[16] M. Gouda e Y. Yu. «Unboundedness detection for a class of communicating finite-state machines». En: *Information Processing Letters* 17.5 (1983), págs. 235-240. DOI: https://doi.org/10.1016/0020-0190(83)90105-9.

[17] J. Hopcroft y J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley series in Computer Science. Addison-Wesley, 1979.

[18] N. Jones, L. Landweber y E. Lien. «Complexity of some problems in Petri nets». En: *Theoretical Computer Science* 4.3 (1977), págs. 277-299. DOI: 10.1016/0304-3975(77)90014-7.

[19] D. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer, 1997.

[20] B. Masson y Ph. Schnoebelen. «On Verifying Fair Lossy Channel Systems». En: *Lecture Notes in Computer Science* (2002). DOI: 10.1007/3-540-45687-2_45.

[21] R. Mayr. «Undecidable problems in unreliable computations». En: *Lecture Notes in Computer Science* 1776 (2003), págs. 377-386. DOI: 10.1007/10719839_37.

[22] T. Neary y D. Woods. «Small weakly universal Turing machines». En: *CoRR* abs/0707.4489 (2007). URL: http://arxiv.org/abs/0707.4489.

[23] J. Pachl. «Reachability problems for communicating finite state machines». En: *Computing Research Repository - CORR* (2003). URL: arxiv.org/abs/cs/0306121v2.

[24] Y. Rogozhin. «Small universal Turing machines». En: *Theoretical Computer Science* 168.2 (1996), págs. 215-240.

[25] Louis E. Rosier y Hsu-Chun Yen. «Boundedness, empty channel detection, and synchronization for communicating finite automata». En: *Theoretical Computer Science* 44 (1986), págs. 69-105. DOI: https://doi.org/10.1016/0304-3975(86)90110-6.

[26] Ph. Schnoebelen. «Verifying lossy channel systems has nonprimitive recursive complexity». En: *Information Processing Letters* 83 (2002), págs. 251-261. DOI: 10.1016/S0020-0190(01)00337-4.

[27] Ph. Schnoebelen. «Verifying lossy channel systems has nonprimitive recursive complexity». En: *Information Processing Letters* 83 (2002), págs. 251-261. DOI: 10.1016/S0020-0190(01)00337-4.

[28] D. A. Wolfram. «Solving generalized Fibonacci recurrences». En: *The Fibonacci Quarterly* 36 (1997), págs. 129-145.