Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales

Trabajo Terminal II
2020-B102

Computación circular mediante colisiones de partículas en autómatas celulares elementales

Presenta:
Oswaldo Leyva Barrientos

Director:
Dr. Genaro Juárez Martínez

Ciudad de México, 17 de enero de 2022

CARTA RESPONSIVA

Que otorga visto bueno y avala la conclusión de documentación del Trabajo Terminal bajo los lineamientos establecidos por la Comisión Académica de Trabajos Terminales (CATT)

CDMX, a 17 de enero de 2022

M EN C. ANDRÉS ORTIGOZA CAMPOS
PRESIDENTE DE LA COMISIÓN ACADÉMICA DE TRABAJOS TERMINALES
P R E S E N T E

EN ATENCIÓN A:
M. EN E. ELIA TZINDEJHÉ RAMÍREZ MARTÍNEZ
SECRETARIA EJECUTIVA

Por medio de la presente, se informa que el Trabajo Terminal Núm. 2020-B102

Que lleva por Título: Computación circular mediante colisiones de partículas en autómatas celulares elementales

Fue concluido satisfactoriamente por:

Leyva Barrientos Oswaldo

Se avala que la documentación entregada mediante discos en formato DVD fue **revisada de manera precisa y exhaustiva** con el propósito de asegurar que los avances desarrollados bajo la supervisión de quien o quienes suscriben, hayan cumplido con lo planteado en el protocolo original, así como en lo establecido por el Documento Rector de Operación y Evaluación para los Trabajos Terminales de la ESCOM.

ATENTAMENTE
"LA TÉCNICA AL SERVICIO DE LA PATRIA"

Dr. Genaro Juárez Martínez
Director

SUBDIRECCIÓN ACADÉMICA
DEPARTAMENTO DE FORMACIÓN INTEGRAL E INSTITUCIONAL
CATT COMISIÓN ACADÉMICA DE TRABAJOS TERMINALES

FORMATO:
CARTA RESPONSIVA

Advertencia

"Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan."

La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen.

Información adicional sobre este reporte técnico podrá obtenerse en:

La Subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n, esq. Miguel Othón de Mendizabal, Unidad Profesional Adolfo López Mateos. Teléfono: 5729-6000, extensión 52000.

A mis profesores,

Por haberme dado las herramientas para construir mi propio sendero, dentro y fuera del salón de clase.

A mi director, el Dr. Genaro Juárez Martínez,

Por su apoyo y guía en mis años de carrera, durante este periodo de clausura y los tiempos por venir.

A mis amigos y compañeros,

Por ayudarme a recorrer este camino en compañía y enseñarme el valor de los lazos humanos.

A mis padres,

Por construir los cimientos de lo que soy ahora. Por su ayuda incondicional y su sacrificio día a día para darme la oportunidad de convertirme en mi mejor versión.

A mi hermana Genny,

Por ser la luz de mi vida y mi motivación para seguir adelante cada día. Y por darme una razón para apreciar el mundo de nuevo.

Gracias.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Resumen

Los autómatas celulares (AC) han sido de mucho interés en la teoría de la computación por su arquitectura simple pero capaz de generar comportamientos dinámicos complejos, y varios de ellos han sido probados como computacionalmente universales. En este trabajo se implementará un visor y constructor de configuraciones para simular un colisionador virtual clásico de partículas en un autómata celular elemental (ACE). A partir de esto, se analizarán las partículas generadas por un ACE para implementar un sistema computable mediante choques.

1.2. Planteamiento del problema

El estudio del comportamiento dinámico de los ACs se ha convertido en un área significante de las matemáticas y la computación teórica en los últimos años. Los ACs proveen un marco de trabajo para una clase de sistemas dinámicos discretos que permiten comportamientos complejos e impredecibles emergentes de las interacciones locales determinísticas de componentes simples actuando en paralelo. [1]

El modelo de AC fue propuesto por John von Neumann en los años 50. Un AC es un modelo matemático definido por un arreglo d-dimensional, teóricamente infinito, en el que cada celda (llamada también célula) contiene un símbolo de un conjunto finito, comunmente enteros. El arreglo es actualizado en pasos finitos, en donde cada celda al tiempo t_i actualiza su valor simultáneamente mediante una función de los valores de ésta y sus celdas contiguas, llamadas vecindad. El AC evoluciona a través de una serie de estados globales mediante iteraciones de este proceso de actualización, llamada regla de transición. [2]

Una particularización de un autómata celular es el autómata celular elemental (ACE). Un ACE es una arquitectura unidimensional de malla finita con periodicidad en sus fronteras, con 2 posibles valores y una vecindad local. En ellos, se puede determinar un total de $2^3 = 8$ posibles combinaciones de estados y, por consiguiente, $2^{2^3} = 256$ posibles reglas para aplicar a la celda objetivo. Una regla podría mapear, por ejemplo $(1,1,1) \to \mathbf{0}, (1,1,0) \to \mathbf{0}, \ldots, (0,0,1) \to \mathbf{0}, (0,0,0) \to \mathbf{1}$, y sería entonces llamada regla $(\mathbf{00}\ldots\mathbf{01})_b$, o regla 1.

Los ACs han sido profundamente estudiados en las investigaciones de Stephen Wolfram, al detectar que pueden soportar comportamientos complejos identificando la existencia de estructuras periódicas bien definidas en su espacio de evoluciones. Wolfram propuso una clasificación de los ACs en 4 clases, numeradas en orden de complejidad:

- Clase I: su comportamiento es simple y casi todas las condiciones iniciales llevan al mismo estado final.
- Clase II: hay diferentes estados finales, pero todos se componen de un conjunto de estructuras simples que no cambian o que se repiten cada cierto número de pasos.
- Clase III: su comportamiento es más complicado, aparentemente aleatorio y pueden llegar a formarse triángulos u otras estructuras pequeñas en algún punto.

- Clase IV: combina orden y aleatoriedad, produciendo estructuras que, si bien pueden ser simples, se mueven e interactúan entre ellas en formas muy complejas. [1]

Algunos de los ACs que pertenecen a la clase IV han sido probados como computacionalmente universales y, por tanto, capaces de realizar operaciones y resolver problemas.

Particularmente, este trabajo se concentrará en el estudio del ACE regla 110.

La regla 110 fue investigada por Matthew Cook, quien demostró que ésta es computacionalmente universal. Cook identificó la formación de patrones periódicos complejos, a los cuales llamó partículas, y los clasificó por su comportamiento y características. A partir de esto construyó una configuración global que lograba emular un sistema tag cíclico con condiciones iniciales y reglas de producción específicas mediante las colisiones de estas partículas. [3] Más tarde, el sistema de Cook fue reinterpretado en el libro A New Kind Of Science, por Stephen Wolfram.

Genaro J. Martínez, Andrew Adamatzky y Harold V. McIntosh idearon un colisionador de partículas de ACE, basándose en el trabajo de 1970 de Fredkin y Toffoli sobre computación basada en interacciones balísticas. [4] Consideraron al ACE como un anillo, en el que partículas con la misma velocidad horizontal giran en una forma análoga a como lo hacen partículas reales en un colisionador clásico. Luego, usando 2 anillos girando en diferentes sentidos, inyectaban las partículas en un anillo final para ver el comportamiento que generaba su choque. [5] [6] El simulador que crearon permitía la visualización individual de los anillos, pero no realizaba el proceso de inyección en el colisionador ni la visualización del sistema de 3 anillos trabajando simultáneamente.

Se plantean, entonces, dos problemáticas a resolver: la necesidad de crear un simulador para visualizar un colisionador completo y, a partir de éste, explorar la reinterpretación de Wolfram sobre el trabajo de Cook para implementar un modelo de sistema computable.

1.3. Objetivos

1.3.1. Objetivo general

Implementar un sistema para simular un colisionador de partículas en un ACE y aplicarlo para ejecutar una computación a partir de choques.

1.3.2. Objetivos específicos

- Desarrollar una interfaz para la visualización de anillos de partículas en un ACE.
- Desarrollar un simulador de colisiones mediante configuraciones de anillos.
- Analizar colisiones de partículas e implementar un sistema computable con ellas.

1.4. Justificación

Los ACs son modelos sencillos, pero muchas de sus reglas pueden llegar a generar comportamientos altamente complejos. Ha sido posible demostrar universalidad en ellas, así que su potencial como un modelo de computación no convencional, es evidente. Este trabajo constituye una propuesta enfocada en encontrar una nueva manera de hacer cálculos computacionales mediante las colisiones entre partículas que se generan.

Así mismo, existen sistemas para visualizar los ACs como anillos, pero al momento no hay programa alguno para visualizar las colisiones en su proceso completo, capaz de construir el modelo de simulación a partir de configuraciones dadas y de sincronizar los anillos para realizar colisiones relevantes. Los resultados de esta investigación aportarán bases para nuevos marcos de computación alternativa, mientras que el simulador a desarrollar servirá como un medio para poder visualizar y comprender estos resultados.

Puesto que es un proyecto de carácter académico y de investigación, el trabajo tiene como sector de interés la comunidad científica, en concreto, los científicos de computación e investigadores en el área de ACs y computación no convencional.

1.5. Productos o resultados esperados

Los resultados a entregar al final del TT-II son:

- Un simulador de colisiones de partículas en ACEs.
- Los resultados de investigación sobre la implementación de un sistema computable utilizando el simulador.

1.6. Estado del arte

Los autómatas celulares son constantemente analizados por diferentes trabajos científicos, con los objetivos de probar su universalidad, conocer el comportamiento de su estructura y buscar potenciales aplicaciones. Entre ellos:

- 2008-0040 – Computación basada en reacción de partículas en un autómata celular hexagonal [7], analiza un tipo particular de ACs: los ACs hexagonales, y realiza un compendio de los gliders que componen a una de sus posibles reglas, así como una observación de su comportamiento en colisiones y una simulación de compuertas básicas.
- 2016-B044 – Máquinas de Turing y el problema de la universalidad como sistemas dinámicos discretos [8], desarrolla una interfaz gráfica para la visualización de máquinas de Turing mediante su representación formal, analiza el comportamiento de algunas de estas máquinas y ofrece una taxonomía estadística inicial.
- 2017-B077 – Simulador de operaciones lógicas desde un autómata celular con comportamiento caótico a su proyección compleja [9], realiza un análisis de otra regla de AC de clase IV, la 126 con memoria de 4 generaciones, en el que enlista las partículas que se forman en su estructura, sus colisiones y una simulación de compuertas básicas, y desarrolla un simulador de operaciones lógicas para la visualización de estas interacciones.

Capítulo 2

Marco teórico

2.1. Teoría de la computación

2.1.1. Gramáticas y la jerarquía de Chomsky

La teoría de la computación es una rama de las ciencias de la computación y las matemáticas, que analiza qué problemas pueden ser resueltos por modelos de computación usando series finitas de instrucciones (algoritmos), qué tan eficiente puede ser su resolución, y qué tan certeramente (mediante soluciones aproximadas). Este campo está dividido en tres principales áreas: teoría de autómatas y gramáticas formales, teoría computacional, y teoría de la complejidad, las cuales buscan responder a la pregunta: *¿Cuáles son las capacidades y limitaciones fundamentales de las computadoras?*. [10]

Uno de los temas más estudiados de este campo es el de las gramáticas formales. Una gramática formal es un concepto que describe cómo crear cadenas de caracteres a partir de un conjunto finito, no vacío, de símbolos, llamado alfabeto. Una gramática está compuesta por un conjunto finito de reglas de producción de la forma $w \to v$, en donde w y v consisten de una secuencia finita de los siguientes símbolos:

- Un conjunto finito de símbolos no terminales, que indican que aún es posible aplicarse alguna regla de producción.
- Un conjunto finito de símbolos terminales, que indican que ya no es posible aplicar reglas de producción.

Una gramática formal genera un lenguaje formal: un conjunto, usualmente infinito, de secuencias finitas (palabras) de símbolos, que pueden ser construidas mediante la aplicación de las reglas de producción a otra palabra (que inicialmente contiene sólo un símbolo inicial). Una regla de producción se aplica reemplazando una ocurrencia de los símbolos de w con su secuencia en v. Así pues, una gramática puede definir un lenguaje como todas las palabras de símbolos terminales que pueden derivarse aplicando reglas de producción desde el símbolo inicial.

Por ejemplo, supóngase el conjunto de símbolos terminales $\{a, b\}$, el conjunto con un único símbolo no terminal $\{S\}$, siendo S el símbolo inicial, y las siguientes reglas de producción:

$$S \to aSb$$
$$S \to \epsilon$$

Donde ϵ es la cadena vacía. A partir de S, esta gramática define el lenguaje de todas las palabras de la forma $a^n b^n$, es decir, n copias de a seguidas por n copias de b, con $n \in \mathbb{Z}_0^+$.

Noam Chomsky, en 1956, propuso una jerarquía que clasifica estas gramáticas en 4 categorías: [11]

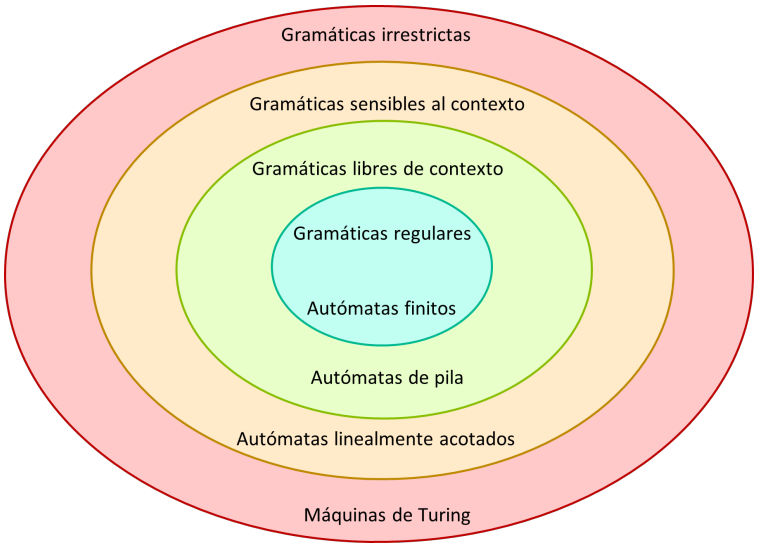

Figura 2.1: Jerarquía de Chomsky para las gramáticas formales.

1. Tipo 3 - Gramáticas regulares: Son definidas por las reglas de las formas $A \to aB$ (regular por la derecha) o $A \to Ba$ (regular por la izquierda), con A, B no terminales y a terminal. Sus lenguajes son reconocidos por los autómatas finitos y pueden ser expresados mediante expresiones regulares, elementos utilizados comúnmente para patrones de búsqueda y estructura léxica.

2. Tipo 2 - Gramáticas libres de contexto: Son definidas por reglas de la forma $A \to \alpha$, con A un no terminal y α una palabra de terminales y/o no terminales. Sus lenguajes son reconocidos por los autómatas de pila y son la base teórica de la mayoría de los lenguajes de programación.

3. Tipo 1 - Gramáticas sensibles al contexto: Son definidas por las reglas de la forma $\alpha A\beta \to \alpha\gamma\beta$, con A un no terminal y α, β y γ palabras de terminales y/o no terminales. Sus lenguajes pueden ser reconocidos por los autómatas linealmente acotados.

4. Tipo 0 - Gramáticas irrestrictas o recursivamente enumerables: No se tiene ninguna restricción en sus reglas. Sus lenguajes pueden ser reconocidos por las máquinas de Turing.

La figura 2.1 muestra la relación entre estas 4 categorías.

2.1.2. Máquina de Turing

Las máquinas de Turing, descritas por primera vez por Alan Turing en 1936, son dispositivos computacionales abstractos simples, usados para investigar el alcance y las limitaciones de lo que puede ser computable, y son considerados actualmente como uno de los modelos fundadores de la teoría de computabilidad y la ciencia de la computación.

Una máquina de Turing es una máquina con un conjunto finito de configuraciones $\{q_1, \ldots, q_n\}$ (sus estados), comenzando en un estado inicial q_0. Ésta usa una cinta, teóricamente infinita, dividida en celdas, cada uno capaz de albergar exactamente un símbolo. En cada momento, el cabezal de la máquina lee el contenido de una celda, ya sea vacío (S_0), o con un símbolo de un conjunto finito $\{S_1, \ldots, S_m\}$. Usando el símbolo obtenido y su estado actual, la máquina realiza tres acciones:

1. Cambiar su estado q_i a un siguiente estado q_j.

2. Imprimir un nuevo símbolo (o el símbolo vacío) en la celda actual.

3. Mover el cabezal una celda hacia la izquierda o hacia la derecha.

Estas acciones serán decididas mediante un conjunto de reglas, llamado función de transición, en el que la máquina sólo se detendrá cuando llegue a uno de sus posibles estados finales. [12]

Formalmente, una máquina de Turing es una 7-tupla

$$M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$$

, donde:

- Q es un conjunto finito, no vacío, de estados,
- Γ es un alfabeto de símbolos,
- $b \in \Gamma$ es el símbolo vacío,
- $\Sigma \subseteq \Gamma \backslash \{b\}$ es el conjunto de símbolos de entrada, es decir, los símbolos que pueden aparecer inicialmente en la cinta,
- $q_0 \in Q$ es el estado inicial,
- $F \subseteq Q$ es el conjunto de estados finales. Se dice que el contenido inicial de la cinta es aceptado por M si ésta eventualmente se detiene al llegar a un estado de F;
- $\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ es la función de transición, que decide si la máquina se moverá hacia la izquierda (L) o hacia la derecha (R). [13]

2.1.3. Computación y la tesis de Church-Turing

La tesis de Church-Turing es una hipótesis sobre la naturaleza de las funciones, formulada en los años 30s por Kurt Gödel, Alonzo Church y Alan Turing. La tesis analiza el concepto de métodos efectivos en la lógica y matemáticas, en donde se dice que un método o procedimiento M es efectivo si y sólo si:

1. M es definido en términos de un número finito de instrucciones exactas, cada una de estas instrucciones expresadas por medio de una cantidad finita de símbolos.
2. De ejecutarse sin errores, M produce un resultado deseado en una serie finita de pasos.
3. M puede (en práctica o en principio) ser ejecutado por un humano sin ayuda mecánica.
4. M no requiere percepción, intuición o ingenio por parte del humano que lo ejecuta. [14]

Un método efectivo para calcular los valores de una función es llamado algoritmo. Equivalentemente, una función para la que existe un método efectivo es denominada efectivamente calculable.

Supóngase entonces el conjunto S de todas las funciones efectivamente calculables. La tesis de Church-Turing dicta que cualquier función $f \in S$ puede ser implemenentada en una máquina de Turing, codificando las entradas y salidas del programa en forma de caracteres en la cinta de la máquina. Dicho de otra manera, es posible definir a un algoritmo como un método implementable en una máquina de Turing.

Completez, equivalencia y universalidad

Un modelo computacional es un modelo que describe cómo se procesa la salida de una función con respecto a una entrada dada. Éste describe la organización de sus unidades de computación, memoria y comunicación.

Sean S y R dos modelos computacionales. Decimos que S es completo con respecto a R si y sólo si S es capaz de simular la estructura y comportamiento de R. Adicionalmente, decimos que S y R son equivalmentes si y sólo si S es completo con respecto a R y R es completo con respecto a S.

La propiedad de completez es particularmente útil en el ámbito de las máquinas de Turing. Supóngase un modelo computacional S. Si S es completo con respecto a la máquina de Turing (o Turing-completo) entonces S puede simular cualquier máquina de Turing. De esta manera, S es capaz de hacer todo lo que una

máquina de Turing hace. Concretamente, por la tesis de Church-Turing, S puede también ejecutar cualquier algoritmo. [15] Decimos entonces que S es un modelo o sistema computacionalmente universal.

Para demostrar que un modelo matemático S es universal, basta con mostrar que, para cualquier programa en una máquina de Turing, existe una forma de realizar ese mismo programa en S. El método más común para lograrlo es presentando un compilador que tome cualquier programa y datos de una máquina de Turing y la transforme en un programa y datos de S. [3]

2.1.4. Sistemas complejos

El concepto de complejidad nació como un paradigma post-Newtoniano para unificar una serie de fenómenos que sucedían en sistemas constituidos por varias subunidades, identificados en áreas tan diversas como la física, la ingeniería, las ciencias ambientales o ciencias sociales. Por un largo tiempo prevaleció la idea de que la percepción de sistemas de este tipo emergía de una falta de información, en conexión con la presencia de un número enorme de variables y parametros que ocultaban posibles regularidades. Sin embargo, con el paso de los años, se fue descubriendo que esta complejidad está profundamente arraigada en las leyes fundamentales de la física. Este hecho abrió camino a un estudio sistemático de la complejidad, la cual hoy constituye una rama de la ciencia altamente interdisciplinaria y en rápido crecimiento, valiéndose de conceptos y herramientas de dinámica no lineal, teoría de la información, análisis de datos y simulación numérica.

Un sistema complejo induce una fenomenología característica, cuya principal cualidad es la multiplicidad de posibles resultaados. Este proceso puede manifestarse de diferentes maneras:

- El surgimiento de rasgos que abarcan al sistema como un todo, el cual no puede ser reducido de ninguna forma a las propiedades de las partes que lo constituyen. Las propiedades emergentes reflejan el rol primordial de la interacción entre sus partes, y se manifiestan mediante la creación de estados auto-organizados de forma modular y jerárquica. Entre los ejemplos clásicos de este comportamiento están los fluidos sometidos a estrés y las reacciones químicas, hasta los sistemas biológicos y genéticos de los seres vivos.

- El entrelazamiento, dentro de un mismo fenómeno, de regularidades de larga escala y tendencias evolucionarias aparentemente erráticas. Esta coexistencia de orden y desorden pone en cuestión la predictibilidad de evoluciones futuras del sistema. Ejemplos típicos aparecen en los fenómenos climáticos y geológicos en relación con la dificultad de producir predicciones precisas, así como sistemas artificiales como el mercado de valores.

La naturaleza diversificada de estos sistemas permite su estudio mediante un enfoque multinivel:

- *Distribuciones probabilísticas.* Gracias a su inherente linearidad y estabilidad, las distribuciones probabilísticas pueden ser usadas para realizar predicciones confiables en la ocurrencia de eventos futuros condicionados por los estados predominantes en un tiempo determinado. Algunos objetivos importantes en este análisis son la predicción de eventos extremos, la recurrencia de estados de un cierto tipo y la aparición de umbrales.

- *Entropía y dimensiones generalizadas.* Un proceso probabilístico puede caracterizarse por una jerarquía de cantidades cuasi-entrópicas que describen la cantidad de datos necesarios para identificar un estado particular del sistema o una secuencia de estados con una resolución específica. Las cantidades cuasi-entrópicas geberab también una jerarquía de cantidades cuasi-dimensionales, generalmente como fractales, las cuales proveen una caracterización geométrica útil en el estudio de la complejidad.

- *Correlaciones.* Una caracterización alterna es en términos de correlaciones: conjuntos de cantidades que describen cómo un sistema mantiene, en espacio y tiempo, la memoria de una perturbación inflingida inicialmente en una de sus partes.

- *Simulaciones.* La simulación directa de un proceso de interés es un elemento indispensable en el estudio de los sistemas complejos. Comenzando con una cantidad mínima de información esencial, es exploran diferentes escenarios compatibles con esta información. Aspactos genéricos de comportamientos

complejos son analizados de esta manera mediante modelos gobernados por reglas locales simples. En su implementación computacional, estos modelos presentan visualizaciones atractivas y permiten la adquisición de información a más profundidad. [16]

2.2. Sistemas tag y sistemas tag cíclicos

2.2.1. Sistema tag

El sistema tag es un tipo de sistema universal, estudiado originalmente por Emil Post en 1920. Es una máquina que opera en una cinta finita, leyendo símbolos del frente de la cinta, y escribiendo símbolos al final de ésta. En cada paso, la máquina remueve una cantidad fija de símbolos, y basándose solamente en el primero de éstos, decide qué debe escribir al final, de acuerdo a una tabla que la define. [17]

Formalmente, un sistema m-tag es una 3-tupla

$$T = (m, \Sigma, P)$$

, donde:

- $m \in \mathbb{Z}^+$ es el número de borrado;
- Σ es un alfabeto de símbolos. Todas las combinaciones $w \in \Sigma^*$ son llamadas palabras;
- $P : \Sigma \to \Sigma^*$ es una función o regla de producción, definida para cada símbolo $a \in \Sigma$.

Una palabra se denomina de parado si su longitud es menor que m. Adicionalmente, puede existir un símbolo especial de parado $H \in \Sigma$, tal que una palabra se considere de parado, también, si comienza con H.

Una transformación t, llamada operación tag, se define en el conjunto de palabras de NO parado, de tal forma que, si a es el primer símbolo de una palabra w, entonces $t(w)$ es el resultado de borrar los primeros m símbolos de w y añadir la palabra $P(a)$ al final.

Una computación en un sistema m-tag es una secuencia finita de palabras producida iterando la transformación t, comenzando con una palabra inicial dada y terminando cuando se produce una palabra de parado. [18]

Por ejemplo, sea un sistema 2-tag con un alfabeto $\{a, b, c, H\}$, H como símbolo de parado, y las siguientes reglas de producción:

$$a \to aa$$
$$b \to aacbH$$
$$c \to aab$$

La figura 2.2 muestra una computación de este sistema sobre la palabra inicial *cbb*.

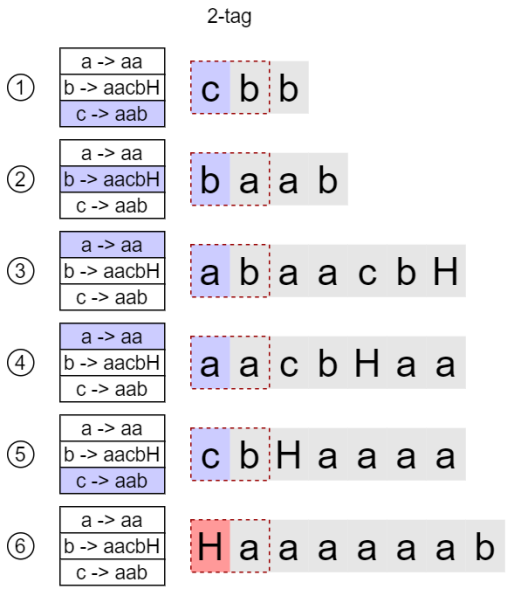

Figura 2.2: Computación en un sistema 2-tag.

Se ha demostrado que los sistemas tag son universales, al ser capaces de simular cualquier máquina de Turing. La demostración completa puede encontrarse en [18], [3] y [19].

Una de las aplicaciones más conocidas de los sistemas tag es la computación de las secuencias de Collatz. En la secuencia de Collatz original, el sucesor de n es $\frac{n}{2}$ (para n par), o $3n+1$ (para n impar). Como $3n+1$ es claramente par para un n impar, podemos inferir que su sucesor siempre será $\frac{3n+1}{2}$. Nos enfocaremos en éste último para las n impares.

El sistema 2-tag que procesará esta secuencia usa el alfabeto $\{a, b, c\}$ con las siguientes reglas de producción:

$$a \to bc$$
$$b \to a$$
$$c \to aaa$$

En este sistema, n se representa mediante una palabra formada por n a's. Por ejemplo, la figura 2.3 muestra la ejecución del sistema comenzando por la palabra inicial aaa, que representa el número 3. [20]

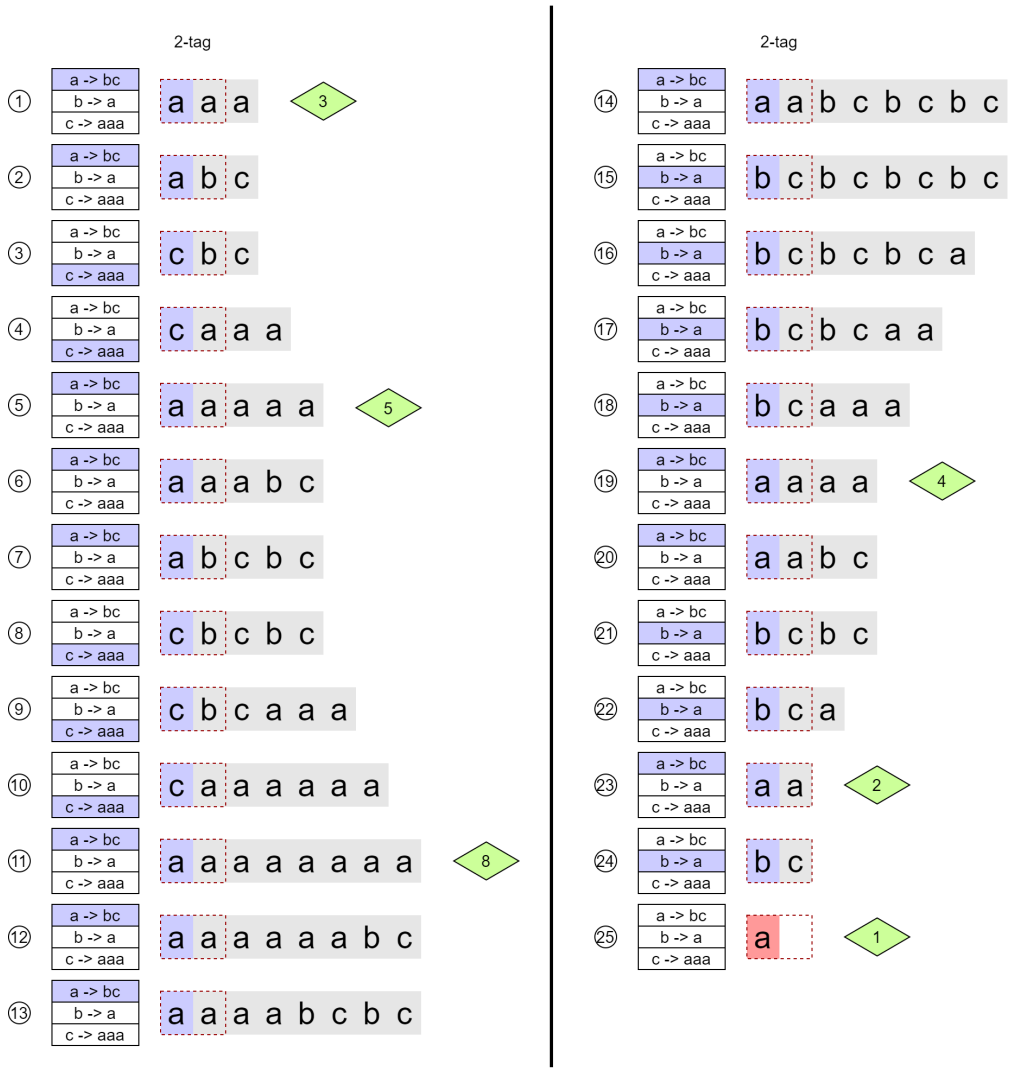

Figura 2.3: Secuencia de Collatz para $n = 3$ funcionando en un sistema 2-tag.

2.2.2. Sistema tag cíclico

Un sistema tag cíclico (CTS, por sus siglas en inglés), es un modelo computacional creado por Matthew Cook para demostrar que el autómata celular elemental con la regla 110 es universal. Un CTS es una modificación de un sistema 1-tag. Su alfabeto consiste de los símbolos 0 y 1, y las reglas de producción se componen como una lista ordenada de producciones elegidas secuencialmente, regresando al principio de la lista después de considerar la última. Comenzando por una palabra inicial, se remueve el primer caracter de la palabra. Si éste es 1, la producción actual se agrega al final de la palabra. De lo contrario, no se agrega nada. Al terminar, se considera la siguiente producción en la lista. El sistema para si y sólo si la palabra se vuelve vacía. [21]

Formalmente, definimos a un CTS como la tupla:

$$C = (k, (p_0, \ldots, p_{k-1}))$$

Donde $k \in \mathbb{Z}^+$ es la longitud del ciclo, y $(p_0, \ldots, p_{k-1}) \in (\{0,1\}^*)^k$ es una k-tupla de producciones. [22]

Definimos a una descripción instantánea (ID) de C como una tupla (v, m), donde $v \in 0, 1^*$ y $m \in \{0, \ldots, k-1\}$. Llamamos a m una fase de la ID.

Luego, definimos a una relación de transición $\Rightarrow$ en el conjunto de las IDs de la siguiente forma: Sean $(v, m), (v', m') \in \{0,1\}^* \times \{0, \ldots, k-1\}$. Entonces:

$$(1v, m) \Rightarrow (v', m') \text{ iff } (m' = m + 1 \bmod k) \wedge (v' = vp_m),$$
$$(0v, m) \Rightarrow (v', m') \text{ iff } (m' = m + 1 \bmod k) \wedge (v' = v)$$

Una secuencia de IDs $(v_0, m_0), \ldots, (v_n, m_n)$ de C es una computación comenzando en $v \in \{0,1\}^*$ si y sólo si:

$$(v_0, m_0) = (v, 0) \wedge (v_i, m_i) \Rightarrow (v_{i+1}, m_{i+1}) \ \forall i \in \{0, \ldots, n-1\}$$

[23]

Se dice que un CTS se detiene en la cinta v si su computación transiciona a la palabra vacía ϵ en una cantidad finita de transiciones.

La figura 2.4 muestra las 100 primeras transiciones de la computación de un CTS con producciones $(1, 10, 110)$, comenzando con la cinta inicial 01100. El símbolo 1 se representa con el color gris, mientras que 0 se representa con el color blanco. Si bien sólo se presenta el principio de la computación, es difícil predecir si en algún momento en el futuro el CTS se detendrá.

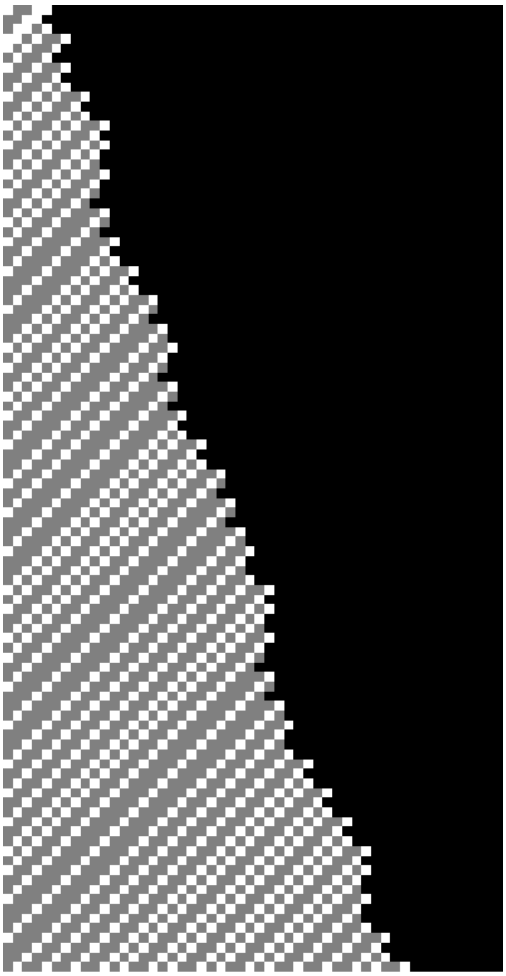

Figura 2.4: Ejemplo de la computación de un sistema tag cíclico con producciones $(1, 10, 110)$ comenzando en la cinta 01100.

Kenichi Morita [24] define una condición de paro extra, determinando a p_0 como una producción de paro. Con esto, decimos que una ID de C es una ID de paro si y sólo si tiene la forma $(1v, 0), v \in \{0,1\}^*$. De

igual manera, decimos que una computación de C comenzando en v es una computación completa si y sólo si (v_n, m_n) es una ID de paro. En otras palabras, si en alguna ID de C, el símbolo al principio de la cinta es 1 y la fase del ID es 0, el sistema se detiene. Es importante notar que la primera producción del CTS, p_0, nunca se va a agregar a la cinta, ya que eso implicaría un paro en el sistema, por lo que esta producción no necesita una definición específica.

La figura 2.5 muestra la computación del CTS del ejemplo anterior, utilizando ahora la condición de paro de Morita. Después de 15 transiciones, el sistema produce la ID de paro $(1101011010, 0)$, con lo que el sistema se detiene.

Figura 2.5: Ejemplo de la computación de un sistema tag cíclico con producciones $(p_0, 10, 110)$ comenzando en la cinta 01100 con condición de paro en p_0.

Es posible simular cualquier sistema m-tag mediante un CTS, por lo que los CTS son sistemas completos con respecto a los sistemas m-tag. Ya que los sistemas m-tag son universales, por la tesis de Church-Turing, los CTS también son universales y por tanto capaces de ejecutar cualquier algoritmo.

La demostración de la completez de los CTS es la siguiente: Sea S un sistema m-tag de $k = |\Sigma|$ símbolos, construimos un CTS de la siguiente forma:

Supóngase una lista ordenada con todos los símbolos de Σ. El i-ésimo símbolo de esta lista se representa por una palabra de longitud k, en la cual el i-ésimo caracter es 1 y el resto son 0. Con esta codificación, es posible traducir la palabra inicial de S y sus producciones. Las k producciones de S se ordenan de acuerdo a la lista de Σ. A esta lista se añaden, además, $(m-1)k$ producciones ϵ, de longitud 0. La lista final de mk producciones define el CTS deseado. [3] [6]

Por ejemplo, sea S un sistema 2-tag con el alfabeto $\Sigma = \{a, b, c\}$ y las reglas de producción $a \to ba,\ b \to cc,\ c \to b$, a procesar la palabra inicial $aabc$. Podemos construir un CTS que simule a S de la siguiente forma:

1. Traducimos los símbolos ordenados de Σ mediante cadenas de longitud $k = 3$: sustituimos a por 100, b por 010 y c por 001.

2. Usando la traducción de Σ, traducimos las reglas de producción ordenadas de la misma forma: ba por 010100, cc por 001001 y b por 010.

3. A la lista de producciones traducida, agregamos $(m-1)k = (2-1)3 = 3$ producciones que generen ϵ. Por lo que la lista de producciones final es $(010100, 001001, 010, \epsilon, \epsilon, \epsilon)$.

4. Traducimos la palabra inicial $aabc$ por 100100010001.

La figura 2.6 muestra la ejecución del CTS que simula a S.

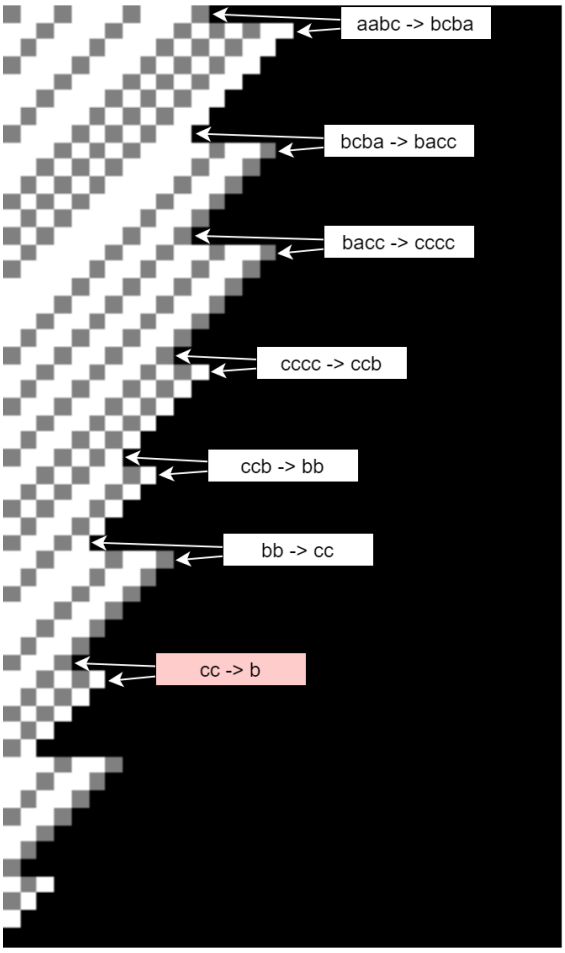

Figura 2.6: Ejemplo de la simulación de un sistema 2-tag mediante un CTS.

De igual manera, es posible recrear la secuencia de Collatz en un CTS, construyendo adecuadamente el sistema 2-tag descrito anteriormente:

1. Traducimos los símbolos ordenados de Σ: a por 100, b por 010 y c por 001.
2. Traducimos las reglas de producción ordenadas: bc por 010001, a por 100 y aaa por 100100100.
3. La lista de producciones final es $(010001, 100, 100100100, \epsilon, \epsilon, \epsilon)$.
4. Traducimos la palabra inicial aaa por 100100100.

La figura 2.7 muestra la ejecución del CTS que genera la secuencia de Collatz.

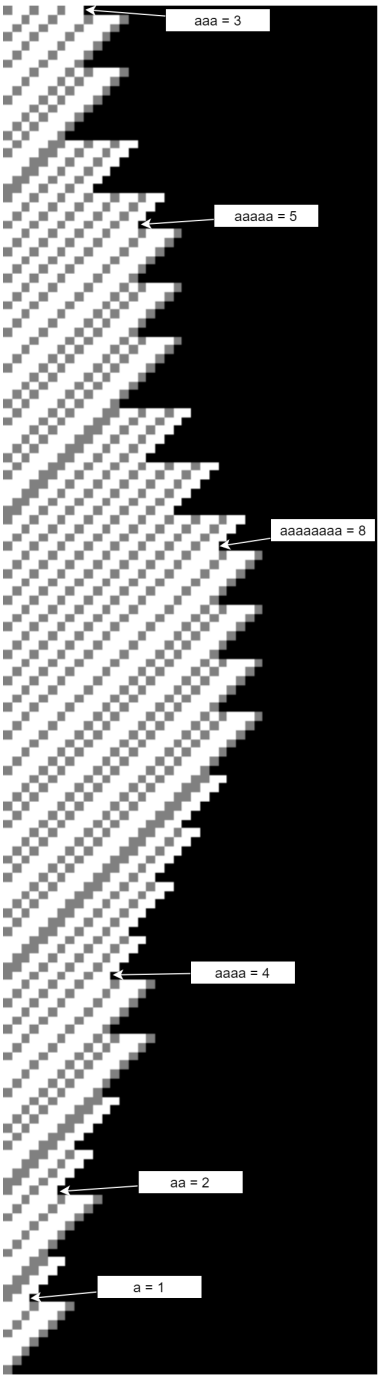

Figura 2.7: Generación de la secuencia de Collatz mediante un CTS.

La universalidad de los CTS es demostrable por un método distinto. Turlough Neary y Damien Woods [25] probaron que el modelo de máquina de Turing en sentido de manecillas de reloj (CTM) puede simular una máquina de Turing, por lo que el conjunto de las CTM es Turing-completo y por consiguiente universal. Luego, implementaron una CTM en un CTS, con lo que demostraron que el conjunto de los CTSs es completo con respecto a los CTM y por tanto también universal.

Kenichi Morita [23] [24] utilizó la universalidad de los CTSs para demostrar la universalidad del modelo de autómata celular reversible unidimensional (RCA), presentando un traductor de CTSs que prueba la completez de los RCAs con respecto a éstos.

2.3. Autómatas celulares

2.3.1. Definición

El autómata celular (AC) es un modelo matemático simple, propuesto en los años 50s por Stanislaw Ulam y John von Neumann en la búsqueda de crear una máquina autorreplicante. [26]

Un AC está compuesto por un arreglo d-dimensional, teóricamente infinito, en el que cada espacio (celda o célula) contiene un símbolo de un conjunto finito de estados, usualmente representados por valores enteros. El patrón de valores sobre el arreglo entero es el estado global del AC en un determinado tiempo. La evolución del sistema es definida por una regla de transición, la cual se aplica simultáneamente a cada una de las celdas. En un tiempo t_i, cada celda cambia su estado dependiendo del estado de sus celdas vecinas en el tiempo t_{i-1}, de acuerdo a la regla. [27]

Formalmente, un AC se define como la 4-tupla:

$$(L, \Sigma, r, \phi)$$

Donde:

- L es un arreglo d-dimensional, teóricamente infinito, de celdas;
- Σ es un alfabeto de estados de cardinalidad k, comúnmente representado con los valores enteros $\{0, 1, \ldots, k-1\}$;
- $r \in \mathbb{Z}^+$ define el tamaño de una vecindad local. Para una celda i, r representa $2r+1$ celdas contiguas: r celdas a la izquierda de i y r celdas a su derecha, y la propia i;
- $\phi : \Sigma^r \to \Sigma$ es una función, llamada regla de transición, que determina el nuevo estado de una celda para un tiempo t_i a partir de los estados de las celdas en su vecindad en un tiempo t_{i-1}. Esta función se aplica de manera síncrona en todo L, generando un nuevo estado global del AC. [28] [27]

Para fines prácticos, L se supone lo suficientemente grande, y sus celdas de frontera se manejan de alguna de las siguientes formas:

1. Frontera periódica: las celdas de frontera comparten vecindad.
2. Frontera cerrada: las celdas de frontera tomarán siempre valores constantes. [2]

Se usará la frontera periódica para la representación de los ACs de este trabajo.

Por ejemplo, sea un AC con un arreglo L unidimensional ($d = 1$) de tamaño $N = 10$ (0-indexado), con $k = 2$ y $r = 1$. Los estados posibles para las celdas son 0 y 1, y la vecindad tiene un tamaño de 3, es decir, para cada celda i, la regla checará el valor de $L[i]$, el de $L[i-1]$ y el de $L[i+1]$.

Describamos la regla para este AC de la siguiente manera:

$$\phi(1,1,1) = 0;\ \phi(1,1,0) = 0;\ \phi(1,0,1) = 0;\ \phi(1,0,0) = 1;$$
$$\phi(0,1,1) = 1;\ \phi(0,1,0) = 1;\ \phi(0,0,1) = 1;\ \phi(0,0,0) = 0$$

La figura 2.8 muestra a ϕ, la cual enlista el comportamiento para todas las posibles combinaciones de 0s y 1s en una vecindad de 3 celdas. En ellas, las celdas blancas (o muertas) representan 0s, y las celdas negras (o vivas) representan 1s. Podemos observar que se tienen $r^k = 3^2 = 8$ posibles combinaciones.

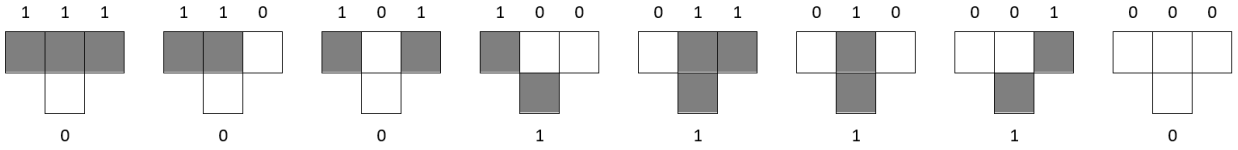

Figura 2.8: Descripción gráfica de las combinaciones de la regla elemental 30 para un AC unidimensional.

Podemos entonces suponer una condición inicial (t_0) para el estado global del AC: 1001100010. La evolución de este AC para t_1 se ilustra en la figura 2.9:

1. Se muestra el estado global del AC en t_0.
2. Para la celda $i = 1$, se tiene que $L[i-1] = L[0] = 1$, $L[i] = L[1] = 0$, y $L[i+1] = L[2] = 0$. Luego, $\phi(1, 0, 0) = 1$, por lo que el valor de $L[i] = L[1]$ para t_1 es 1.
3. Para la celda $i = 2$, se tiene que $L[i-1] = L[1] = 0$, $L[i] = L[2] = 0$, y $L[i+1] = L[3] = 1$. Luego, $\phi(0, 0, 1) = 1$, por lo que el valor de $L[i] = L[2]$ para t_1 es 1. Se continúa el procedimiento para todas las celdas de L.
4. Para las fronteras de L, se usa una condición periódica. En el ejemplo, para $i = 0$, se considera como vecina izquierda a la celda $N - 1 = 9$.
5. Finalmente, se muestra el estado global del AC en t_1.

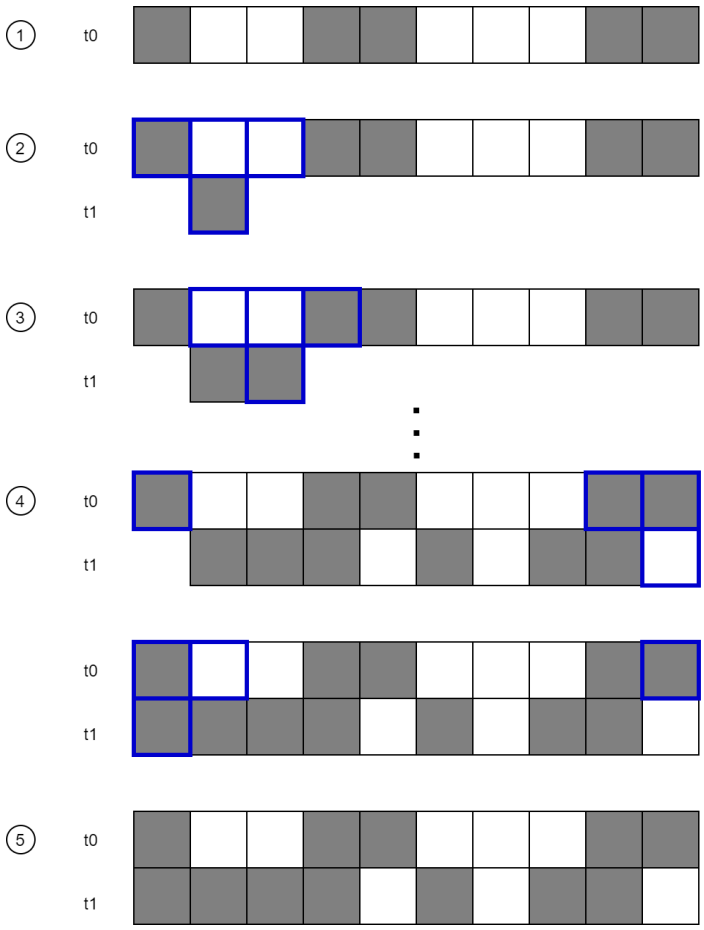

Figura 2.9: Pasos en la evolución del estado global del AC de ejemplo.

Este procedimiento puede repetirse aplicando ϕ las veces que sea, generando una malla con todos los estados globales que se van obteniendo, uno debajo del otro. La figura 2.10 muestra un espacio de 200 evoluciones de la regla ϕ para un AC con $N = 400$ y una condición inicial aleatoria.

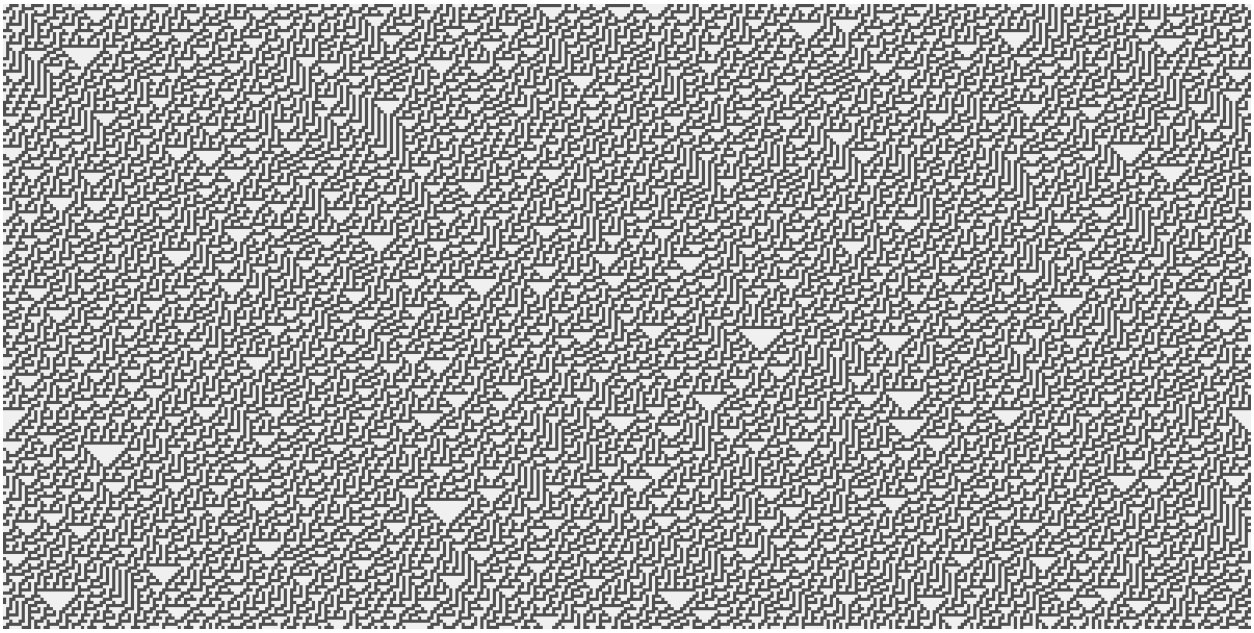

Figura 2.10: Ejemplo de evolución de un AC unidimensional con $N = 400$ usando la regla elemental 30.

Un AC unidimensional con condición de frontera periódica puede representarse también en forma de anillo, uniendo la primera celda de L con la última, y reproduciendo las evoluciones hacia abajo, creando un cilindro en el que es posible visualizar el comportamiento del AC sin las limitaciones de un espacio plano. Esto se ilustra en la figura 2.11 y será un punto importante en el desarrollo de este trabajo.

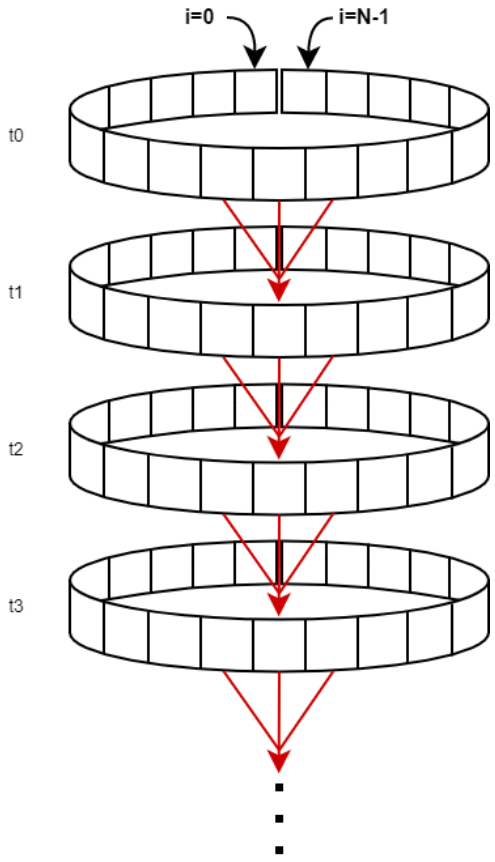

Figura 2.11: Representación en forma de anillo de un AC unidimensional. [2]

2.3.2. Autómata celular elemental

Un autómata celular elemental (ACE) es la clase más simple de AC: un AC unidimensional ($d = 1$) que utiliza valores binarios en las celdas ($k = 2$) y una vecindad de 3 celdas ($r = 1$). El ejemplo de AC unidimensional en la sección anterior también es un ejemplo de un ACE.

Podemos inferir que existen 8 posibles estados para la vecindad de un ACE y, por tanto, $2^8 = 256$ reglas elementales. Si las combinaciones de una regla están ordenadas por el valor binario de su vecindad, es posible codificar los estados resultantes de esta regla para darles un identificador único. Por ejemplo, de la figura 2.8, los estados ordenados son $0, 0, 0, 1, 1, 1, 1, 0$. De esta forma, tenemos la regla $(00011110)_b$, o bien, regla elemental 30.

La figura 2.12 muestra el comportamiento de algunas reglas que presentan patrones interesantes a partir de una condición inicial de una sola célula viva en el centro de su arreglo.

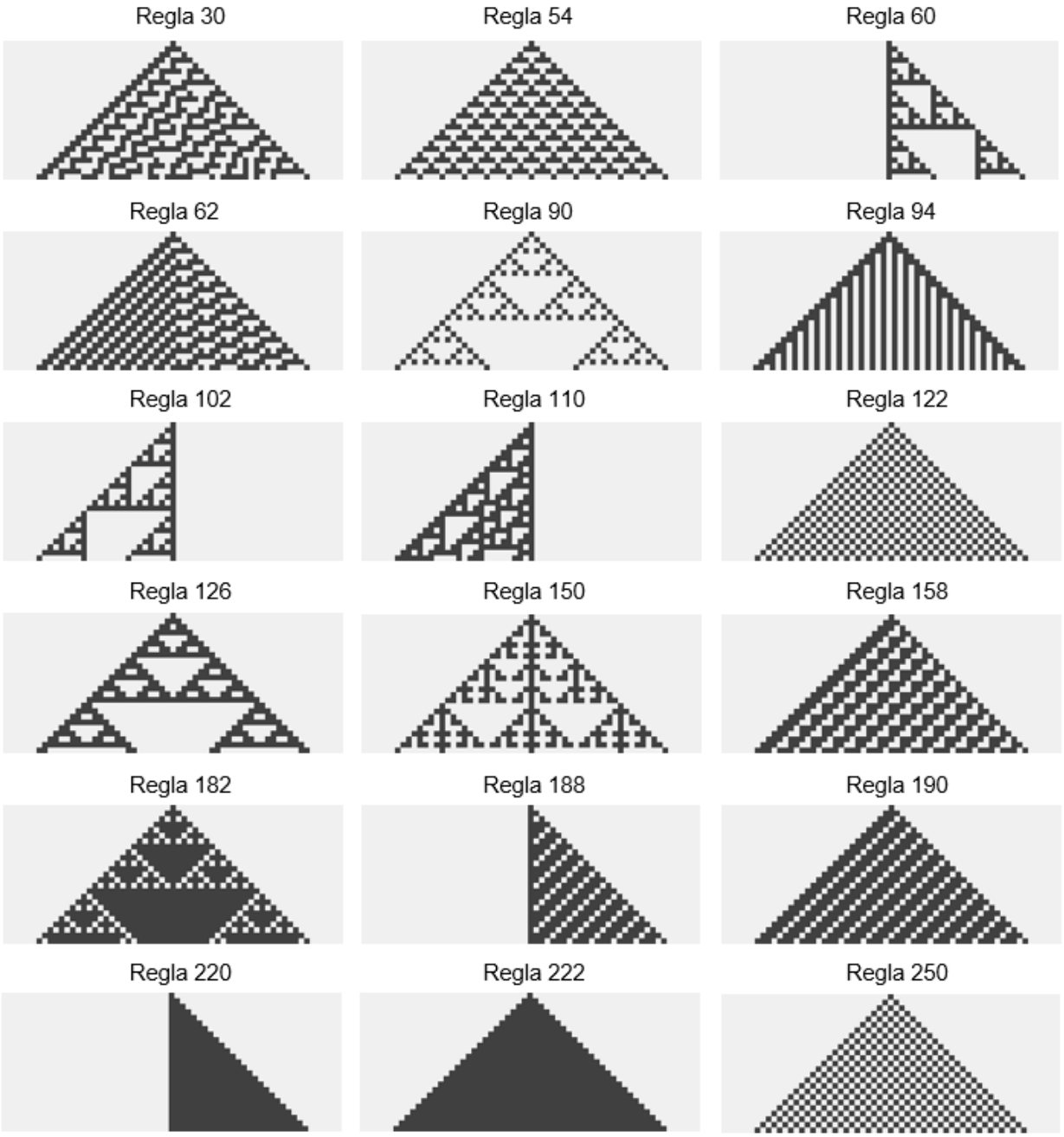

Figura 2.12: Ejemplos de reglas elementales con condición inicial $00\ldots010\ldots00$.

2.3.3. Clasificación de Wolfram

No todos los ACEs, y en general no todos los ACs producen comportamientos complejos. La dinámica de un AC surge a través del paso del tiempo, durante el transcurso de varias evoluciones. Por lo tanto, no es fácil analizar las propiedades de un AC desde su comienzo. Entre las reglas que puede tener un AC, existen algunas que producen un comportamiento simple y trivial, mientras que otras pueden presentar comportamientos complicados pero predecibles, hasta otras que pueden presentar la cualidad de sistema universal. [9]

Stephen Wolfram, el autor de la nomenclatura para las reglas elementales, estudió el comportamiento de los ACs y presentó una clasificación en 4 clases de acuerdo a la dinámica de sus evoluciones, independiente-

mente de sus condiciones iniciales: [1]

Clase I: Evolución a un estado uniforme

Después de transcurrido un número finito de generaciones, todas las celdas del AC convergen a un solo estado. La figura 2.13 muestra un ACE de clase I: la regla elemental 160.

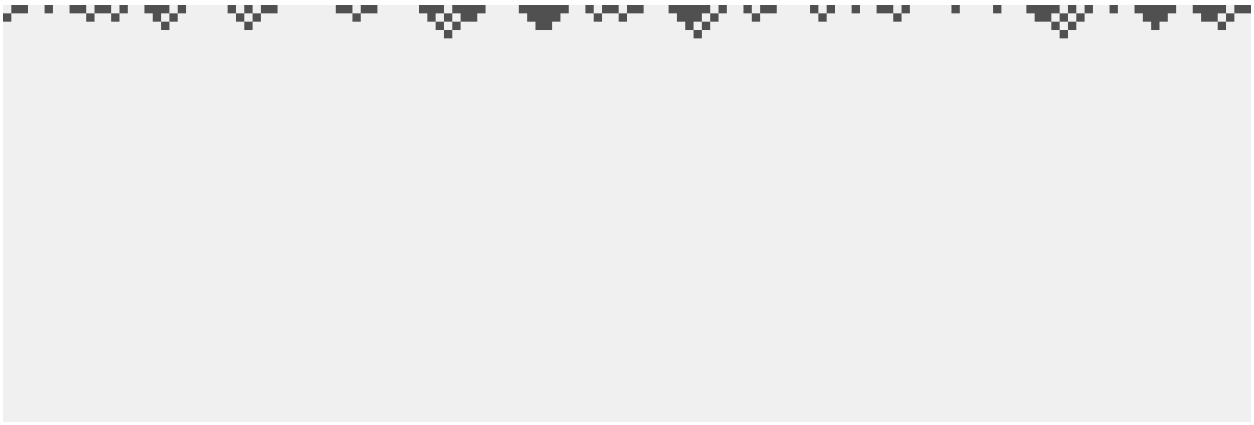

Figura 2.13: Regla elemental 160, un ejemplo de AC de clase I.

Clase II: Evolución a estados cíclicos aislados

Durante la evolución del AC, existen ciertos patrones de comportamiento que se repiten sistemáticamente. Estos patrones se pueden distinguir claramente sobre un fondo, representado por un solo estado, el cual es opuesto al de las células que representan el patrón de comportamiento cíclico. La figura 2.14 muestra un ACE de clase II: la regla elemental 108.

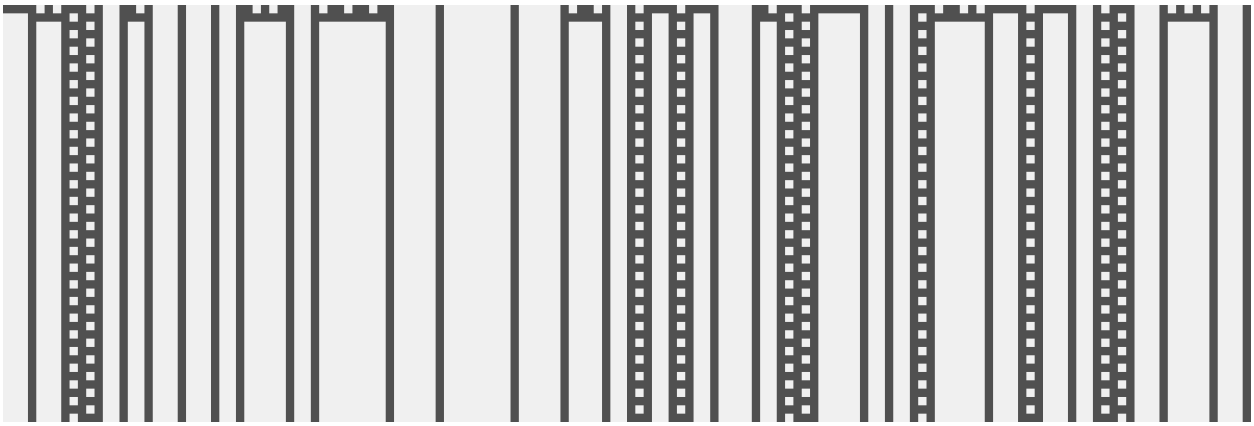

Figura 2.14: Regla elemental 108, un ejemplo de AC de clase II.

Clase III: Evolución a estados cíclicos amplios

Al igual que en la clase II, existen patrones de comportamiento repetitivos, aunque no son tan fácilmente identificables, ya que el comportamiento de éstos puede llegar a ser caótico. Por tanto, su análisis es bastante más complicado. La figura 2.15 muestra un ACE de clase III: la regla elemental 126.

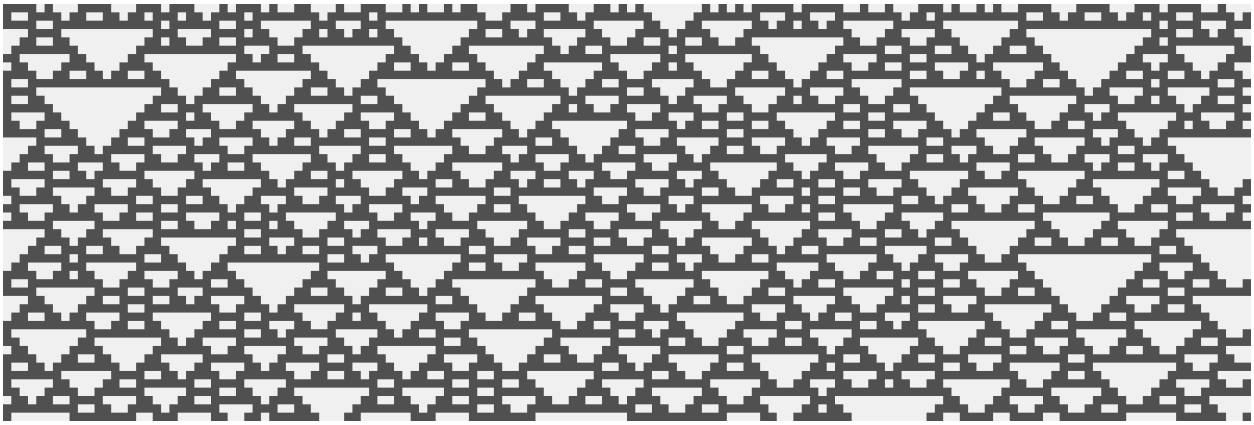

Figura 2.15: Regla elemental 126, un ejemplo de AC de clase III.

Clase IV: Evolución a estados complejos aislados

Esta clase es una combinación de las clases I, II y III. Presenta comportamientos cíclicos aislados como la clase II, lo que lo hace distinguible en un fondo uniforme. Sin embargo, los patrones que se presentan tienen una naturaleza caótica, similar a la clase III. Se ha demostrado que algunos de los ACs de esta clase tienen la capacidad de realizar computación universal, pero aún no se sabe con certeza si todos los ACs de clase IV comparten esta cualidad. La figura 2.16 muestra un ACE de clase IV: la regla elemental 110.

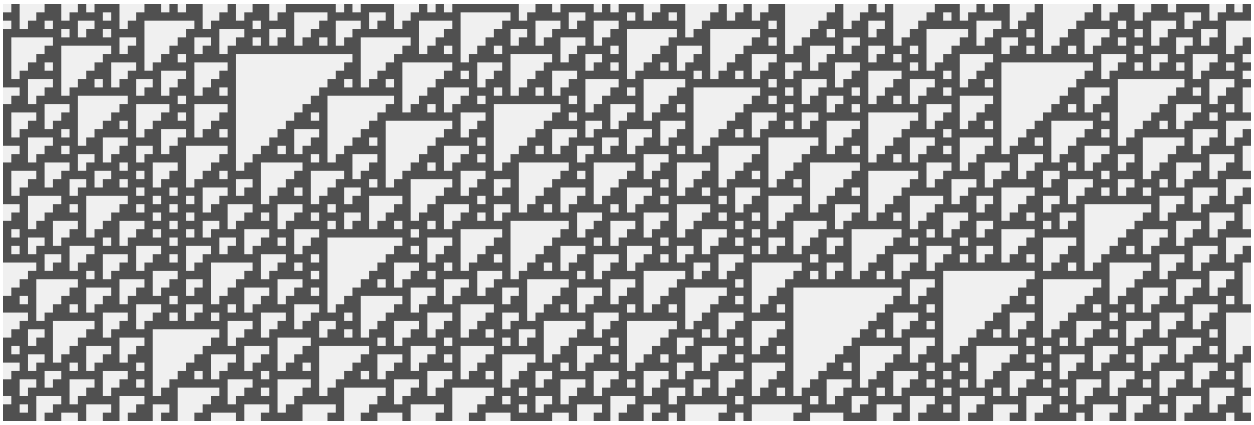

Figura 2.16: Regla elemental 110, un ejemplo de AC de clase IV.

2.3.4. El Juego de la Vida

Uno de los ACs de clase IV más famosos es el Juego de la Vida, creado por John Conway en 1970.

El Juego de la Vida es un AC bidimensional con dos estados, vivo y muerto. La vecindad de cada celda se compone de las 8 celdas que la rodean, llamada vecindad de Moore. La función de transición del sistema se define coloquialmente de la siguiente forma:

- Si una célula viva tiene 2 o 3 vecinos vivos, permanece viva. De otra manera muere.
- Si una célula muerta tiene exactamente 3 vecinos vivos, ésta vuelve a la vida. [29]

Haciendo uso de estas simples reglas, es posible crear patrones complejos a partir de condiciones iniciales específicas. El patrón más conocido es el *Glider*, cuyo comportamiento se muestra en las evoluciones de la figura 2.17. [30]

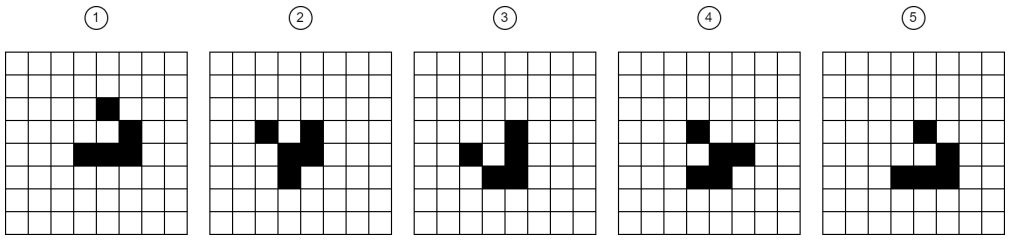

Figura 2.17: Evoluciones de un Glider en el Juego de la Vida.

Una demostración de la universalidad del Juego de la Vida se encuentra en [31]. En ella, se presenta la construcción de una malla que emula una máquina de Turing, probando que el Juego es Turing-completo y, por la tesis de Church-Turing, capaz de ejecutar cualquier algoritmo. La figura 2.18 muestra el espacio inicial para esta simulación.

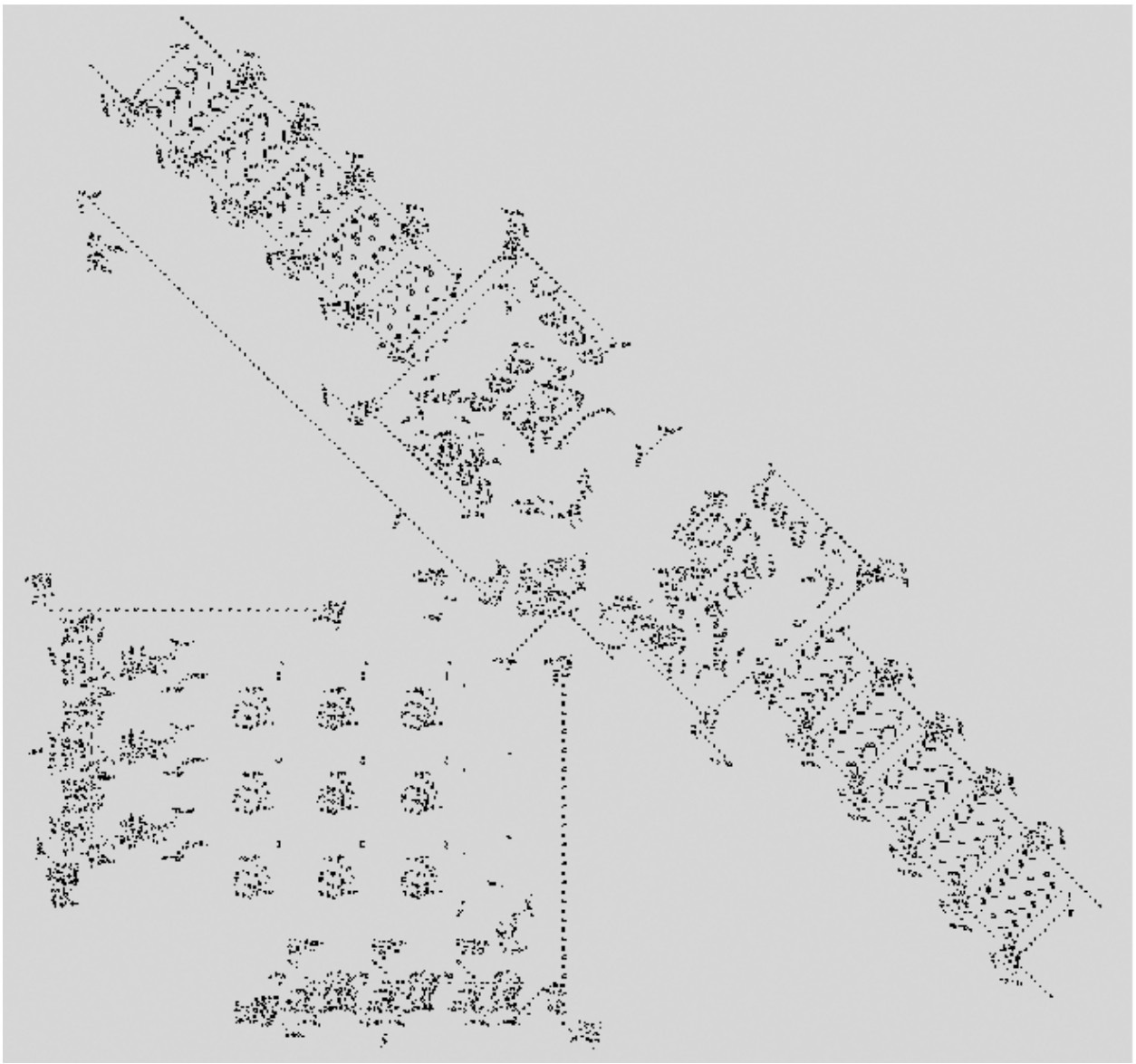

Figura 2.18: Simulación de una máquina de Turing en el Juego de la Vida. [31]

2.4. Regla 110

2.4.1. Análisis de la regla 110

Este trabajo se enfocará en el estudio de una regla elemental específica: la regla 110, cuya representación gráfica se muestra en la figura 2.19. Fue estudiada originalmente por Stephen Wolfram en 1983, quien la clasificó como una regla de clase IV por su comportamiento caótico. Wolfram conjeturó que tenía el potencial de ser un sistema universal, pero no fue sino hasta 1998 que Matthew Cook publicó un artículo que lo demostraba. La simpleza de la regla 110 la convierte en una poderosa herramienta para demostrar la universalidad de otros modelos matemáticos. La regla 110 puede resultar ser más sencilla de emular por otros sistemas que, por ejemplo, una máquina de Turing. [3]

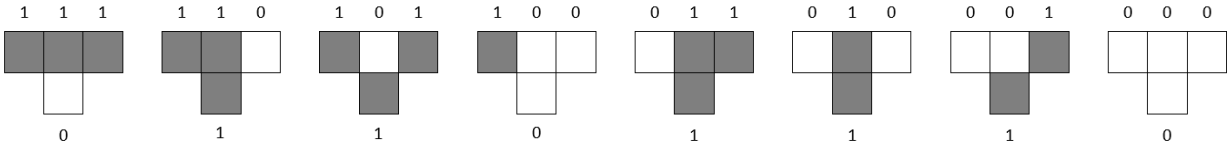

Figura 2.19: Descripción gráfica de las combinaciones de la regla elemental 110.

2.4.2. Partículas

Cook analizó el espacio de evoluciones de la regla 110 a partir de condiciones iniciales aleatorias, y descubrió que, conforme pasa el tiempo, las celdas dejan de tener un comportamiento totalmente aleatorio, y es posible identificar visualmente una serie de patrones periódicos en tiempo, que parecen moverse a través de un fondo de pequeños mosaicos triangulares, como se muestra en la figura 2.20. Estos patrones periódicos reciben el nombre de partículas o gliders, mientras que el fondo de mosaicos se denomina informalmente como *éter*.

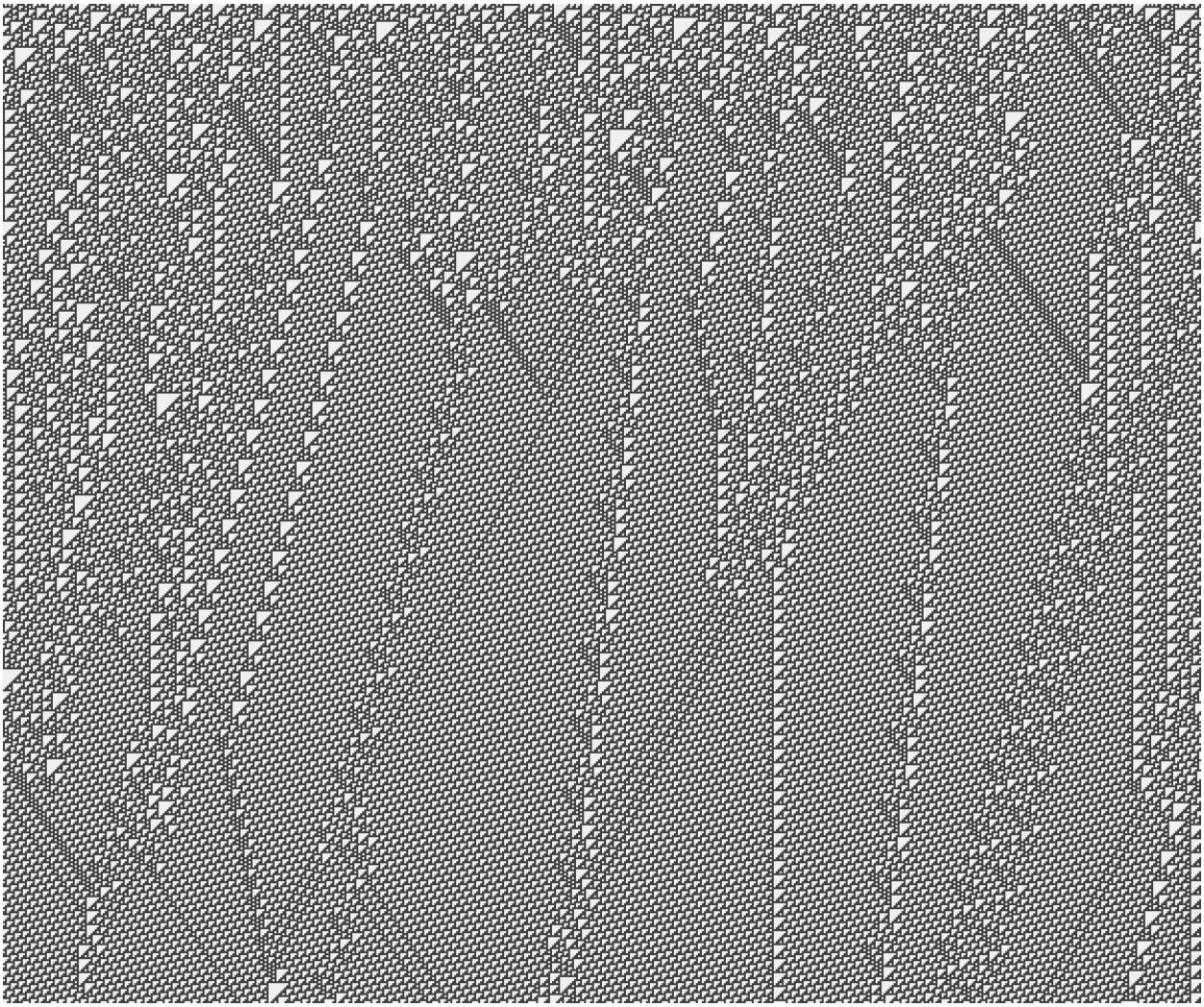

Figura 2.20: Evolución de la regla 110.

Cook realizó una descripción de todas las partículas que descubrió en la evolución de la regla 110. La figura 2.21 muestra todas las partículas conocidas, así como unos generadores de partículas, llamados *glider guns*. Cada partícula presenta características únicas. El conjunto de las partículas de la regla 110 se define como:

$$\Gamma = \{A^n, B, \bar{B}^n, \hat{B}^n, C_1, C_2, C_3, D_1, D_2, E^n, \bar{E}, F, G^n, H, gun^n\}$$

Donde $n \in \mathbb{Z}^+$ denota que una partícula puede ser extendible infinitamente. Adicionalmente, cg representa a una partícula g repetida c veces. [6]

El cuadro 2.1 recopila características claves de las partículas:

- El identificador de la partícula.
- El número de márgenes periódicos en la partícula. Para una partícula desplazándose hacia la derecha, *ems* indica el número de puntos de contacto que puede tener desde la izquierda o desde la derecha. Similarmente, para una partícula desplazándose hacia la izquiera, *oms* indica sus puntos de contacto, desde la izquierda o desde la derecha.
- La velocidad de la partícula, de la forma $\frac{x}{t}$, con x un desplazamiento lateral y t un desplazamiento hacia abajo. Se dice que una partícula con un movimiento hacia la derecha tiene una velocidad positiva, y

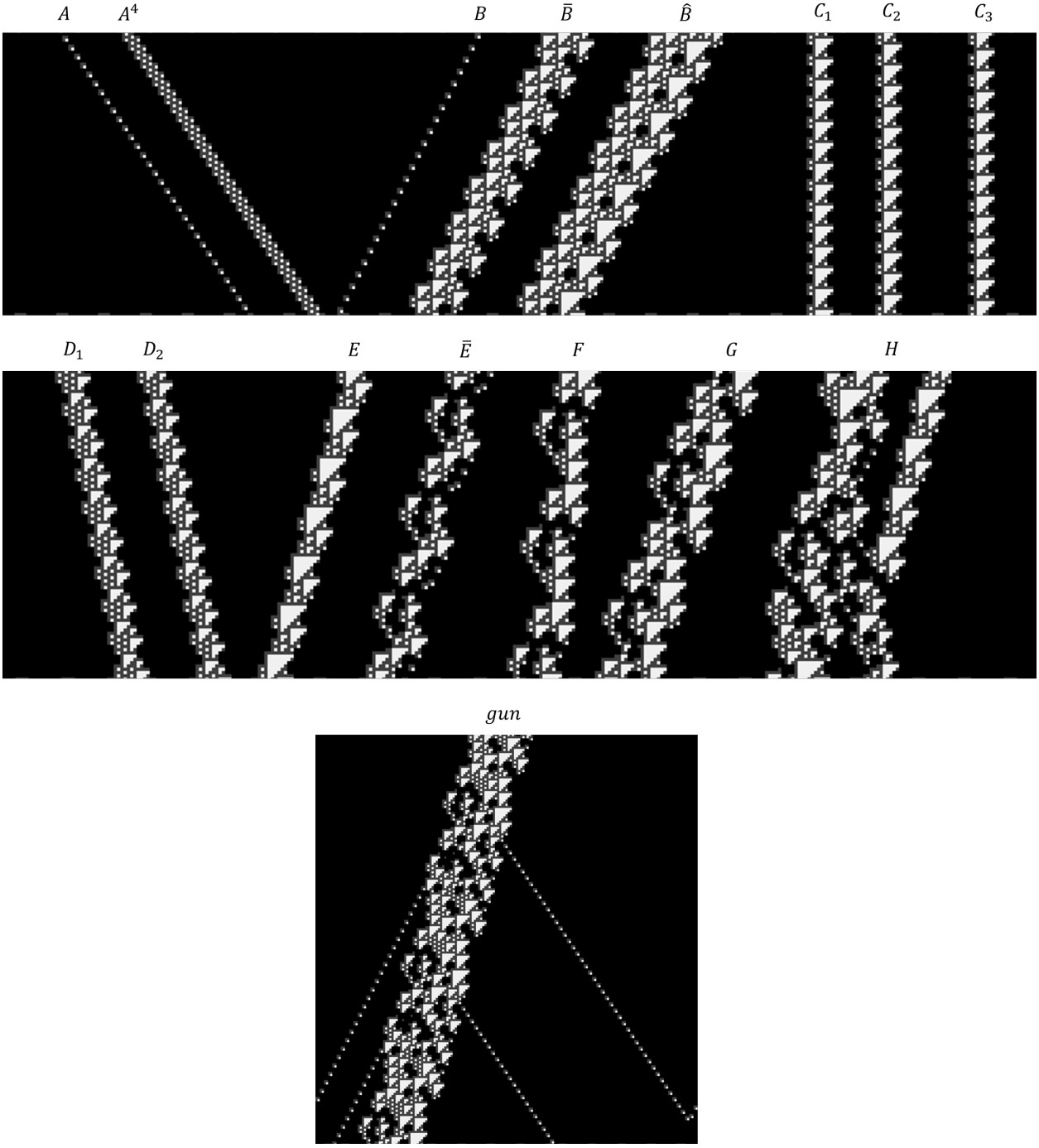

Figura 2.21: Partículas de la regla 110.

una con movimiento hacia la izquierda, velocidad negativa. Es importante notar que, para todo $g \in \Gamma$, $v_{e_l} \leq v_g \leq v_{e_r}$

- El ancho o volumen lineal de la partícula, es decir, el número de celdas que ocupa. Representado también mod 14 (el ancho del éter).

Se incluyen dos estructuras adicionales: e_r y e_l, las cuales representan la inclinación del patrón de éter. La figura 2.22 ilustra estas propiedades para e_r y e_l.

Estructura	**Márgenes**				**Velocidad**	**Volumen Lineal**
	Izquierda		**Derecha**			
	ems	**oms**	**ems**	**oms**		
e_r	.	1	.	1	$2/3 \approx 0{,}666666$	14
e_l	1	.	1	.	$-1/2 = -0{,}5$	14
A	.	1	.	1	$2/3 \approx 0{,}666666$	6
B	1	.	1	.	$-2/4 = -0{,}5$	8
$\bar{B}^n$	3	.	3	.	$-6/12 = -0{,}5$	22
$\hat{B}^n$	3	.	3	.	$-6/12 = -0{,}5$	39
C_1	1	1	1	1	$0/7 = 0$	9-23
C_2	1	1	1	1	$0/7 = 0$	17
C_3	1	1	1	1	$0/7 = 0$	11
D_1	1	2	1	2	$2/10 = 0{,}2$	11-25
D_2	1	2	1	2	$2/10 = 0{,}2$	19
E^n	3	1	3	1	$-4/15 \approx -0{,}266666$	19
$\bar{E}$	6	2	6	2	$-8/30 \approx -0{,}266666$	21
F	6	4	6	4	$-4/36 \approx -0{,}111111$	15-29
G^n	9	2	9	2	$-14/42 \approx -0{,}333333$	24-38
H	17	8	17	8	$-18/92 \approx -0{,}195652$	39-53
Glider Gun	15	5	15	5	$-20/77 \approx -0{,}259740$	27-55

Cuadro 2.1: Propiedades de las partículas de la regla 110. [6]

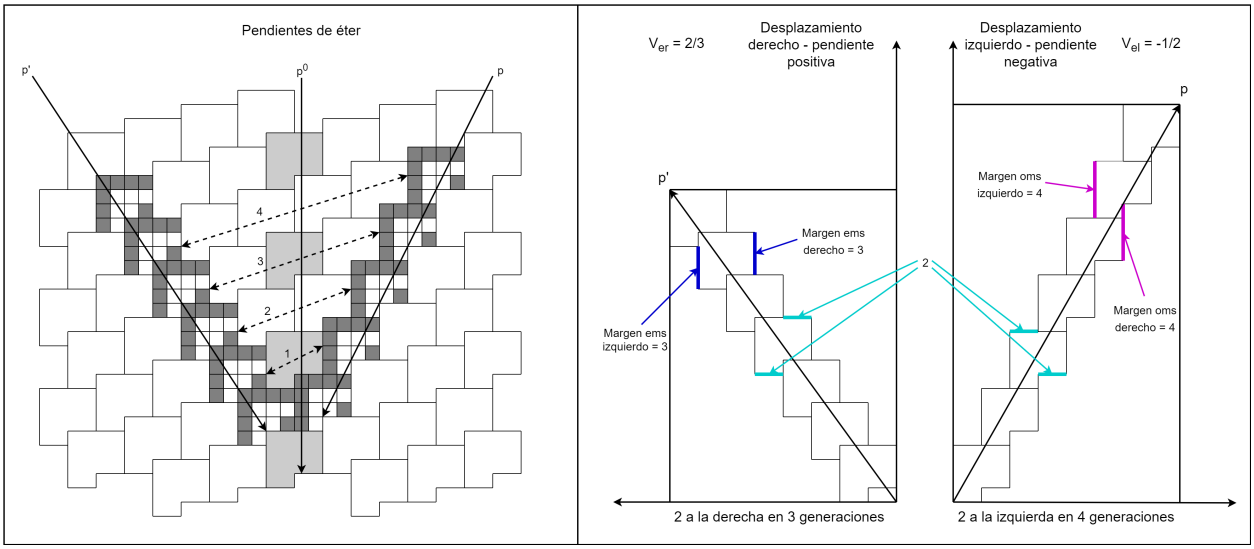

Figura 2.22: Propiedades del éter. [32]

2.4.3. Mosaicos

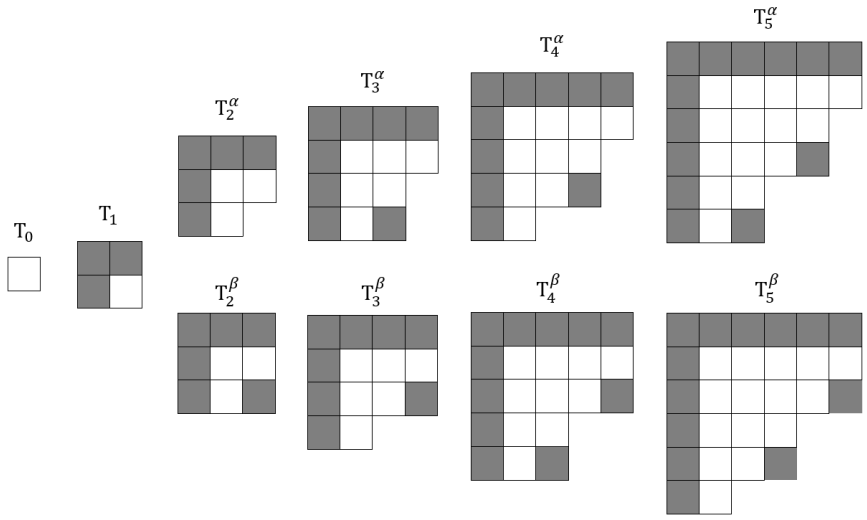

Figura 2.23: Primeros 6 tipos de mosaicos de la regla 110.

La regla 110 puede generar diferentes mosaicos triangulares, de la forma T_n, donde $n \in \mathbb{Z}_0^+$ representa el tamaño del triángulo interno. Para $n \geq 2$, los mosaicos se dividen en tipos α y β. Éstos se ilustran en la figura 2.23. El mosaico más grande que puede construirse en el espacio de la regla 110 es el T_{45}. [33]

De todos, los mosaicos T_3^β son los más recurrentes, por ser los elementos principales del éter. De aquí en adelante nos referiremos a ellos simplemente como T_3.

2.4.4. Fases

Es posible representar cualquiera de las partículas como expresiones regulares. Éstas son obtenidas mediante lo que se conoce como diagrama de De Brujin. Una explicación a fondo puede encontrarse en [6], [5], [34] y [33].

Es evidente que una partícula tiene diferentes representaciones, tantas como su periodo. Conocer estas alineaciones es de gran utilidad para crear instancias aisladas de la partícula. Las alineaciones, también llamadas fases, son agrupadas en secciones de 4. La fase de una partícula tiene la siguiente notación:

$$g(p, f_i\_1)$$

Donde:

- $g \in \Gamma$ es una partícula,
- $p \in \mathbb{Z}^+$ es una sección de 4 fases de g para un g con periodo mayor a 4. Es común usar también la notación ω_m, con $\omega \in \Omega = [A, B, C, \ldots, H]$ (1-indexado) y $m \in \mathbb{Z}^+$, para denominar la sección $IndexOf(\omega) * m$. Si $m = 1$, se puede omitir de la notación;
- $i \in 1, 2, 3, 4$ es el número de la fase de g en la sección p.

Por ejemplo, el éter contiene 4 fases, las cuales son:

1. $e(f_1\_1) = 11111000100110$
2. $e(f_2\_1) = 10001001101111$
3. $e(f_3\_1) = 10011011111000$
4. $e(f_4\_1) = 10111110001001$

Una lista con todas las fases para las partículas de la regla 110 puede consultarse en [35]. Es importante mencionar que $e(f_1\_1)$ es la fase más directa, y es regularmente usada en todas las expresiones, por lo que es usual denotarla solamente por e.

2.4.5. Universalidad de la regla 110

Matthew Cook creó los CTSs como una herramienta para implementar computaciones en la regla 110 y demostrar su universalidad. En su demostración, propuso un diagrama esquemático de conjuntos de partículas en la regla 110 para emular el comportamiento de un CTS con una lista de producciones $(111, 0)$. La figura 2.24 muestra la computación de las primeras 100 transiciones de este CTS con una cinta inicial 1, mientras que la figura 2.25 muestra el diagrama esquemático que propuso para este fin.

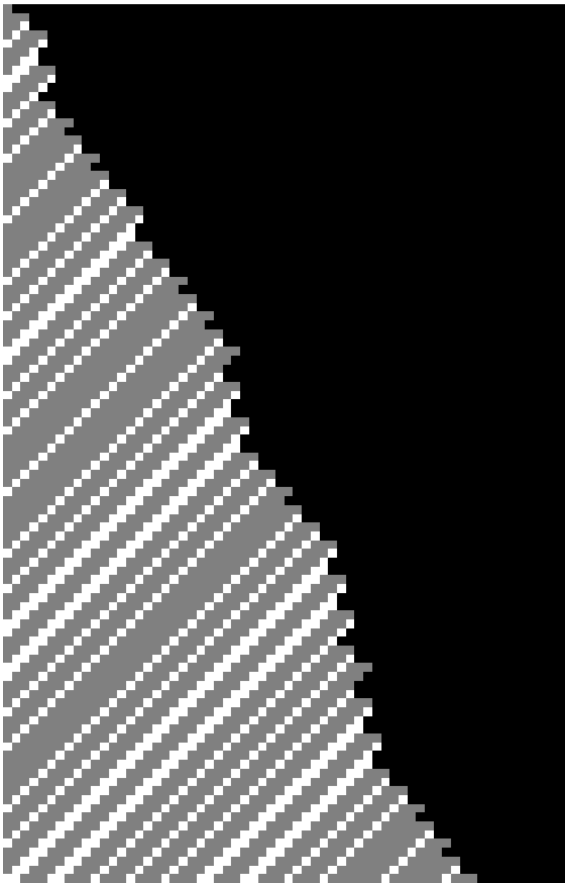

Figura 2.24: Computación del CTS $(111, 0)$ comenzando con la cinta 1.

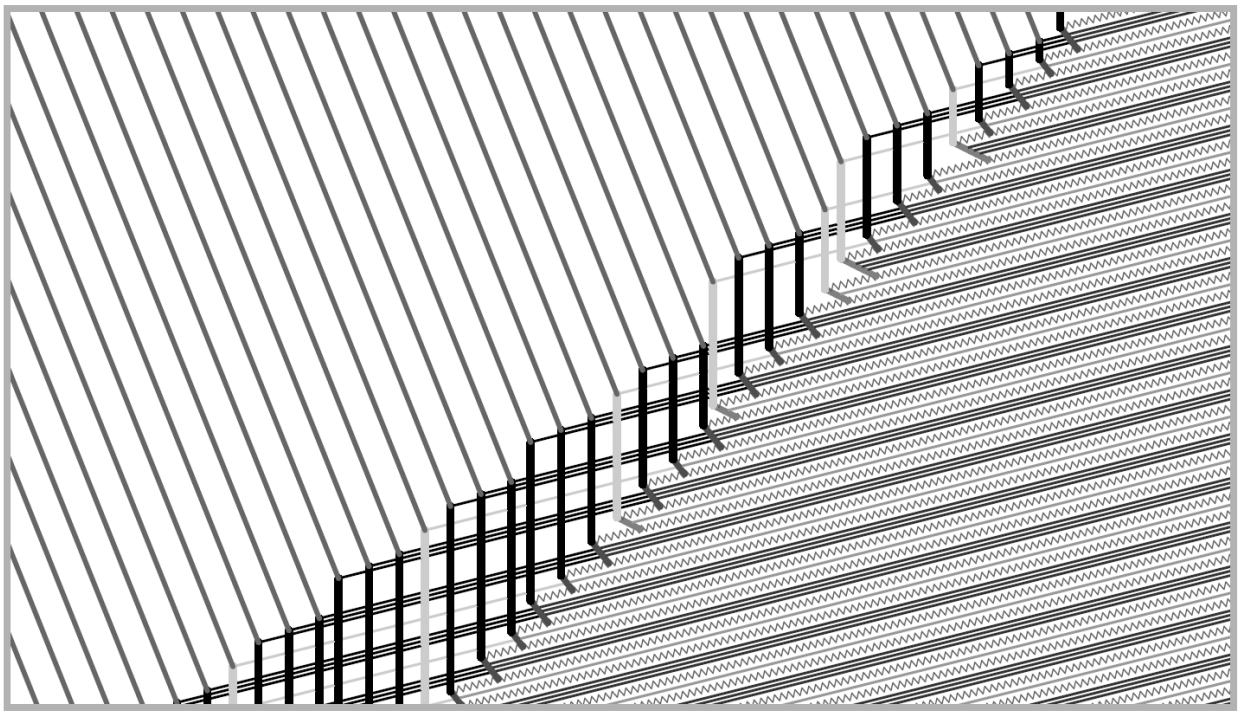

Figura 2.25: Diagrama esquemático del CTS $(111, 0)$ emulado por la regla 110. [3]

El artículo original de Cook fue reinterpretado en el libro A New Kind Of Science, por Stephen Wolfram, para una lista de producciones $(1, 101)$. [1] La figura 2.26 muestra la computación de las primeras 100 transiciones de este CTS con una cinta inicial 1, mientras que la figura 2.27 muestra el diagrama esquemático correspondiente.

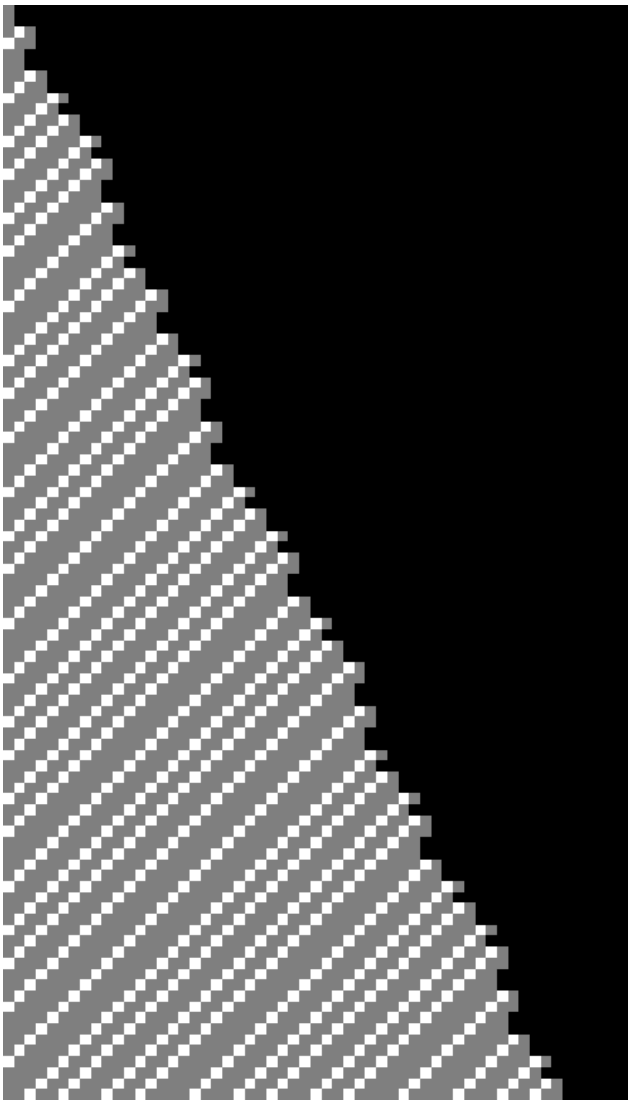

Figura 2.26: Computación del CTS $(1, 101)$ comenzando con la cinta 1.

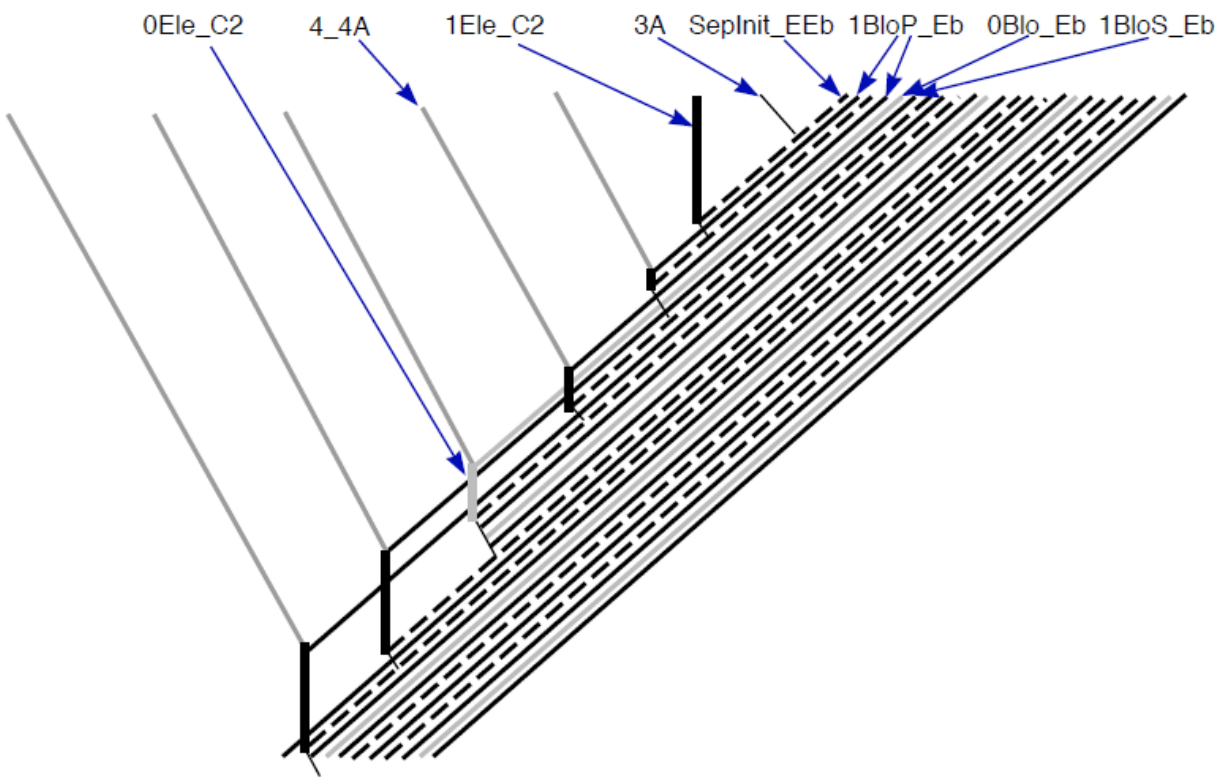

Figura 2.27: Diagrama esquemático del CTS $(1, 101)$ emulado por la regla 110. [6] [5] [34]

Este trabajo busca realizar la simulación de éste último. En la siguiente sección se explicarán a profundidad los conjuntos de partículas usados para lograrlo.

2.4.6. Conjuntos de partículas

Conjunto 4_A^4

Define 4 trenes de partículas A^4, los cuales cambian de fase periódicamente:

$$4\_A^4(F_3) = A^4(f_3\_1) - 27e - A^4(f_2\_1) - 23e - A^4(f_1\_1) - 25e - A^4(f_3\_1)$$

$$4\_A^4(F_2) = A^4(f_2\_1) - 27e - A^4(f_1\_1) - 23e - A^4(f_3\_1) - 25e - A^4(f_2\_1)$$

$$4\_A^4(F_1) = A^4(f_1\_1) - 27e - A^4(f_3\_1) - 23e - A^4(f_2\_1) - 25e - A^4(f_1\_1)$$

Una extensión de este conjunto es útil en las colisiones del CTS:

$$\{649e - 4\_A^4(F_2) - 649e - 4\_A^4(F_1) - 649e - 4\_A^4(F_3)\}*$$

Conjuntos $1Ele\_C_2$ **y** $0Ele\_C_2$

El conjunto $1Ele\_C_2$ representa un 1 en la palabra del CTS, mientras que $0Ele\_C_2$ representa un 0.

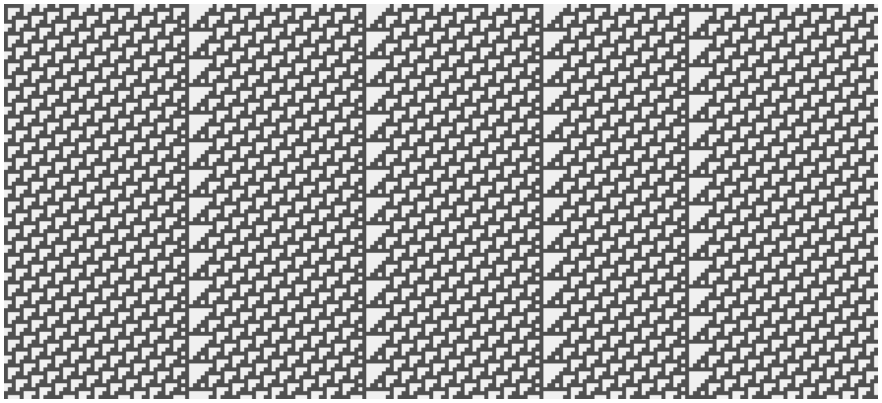

Figura 2.28: Conjunto $1Ele\_C_2$.

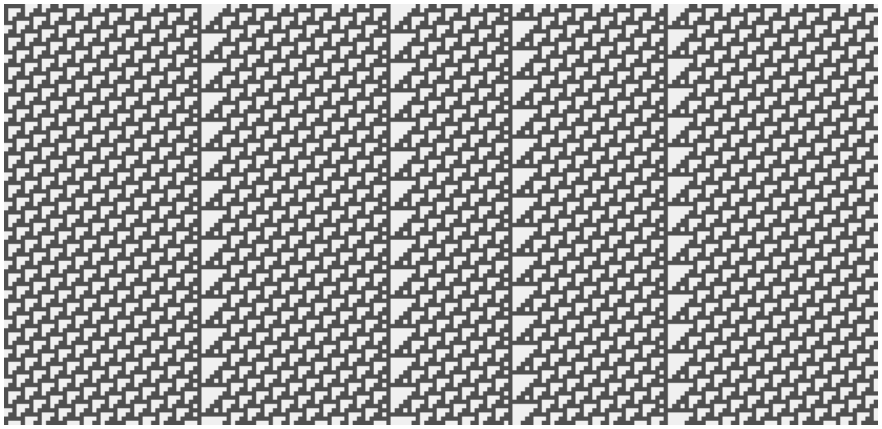

Figura 2.29: Conjunto $0Ele\_C_2$.

Conjunto $0Blo\_\bar{E}$**.**

El conjunto $0Blo\_\bar{E}$ almacena bloques de data en su parte izquierda y produce el conjunto $0Add\_\bar{E}$.

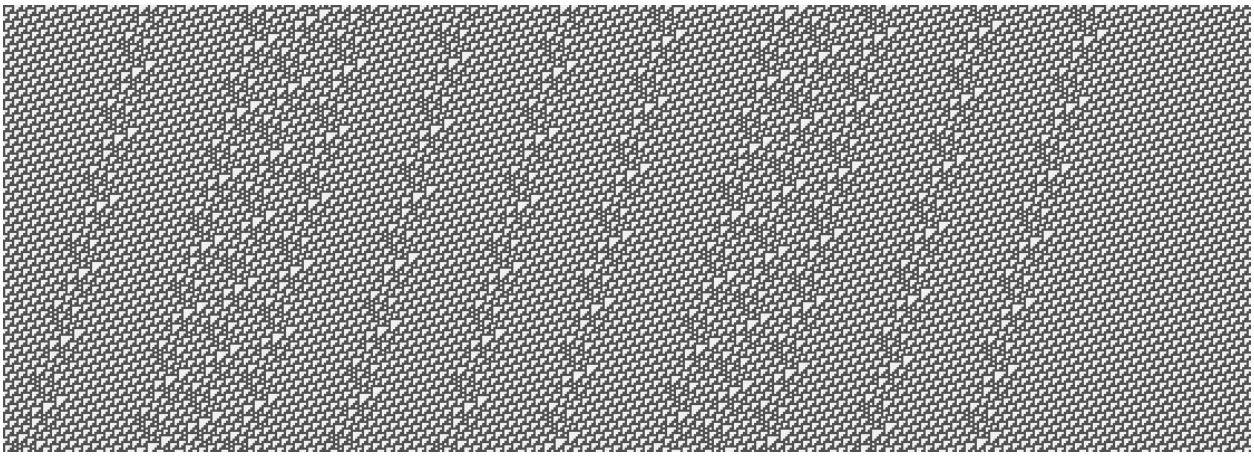

Figura 2.30: Conjunto $0Blo\_\bar{E}$.

Conjuntos $1BloP\_\bar{E}$ **y** $1BloS\_\bar{E}$

Debido a la evolución asimétrica de la regla 110, es necesario usar dos distintos conjuntos para escribir 1s en el CTS: $1BloP\_\bar{E}$ (primario) y $1BloS\_\bar{E}$ (estándar). Ambos conjuntos producen el mismo conjunto $1Add\_\bar{E}$.

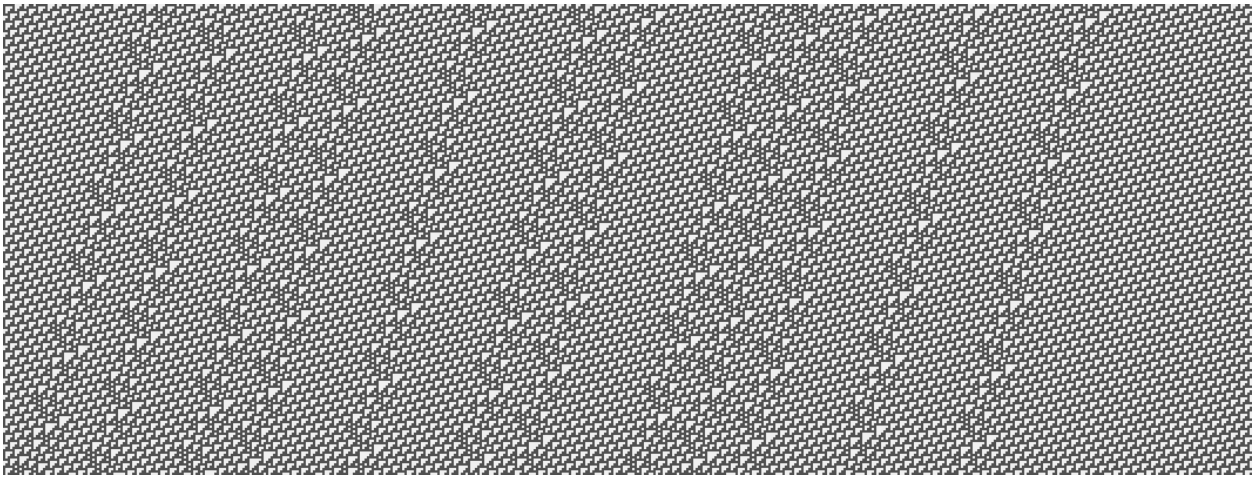

Figura 2.31: Conjunto $1BloP\_\bar{E}$.

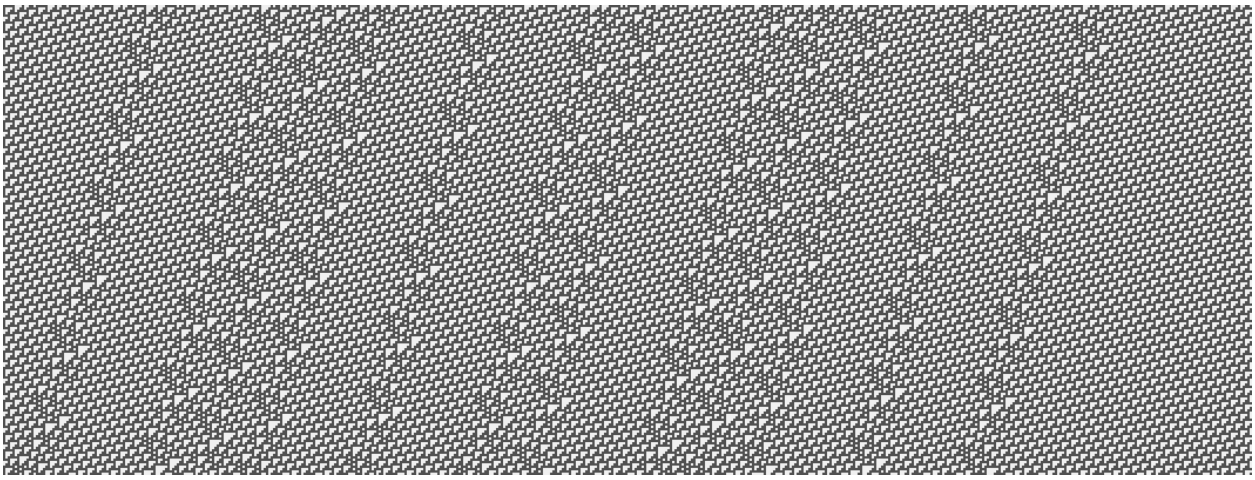

Figura 2.32: Conjunto $1BloS\_\bar{E}$.

Conjunto $SepInit\_E\bar{E}$

El conjunto $SepInit\_E\bar{E}$ actúa como el componente líder para las interacciones de los demás conjuntos. Es esencial para separar trenes de datos y para determinar la incorporación de datos en la palabra del CTS, al discriminar los datos que se agregarán o eliminarán.

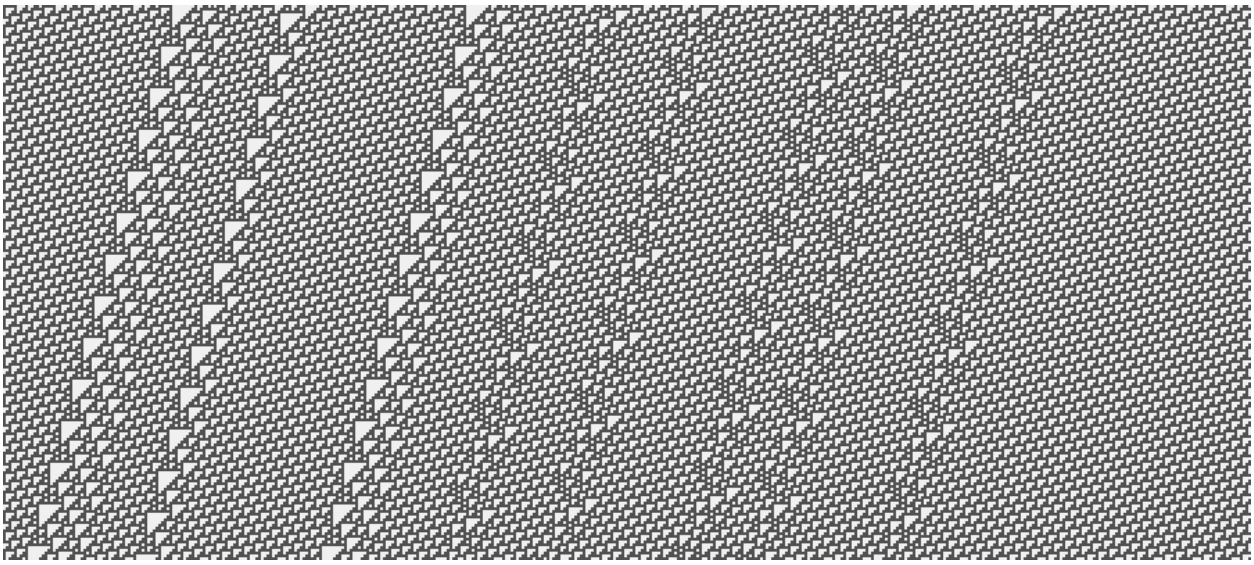

Figura 2.33: Conjunto $SepInit\_E\bar{E}$.

Conjuntos $1Add\_\bar{E}$ **y** $0Add\_\bar{E}$

El conjunto $1Add\_\bar{E}$ es generado por $1BloP\_\bar{E}$ y $1BloS\_\bar{E}$, mientras que $0Add\_\bar{E}$ es generado por $0Blo\_\bar{E}$.

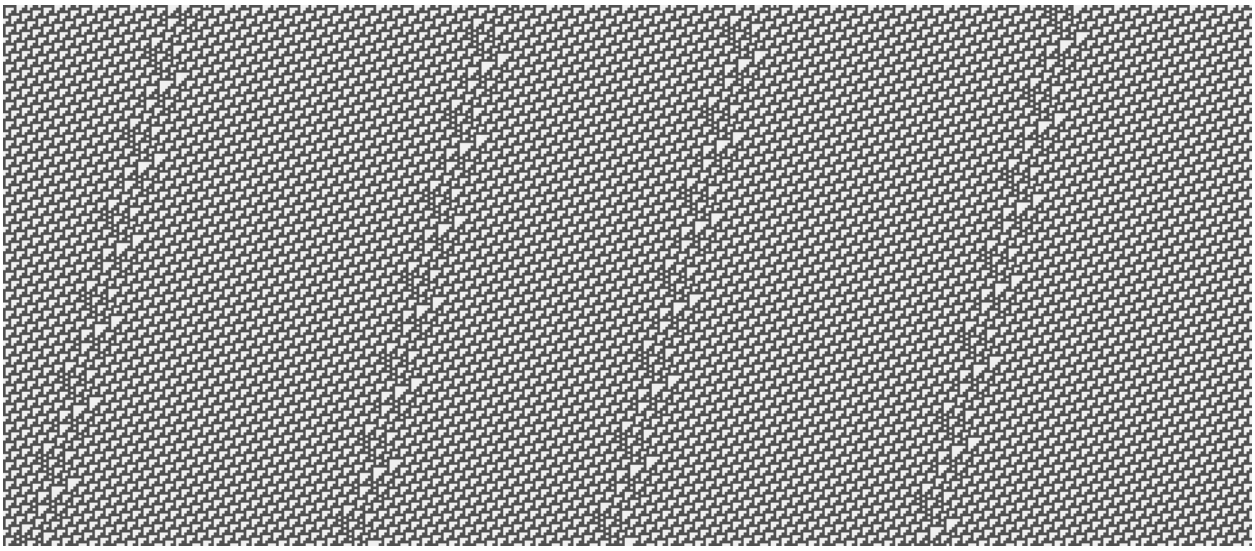

Figura 2.34: Conjunto $1Add\_\bar{E}$.

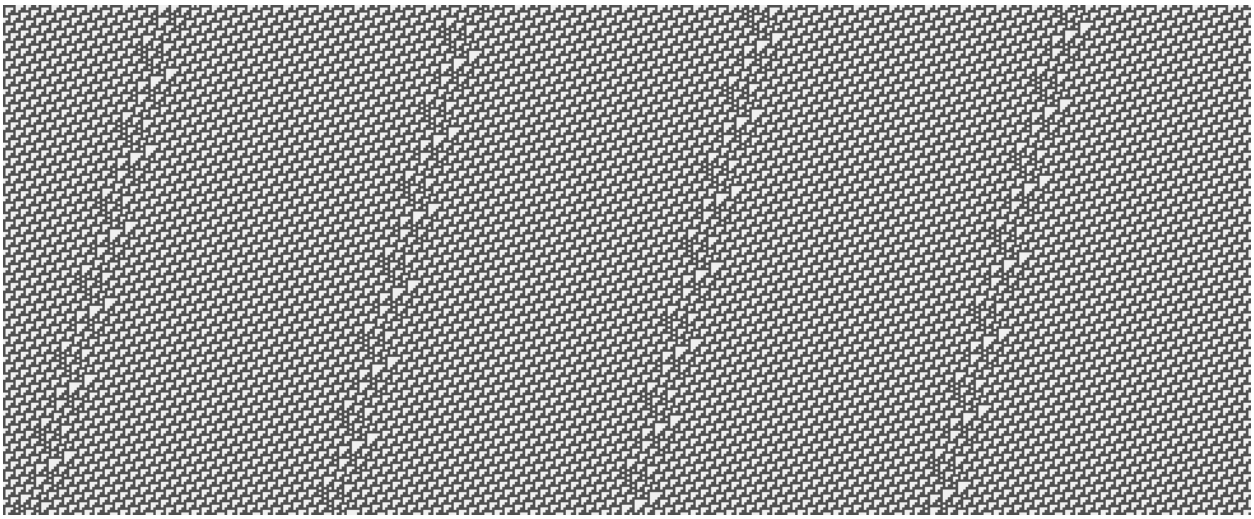

Figura 2.35: Conjunto $0Add\_\bar{E}$.

2.4.7. Funcionamiento del CTS en la regla 110

El sistema completo para simular el CTS $(1, 101)$ con una palabra inicial 1, se puede dividir en 3 componentes principales:

$$
\begin{aligned}
Izquierda: \quad & \ldots - 217e - 4\_A^4(F_2) - 649e - 4\_A^4(F_1) - 649e - 4\_A^4(F_3) - 649e - 4\_A^4(F_2) - \\
& 649e - 4\_A^4(F_1) - 649e - 4\_A^4(F_3) - 216e - \\
Centro: \quad & 1Ele\_C_2(A, f_1\_1) - e - A^3(f_1\_1) - \\
Derecha: \quad & SepInit\_E\bar{E}(C, f_3\_1) - 1BloP\_\bar{E}(C, f_4\_1) - SepInit\_E\bar{E}(C, f_3\_1) - \\
& 1BloP\_\bar{E}(C, f_4\_1) - 0Blo\_\bar{E}(C, f_4\_1) - 1BloS\_\bar{E}(A, f_4\_1) - \\
& SepInit\_E\bar{E}(A, f_2\_1) - 1BloP\_\bar{E}(F, f_1\_1) - SepInit\_E\bar{E}(A, f_3\_1) - \\
& 1BloP\_\bar{E}(F, f_1\_1) - 0Blo\_\bar{E}(E, f_4\_1) - 1BloS\_\bar{E}(C, f_4\_1) - e - \\
& SepInit\_E\bar{E}(B, f_1\_1) - 1BloP\_\bar{E}(F, f_3\_1) - e - \\
& SepInit\_E\bar{E}(B, f_1\_1) - 217e - \ldots
\end{aligned}
$$

- La sección izquierda, compuesta por trenes de partículas 4_A^4, moviéndose a la derecha, controla la producción final de valores que se agregan a la cinta del CTS.
- La sección central contiene alguno de los conjuntos estáticos $1Ele\_C_2$ o $0Ele\_C_2$, que simbolizan una cinta inicial 1 o 0, respectivamente. Además, contiene una partícula A^3 que servirá como un activador para el sistema de la sección derecha.
- La sección derecha se mueve con dirección izquierda y se encarga del procesamiento de datos. Cada conjunto $SepInit\_E\bar{E}$ marca el comienzo de un sector de procesamiento, en el que cada uno de los conjuntos $Blo\_\bar{E}$ producirá los elementos necesarios para añadir nuevos valores a la cinta del CTS, dependiendo de los valores encontrados al momento.

La computación del CTS, indicada en la figura 2.27, procesa su cinta de derecha a izquierda mediante los siguientes pasos:

1. El conjunto $SepInit\_E\bar{E}$ que lidera el primer sector de procesamiento de la sección derecha, colisiona con la partícula central A^3 para entrar en modo de activación.
2. El conjunto $SepInit\_E\bar{E}$ activado colisiona con el conjunto estático de la sección central y lo destruye, simulando la eliminación del valor en la cinta.

a) Si el conjunto estático removido es un $0Ele\_C_2$, todos los bloques $Blo\_\bar{E}$ a la derecha también serán destruidos. Esta acción representa la lectura de un valor 0 en la cinta, con la cual no se agrega ningún nuevo dato.

b) Si el conjunto removido es un $1Ele\_C_2$, los bloques $Blo\_\bar{E}$ a la derecha colapsan uno a uno para crear conjuntos de adición de datos $Add\_\bar{E}$.

1) Si el bloque a colapsar es un $0Blo\_\bar{E}$, la colisión resultante creará un conjunto de adición $0Add\_\bar{E}$.
2) Si el bloque a colapsar es un $1BloP\_\bar{E}$ o $1BloS\_\bar{E}$, la colisión resultante creará un conjunto de adición $1Add\_\bar{E}$. Debido a la asimetría del sistema global, es necesario tener dos diferentes bloques para la correcta creación del conjunto de adición. El bloque primario, $1BloP\_\bar{E}$, debe ser siempre el primero de su sector, mientras que el resto serán bloques estándar, $1BloS\_\bar{E}$.

3. En el proceso de destrucción de los conjuntos Ele_C_2, se generarán además partículas $\bar{E}$ individuales como residuos de la colisión. Si todas las partículas a su izquierda están adecuadamente sincronizadas en fase y distancia, estos residuos deberían atravesarlos sin alterar su estructura, efectivamente actuando como solitones en el sistema.

4. Al terminar el proceso de eliminación o adición, la última partícula generada es un A^3, que servirá de activador para el $SepInit\_E\bar{E}$ del siguiente sector.

5. Finalmente, los conjuntos de adición generados en el sector colisionan con el tren de partículas 4_A^4 de la sección izquierda, generando un $0Ele\_C_2$ a partir de un $0Add\_\bar{E}$, o bien, generando un $1Ele\_C_2$ a partir de un $1Add\_\bar{E}$. Esto representa la adición de un nuevo valor 0 o 1, respectivamente, al final de la cinta.

Para poder generar las transiciones $1 \Rightarrow 1 \Rightarrow 101 \Rightarrow 011$, se requieren 57,400 iteraciones y una configuración inicial de 56,240 celdas. El espacio de evoluciones total contiene 3,228,176,000 celdas. [6] [5] [34]

2.5. Colisionador

2.5.1. Teoría de choques

En los años 70s, Fredkin y Toffoli propusieron un modelo de computación que priorizaba la conservación de energía, basado en las interacciones balísticas entre cuantos de información, los cuales eran representados por partículas abstractas. [6] Este modelo, llamado modelo de bolas de billar, demostró que un contenedor de partículas dado es capaz de realizar cualquier tipo de computación. Más tarde, Margolus aplicó este concepto a la teoría de autómatas celulares mediante la implementación de un modelo de vecindad que componía un sistema de bolas de billar. [4]

Podemos definir de manera no rigurosa una función de colisión clásica $f : \Gamma \times \Gamma \to \Gamma^*$.

Sean $u, v, w, w_1, \ldots, w_k \in \Gamma$. La función de colisión puede presentar alguno de los siguientes comportamientos, ilustrados en la imagen 2.36:

- Destrucción: $f(u, v) = \epsilon$ (figura 2.36a).
- Fusión: $f(u, v) = w$ (figura 2.36b).
- Interacción y producción de nuevas partículas: $f(u, v) = (w_1, \ldots, w_k)$ (figura 2.36c).
- Identidad o colisión solitónica: $f(u, v) = f_i(u, v) = (u, v)$ (figura 2.36d).
- Reflexión o colisión elástica: $f(u, v) = f_r(u, v) = (v, u)$ (figura 2.36e).

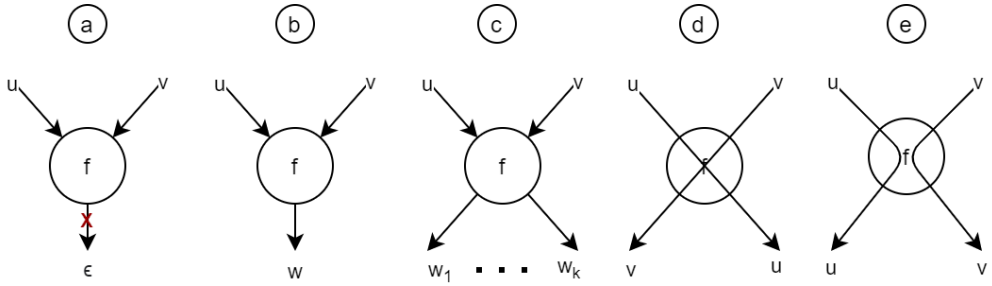

Figura 2.36: Tipos de colisiones entre partículas en un AC.

Para representar una colisión en un AC, utilizamos la noción de partículas idealizadas $p \in \Gamma$ (sin energía ni potencial), las cuales se mueven a través de un espacio con desplazamiento lateral positivo (p^+), negativo (p^-) o neutro (p^0). La figura 2.37 muestra diferentes escenarios del movimiento de dos partículas hacia un punto de colisión (marcado con azul y contorno punteado):

- Dos partículas, p^+ y p^-, se mueven en direcciones opuestas hasta chocar en el punto de colisión (figura 2.37a).
- Dos partículas, p_F y p_S, se mueven en la misma dirección, y p_F posee una mayor velocidad lateral que p_S. Por tanto, el punto de colisión marca el momento en el que p_F alcanza a p_S (figuras 2.37b y 2.37c).
- Una partícula p se mueve hacia una partícula neutra p_0. El punto de contacto se encuentra en p_0 (figuras 2.37d y 2.37e).

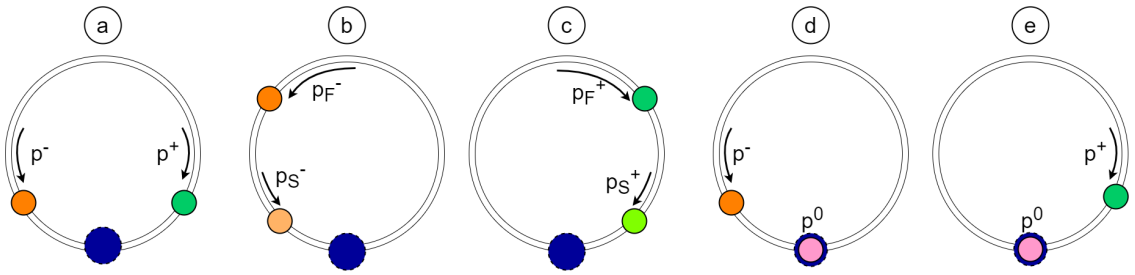

Figura 2.37: Posibles escenarios en la colisión de dos partículas en un AC.

Podemos encontrar todo tipo de partículas en un AC complejo, con desplazamientos positivos, negativos o neutros, así como partículas compuestas de otras partículas en fases determinadas. Podemos apreciar diferentes tipos y escenarios de colisiones con partículas de la regla elemental 110 en las siguientes figuras:

- Colisión destructiva en escenario positivo-negativo: $f(p_A^+, p_B^-) = \epsilon$ (figura 2.38).
- Colisión de fusión en escenario positivo-neutro: $f(p_{A^4}^+, p_{C_1}^0) = (p_E^-)$ (figura 2.39).
- Colisión solitónica en escenario neutro-negativo: $f(p_{C_2}^0, p_F^-) = (p_{C_2}^0, p_F^-)$ (figura 2.40).
- Colisión de producción en escenario negativo-negativo: $f(p_F^-, p_E^-) = (p_B^-, p_A^+, p_A^+)$ (figura 2.41).

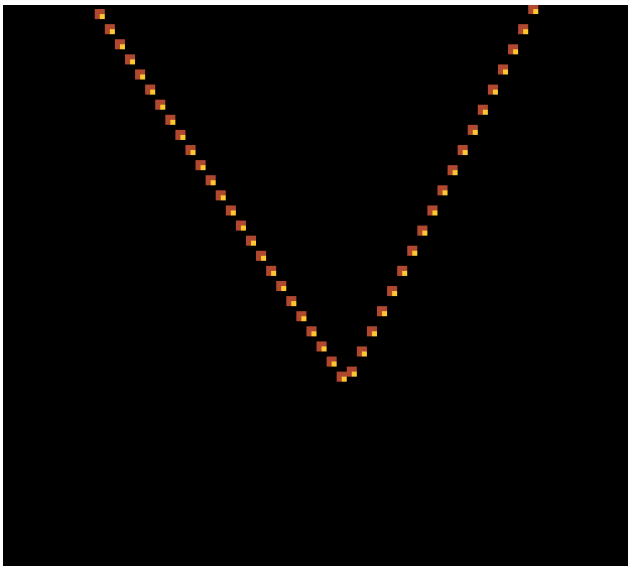

Figura 2.38: Ejemplo de colisión destructiva $f(p_A^+, p_B^-) = \epsilon$.

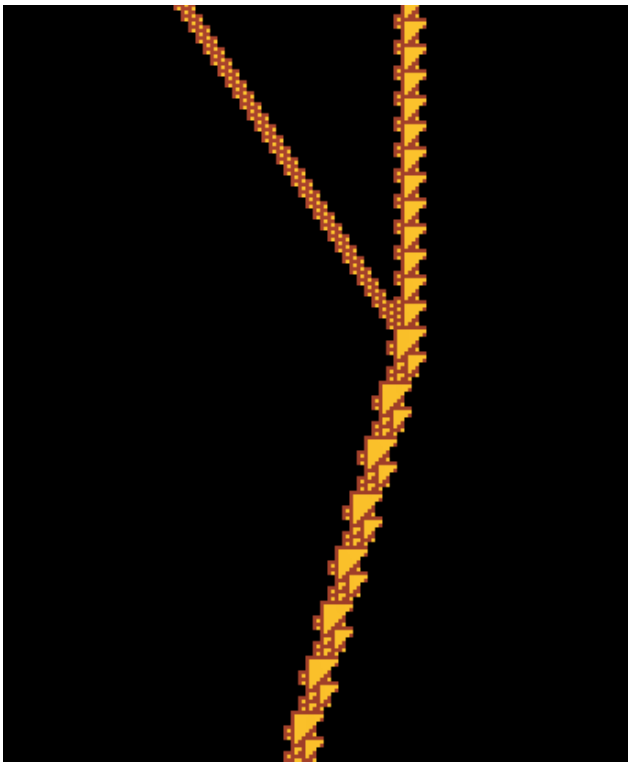

Figura 2.39: Ejemplo de colisión de fusión $f(p_{A^4}^+, p_{C_1}^0) = (p_E^-)$.

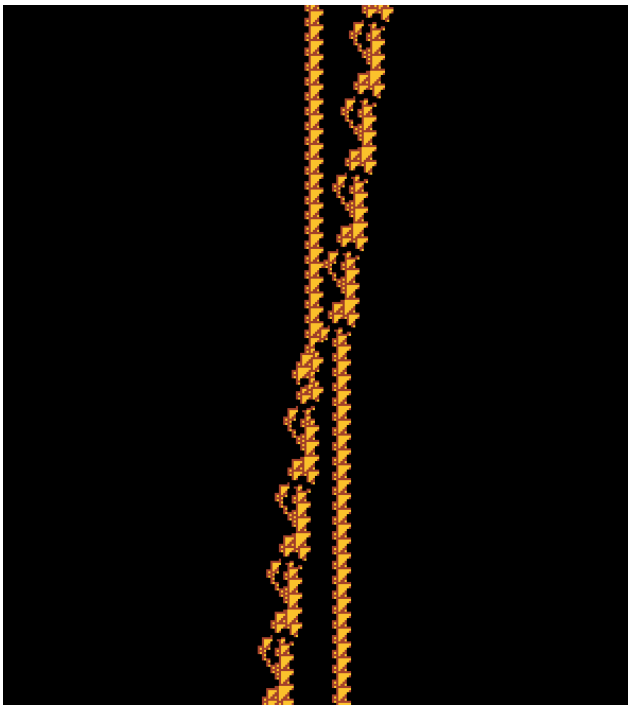

Figura 2.40: Ejemplo de colisión solitónica $f(p^0_{C_2}, p^-_F) = (p^0_{C_2}, p^-_F)$.

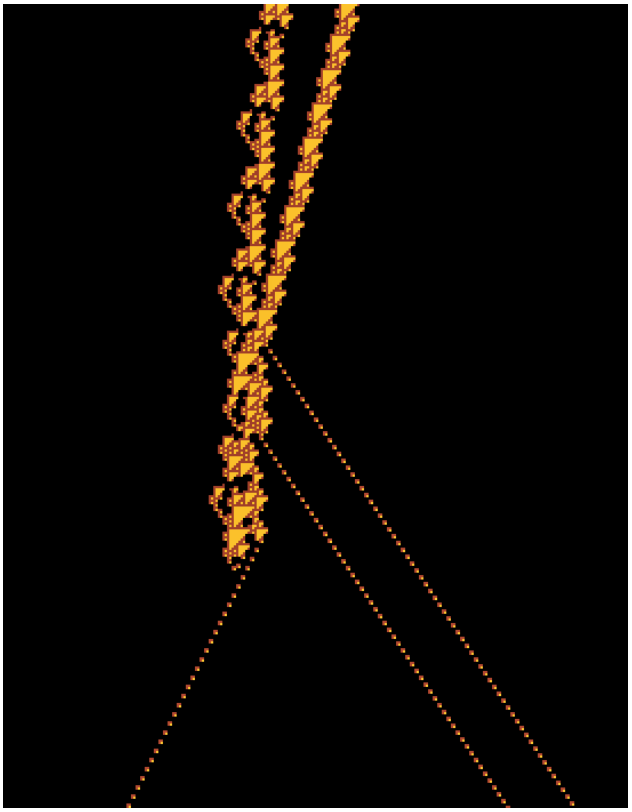

Figura 2.41: Ejemplo de colisión de producción $f(p^-_F, p^-_E) = (p^-_B, p^+_A, p^+_A)$.

2.5.2. Ciclotrones y enrutamiento de haces de partículas

Al suponer condiciones de frontera periodicas, el espacio del AC presenta un comportamiento de anillo, como se mostró en la figura 2.11. De esta forma, partículas que se mueven a lo largo del espacio podrán transmitirse e interactuar de manera indefinida, asemejando el comportamiento de un ciclotrón virtual.

Mediante un correcto posicionamiento en distancias y fases, es posible enrutar haces de partículas para secuenciar colisiones en un ciclotrón. Por ejemplo, la figura 2.42 muestra el diseño de un enrutamiento de las partículas $p^+_{A^4}$ y $p^-_{\bar{E}}$. Al colisionar, se producen partículas p^-_B, $p^-_{\bar{B}}$ y p^-_F, las cuales conforman un nuevo enrutamiento. La partícula p^-_F es rápidamente alcanzada por las otras dos, colisionando y produciendo las partículas $p^+_{A^4}$ y $p^-_{\bar{E}}$ del enrutamiento original. Esta transición se repite infinitamente. La figura 2.43 muestra la dinámica en malla de esta transición.

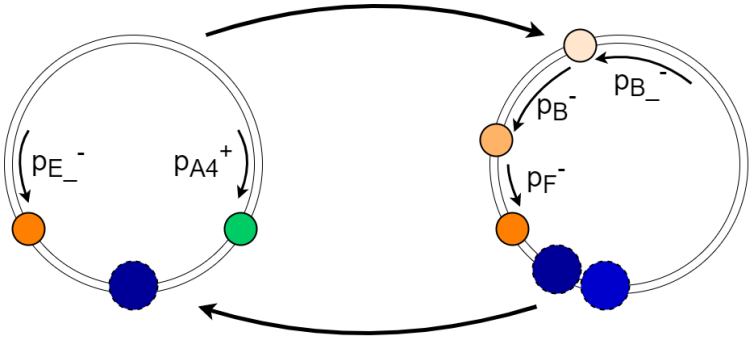

Figura 2.42: Transición entre dos enrutamientos sincronizados en reacciones múltiples para las partículas $p^+_{A^4}$ y $p^-_{\bar{E}}$.

.

Decimos, entonces, que la primera transición entre los enrutamientos se conforma por:

$$(p^+_{A^4} \Leftrightarrow p^-_{\bar{E}}) \rightarrow (p^-_F \Leftrightarrow p^-_B, p^-_{\bar{B}})$$

Mientras que la segunda transición se conforma por:

$$(p^-_F \Leftrightarrow p^-_B, p^-_{\bar{B}}) \rightarrow (p^+_{A^4} \Leftrightarrow p^-_{\bar{E}})$$

2.5.3. Computación en ciclotrones y colisionador virtual

La representación de transiciones de enrutamientos nos permite diseñar un modelo para las colisiones necesarias en la computación del CTS en un ciclotrón. La figura 2.44 muestra la representación en enrutamientos de cada uno de los conjuntos de partículas definidos en la regla 110.

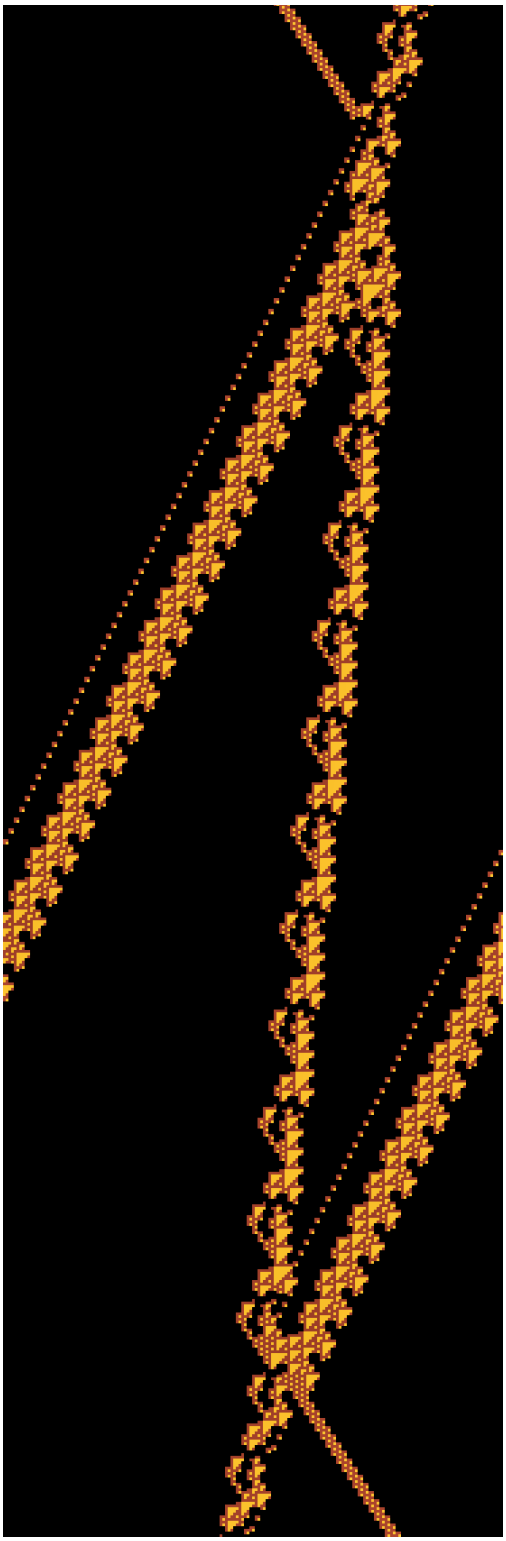

Figura 2.43: Visualización en malla de la transición de enrutamientos para las partículas $p^{+}_{A^4}$ y $p^{-}_{\bar{E}}$.

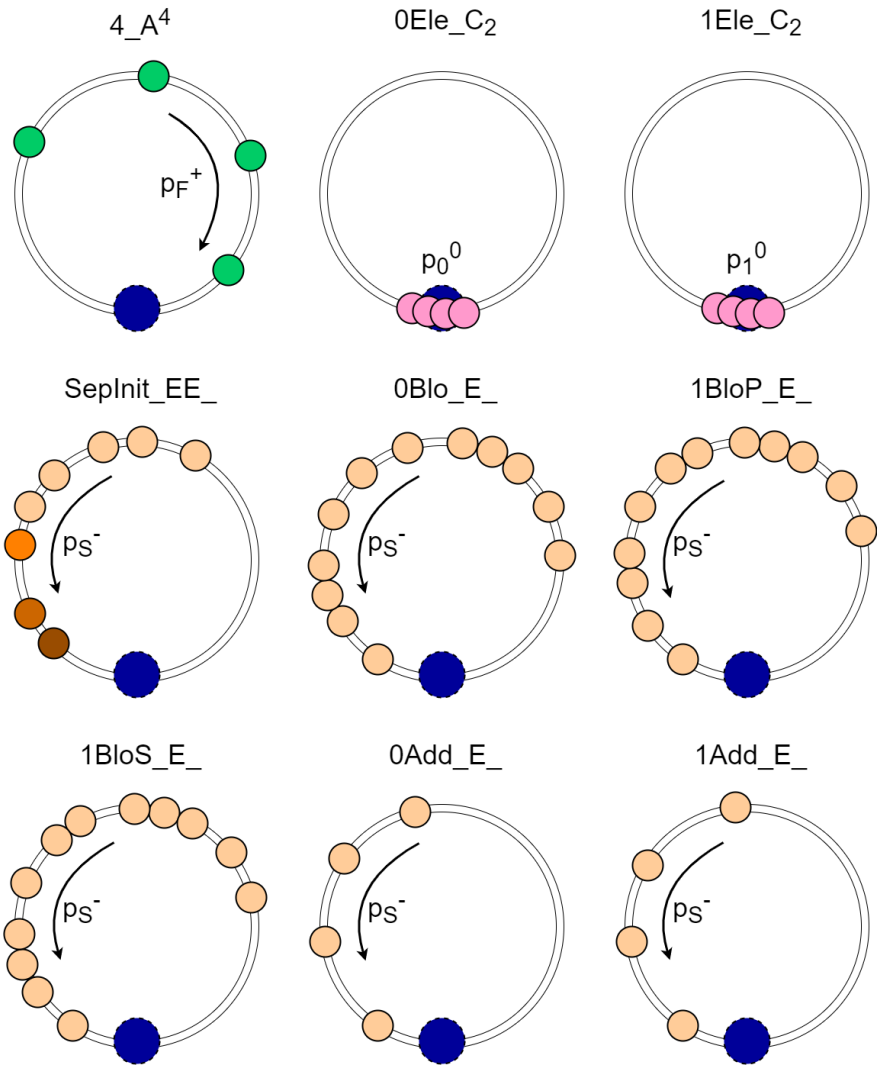

Figura 2.44: Enrutamiento de los conjuntos de partículas de la regla 110.

.

De esta manera, es posible codificar los pasos para la computación del CTS, descritos en la subsección *Funcionamiento del CTS en la regla 110*, usando transiciones de enrutamientos. La figura 2.45 muestra estas transiciones para esquematizar el simulador mediante colisiones.

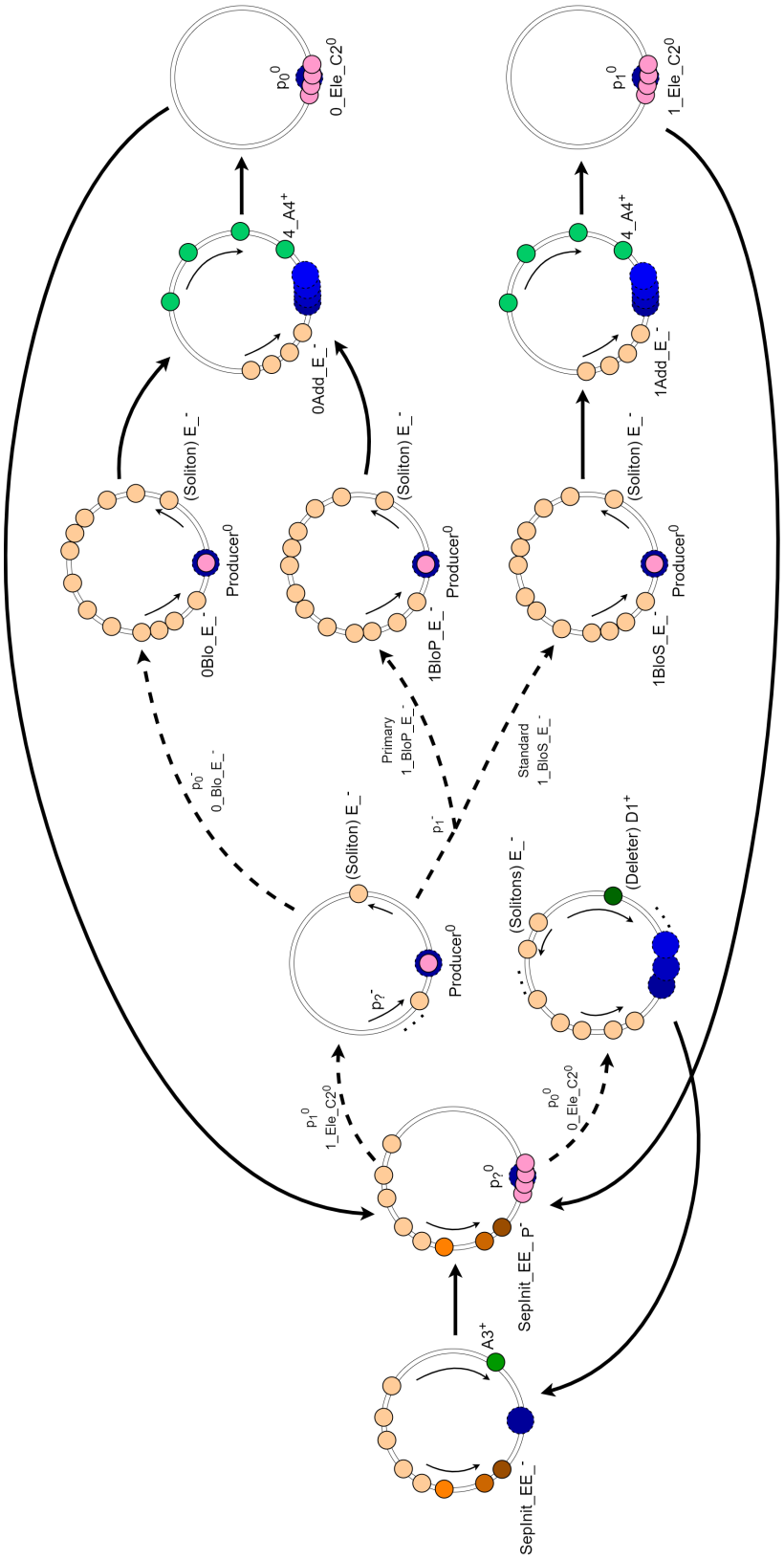

Figura 2.45: Transición de las colisiones del simulador de CTS. [6]

.

Para construir las interacciones en el sistema global, es necesario unir los espacios de los ciclotrones en un súper-ciclotrón. Los puntos de contacto entre ciclotrones transmitirán haces de partículas para comenzar el proceso de colisión. Si se sincronizan adecuadamente las fases y distancias en cada ciclotrón, podemos modelar el sistema de súper-ciclotrón como un colisionador virtual clásico:

- El anillo izquierdo inyectará al anillo central partículas $p^+_{A^4}$.
- El anillo derecho inyectará al anillo central partículas $p^-_{\bar{E}}$ y $p^-_{E^k}$.
- El anillo central llevará a cabo todas las colisiones de las partículas inyectadas.

La configuración total del sistema del colisionador se define mediante la 7-tupla:

$$(S_l, S_r, S_c, i_l, i_r, i_{cl}, i_{cr})$$

Donde:

- $S_l \in \Sigma^+$ es la condición inicial para el anillo izquierdo.
- $S_r \in \Sigma^+$ es la condición inicial para el anillo derecho.
- $S_c \in \Sigma^+$ es la condición inicial para el anillo central.
- $i_l \in \{0, \ldots, |S_l|\}$ es el punto de contacto desde el anillo izquierdo hacia el central.
- $i_r \in \{0, \ldots, |S_r|\}$ es el punto de contacto desde el anillo derecho hacia el central.
- $i_{cl}, i_{cr} \in \{0, \ldots, |S_c|\}$ son los puntos de contacto del anillo central hacia el izquierdo y el derecho, respectivamente.

Para cada punto de contacto $i \in \{i_l, i_r, i_{cl}, i_{cr}\}$ correspondiente a un anillo S, el intercambio de células se lleva a cabo entre las celdas i e $i - 1 \bmod |S|$. Comúnmente, se supone además que $i_{cl} \neq i_{cr}$.

Para asegurar una correcta sincronización entre los ciclotrones, la configuración de los puntos i debe cumplir con dos condiciones:

1. $i \bmod 14 = 0$, siendo 14 el ancho del éter.
2. i es el punto medio entre dos partículas o cadenas de éter.

La figura 2.46 muestra de manera simplificada el funcionamiento del colisionador virtual.

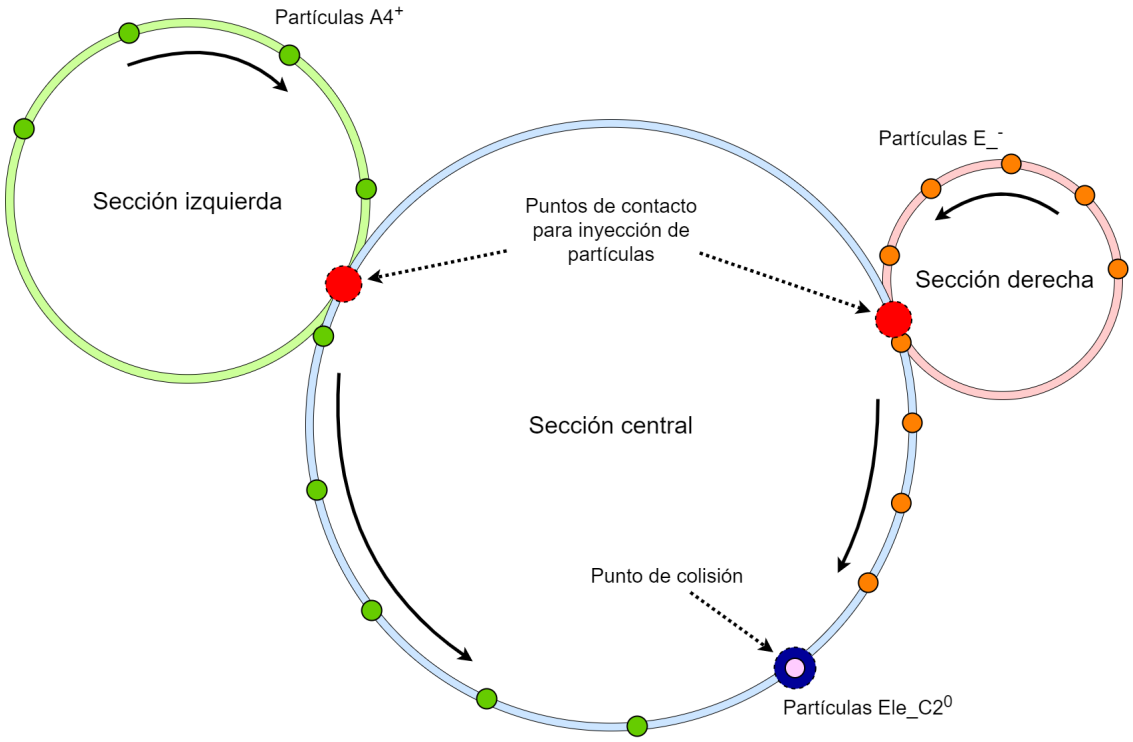

Figura 2.46: Diagrama simplificado de un colisionador virtual de partículas de AC. [6]

.

2.6. Reconocimiento de patrones en cadenas

2.6.1. Reconocimiento por fuerza bruta

Un algoritmo simple de búsqueda escanea ocurrencias de una palabra w, en un texto S, con $|w| = m$ y $|S| = n$. En cada posición $i \in \{0, \ldots, n-m-1\}$, el algoritmo checa igualdad entre $S[i]$ con el primer caracter de w, es decir, si $S[i] == w[0]$. Si coinciden, se comparan los caracteres sucesivos, $S[i+1] == w[1], S[i+2] == w[2], \ldots, S[i+m] == w[m-1]$. Si todas las comparaciones son correctas, se ha encontrado una ocurrencia de w en S comenzando en la posición i. El algoritmo tiene, entonces, una complejidad temporal de $O(nm)$.

La figura 2.47 muestra un ejemplo de la ejecución de este algoritmo para una $S =$ "$AABAAABAA$" y $w =$ "$ABAA$".

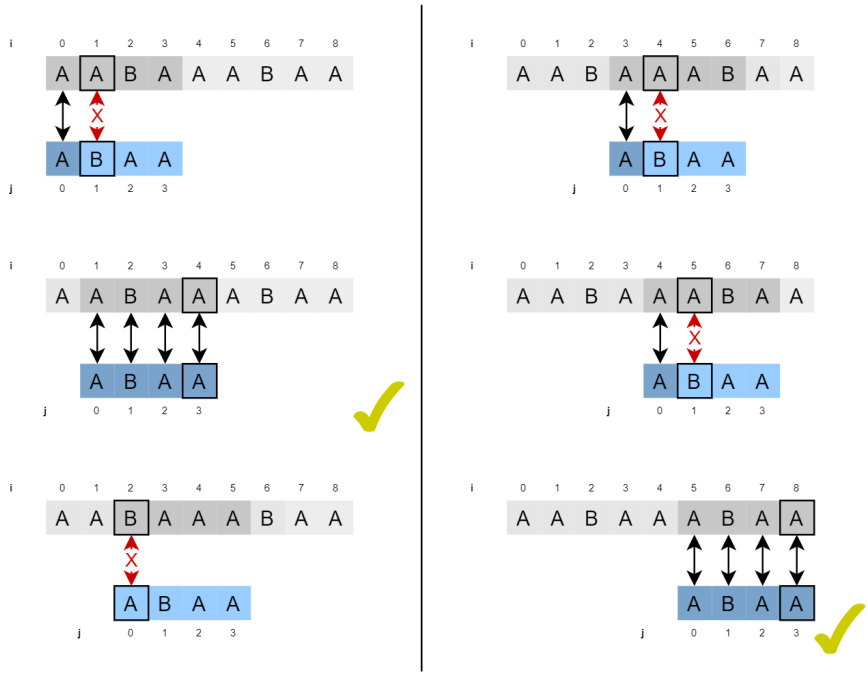

Figura 2.47: Ejemplo del algoritmo de reconocimiento de patrones en cadena por fuerza bruta.

2.6.2. Algoritmo Knuth-Morris-Pratt

El algoritmo de búsqueda Knuth-Morris-Pratt, también llamado KMP, posee una ventaja sobre una búsqueda por fuerza bruta cuando los posibles caracteres en las cadenas tienen una alta probabilidad de repetirse. [36]

Cuando ocurre una disparidad, w tiene suficiente información como para determinar dónde podría iniciar la siguiente coincidencia, efectivamente saltando la re-examinación de caracteres previamente coincididos. Específicamente, si encontramos una coincidencia parcial de los primeros j caracteres empezando en la posición i, sabemos qué caracteres hay en las posiciones i a $i+j-1$, y podemos usar este hecho para evitar probar caracteres que ya sabemos que pueden coincidir. [37]

Definimos a un prefijo de w como una subcadena de w que empieza en $w[0]$, y a un sufijo como una subcadena de w que termina en $w[m-1]$. Decimos que un prefijo o un sufijo de w es propio cuando éste tiene una longitud menor a w. Un caso especial es la cadena vacía, ϵ, la cual se considera un prefijo propio y un sufijo propio de cualquier otra cadena.

Es posible hacer un preprocesamiento de w construyendo un arreglo auxiliar de tamaño m al que llamaremos LPS. Para cada $j \in \{0, \ldots, m-1\}$, $LPS[j]$ contiene la longitud del prefijo propio de de $w[0..j]$ que es también un sufijo de $w[0..j]$.

Por ejemplo, supongamos S y w del ejemplo anterior. Obtenemos LPS de la siguiente manera:

1. Para $j = 0$, debemos encontrar un prefijo propio de la subcadena "A" que sea también un sufijo de sí misma. Pero el único prefijo propio de "A" es ϵ, por lo que $LPS[0] = |\epsilon| = 0$.

2. Para $j = 1$, "AB" tiene dos prefijos propios, "A" y ϵ, pero "A" no es un sufijo de "AB", por lo que $LPS[1] = |\epsilon| = 0$.

3. Para $j = 2$, "ABA" tiene dos prefijos propios, "A" y ϵ, y ambos son sufijos de "ABA", por lo que tomamos la longitud del más grande: $LPS[2] = |\text{"}A\text{"}| = 1$.

4. Para $j = 3$, "$ABAA$" tiene dos prefijos propios, "A" y ϵ, y ambos son sufijos de "$ABAA$", por lo que tomamos el más grande: $LPS[3] = |\text{"}A\text{"}| = 1$.

De tal manera que $LPS = [0, 0, 1, 1]$. Esto nos asegura que, al suceder una disparidad, los j caracteres ya coincididos forman un prefijo propio de longitud $LPS[j-1]$, y por tanto, puede evitarse su revisión en una siguiente iteración. [38]

Es importante mencionar que la creación de LPS puede realizarse también usando las longitudes de los sufijos propios de de $w[0..j]$ que también sean prefijos de $w[0..j]$. Ambos enfoques son equivalentes.

La figura 2.48 muestra la ejecución de este algoritmo con las cadenas mencionadas. A diferencia del algoritmo de fuerza bruta, las optimizaciones en las comparaciones, marcadas en color gris claro, permiten a KMP tener una complejidad temporal lineal en la longitud de w y S: $O(n+m)$. Naturalmente, la memoria necesaria para almacenar LPS provoca que la complejidad espacial aumente de $O(1)$ a $O(m)$, pero si w es constante, LPS puede ser preprocesado una sola vez y nunca necesitar espacio extra en futuras ejecuciones.

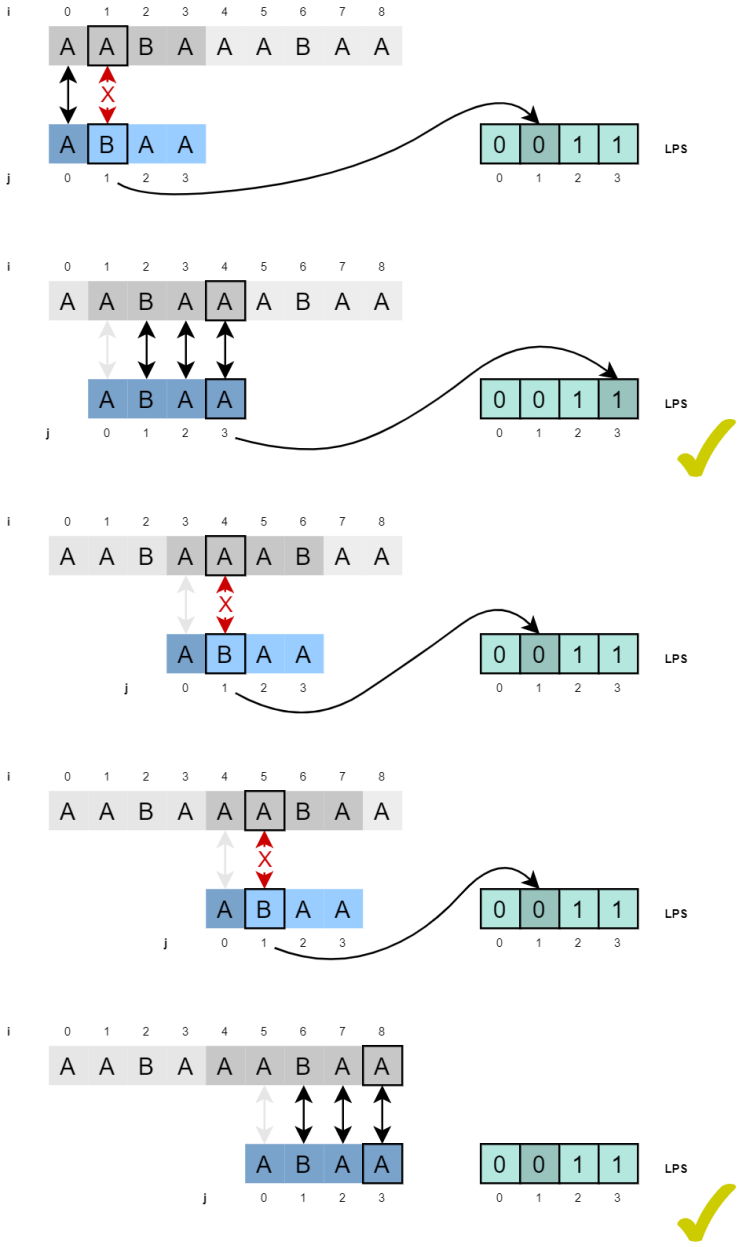

Figura 2.48: Ejemplo del algoritmo KMP.

2.6.3. Algoritmo Aho-Corasick

El algoritmo de Aho-Corasick es una generalización de KMP, en el que es posible buscar un conjunto de patrones simultáneamente. Éste conjunto, al que llamaremos diccionario, se preprocesa en una estructura basada en un trie.

Un trie es un tipo de árbol de búsqueda eficiente en la recuperación de subcadenas a partir de un diccionario. La información de una determinada subcadena esta codificada en un nodo, pero no es directamente accesible mediante éste, sino mediante los enlaces en la rama a la que pertenece. Para poder recuperar una subcadena, es necesario recorrer el trie por profundidad, siguiendo los enlaces entre nodos que representan cada uno de los caracteres de la subcadena a localizar. [39]

Un nodo puede tener tantos hijos como caracteres haya en el alfabeto. Todos estos hijos tienen un prefijo común de la cadena asociada con el nodo padre, siendo el nodo raíz asociado con ϵ. Además, cada nodo puede catalogarse como terminal, lo que implica que la rama a la que pertenece forma una cadena del diccionario. [40]

La figura 2.49 muestra la creación de un trie a partir del diccionario [“$ABAB$”, “$ABAC$”, “ABC”, “BC”].

Aho-Corasick expande este trie, agregando enlaces de fallo para los nodos, exceptuando la raíz. Para cada nodo x, la rama de la raíz a x representa el prefijo común u de alguno de los patrones del diccionario. Para crear un enlace de fallo para x, debemos encontrar el sufijo propio de u de mayor longitud que aparezca en el trie mediante un recorrido en profundidad. [41] La figura 2.50 muestra la creación de estos enlaces de fallo para cada uno de los nodos de la estructura anterior.

De esta manera, la estructura está preparada para la ejecución del algoritmo de búsqueda. Para encontrar las coincidencias de los patrones del diccionario en una S dada, debemos recorrer S y procesar el nodo actual, c, acordemente: [39]

1. Sea c el nodo raíz.
2. Para cada $i \in \{0, \ldots, n\}$:
   a) Checamos si c tiene un enlace con $S[i]$:
      - Si lo tiene, movemos c al nodo que apunta dicho enlace.
      - De lo contrario movemos c al nodo que apunta el enlace de fallo de c y repetimos *2a*.
        - Si c no tiene un enlace de fallo, continuamos con la siguiente i.

   b) Si c es un nodo terminal, hemos encontrado una coincidencia en S del patrón con rama de la raíz a c.

La figura 2.51 muestra la ejecución de este algoritmo para analizar una $S =$ “$ABAABABBC$” con la estructura del ejemplo anterior. Aho-Corasick utiliza una complejidad temporal $O(n + m_{max} + z)$, donde m_{max} es la longitud del patrón más grande del diccionario y z es el número de ocurrencias del diccionario en S. La complejidad espacial se describe por $O(km_{max})$, donde k es el número de patrones en el diccionario. No obstante, si el diccionario es constante, sólo es necesario preprocesar la estructura una única vez. [42]

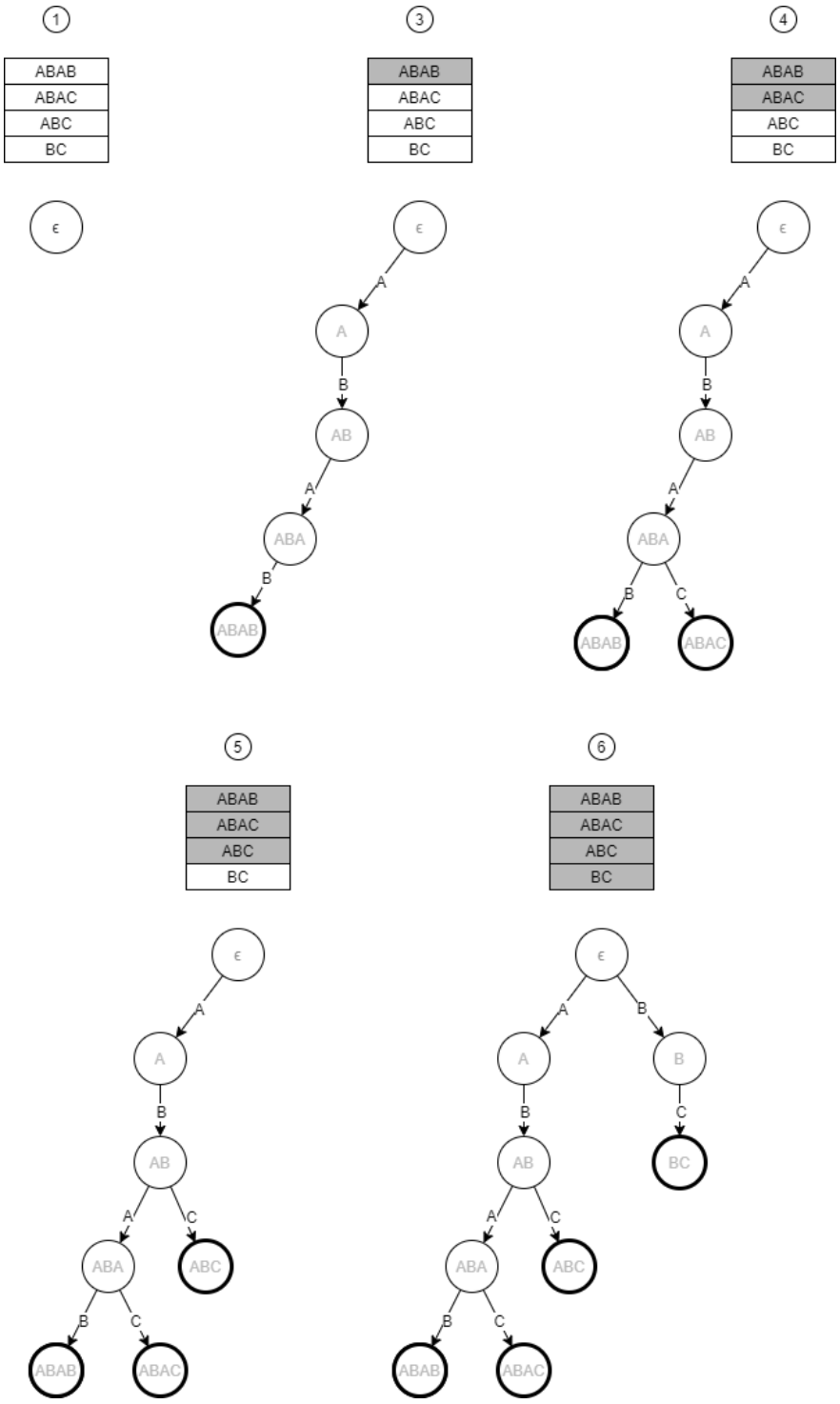

Figura 2.49: Creación de un trie.

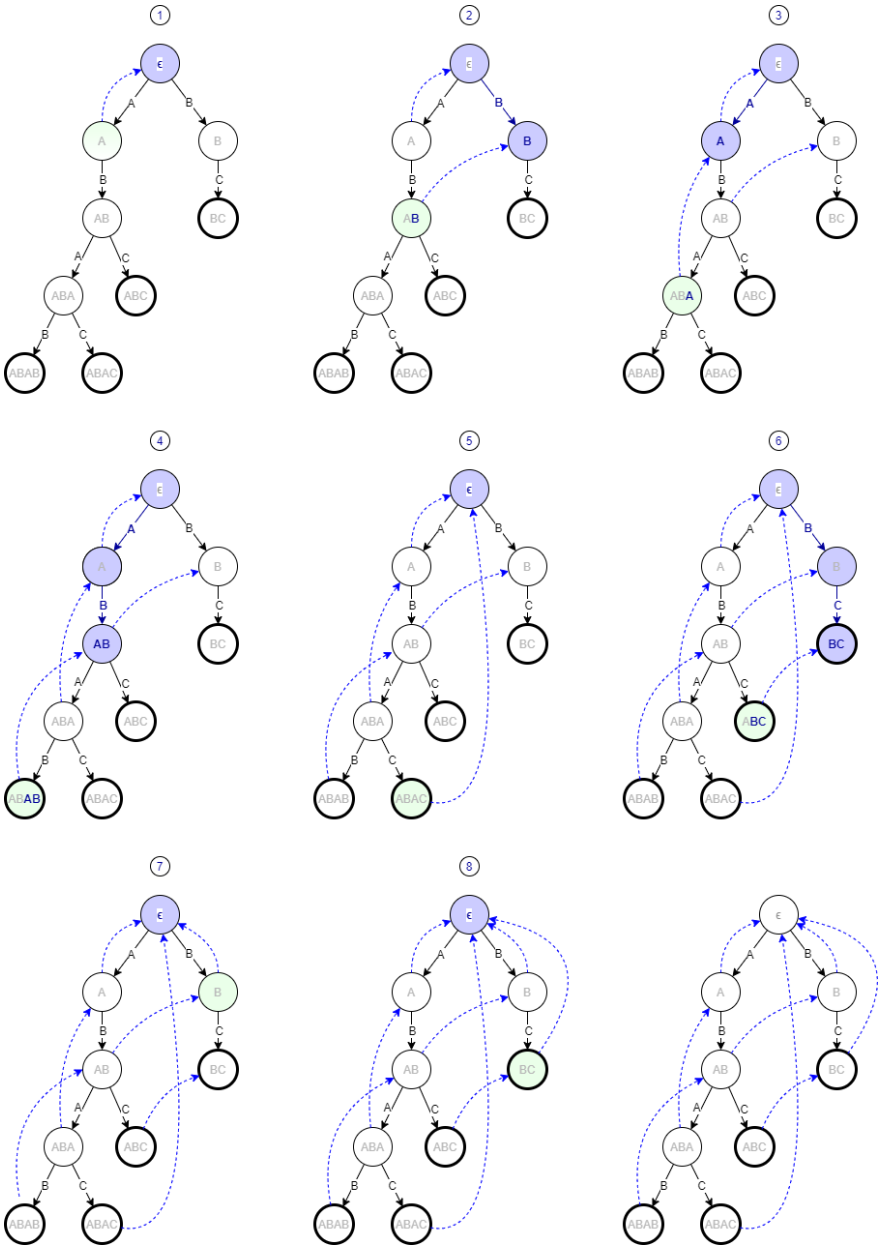

Figura 2.50: Creación de enlaces de fallo para un trie en Aho-Corasick.

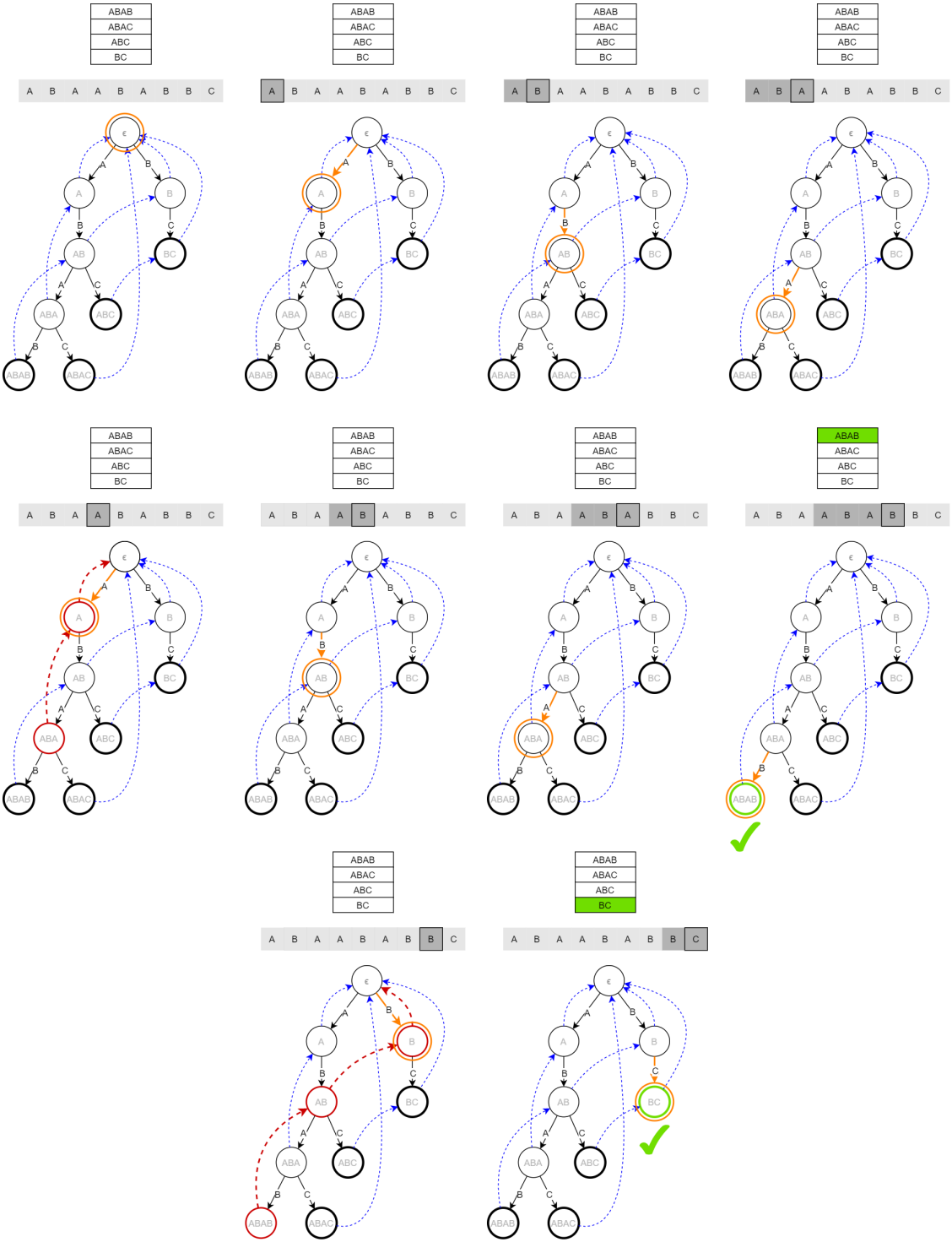

Figura 2.51: Ejecución del algoritmo Aho-Corasick.

2.6.4. Algoritmo Rabin-Karp

El algoritmo de Rabin-Karp es un algoritmo de búsqueda de patrones que utiliza una función rolling hash para filtrar tempranamente porciones del texto que no coincidan con el patrón, y después checa coincidencias en las posiciones restantes.

Para fines del algoritmo, un hash se define como una función inyectiva $h : \Sigma^+ \to X \subseteq \mathbb{Z}$, es decir, una función que asigna a cualquier cadena un valor numérico, llamado valor hash. Al ser una función inyectiva, se sigue que, para $w_1, w_2 \in \Sigma^+$, si $w_1 = w_2$, entonces $h(w_1) = h(w_2)$. Sin embargo, si dos cadenas tienen el mismo valor hash, no se puede concluir que ambas cadenas sean iguales. Una función hash bien diseñada debería evitar lo más posible que dos cadenas diferentes tengan el mismo valor hash.

Es fácil notar que una función hash requiere procesar una cadena por completo para poder computar su valor hash, lo cual puede resultar altamente costoso para cadenas largas. Un rolling hash, también llamado hash recursivo o rolling checksum, es una función hash en la que la entrada se procesa en una ventana que se mueve a lo largo de la entrada. Conforme la ventana se mueve, el nuevo valor hash se calcula utilizando el valor hash de la ventana anterior y el nuevo caracter añadido a la ventana, evitando la necesidad de reprocesar toda la ventana de nuevo. [39]

Rabin-Karp aprovecha esta propiedad para procesar, en cada posición i de S, un valor hash de la subcadena $s_i = S[i]...S[i+m]$. Si $h(s_i) = h(w)$, procedemos a hacer una comparación explicita para encontrar una coincidencia. Para que el algoritmo funcione correctamente, el rolling hash debe ser elegido de forma que se reduzca la probabilidad de producir falsos positivos, es decir, reducir lo más posible que la probabilidad de que, después de comparar los valores hash, s_i no sea igual a w. Para la iteración $i + 1$, el rolling hash sólo necesita $h(s_i)$ y $S[i + 1]$ para procesar $h(s_{i+1})$, evitando así recalcular toda s_{i+1}. [43]

Uno de los rolling hash más usados por su eficiencia y simplicidad es el Rabin Fingerprint. Sean $a = |\Sigma|$ y $r : \Sigma \to \mathbb{Z}$ una función biyectiva de mapeo de caracteres de Σ a enteros. [44] Para calcular el valor hash de una cadena $x \in \Sigma^+$, con $|x| = k$, Rabin Fingerprint se define como

$$h_F(x) = r(x[0])a^{k-1} + r(x[1])a^{k-2} + \ldots + r(x[k-1])a^0 = \sum_{j=0}^{k-1} r(x[j])a^{k-j-1}$$

.

Por simplicidad, definiremos r trivialmente ordenando Σ. Por ejemplo, para el alfabeto tradicional $'A', \ldots, 'Z'$, $r('A') = 1, r('B'), \ldots r('Z') = 26$

Podemos mencionar propiedades interesantes para un rolling hash. En concreto, para el Rabin Fingerprint:

- Sea $x = x[0] \ldots x[j-1]x[j]x[j+1] \ldots x[k-1]$. Entonces $h_F(x[0] \ldots x[j-1]x[j+1] \ldots x[k-1]) = h_F(x) - r(x[j])a^{k-j-1}$.
- Sean $x = x[0] \ldots x[k-1]$ y $c \in \Sigma$. Entonces $h_F(x[0] \ldots x[k-1]c) = ah_F(x) + r(c)$.

Podemos calcular fácilmente el valor hash de la subcadena de longitud k en la posición i para un texto S, si tenemos el valor hash de la subcadena en la posición $i - 1$. El nuevo valor estará dado por:

$$h_F(S[i..i+k]) = a \cdot [h_F(S[i-1..i-1+k]) - r(S[i-1])a^{k-1}] + r(S[i])$$

Utilizaremos Rabin Fingerprint para localizar ocurrencias de un patrón w, con $|w| = m$, en el texto S, con una ventana de tamaño $k = m$. Sean $p = h_F(w)$ y $t_i = h_F(S[i..i+k])$. Seguiremos los siguientes pasos:

1. Calculamos p y t_0.
2. Para $i \in 0, \ldots, n-m-1$:
   a) Si $t_i = p$, checamos si $S[i..i+m] = w$. Si son iguales, encontramos una coincidencia.
   b) Procesamos $t_{i+1} = a \cdot [t_i - r(S[i])a^{m-1}] + r(S[i+1])$.

La figura 2.52 muestra el análisis del texto $S =$ "$CABABAC$" con un patrón $w =$ "ABA". Se supone un alfabeto tradicional $'A', \ldots, 'Z'$, por lo que $a = 26$, y una r trivial.

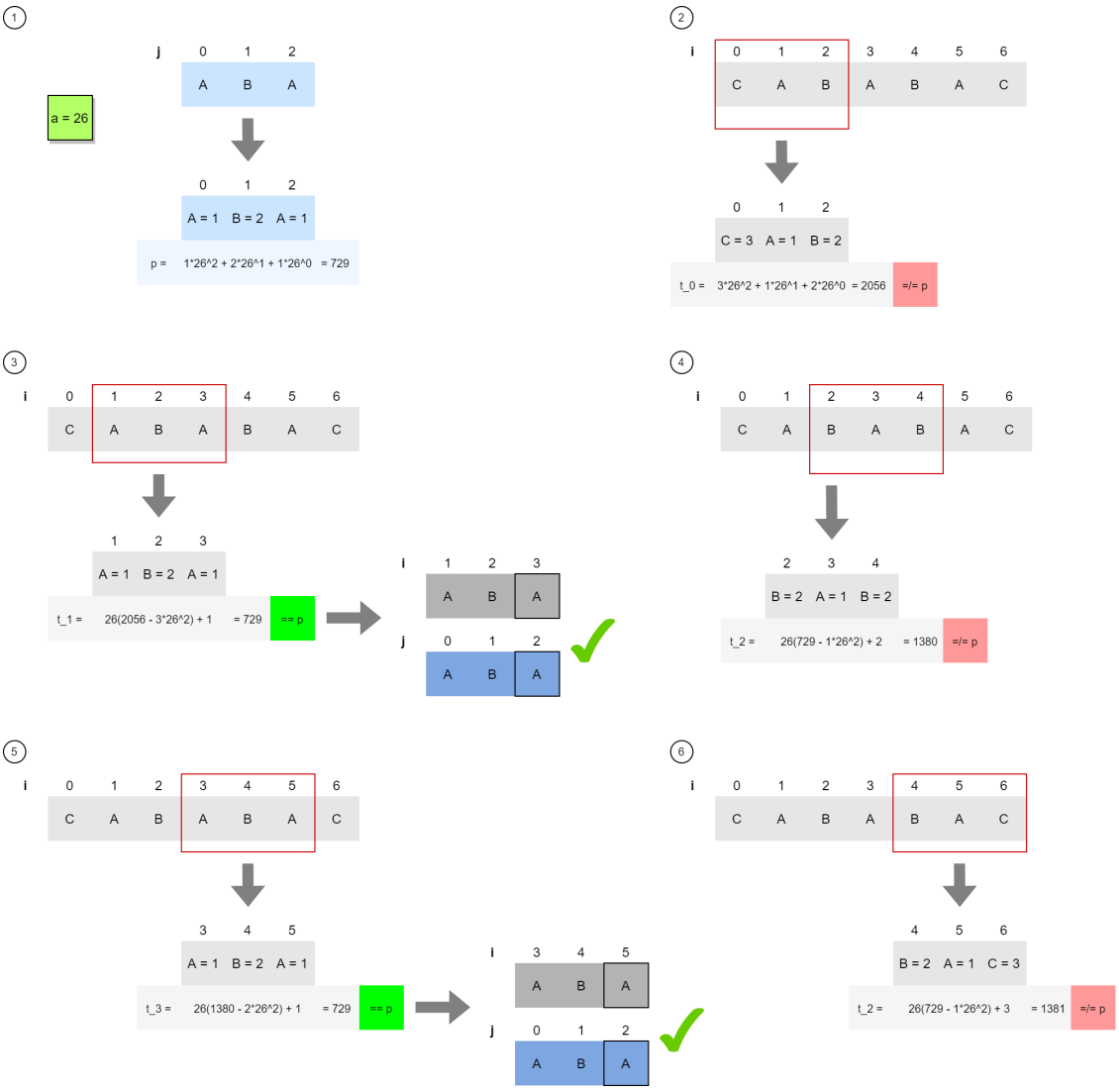

Figura 2.52: Ejecución del algoritmo Rabin-Karp.

A pesar de que Rabin-Karp posee una complejidad $O(nm)$ para su peor caso, similar a un algoritmo por fuerza bruta, su mejor caso y su promedio son de $O(n+m)$. El peor caso sucede cuando todas las subcadenas de tamaño m de S comparten el mismo valor hash con w. Al seleccionar un rolling hash que evite colisiones de valores, la probabilidad de que esto suceda se reduce. La complejidad espacial es $O(1)$, por lo que es un algoritmo eficiente en memoria. [45]

Para localizar ocurrencias en un diccionario de patrones de longitud m_{max}, basta con obtener los valores hash de cada patrón al principio. Así, al procesar el valor hash de cada subcadena de S, podemos revisar si coincide con alguno de los valores del diccionario.

Capítulo 3

Arquitectura

3.1. Metodología

Se decidió el uso de la metodología de desarrollo iterativo e incremental para la construcción del simulador de colisiones. La estructura modular del sistema a desarrollar, la condición poco cambiante del proyecto y el hecho de que el simulador tiene como público objetivo a los investigadores en el área de ACs, permite que la metodología sea fácil de integrar a las necesidades de este trabajo. Adicionalmente, la composición de módulos funcionales para cada iteración permitirá un avance constante en el estudio del comportamiento de los ACEs y la regla 110. La figura 3.1 ilustra las iteraciones del proyecto.

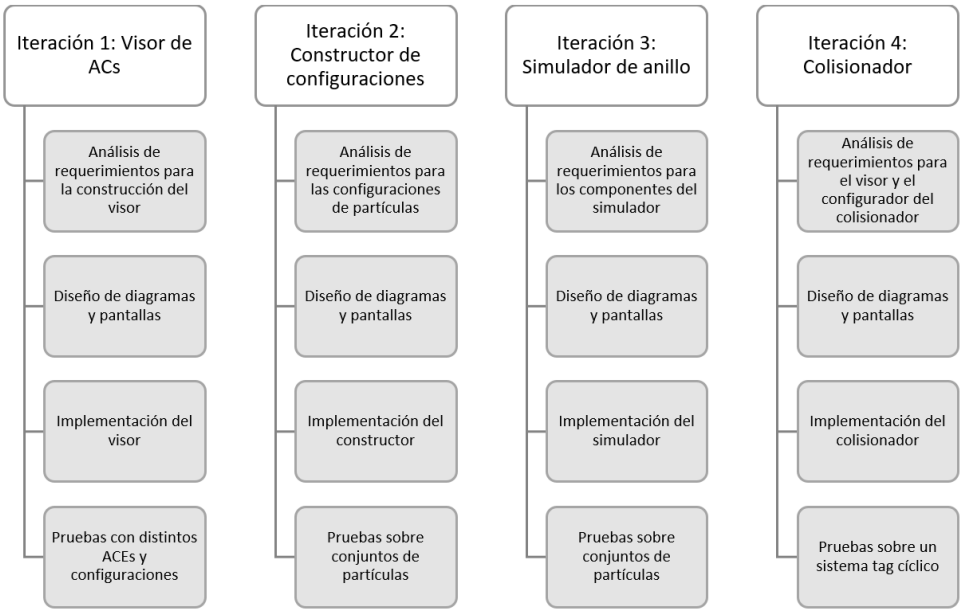

Figura 3.1: Metodología iterativa e incremental adaptada al desarrollo del simulador de colisiones.

La primera iteración se enfocará en construir un visor genérico para ACEs. Éste se utilizará en un primer análisis sobre el comportamiento dinámico del espacio de evoluciones de un ACE, y en concreto de la regla 110.

La segunda iteración constituirá un constructor de configuraciones, una funcionalidad específica para la regla 110. El sistema tendrá un módulo de procesamiento para partículas específicas de la regla 110, el cual será utilizado por el visor como un traductor y validador de sintaxis para conjuntos de partículas.

La tercera iteración se encargará de implementar la funcionalidad de anillo para los ACEs del visor. De igual manera, el módulo implementará un sistema eficiente para el filtrado de éter para facilitar la distinción de partículas en el visor.

La cuarta y última iteración constituirá el simulador de colisiones, que se encargará de orquestar el comportamiento de inyección de partículas de los anillos laterales para su posterior colisión en el anillo central.

Utilizando el sistema de colisiones, se realizará un análisis del comportamiento del espacio de evoluciones de la regla 110. Con estos resultados, se investigará la factibilidad de la regla 110 como un modelo de computación universal mediante la simulación de sistemas tag cíclicos.

3.2. Cronograma

La figura 3.2 muestra el cronograma planeado para el desarrollo de este proyecto.

Actividad		FEB	MAR	ABR	MAY	JUN	JUL	AGO	SEP	OCT	NOV	DIC
Visor de ACs	Análisis											
	Diseño											
	Implementación											
	Pruebas											
Constructor de configuraciones	Análisis											
	Diseño											
	Implementación											
	Pruebas											
Simulador de anillo	Análisis											
	Diseño											
	Implementación											
	Pruebas											
Colisionador	Análisis											
	Diseño											
	Implementación											
	Pruebas											
Presentación TT-I												
Implementación de sistema computable	Estudio de partículas											
	Estudio de conjuntos de partículas											
	Simulación de colisiones											
	Descripción de operaciones por colisiones											
	Análisis de resultados											
Presentación TT-II												

Figura 3.2: Cronograma con las fechas estimadas para el proyecto.

Capítulo 4

Iteración 1: Visor de ACEs

4.1. Análisis

A continuación se describirá el ciclo de desarrollo de la primera iteración del sistema: el visor de ACEs.

4.1.1. Especificación de requerimientos

Requerimientos funcionales

Id	Requerimiento	Descripción	Prioridad
RF1-01	Configurar ACEs	El usuario construye el ACE que desea visualizar: regla elemental a utilizar, condición inicial de la cinta y tamaño de la cinta. De igual manera, configura las opciones de visualización: tamaño por celda, iteraciones del ACE a mostrar, y color de los estados.	Alta
RF1-02	Navegar el espacio de evoluciones	El visor permite al usuario avanzar en las iteraciones del ACE, así como reiniciar desde la condición inicial o resetear por completo el ACE con una condición inicial aleatoria, así como guardar el espacio actual en un archivo de imagen.	Alta

Cuadro 4.1: Requerimientos funcionales para la iteración 1.

Requerimientos no funcionales

Id	Requerimiento	Descripción
RNF1-01	Usabilidad	La aplicación es intuitiva y entendible para cualquier persona con conocimiento de ACEs.
RNF1-02	Eficiencia	El visor muestra el espacio de evoluciones en un tiempo y con una cantidad de recursos de memoria proporcionales al tamaño del espacio seleccionado. El sistema libera los recursos que ya no estén en uso.

Cuadro 4.2: Requerimientos no funcionales para la iteración 1.

4.2. Diseño

Para el desarrollo de este trabajo, se decidió utilizar el lenguaje de programación *C++*. Su versatilidad en el manejo de memoria lo hace ideal para la implementación de los ACEs, entidades inherentemente pesadas espacialmente.

Para el dibujo de las interfaces gráficas, se hará uso de la biblioteca de desarrollo de GUI *wxWidgets*.

Finalmente, el desarrollo de la aplicación se llevará a cabo en el IDE *Visual Studio 2019*, utilizando un control de versiones en *Git* hacia un repositorio en *Github*.

4.2.1. Diagrama de casos de uso

La figura 4.1 muestra el diagrama de casos de uso de la iteración 1. El sistema consta de dos módulos:

1. *Configurar ACEs*: El usuario realiza la configuración del ACE que desea visualizar: la regla elemental que se aplicará, el tamaño de la cinta que éste utilizará, y la condición inicial de la cinta, en conjuntos de 0s y 1s. Si así lo desea, el usuario puede seleccionar una condición inicial aleatoria, o ajustar el número de celdas al tamaño de la condición inicial que haya escrito. De igual manera, el usuario puede cambiar las opciones de visualización: el tamaño de cada celda (en pixeles), el número de iteraciones del ACE a mostrar, y los colores de cada respectivo valor de las celdas. Al presionar el botón de aceptar, el sistema valida las opciones, crea un controlador de ACE y abre una ventana de navegación. El usuario puede crear tantas ventanas de navegación como desee.

2. *Navegar el espacio de evoluciones*: El navegador hace uso del ACE creado para mostrar el espacio de evoluciones. Empezando por la condición inicial, el sistema aplica la regla seleccionada la cantidad de veces que se haya indicado, y pintará cada cinta resultante debajo de la anterior. El usuario puede elegir mediante los botones de la barra de herramientas, o mediante comandos del teclado, diferentes opciones:

   - Avanzar al siguiente espacio de evoluciones, procesando las siguientes iteraciones del ACE.
   - Reiniciar el espacio, comenzando desde la condición inicial.
   - Resetear el espacio con una condición inicial aleatoria.
   - Guardar el espacio actual en un archivo de imagen.

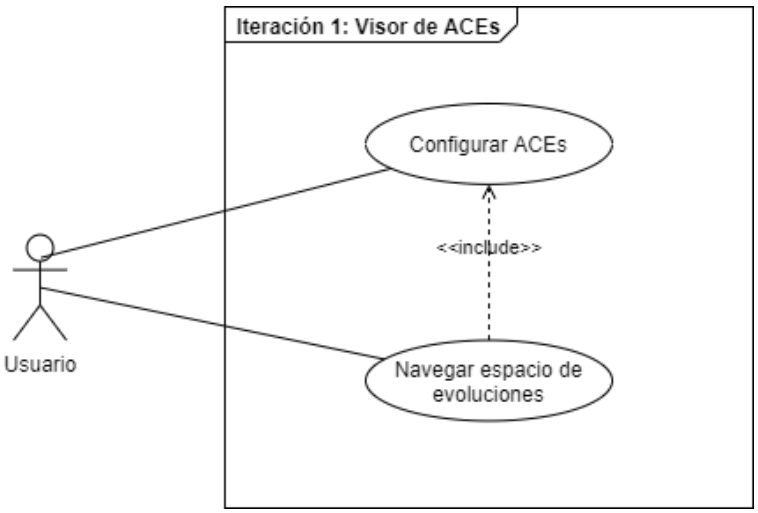

Figura 4.1: Diagrama de caso de uso para la iteración 1.

4.2.2. Diagramas de actividades

La figura 4.2 muestra el diagrama de actividades correspondiente al caso de uso *Configurar ACEs*.

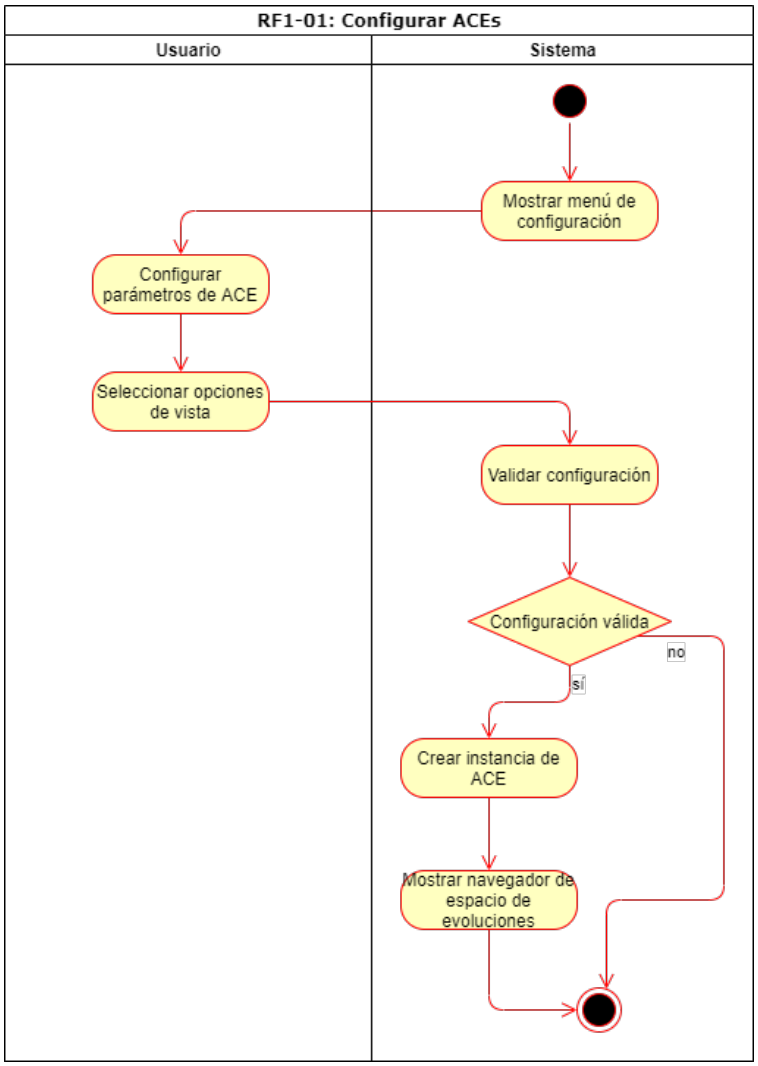

Figura 4.2: Diagrama de actividades del caso de uso *Configurar ACEs*.

La figura 4.3 muestra el diagrama de actividades correspondiente al caso de uso *Navegar espacio de evoluciones*.

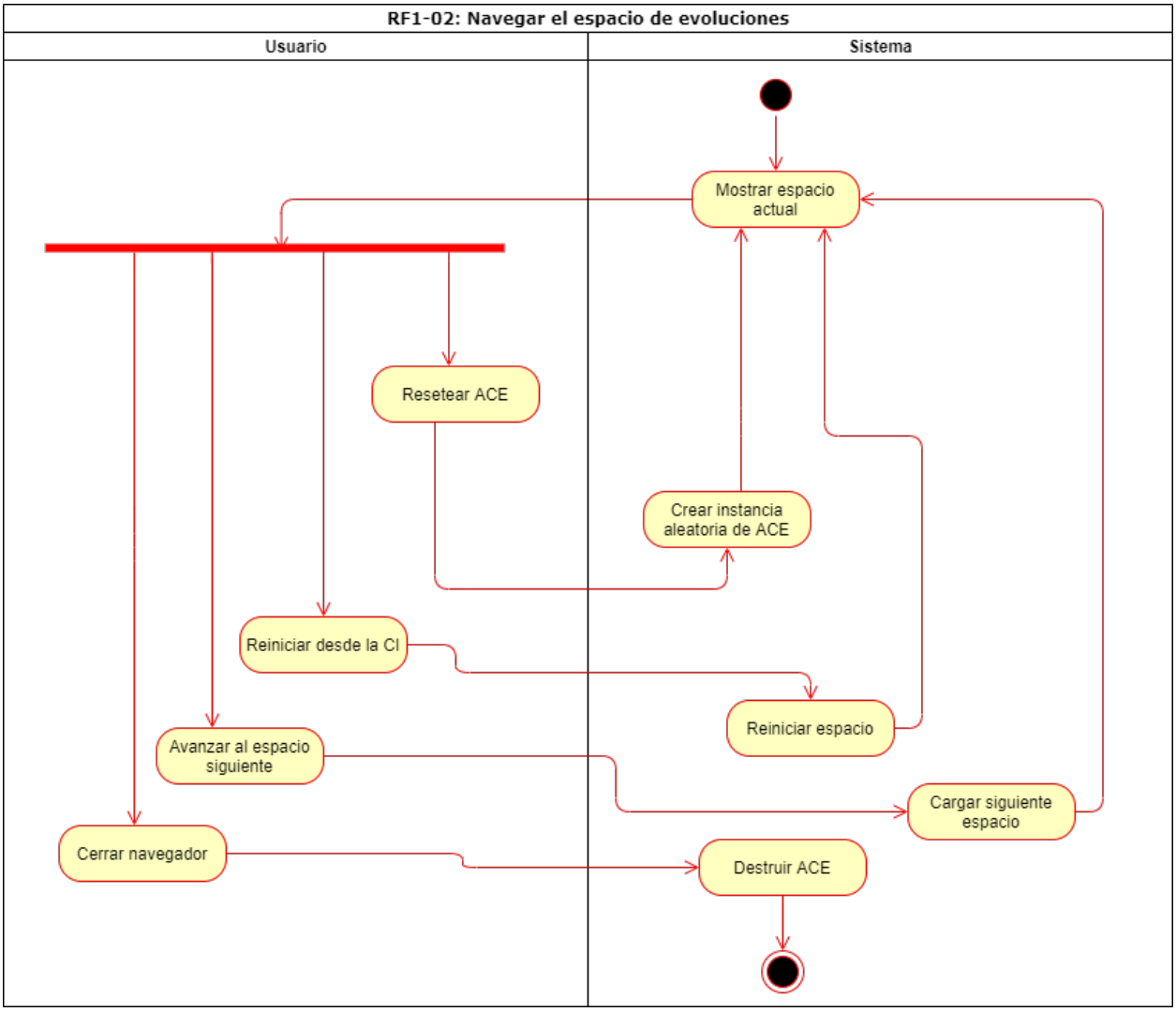

Figura 4.3: Diagrama de actividades del caso de uso *Navegar espacio de evoluciones*.

4.2.3. Diagrama de clases

La figura 4.4 muestra el diagrama de clases para la iteración 1.

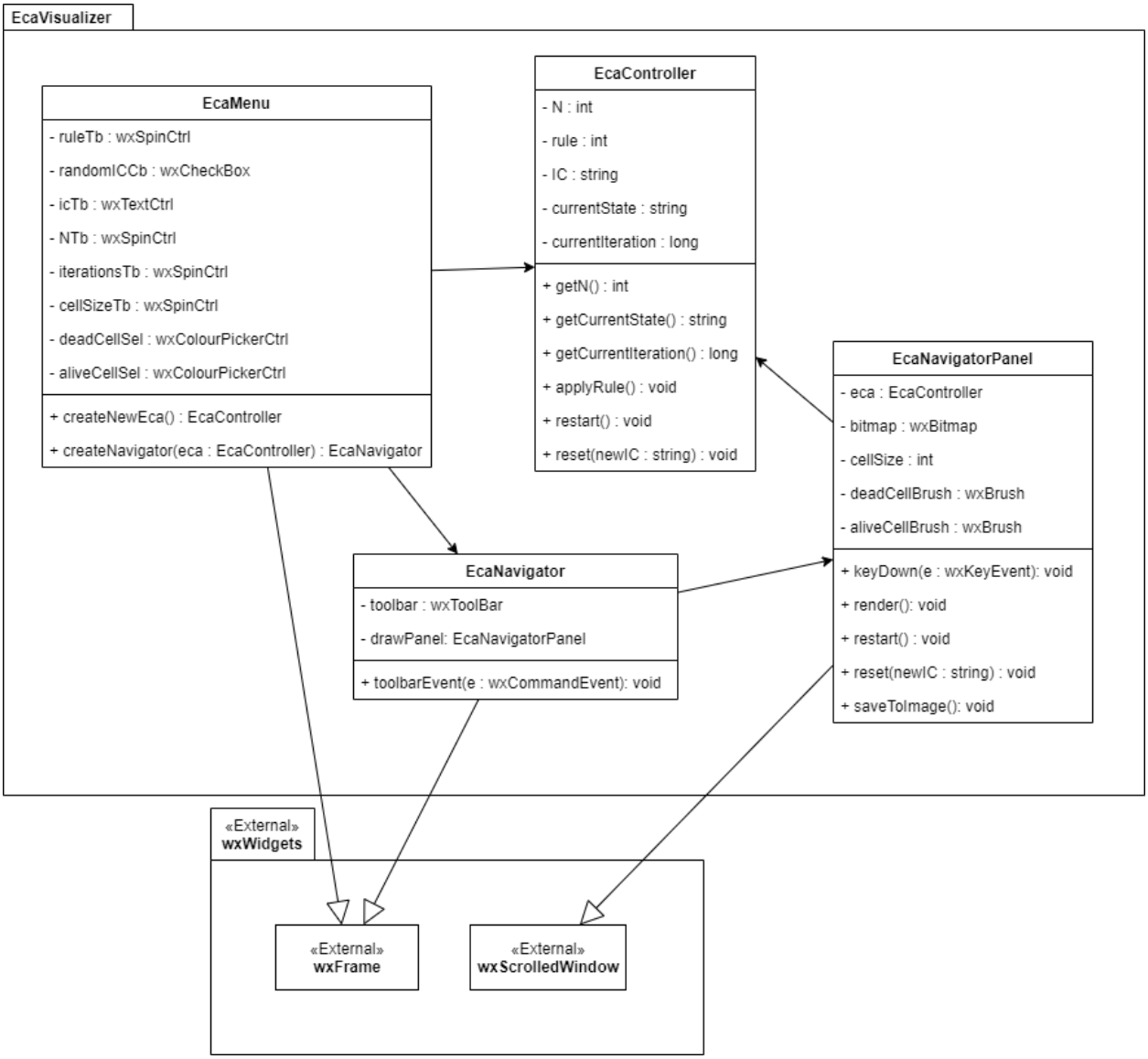

Figura 4.4: Diagrama de clases para la iteración 1.

EcaMenu, *EcaNavigator* y *EcaNavigatorPanel* se encargan de las vistas, y *EcaController* de la lógica del ACE.

EcaMenu y *EcaNavigator* heredan de *wxFrame*, y *EcaNavigatorPanel* de *wxScrolledWindow*. *wxFrame* y *wxScrolledWindow* provienen de la biblioteca externa **wxWidgets**. Además, las 3 clases nativas usan diferentes controles de **wxWidgets**, pero se omiten del diagrama de clases por motivos de visibilidad.

4.3. Implementación y pruebas

La figura 4.5 muestra la captura de pantalla del menú de configuración de ACE.

La figura 4.6 muestra la captura de pantalla del navegador, en el cual se dibujaron las primeras 350 iteraciones de un ACE usando la regla elemental 110, con $N = 400$ y una condición inicial $00\ldots01$.

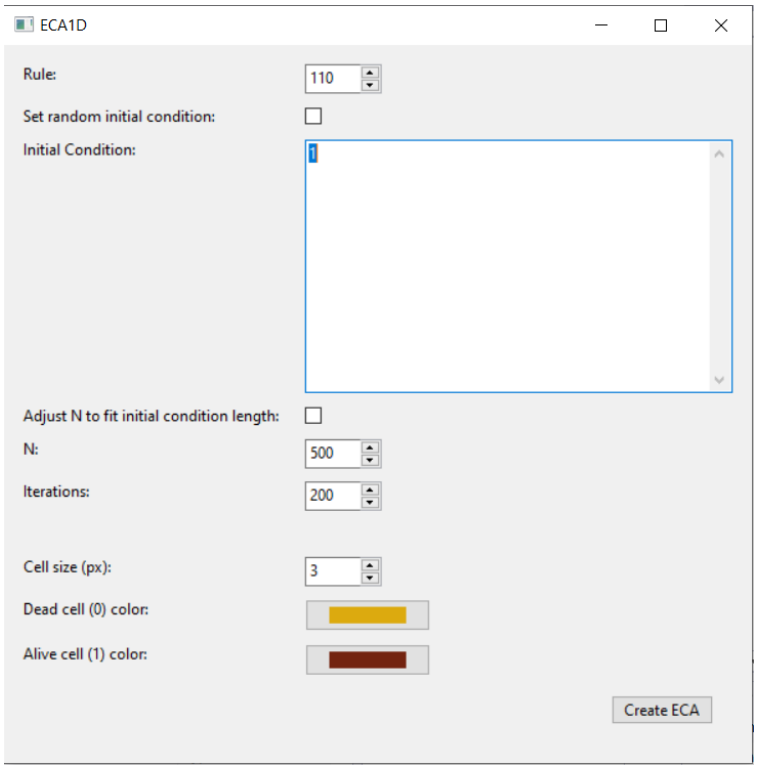

Figura 4.5: Captura de pantalla del menú de configuración de ACE.

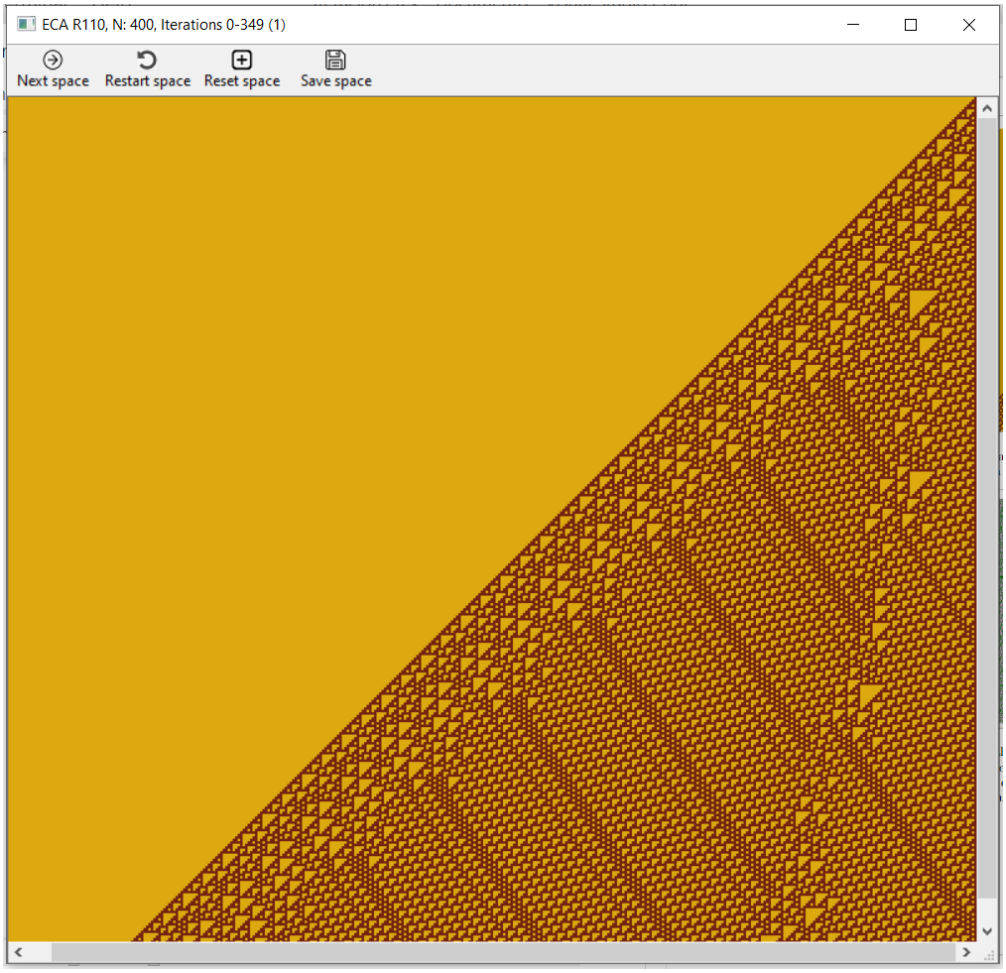

Figura 4.6: Captura de pantalla del navegador, mostrando las primeras 350 iteraciones de la regla 110 con $N = 400$ y condición inicial $00\ldots01$.

La figura 4.7 muestra las primeras 650 iteraciones de otro ACE, la regla 30, con $N = 2000$ y una condición inicial aleatoria. Además, se usaron colores alternos para los valores de las celdas. Es de notarse que, al ser un espacio muy grande, es necesario desplazarse mediante scrollers para poder visualizar diferentes áreas. La figura 4.8 muestra el resultado de utilizar el botón de guardado, un archivo PNG con el espacio de evoluciones completo para su posterior estudio.

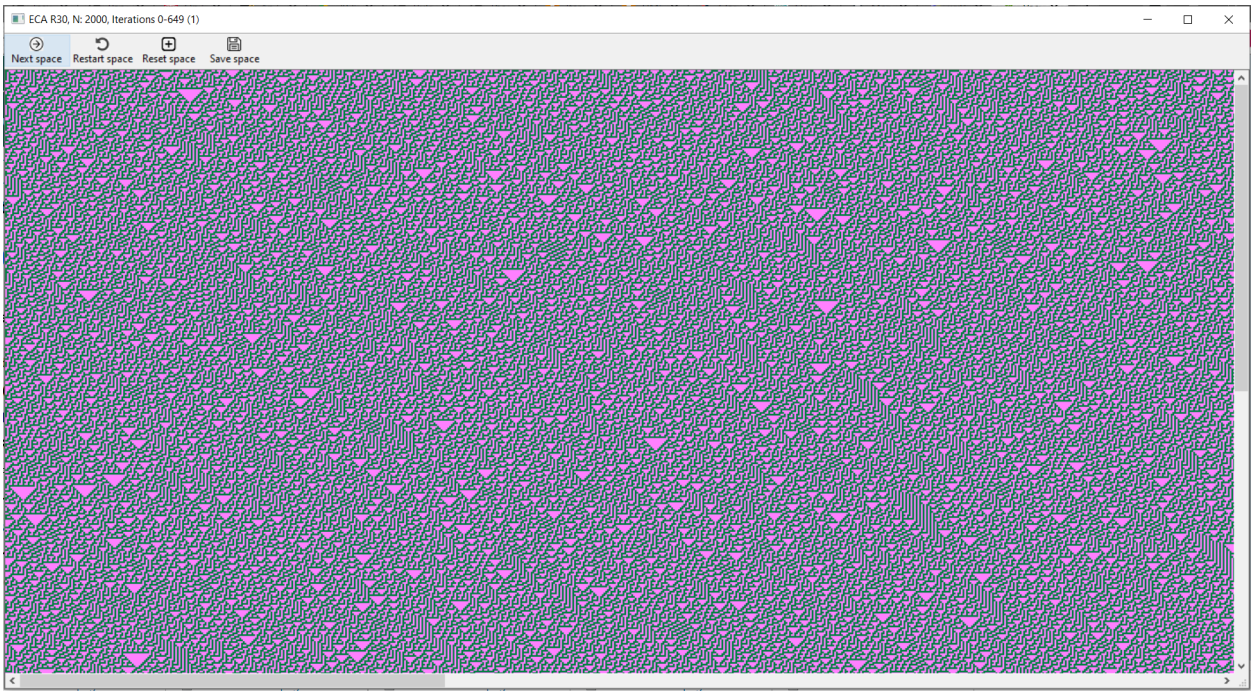

Figura 4.7: Captura de pantalla del navegador, con opciones de vista no predefinidas, mostrando las primeras 650 iteraciones de la regla 30 con $N = 2000$ y condición inicial aleatoria.

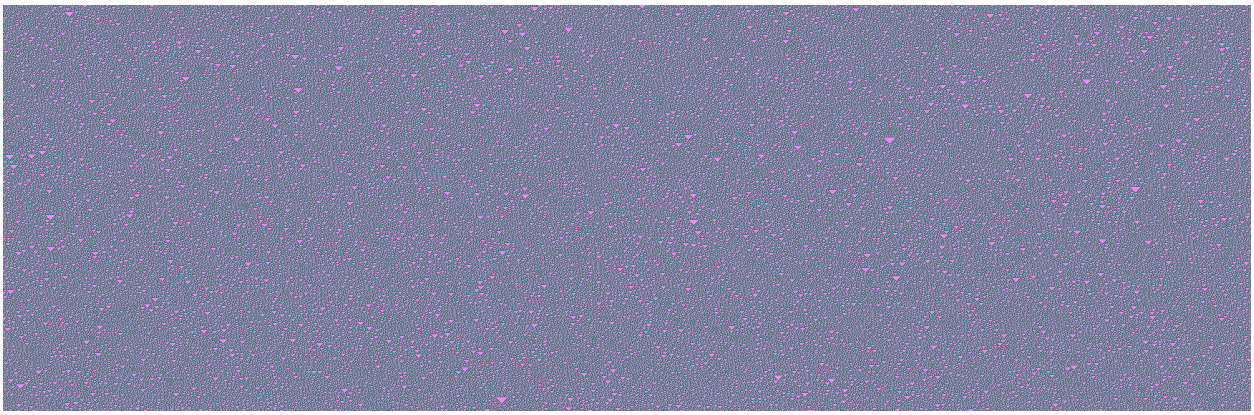

Figura 4.8: Espacio de evoluciones completo del ACE de la figura 4.7 a partir del archivo de imagen descargado del navegador.

Capítulo 5

Iteración 2: Constructor de configuraciones

5.1. Análisis

A continuación se describirá el ciclo de desarrollo de la segunda iteración del sistema: el constructor de configuraciones.

5.1.1. Especificación de requerimientos

Requerimientos funcionales

Id	Requerimiento	Descripción	Prioridad
RF2-01	Traducir expresión R110	El usuario ingresa una cadena al sistema. El traductor checa la validez de esta cadena como una expresión de la regla 110: primero léxicamente como tokens válidos, luego sintácticamente para cada token, y después semánticamente con los compuestos y sus fases. Si la expresión es válida, es traducida a una cadena binaria. De otro modo, se arroja la excepción pertinente al usuario.	Alta

Cuadro 5.1: Requerimientos funcionales para la iteración 2.

Requerimientos no funcionales

Id	Requerimiento	Descripción
RNF2-01	Usabilidad	El sistema es claro para notificar problemas en el parseo de las expresiones.
RNF2-02	Eficiencia	El sistema valida y traduce expresiones en un tiempo mínimo.

Cuadro 5.2: Requermientos no funcionales para la iteración 2.

5.2. Diseño

5.2.1. Diagrama de casos de uso

La figura 5.1 muestra el diagrama de casos de uso de la iteración 2, la cual posee un único módulo:

1. *Traducir expresión R110.* El usuario provee una cadena que contiene una expresión válida en la regla 110. Esta expresión puede ser:

- $(0|1)+$: cadena binaria.
- e: éter fase f_1_1 de la regla 110 (11111000100110).
- $\boldsymbol{ID}(\boldsymbol{PHASE}, f\boldsymbol{p}\_1)$: compuesto (una partícula o un conjunto de partículas), donde $\boldsymbol{ID}$ es el identificador del compuesto, $\boldsymbol{PHASE}$ es una fase válida del compuesto, y $\boldsymbol{p} \in \{1, 2, 3, 4\}$.
- Secuencia de cadenas binarias, éters y/o compuestos, separados con $-$.
- $\boldsymbol{k}\{\boldsymbol{SEQ}\}$: múltiplo de una secuencia, donde $\boldsymbol{k} \in \mathbb{Z}^+$ y $\boldsymbol{SEQ}$ es una secuencia.

El sistema checa la validez de la expresión. De ser correcta, la traduce a una cadena binaria. De otro modo, arroja una excepción de acuerdo al error encontrado:

- Error de parseo: si se suscita un problema léxico durante la obtención de tokens.
- Error de token inválido: si algún token no tiene la sintaxis correcta.
- Error de compuesto no encontrado: si el $\boldsymbol{ID}$ no existe.
- Error de fase no encontrada: si la fase del compuesto, determinada por $\boldsymbol{PHASE}$ y $\boldsymbol{p}$, no existe.

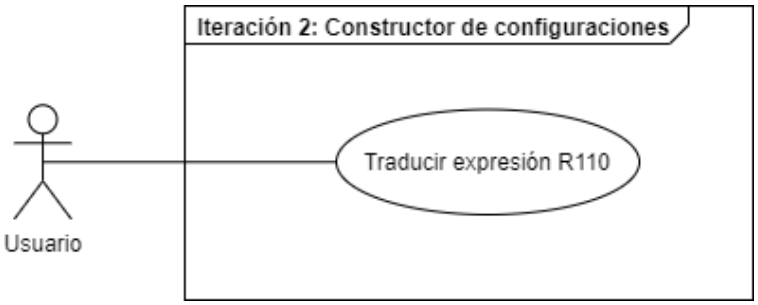

Figura 5.1: Diagrama de caso de uso para la iteración 2.

5.2.2. Diagramas de actividades

La figura 5.2 muestra el diagrama de actividades correspondiente al caso de uso *Traducir expresión R110*.

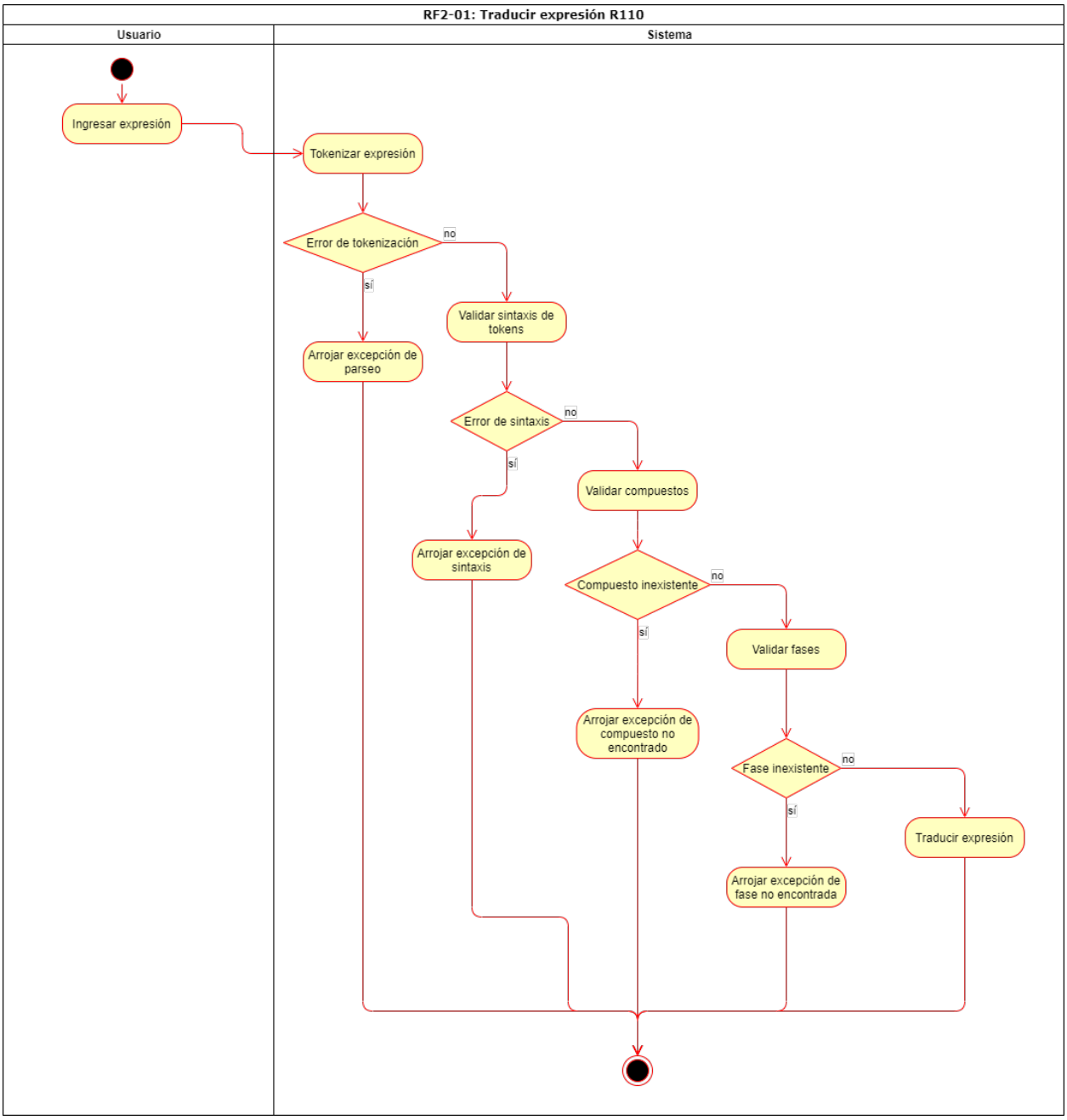

Figura 5.2: Diagrama de actividades del caso de uso *Traducir expresión R110*.

5.2.3. Diagrama de clases

La figura 5.3 muestra el diagrama de clases para la iteración 2.

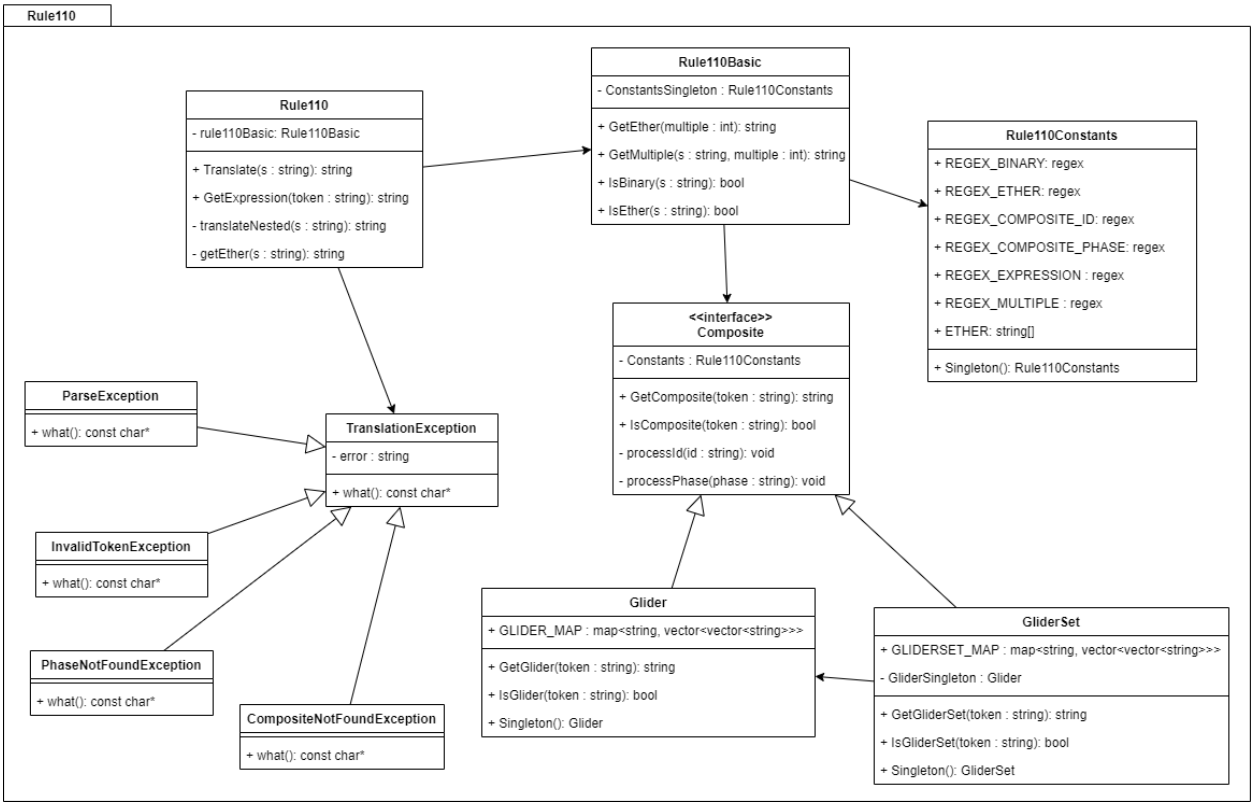

Figura 5.3: Diagrama de clases para la iteración 2.

Rule110 es la clase principal, la cual recibe y tokeniza la expresión original. Hace uso los métodos de utilidades de *Rule110Basic*, la cual a su vez hace uso de singletons de *Rule110Constants* y de los *Composites*. Así mismo, contiene cuatro tipos de *TranslationExceptions*: *ParseException* (para errores en la tokenización), *InvalidTokenException* (para errores sintácticos), *CompositeNotFoundException* (para errores de IDs de compuestos inexistentes), y *PhaseNotFoundException* (para fases del compuesto inexistentes).

Rule110Constants posee las constantes de validación sintáctica de cada expresión.

Composite es una interfaz que ayuda en la interpretación de cada token para su posterior traducción a cadenas binaraias. Dos clases heredan de *Composite*: *Glider* y *GliderSet*. *Glider* es un singleton que contiene todas las traducciones de partículas a cadenas binarias, así como los métodos estáticos necesarios para comprobar su semántica. *GliderSet* es una clase equivalente para los conjuntos de partículas.

Es posible agregar el módulo **Iteración 2 - Constructor De Configuraciones** de forma directa al módulo **Iteración 1 - Visor de ACEs**, simplemente agregando la clase *Rule110* como validador de la cadena ingresada en el área de texto *icTb* antes de crear el ACE y enviarlo a *EcaNavigator*, como se muestra en el diagrama de clases general de la figura 5.4.

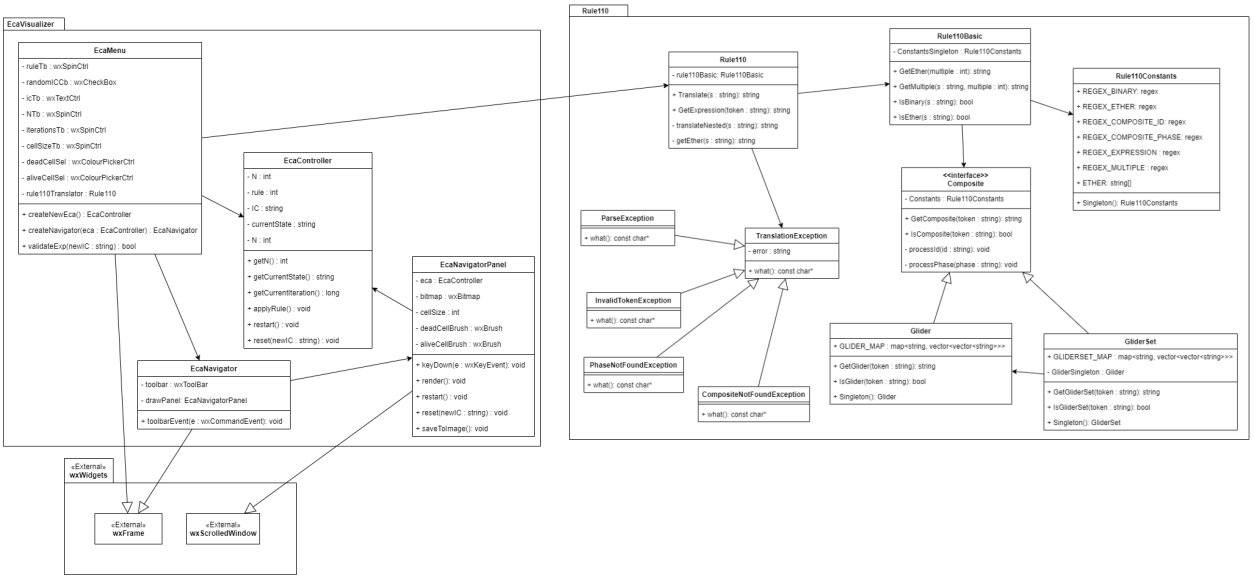

Figura 5.4: Diagrama de clases general hasta la iteración 2.

5.3. Implementación y pruebas

Las figuras 5.5 y 5.6 muestran el controlador de configuraciones funcionando en el visor de ACEs. Se ingresa la expresión `5{5e-A(f1_1)}` la cual se traduce correctamente en 5 partículas $A(f_1\_1)$, separadas por 5 éters.

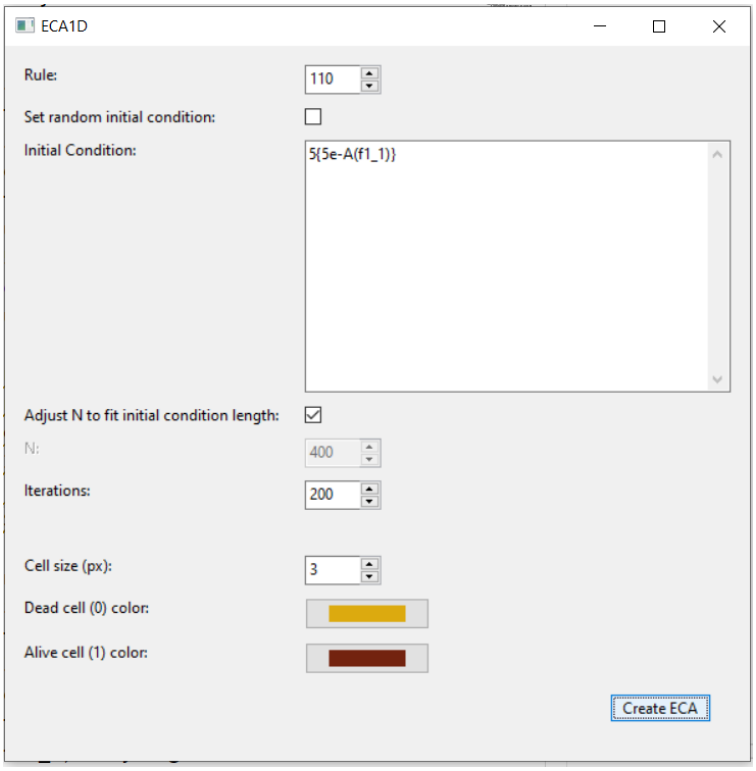

Figura 5.5: Expresión `5{5e-A(f1_1)}` ingresada en la condición inicial del visor de ECAs.

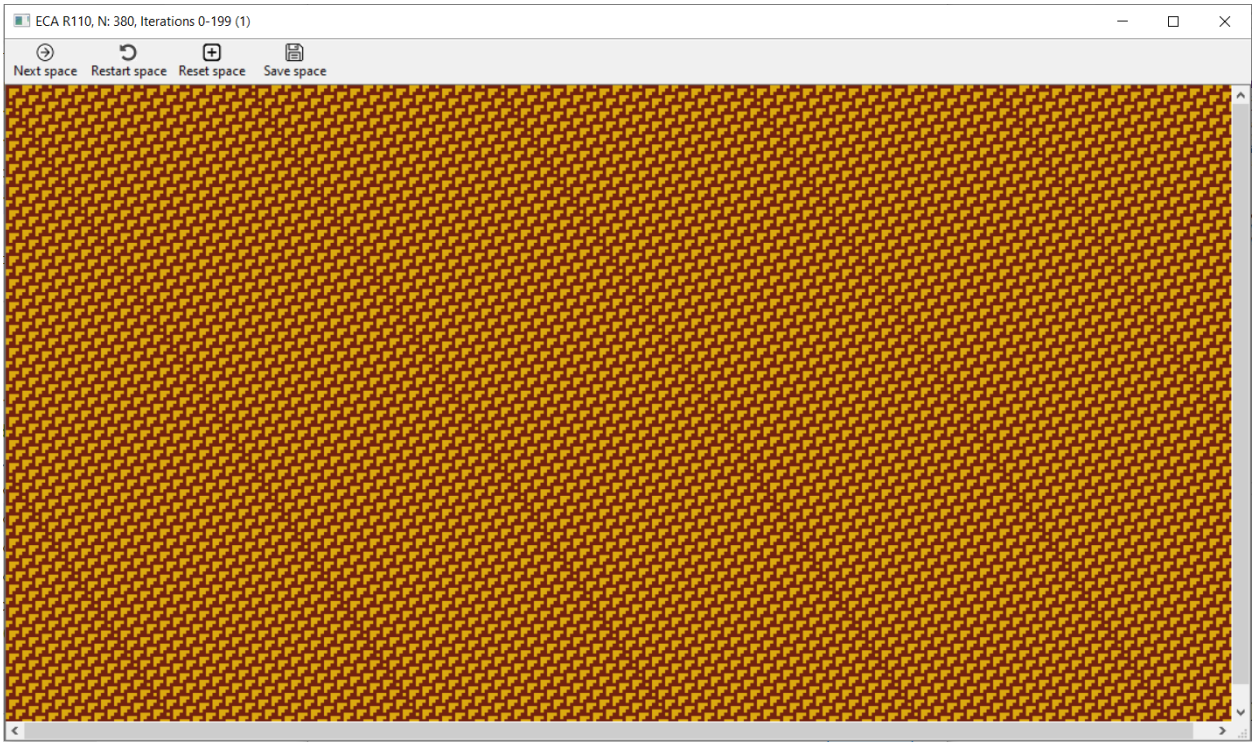

Figura 5.6: Navegador mostrando el espacio de evoluciones creado por la expresión `5{5e-A(f1_1)}`.

Las figuras 5.7 y 5.8 muestran la correcta traducción de la expresión `10e-SepInit_EE_(A1,f1_1)-10e` en el conjunto de partículas $SepInit\_E\bar{E}(A_1, f_1\_1)$, separado de ambos lados por 10 éters.

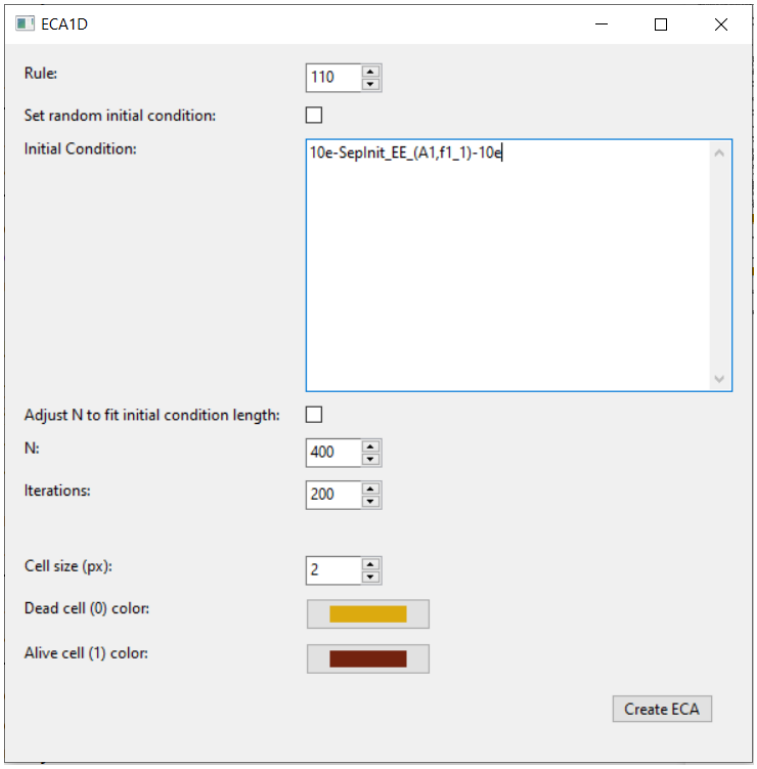

Figura 5.7: Expresión `10e-SepInit_EE_(A1,f1_1)-10e` ingresada en la condición inicial del visor de ECAs.

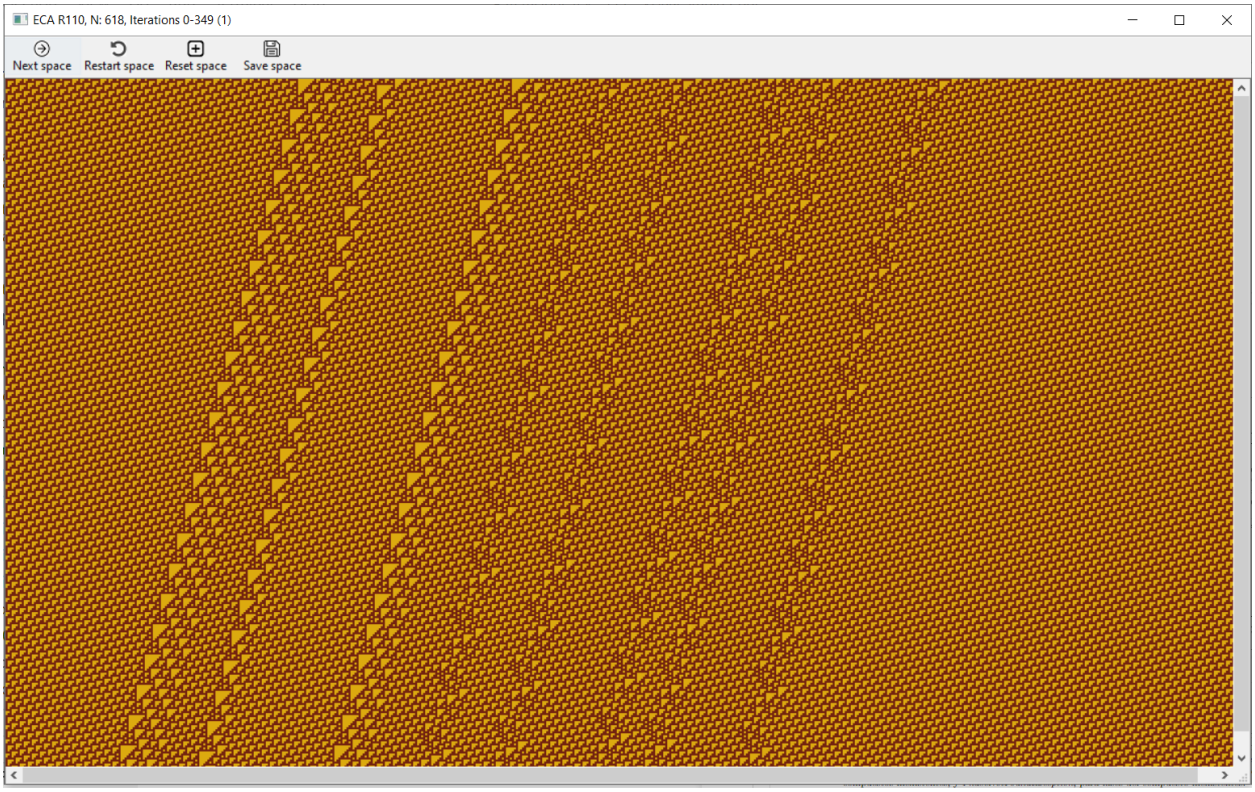

Figura 5.8: Navegador mostrando el espacio de evoluciones creado por la expresión `10e-SepInit_EE_(A1,f1_1)-10e`.

La figura 5.9 muestra las excepciones que puede lanzar el traductor. De izquierda a derecha, de arriba a abajo:

1. Excepción de parseo en una llave sin cerrar, en `5e-{`.
2. Excepción de token inválido en sintaxis incorrecta de un compuesto, en `E(q)`.
3. Excepción de compuesto no encontrado en el id desconocido k, en `k(f1_1)`.
4. Excepción de fase no encontrada, para la fase $(b4, f_1\_1)$ de la partícula C_1, en `c1(b4,f1_1)`.

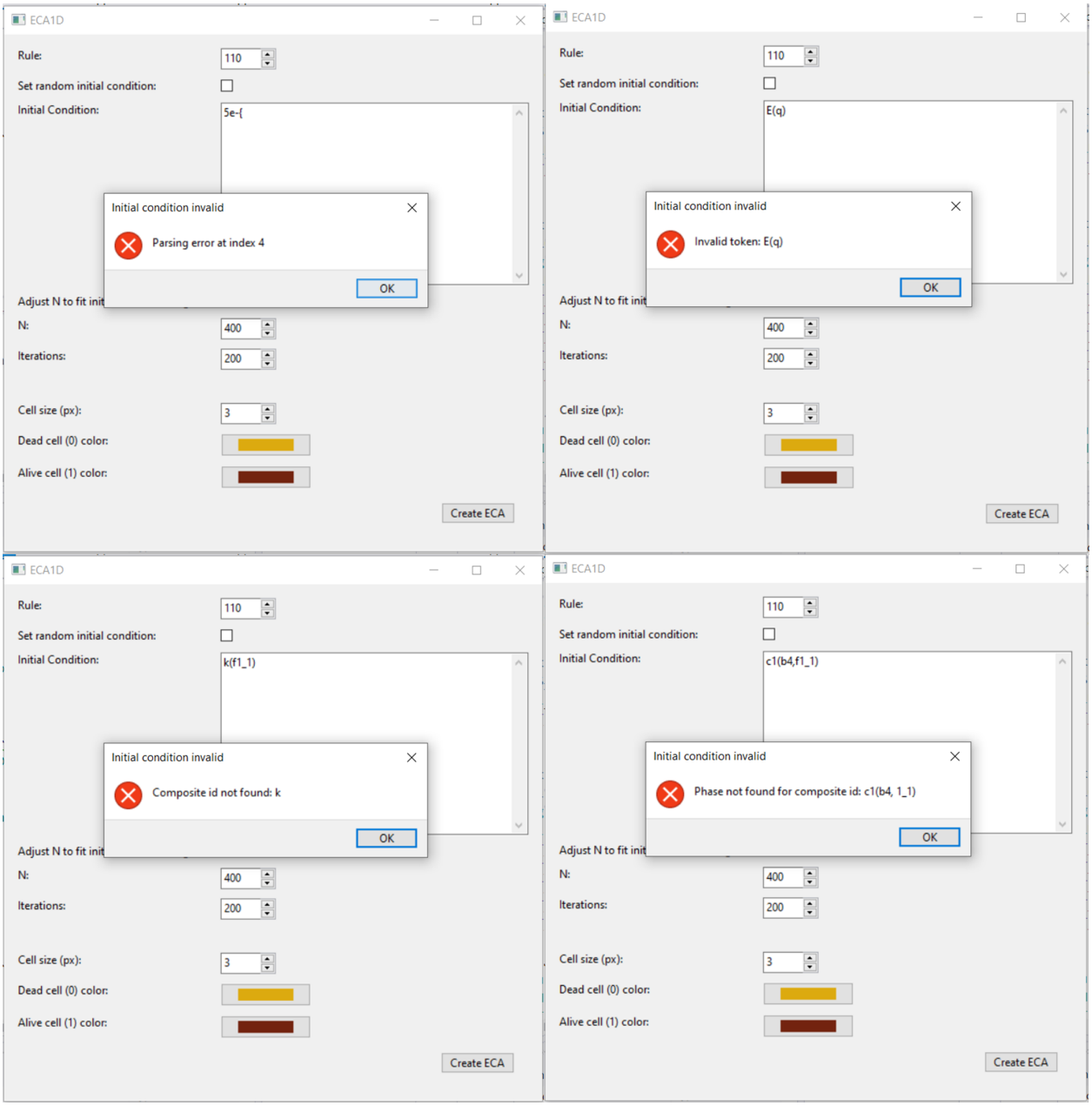

Figura 5.9: Diferentes tipos de excepciones del traductor.

Capítulo 6

Iteración 3: Simulador de anillo

6.1. Análisis

A continuación se describirá el ciclo de desarrollo de la tercera iteración del sistema: el simulador de anillo.

6.1.1. Especificación de requerimientos

Requerimientos funcionales

Id	Requerimiento	Descripción	Prioridad
RF3-01	Configurar ACEs 2.0	El usuario construye el ACE que desea visualizar: regla elemental a utilizar, condición inicial de la cinta, tamaño de la cinta y color de los estados. Escoge también si desea aplicar un filtro a los mosaicos T3 de la regla 110 y un offset opcional a la condición inicial. Además, configura la representación del ACE que desea y selecciona las opciones correspondientes: para la representación de malla, el número de iteraciones del espacio y el tamaño de las celdas; para la representación de ciclotrón, el ancho y radio del anillo, la velocidad de refresco, y el offset del anillo en el modo 3D.	Alta
RF3-02	Navegar el ciclotrón	El navegador de ciclotrón muestra la evolución del ACE en tiempo real, dibujando su cinta en forma de anillo, y filtrando mosaicos T3 si el usuario así lo seleccionó. El usuario puede pausar o reanudar la evolución, así como activar o desactivar el modo 3D del ciclotrón. Las opciones de reiniciar, resetear aleatoriamente y guardar imagen también están disponibles.	Alta

Cuadro 6.1: Requerimientos funcionales para la iteración 3.

Requerimientos no funcionales

Id	Requerimiento	Descripción
RNF3-01	Usabilidad	La aplicación es intuitiva y con una mínima curva de aprendizaje con respecto a *RF1-01*.
RNF3-02	Eficiencia	El visor muestra el espacio de evoluciones en un tiempo y con una cantidad de recursos de memoria proporcionales al tamaño del espacio seleccionado. El sistema libera los recursos que ya no estén en uso.

Cuadro 6.2: Requermientos no funcionales para la iteración 3.

6.2. Diseño

6.2.1. Diagrama de casos de uso

La figura 6.1 muestra el diagrama de casos de uso de la iteración 3. El sistema consta de tres módulos:

1. *Configurar ACEs 2.0*: El usuario realiza la configuración del ACE similar a como lo hacía en *RF1-01*. El usuario puede ahora seleccionar la representación del ECA, es decir, un navegador en forma de malla como en *RF1-02*, o un navegador en forma de ciclotrón, para el cual muestra opciones de visualización específicas: grosor y radio del anillo, velocidad de refresco, y offset con respecto al anillo anterior (en modo 3D). Además, el usuario puede activar un filtro específico para la regla 110, el cual elimina todos los mosaicos T3 del espacio de evoluciones en el navegador, así como realizar un offset a la condición inicial dada. El usuario selecciona sus preferencias y, al finalizar, presiona el botón de aceptar. El sistema valida las opciones, crea un controlador de ACE y abre una ventana de navegación acorde a la representación que haya elegido.

2. *Navegar ciclotrón*: El navegador hace uso del ACE creado para mostrar el espacio de evoluciones, dibujando la cinta en forma de anillo en tiempo real. Si el usuario activó el filtro de mosaicos, éstos no se dibujarán en el anillo. El usuario puede elegir mediante los botones de la barra de herramientas, o mediante comandos del teclado, diferentes opciones:

   - Pausar o continuar el avance del ACE.
   - Activar o desactivar el modo 3D del ciclotrón.
   - Reiniciar el espacio, comenzando desde la condición inicial.
   - Resetear el espacio con una condición inicial aleatoria.
   - Guardar el espacio actual en un archivo de imagen.

3. *«RF1-02» Navegar el espacio de evoluciones*.

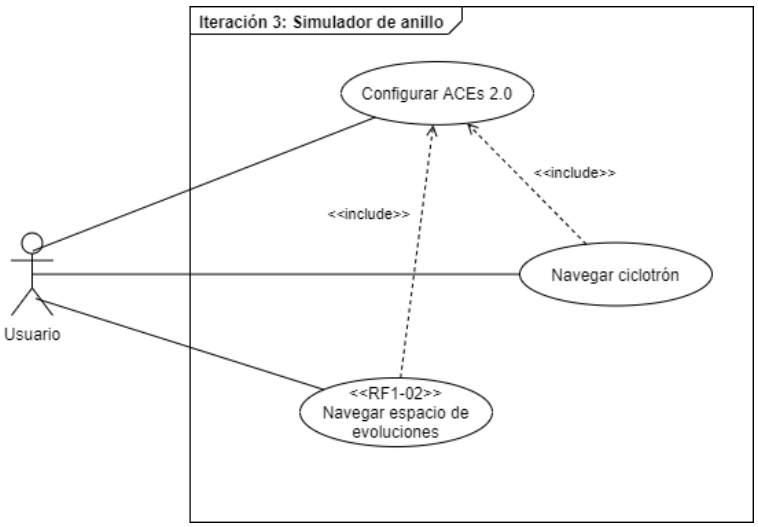

Figura 6.1: Diagrama de caso de uso para la iteración 3.

6.2.2. Diagramas de actividades

La figura 6.2 muestra el diagrama de actividades correspondiente al caso de uso *Configurar ACEs 2.0*.

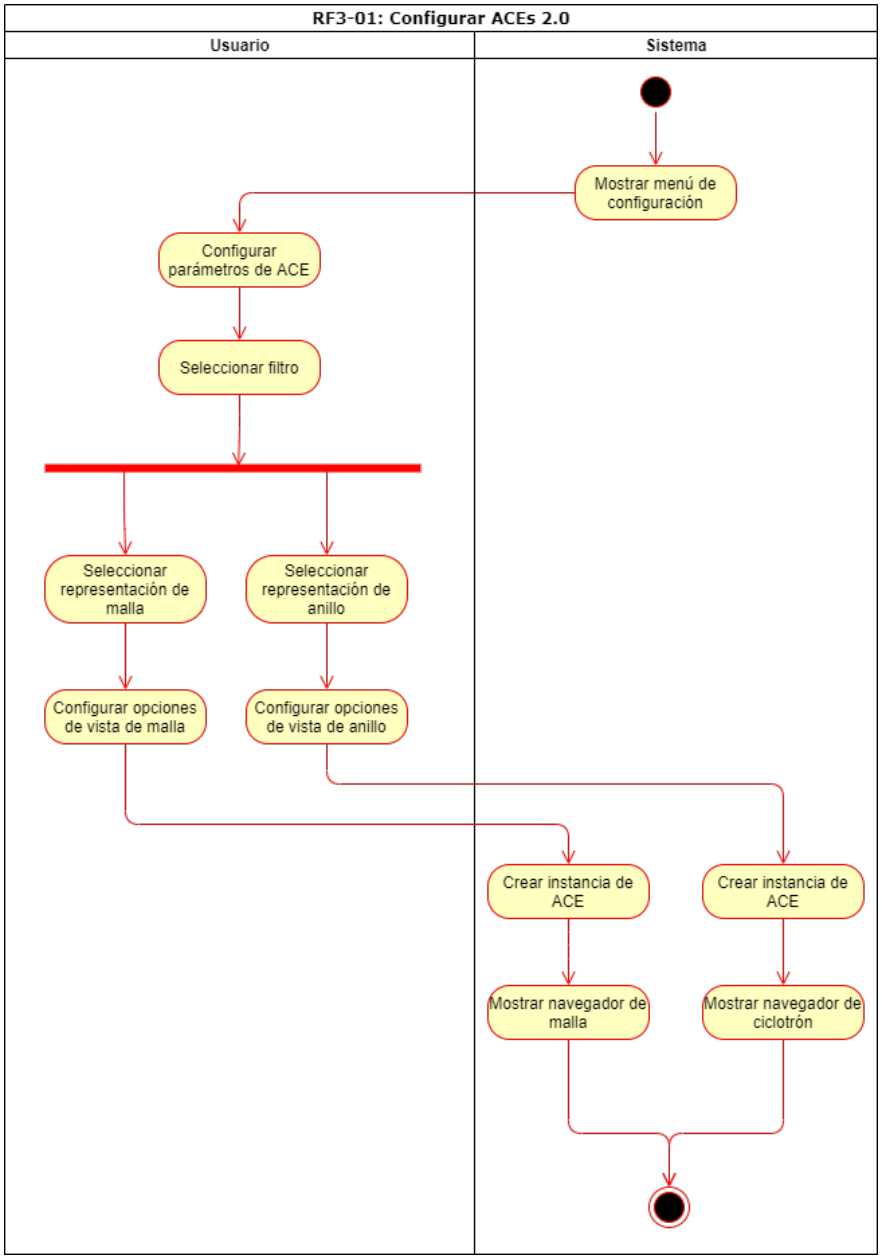

Figura 6.2: Diagrama de actividades del caso de uso *Configurar ACEs 2.0*.

La figura 6.3 muestra el diagrama de actividades correspondiente al caso de uso *Navegar ciclotrón*.

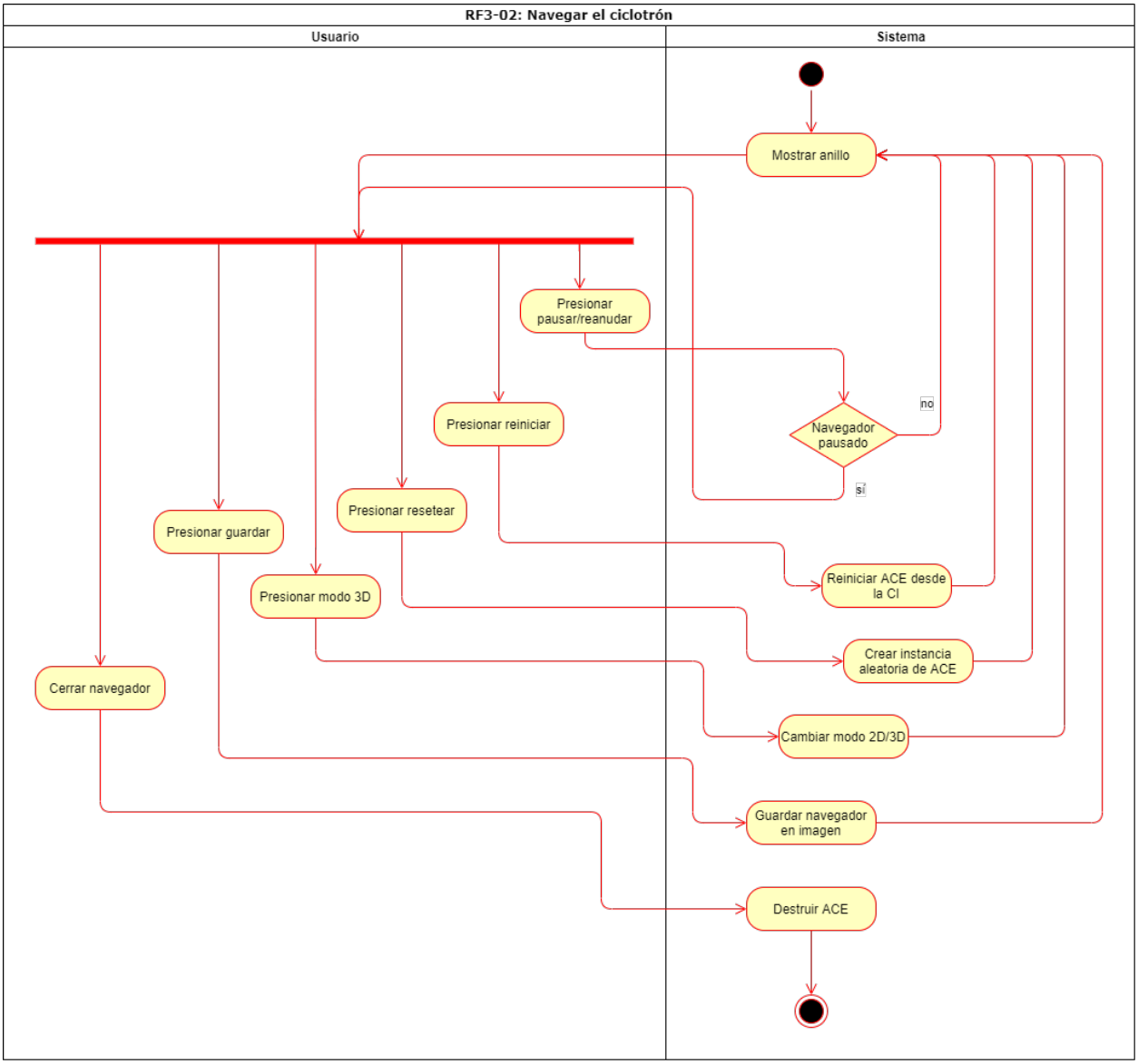

Figura 6.3: Diagrama de actividades del caso de uso *Navegar ciclotrón*.

6.2.3. Diagrama de clases

La figura 6.4 muestra el diagrama de clases para la iteración 3.

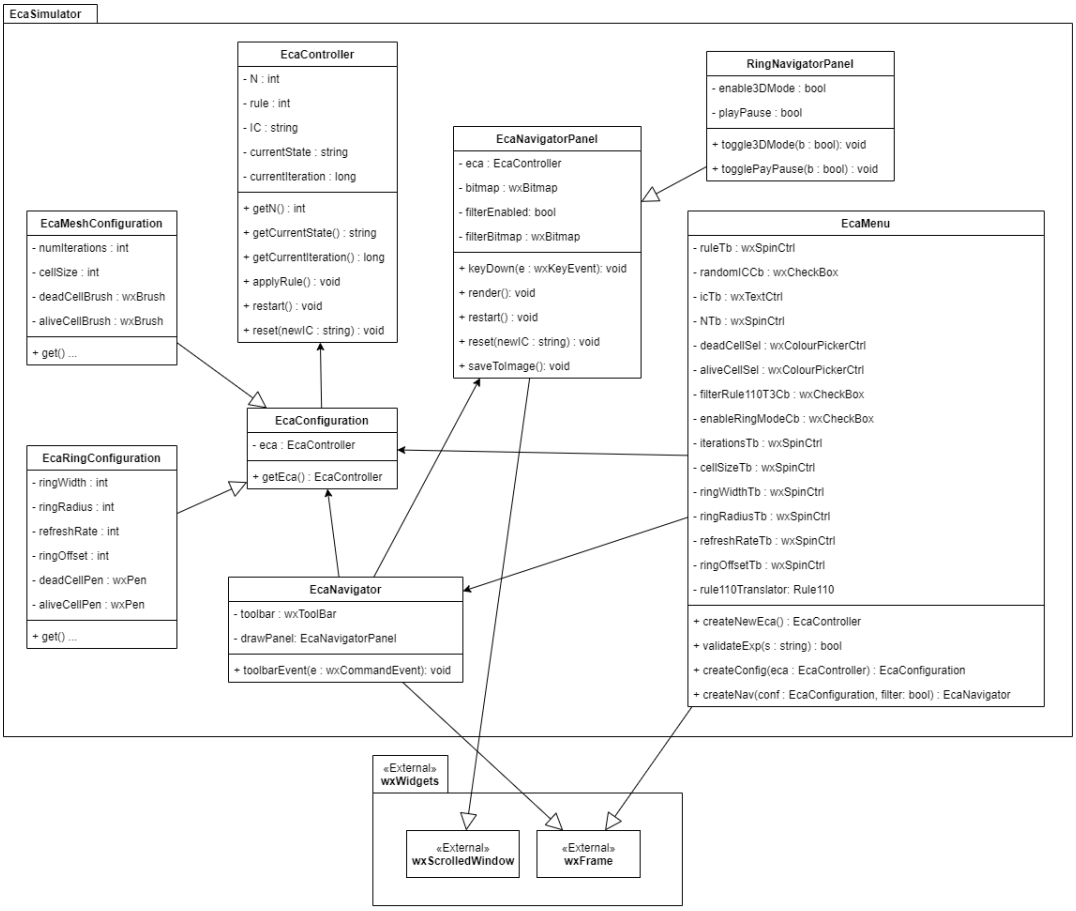

Figura 6.4: Diagrama de clases para la iteración 3.

Se realiza una reestructuración con respecto al módulo original:

- *EcaMenu* presenta una opción de filtro de mosaicos T3 para el ACE. Además, ahora es posible seleccionar el tipo de navegador a utilizar, dependiendo de la representación deseada: de malla o de ciclotrón.
- En vez de interactuar directamente con *EcaController*, éste ahora está encapsulado en la iterfaz *EcaConfiguration*, la cual también contiene opciones de visualización para sus respectivas implementaciones: *EcaMeshConfiguration*, para la representación de malla, y *EcaRingConfiguration*, para la representación de ciclotrón.
- Se añade una especialización de *EcaNavigatorPanel*: *RingNavigatorPanel*. Éste tiene toda la lógica necesaria para procesar el anillo en tiempo real, así como para manejar las opciones específicas del ciclotrón: activar/desactivar el modo 3D, y pausar/reanudar la evolución del ACE.

La figura 6.5 muestra el diagrama general del sistema hasta la iteración 3, incluyendo el módulo de traductor de expresiones R110.

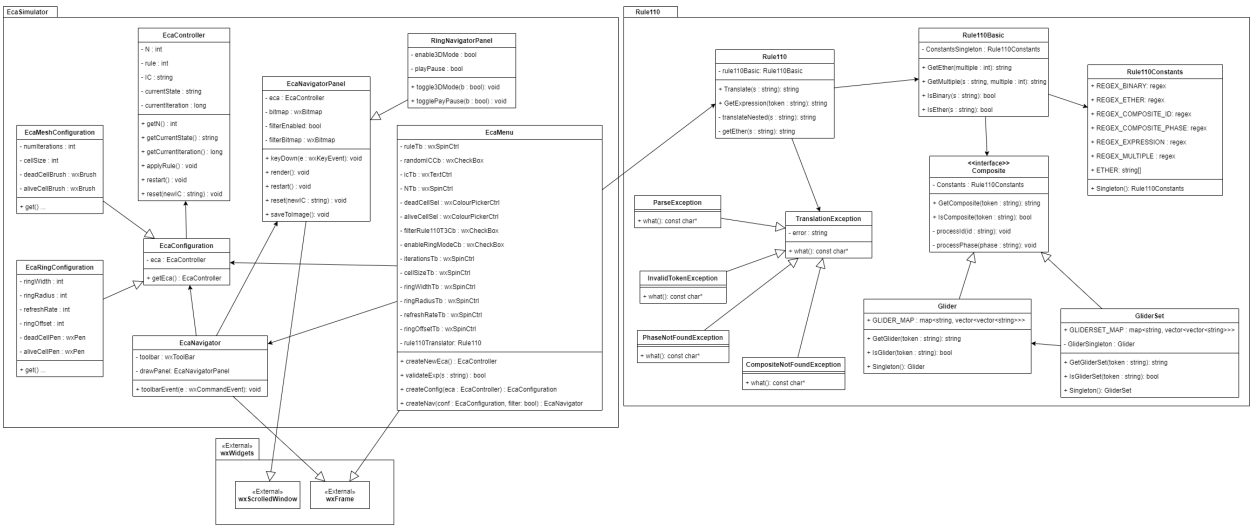

Figura 6.5: Diagrama de clases general hasta la iteración 3.

6.3. Implementación y pruebas

La figura 6.6 muestra la versión 2 del configurador de ACEs, con una interfaz gráfica renovada. Las opciones de configuración cambian dependiendo del modo de representación que se seleccione.

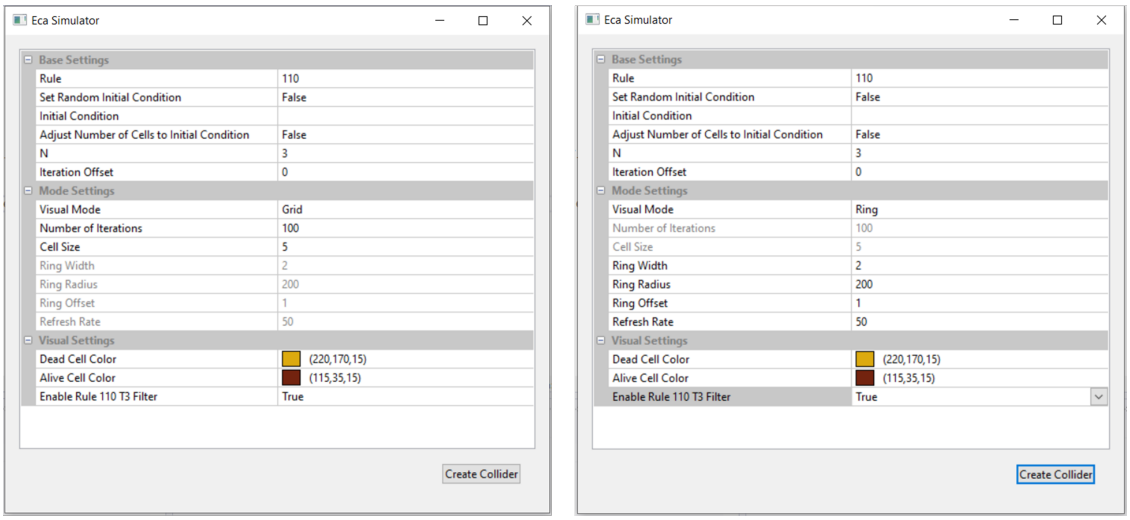

Figura 6.6: Capturas de pantalla del menú de configuración de ACEs 2.0. Del lado izquierdo, las opciones para un navegador de malla. Del lado derecho, las opciones para un navegador de ciclotrón.

La figura 6.7 muestra el navegador de ciclotrón en funcionamiento con una condición inicial aleatoria y opciones de visualización por defecto. Si se supone el anillo como una circunferencia con centro en el origen, el principio de la cinta se encuentra en el eje positivo x, y recorre el perímetro de la circunferencia en sentido antihorario. La figura 6.8 muestra el mismo ciclotrón con el modo 3D activado.

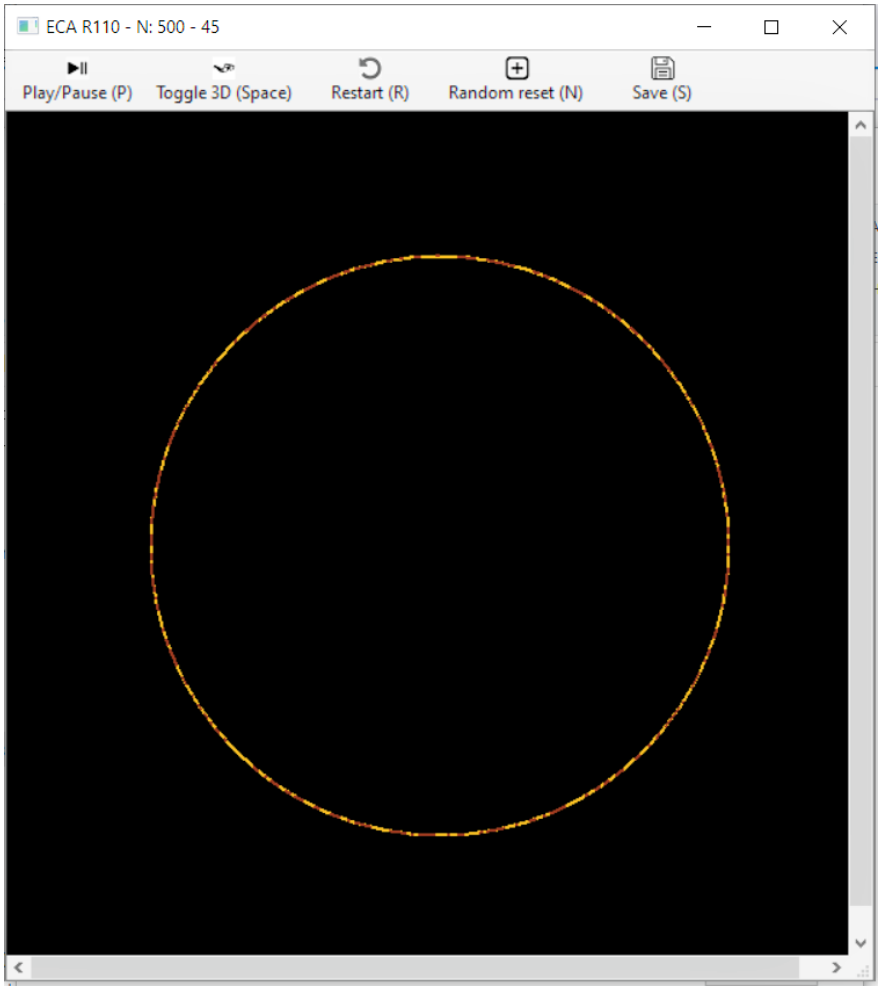

Figura 6.7: Captura de pantalla del navegador de ciclotrón con condición inicial aleatoria y opciones de visualización por defecto.

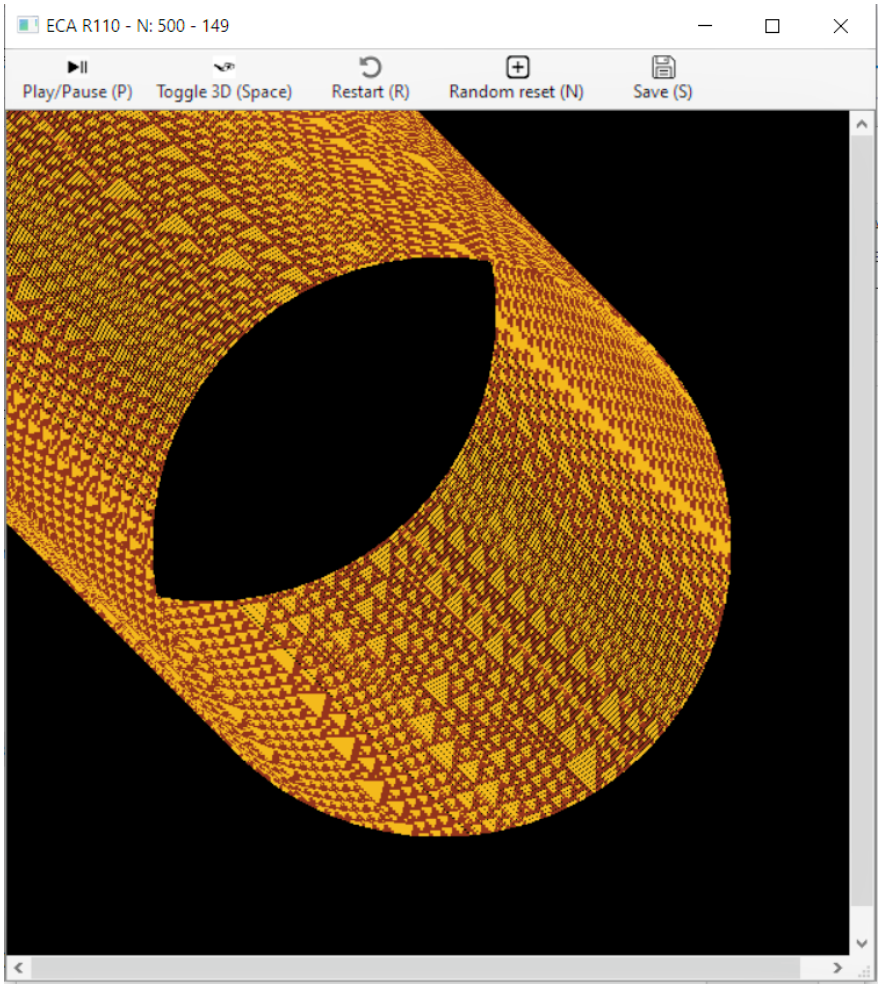

Figura 6.8: Modo 3D activado para el ACE de la figura 6.7.

La figura 6.9 muestra otro ciclotrón en modo 3D, con opciones de visualización elegidas por el usuario: distintos colores de estados, mayor grosor de anillo, y offset diferente al recomendado.

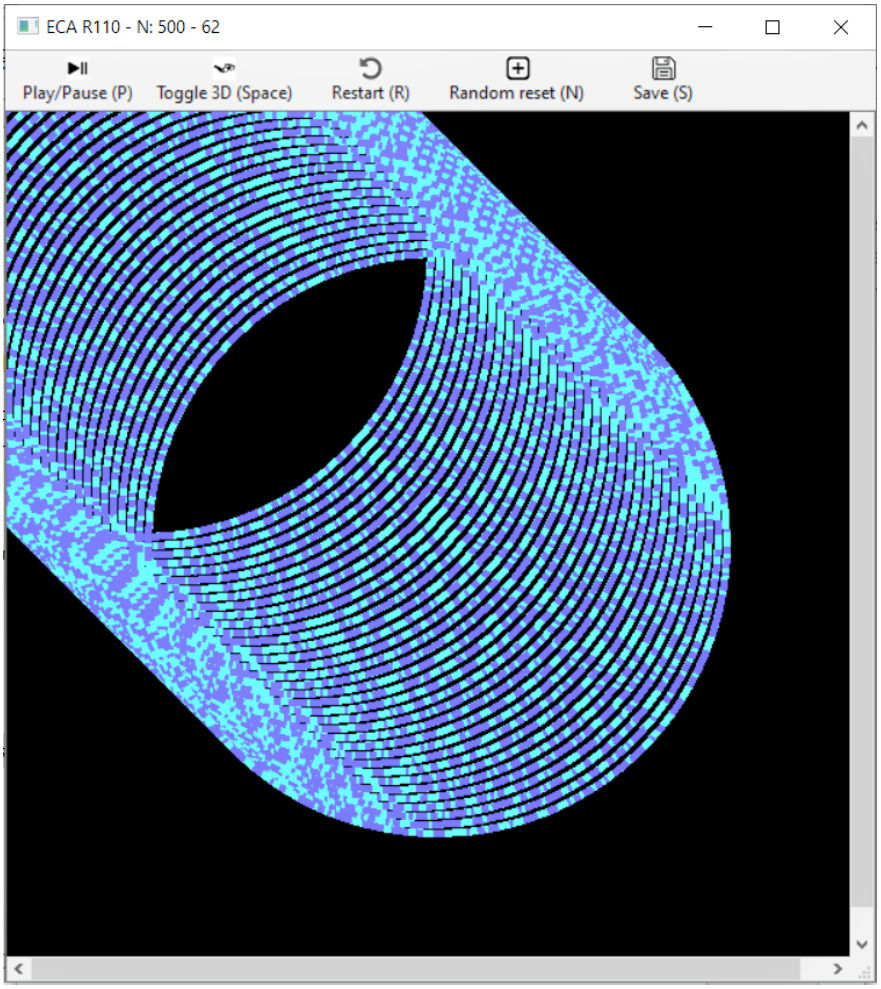

Figura 6.9: Captura de pantalla del navegador de ciclotrón con condición inicial aleatoria y opciones de visualización especificadas.

La figura 6.10 muestra un ciclotrón al cual se le ha aplicado un filtro de mosaicos T3 de la regla 110. En cada evolución del anillo, si se localiza un mosaico, éste no se dibuja. La figura 6.11 muestra este filtro funcionando también para un navegador de malla.

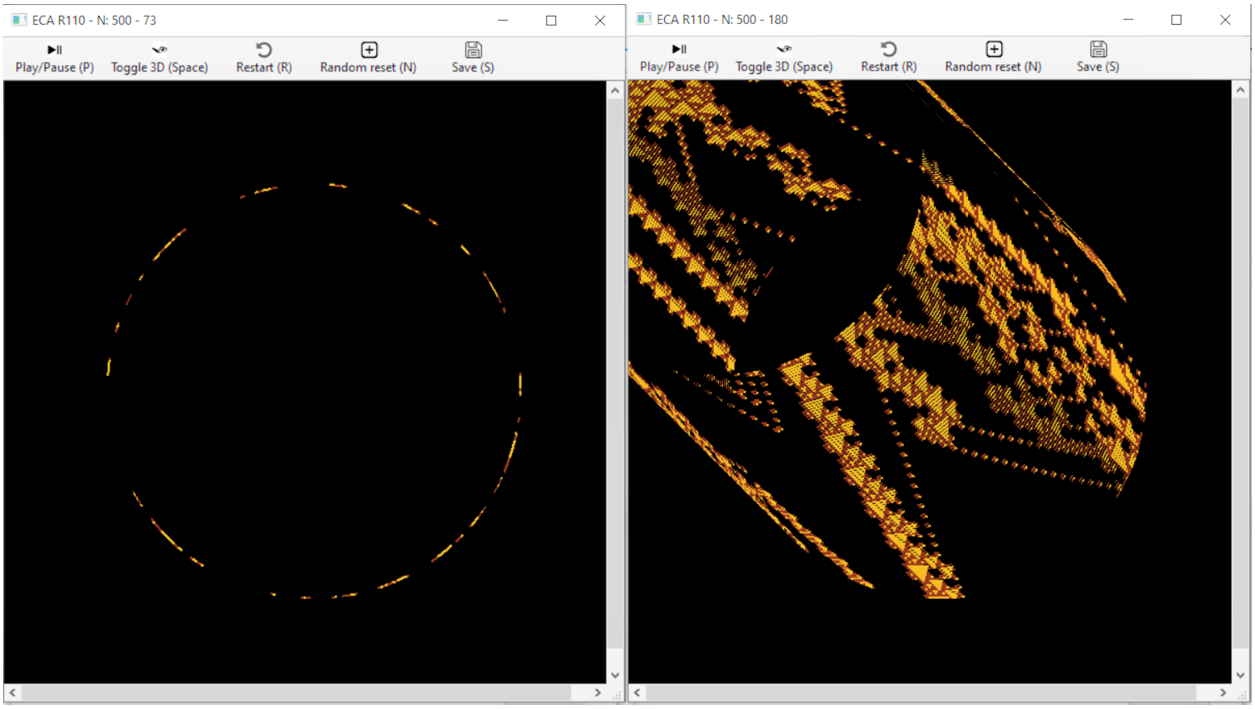

Figura 6.10: Navegador de ciclotrón con filtro de mosaicos T3.

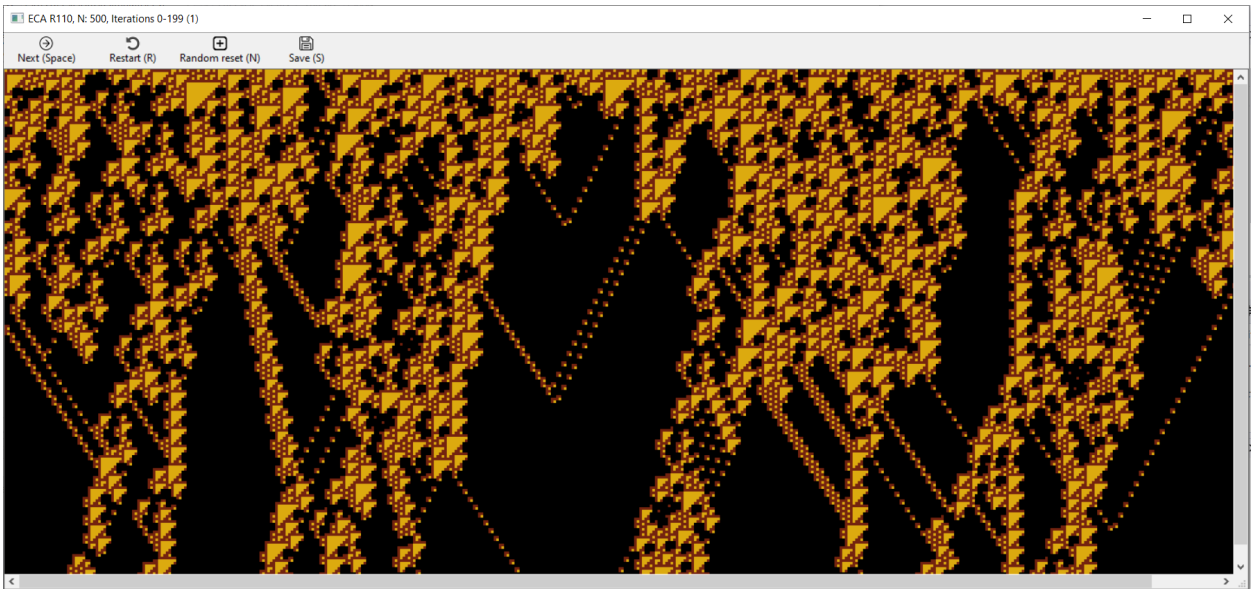

Figura 6.11: Navegador de malla con filtro de mosaicos T3.

Capítulo 7

Iteración 4: Colisionador

7.1. Análisis

A continuación se describirá el ciclo de desarrollo de la cuarta iteración del sistema: el colisionador virtual.

7.1.1. Especificación de requerimientos

Requerimientos funcionales

Id	Requerimiento	Descripción	Prioridad
RF4-01	Configurar el colisionador	El usuario realiza la configuración del colisionador: el estado inicial de cada uno de los 3 anillos, la posición de los puntos de interacción entre los anillos, el tamaño del anillo central y el ratio de refresco de las iteraciones. Además, especificará el comportamiento de los anillos durante la ejecución, con una lista que indica cómo deben irse habilitando o desabilitando los anillos o los puntos de interacción. Si así lo desea, el usuario puede moverse al menú de configuración de simulador de ACE original.	Alta
RF1-02	Navegar el colisionador	El navegador de ciclotrón muestra la evolución del colisionador en tiempo real, dibujando los estados de los 3 anillos, filtrados de sus mosaicos T3, así como sus interacciones. El usuario puede pausar o reanudar la evolución, o saltar a una iteración específica del estado de los anillos.Puede también decidir si mostrar u ocultar cualquiera de los 3 anillos del colisionador. Además, puede guardar la imagen de la iteración actual, o reiniciar la ejecución del colisionador si así lo desea.	Alta

Cuadro 7.1: Requerimientos funcionales para la iteración 4.

Requerimientos no funcionales

Id	Requerimiento	Descripción
RNF4-01	Usabilidad	La aplicación es intuitiva y entendible para cualquier persona con conocimiento del ACE con regla 110.
RNF4-02	Eficiencia	El visor muestra el colisionador en un tiempo y con una cantidad de recursos de memoria proporcionales al tamaño de los anillos. El sistema libera los recursos que ya no estén en uso.

Cuadro 7.2: Requerimientos no funcionales para la iteración 4.

7.2. Diseño

7.2.1. Diagrama de casos de uso

La figura 7.1 muestra el diagrama de casos de uso de la iteración 4. El sistema consta de dos módulos:

1. *Configurar colisionador*: El usuario realiza la configuración del colisionador. Escribe la condición inicial para cada uno de los anillos, haciendo uso del traductor de *RF2-01*, y selecciona los puntos de contacto que transmitirán las células entre los anillos laterales y el anillo central. De igual manera, puede seleccinar el tamaño que tendrá el anillo central y el ratio de refresco de la ejecución. El usuario puede, además, escribir una lista de acciones que el colisionador ejecutará, de la forma i-$<ACTION>$, en donde i es un número que indica la iteración en la que se realizará la acción dictada por $<ACTION>$: encender o apagar cualquiera de los anillos $((DISABLE \mid ENABLE)\_(LEFT \mid CENTRAL \mid RIGHT)\_RING)$, o como habilitar o desabilitar alguno de los puntos de contacto entre anillos $((DISABLE \mid ENABLE)\_(LEFT \mid RIGHT)\_CONTACT)$. Al finalizar, el usuario presiona el botón de aceptar, con lo que el sistema valida las opciones, crea un controlador de colisionador y abre una ventana de navegación. Finalmente, es posible presionar el botón de Ir a Simulador para saltar al menú de simulador de ACE original.

2. *Navegar colisionador*: El navegador hace uso del controlador creado para mostrar el sistema de colisiones, dibujando los anillos, filtrados de sus mosaicos T3, en tiempo real. El usuario puede elegir mediante los botones de la barra de herramientas, o mediante comandos del teclado, las siguientes opciones:

   - Pausar o continuar el avance del colisionador.
   - Saltar a una iteración específica del colisionador.
   - Mostrar u ocultar cualquiera de los 3 anillos del colisionador.
   - Guardar el estado actual en un archivo de imagen.
   - Reiniciar la ejecución del colisionador.

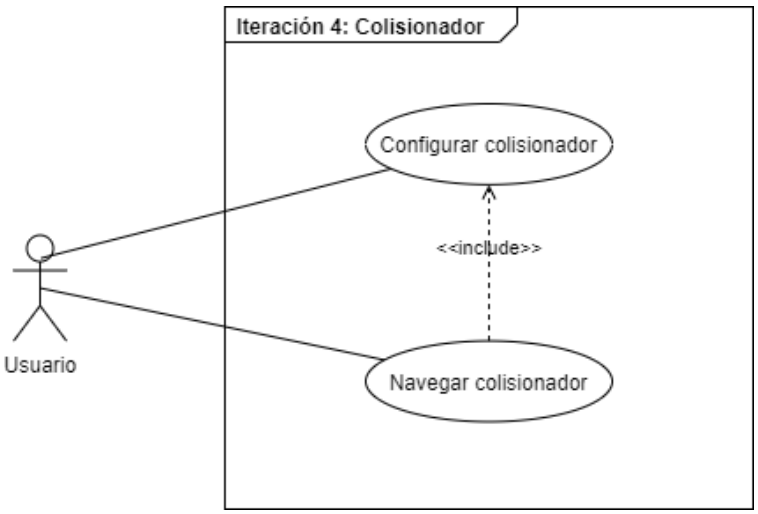

Figura 7.1: Diagrama de caso de uso para la iteración 4.

7.2.2. Diagramas de actividades

La figura 7.2 muestra el diagrama de actividades correspondiente al caso de uso *Configurar colisionador*.

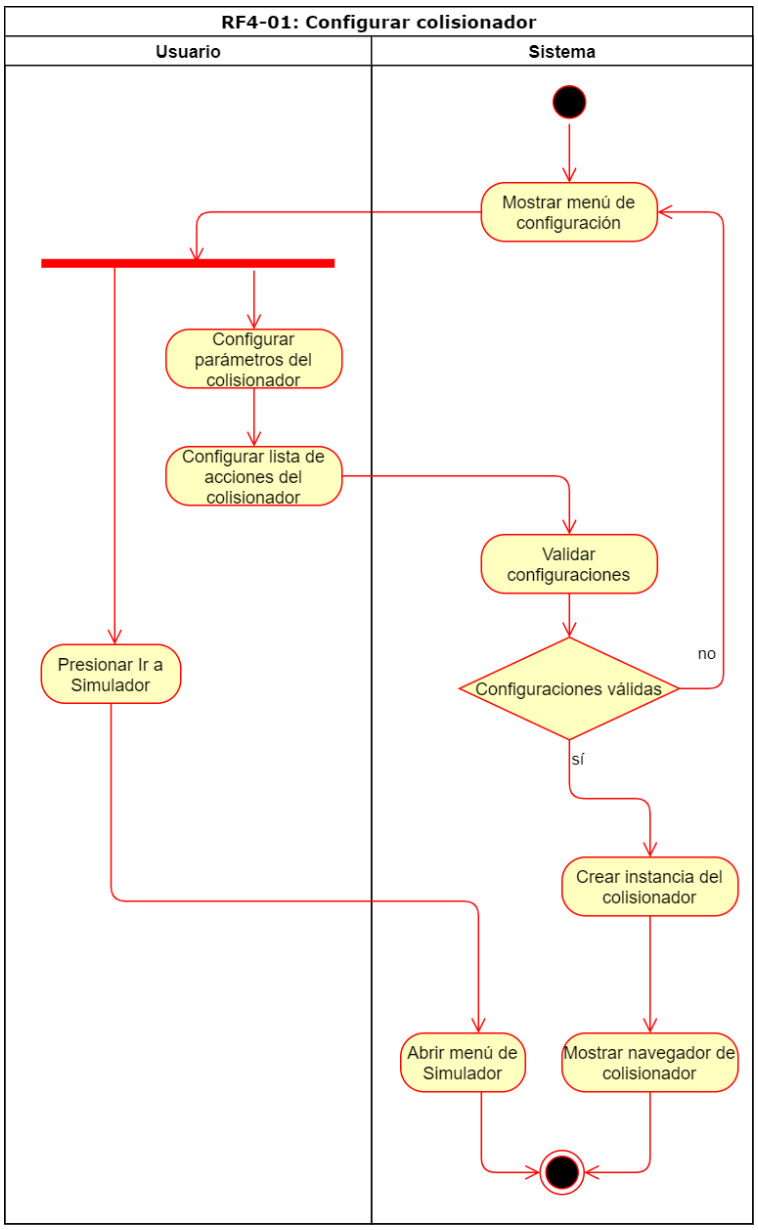

Figura 7.2: Diagrama de actividades del caso de uso *Configurar colisionador*.

La figura 7.3 muestra el diagrama de actividades correspondiente al caso de uso *Navegar colisionador*.

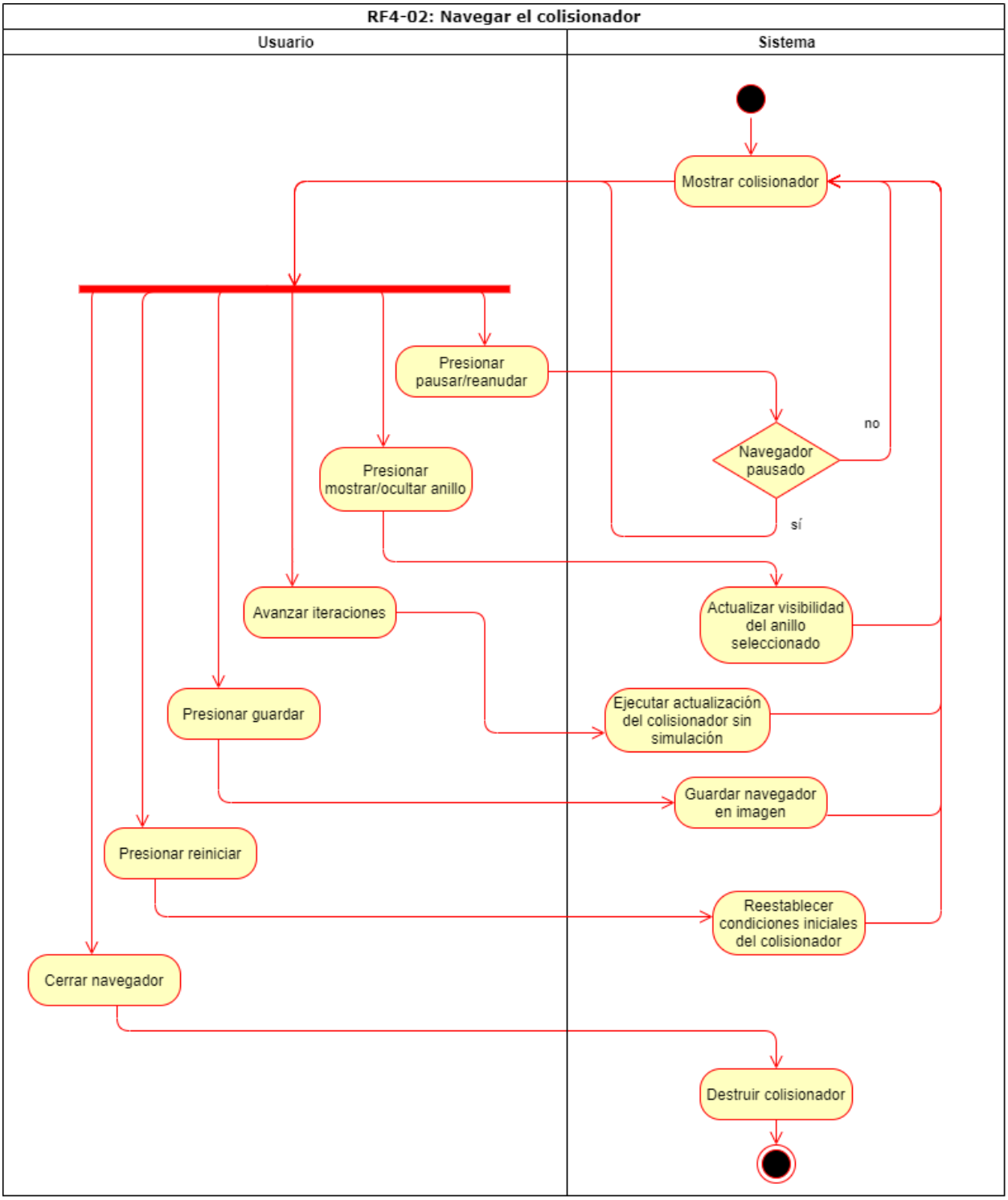

Figura 7.3: Diagrama de actividades del caso de uso *Navegar colisionador*.

7.2.3. Diagrama de clases

La figura 7.4 muestra el diagrama de clases para la iteración 4.

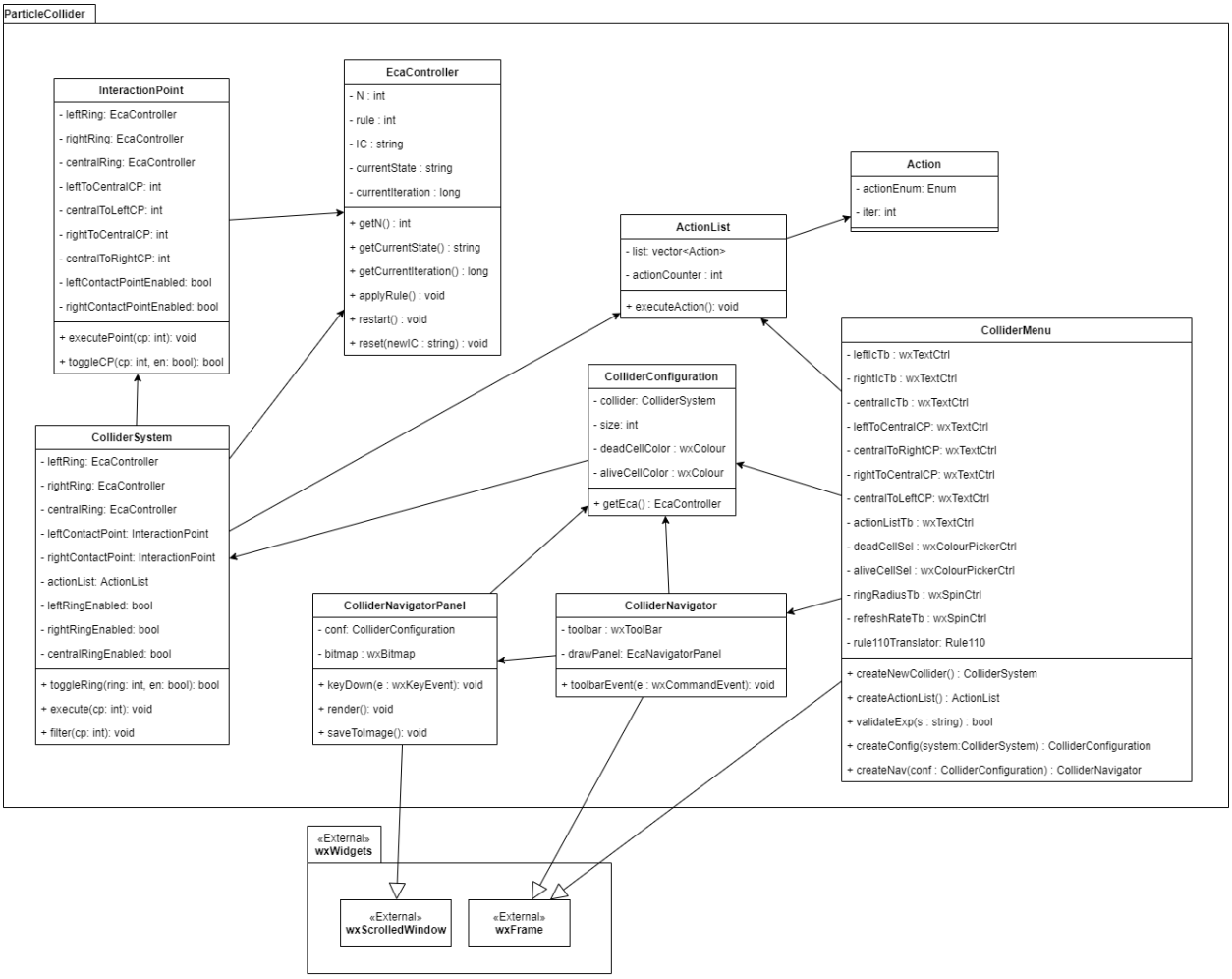

Figura 7.4: Diagrama de clases para la iteración 4.

Similares a sus contrapartes en la iteración 3, *ColliderMenu*, *ColliderNavigator* y *ColliderNavigatorPanel* se encargan de las vistas,las cuales utilizan los parámetros dados por *ColliderConfiguration*.

La lógica es controlada por *ColliderSystem*, un objeto que contiene los *EcaController*s para cada uno de los anillos del colisionador, así como dos *InteractionPoint*s: entidades de ejecución para el intercambio de células entre anillos. De igual manera, posee una *ActionList*, una lista de acciones a ejecutar a lo largo del procesamiento del colisionador: encender o apagar cualquiera de los anillos, o habilitar o desabilitar alguno de los puntos de contacto entre anillos.

La figura 7.5 muestra el diagrama general del sistema en la iteración 4, incluyendo el módulo de traductor de expresiones R110. Se omite el diagrama de la iteración 3 para mejor visibilidad.

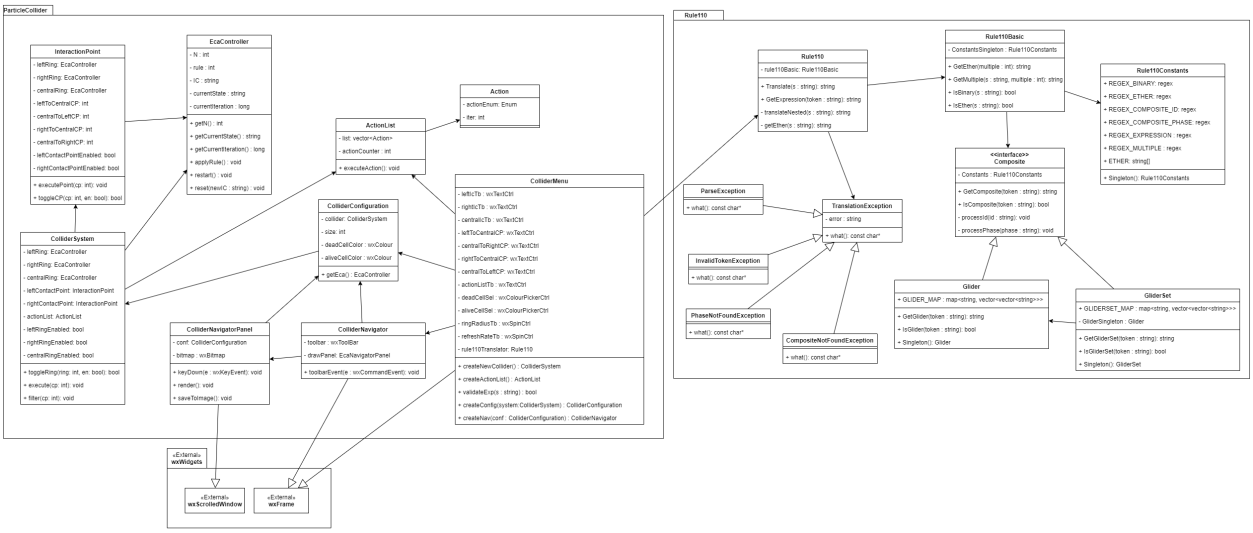

Figura 7.5: Diagrama de clases general de la iteración 4.

7.3. Implementación y pruebas

La figura 7.6 muestra el menú de configuración del simulador original. Con un botón, el usuario puede abrir el nuevo menú de configuración del colisionador.

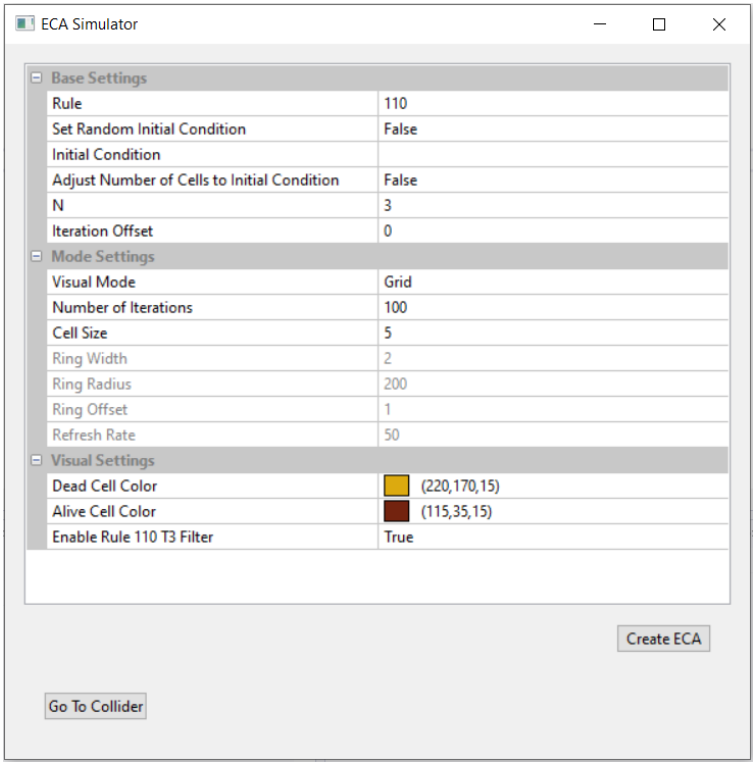

Figura 7.6: Captura de pantalla del nuevo menú de configuración de simulador.

La figura 7.7 muestra el menú de configuración del colisionador.

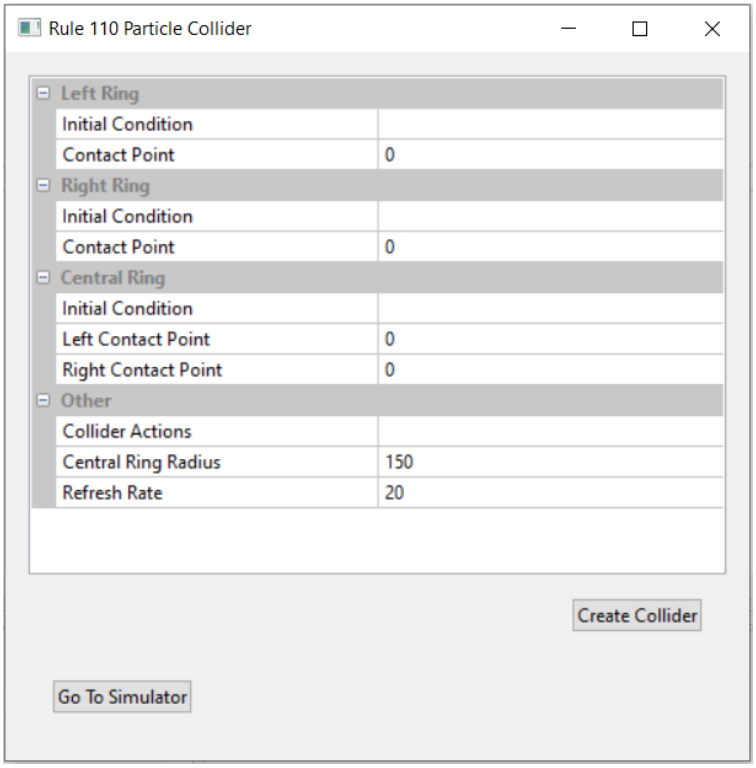

Figura 7.7: Captura de pantalla del menú de configuración de colisionador.

La figura 7.8 utiliza la configuración de la figura anterior para mostrar la ejecución de un colisionador que transporta una partícula $\bar{E}$ a lo largo de anillos de éter. De acuerdo a las acciones dictadas en el configurador, el punto de interacción del anillo izquierdo está desactivado.

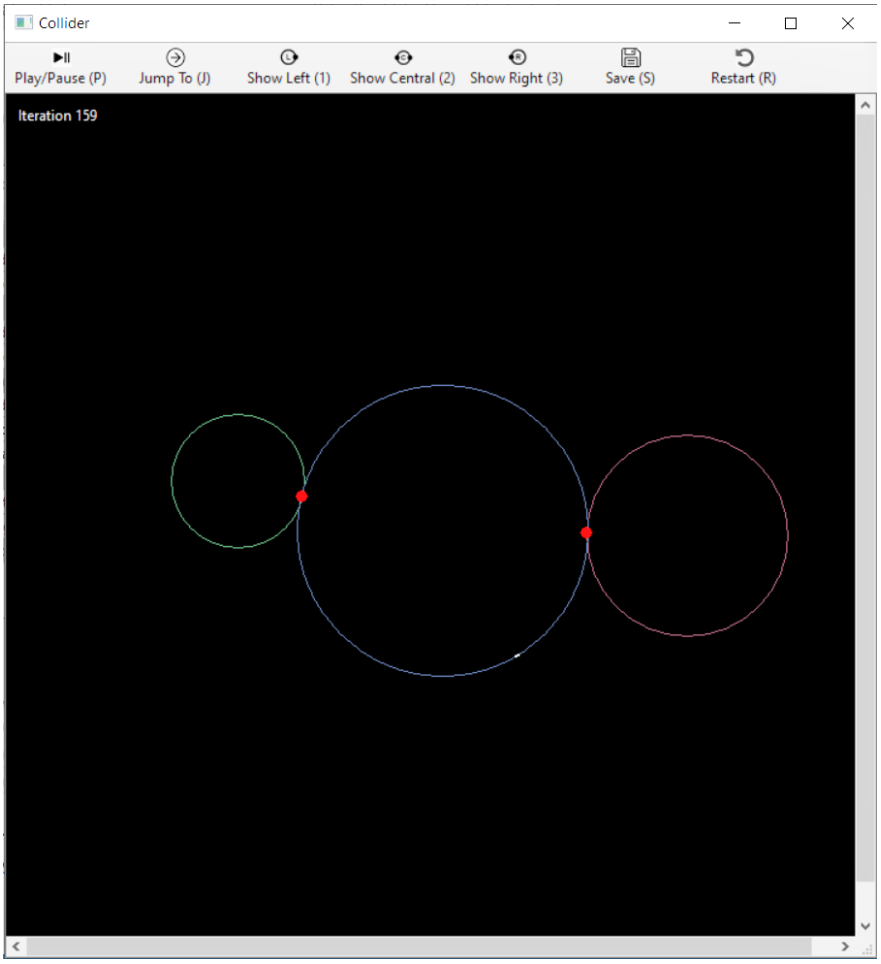

Figura 7.8: Captura de pantalla del navegador de colisionador transportando una partícula $\bar{E}$.

La figura 7.9 muestra la reactivación del punto de interacción izquierdo, permitiendo a la partícula atravesar el anillo.

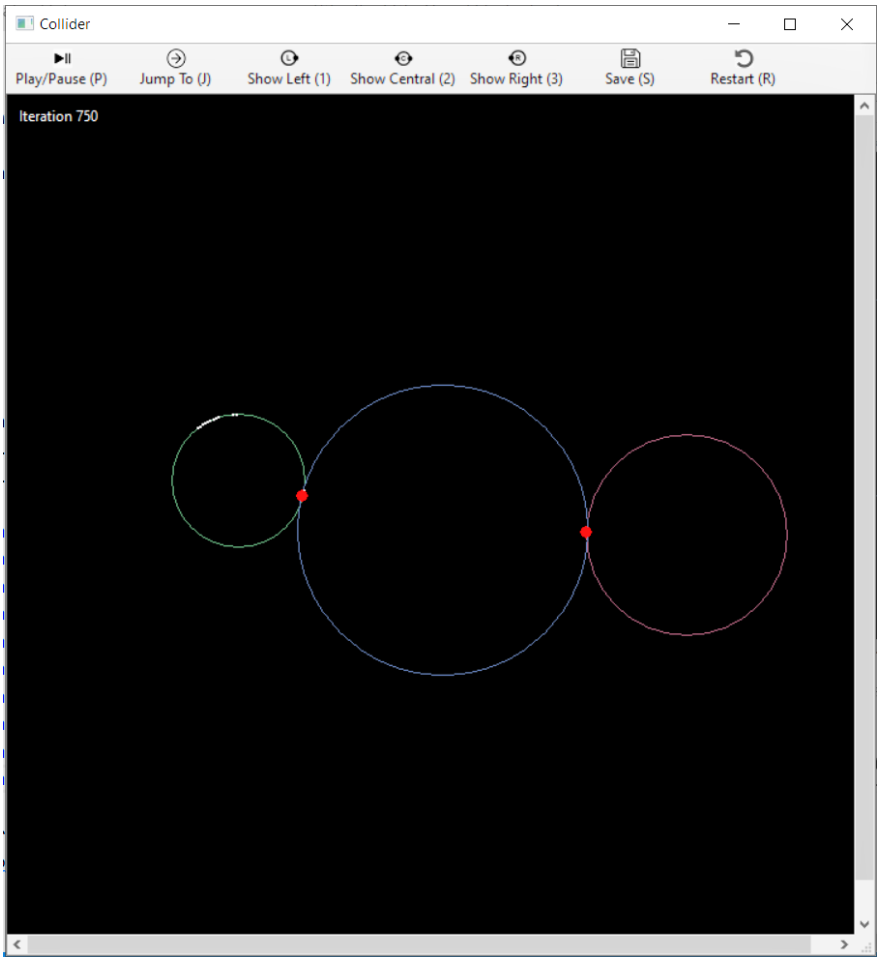

Figura 7.9: Captura de pantalla del navegador de colisionador transmitiendo una partícula $\bar{E}$ a su anillo izquierdo.

Finalmente, la figura 7.10 muestra la partícula transmitida al anillo derecho. El anillo izquierdo ha sido ocultado por el usuario mediante la opción de visibilidad respectiva.

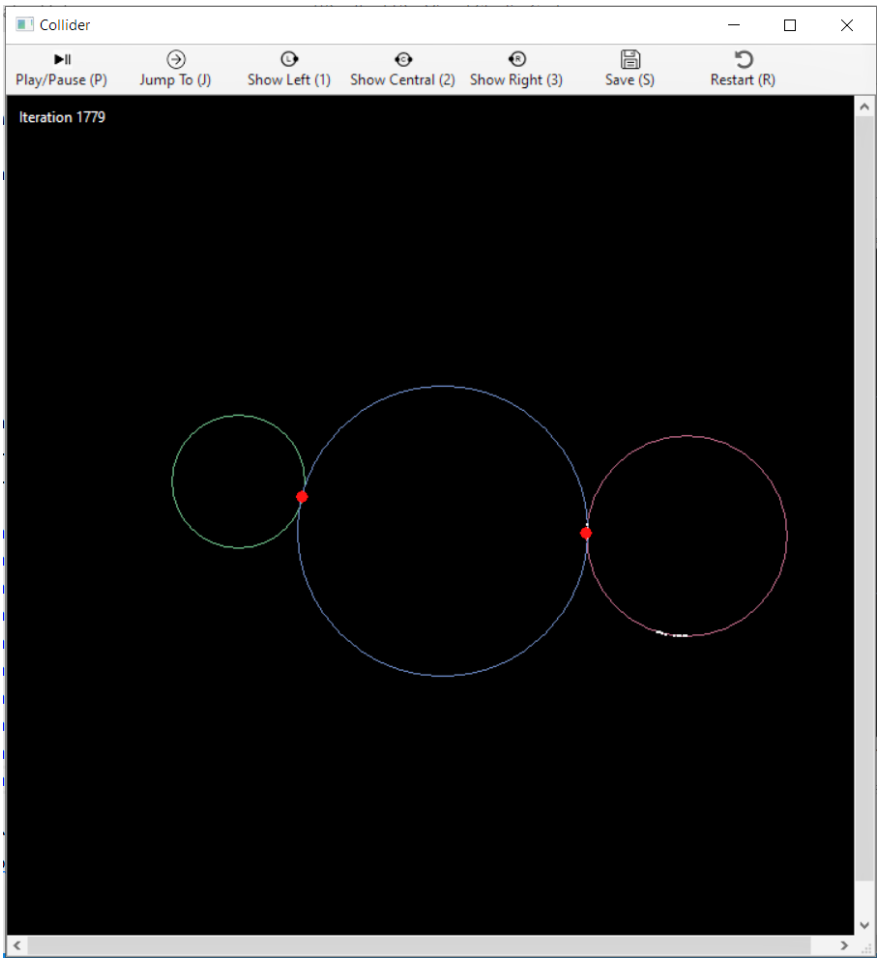

Figura 7.10: Captura de pantalla del navegador de colisionador transmitiendo una partícula $\bar{E}$ a su anillo derecho, con anillo izquierdo oculto.

Capítulo 8

Iteración 4.5: Optimización

8.1. Preámbulo

Durante la implementación de las iteraciones 3 y 4, se pudo observar una notable ralentización en el procesamiento de ACEs de gran tamaño, sobre todo cuando es necesario filtrar mosaicos T3. A continuación se enlistarán diferentes técnicas de optimización para el ACE y cómo se adaptaron a este trabajo.

8.1.1. Procesamiento in-place

La forma trivial de almacenar la cinta de un ACE es con una cadena de ceros y unos representando células. Para procesar una iteración del ACE, se analiza cada célula junto con su vecindad, y aplicando la regla del ACE, se obtiene una cadena con el nuevo estado. Ya que deben revisarse constantemente células anteriores a las iteradas, se utiliza un búfer de respaldo para almacenar el nuevo estado sin sobreescribir el anterior. Al terminar la iteración, el búfer se copia de vuelta al estado del autómata, una acción que resulta costosa para ACs de gran tamaño.

Una solución simple es realizar un intercambio de búfers. En vez de tener un estado fijo, se usa un apuntador de estado para intercalar entre dos búfers. Si bien esto resuelve el problema de copiado, se requiere duplicar la cantidad de memoria necesaria para procesar el ACE. [46]

Podemos notar que después de procesar una célula, ésta puede ser colocada en áreas que ya hayan sido procesadas. El truco está en realizar una rotación de la cinta de la siguiente manera:

1. Guardar el valor de la primera y la última célula de la cinta.
2. Para cada una de las demás células, procesar su vecindad con normalidad, y almacenar la célula resultante una posición a la izquierda, efectivamente rotando la cinta un lugar.
3. Para procesar la primera y la última célula, utilzar en su vecindad el valor correspondiente guardado en el paso 1.
4. Ahora que el comienzo de la cinta ya no está en la posición inicial, es necesario tener una variable que apunte al nuevo inicio de la cinta, el cual deberá recorrerse una posición a la izquierda con cada iteración.

De esta manera, se garantiza un procesamiento in-place del estado del ACE. La figura 8.1 muestra la ejecución del algoritmo de procesamiento in-place para un ACE regla 110 con una cinta de tamaño 4.

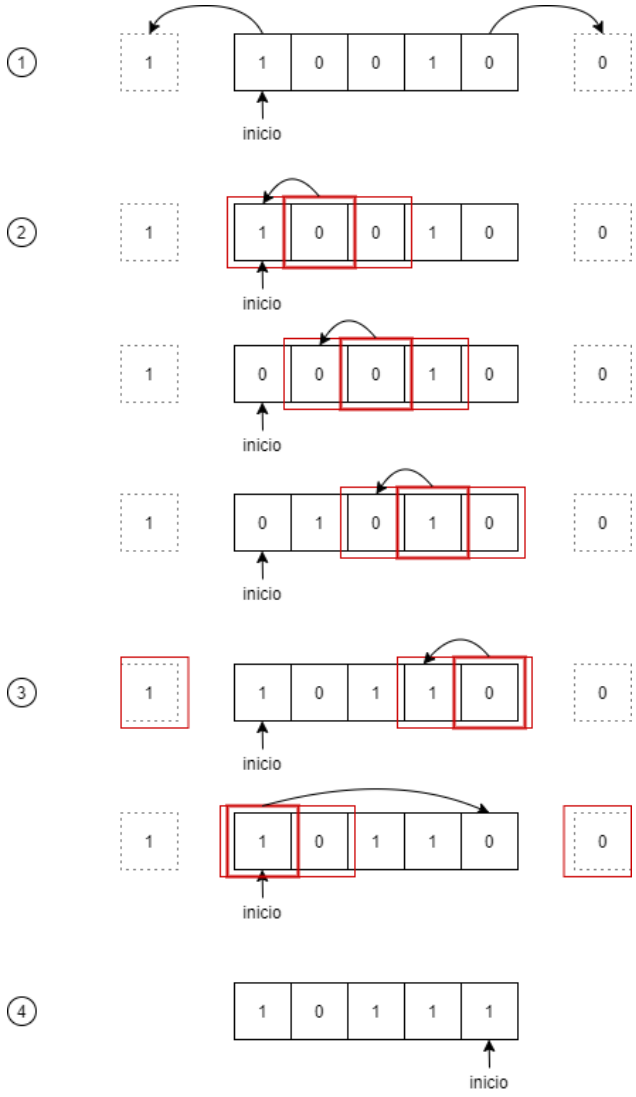

Figura 8.1: Ejemplo del algoritmo de procesamiento in-place para un ACE regla 110.

8.1.2. Compresión de células

Ya que el objeto de este trabajo es el análisis de ACEs de dos estados (cero y uno), podemos optimizar el almacenamiento de las células del autómata. Es posible comprimir un conjunto de m células, siendo m una potencia de 2, en bits de una variable de tamaño respectivo; por ejemplo, comprimiendo un bloque de 32 células en un solo valor entero de 32 bits, e.g. una variable *int* en C++. Así, una cinta de tamaño N puede comprimirse hasta $N \ mod \ m$. Es necesario mantener el número de bits útiles del último bloque de la cinta, en caso de que $N \ mod \ m$ sea distinto a cero. [46]

La figura 8.2 muestra el método de compresión de la cinta de un ACE de 14 células en bloques de 4 bits.

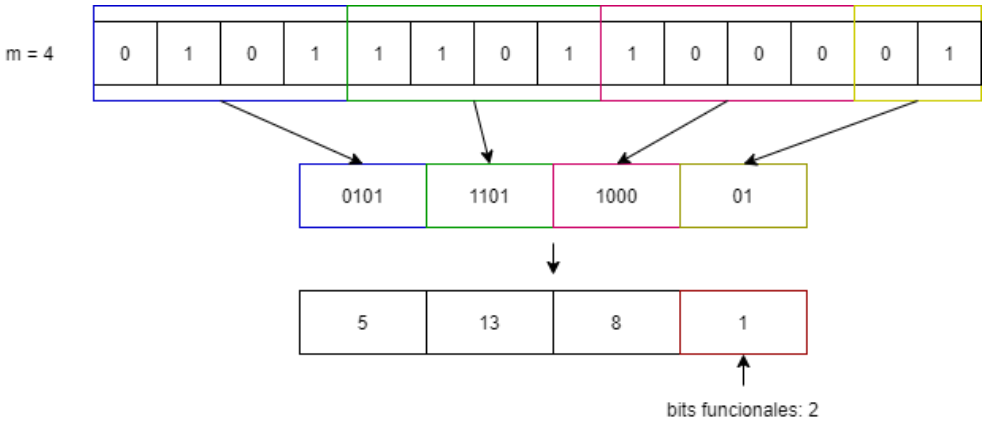

Figura 8.2: Ejemplo del método de compresión para un ACE de dos estados con $m = 4$.

Naturalmente, es necesario agregar la lógica de extracción de bits de cada bloque, un asunto trivial y bastante eficiente mediante operaciones a nivel de bits. De igual forma, es fácil ver que la compresión afecta mínimamente al procesamiento in-place anteriormente implementado, recorriendo por segmentos de tamaño m para el procesamiento y comprimiendo para su almacenamiento in-place.

8.1.3. Lookup tables

La naturaleza determinista de los ACEs permite predecir el estado de un conjunto de células para una generación siguiente. Podemos aprovechar este hecho para crear una lookup table de tamaño 2^s, preprocesada con los bloques resultantes de cualquier conjunto de s células. De tal manera, el procesamiento de una cinta se puede hacer en bloques de tamaño s, evitando así un cálculo 1 a 1 de cada célula.

La implementación de la lookup table es relativamente sencilla si suponemos $s = m$, con m de la sección anterior. Podemos crear una lookup table tridimensional, de dimensiones $m \times 2 \times 2$. Así, para cada bloque de tamaño m, su siguiente estado estará dado por el valor comprimido del bloque, el valor de la última célula del bloque a su izquierda, y la primera célula del bloque a su derecha. [46] La lógica del procesamiento in-place sigue siendo la misma.

La figura 8.3 muestra el funcionamiento esta lookup table para una cinta dividida en bloques de tamaño 4.

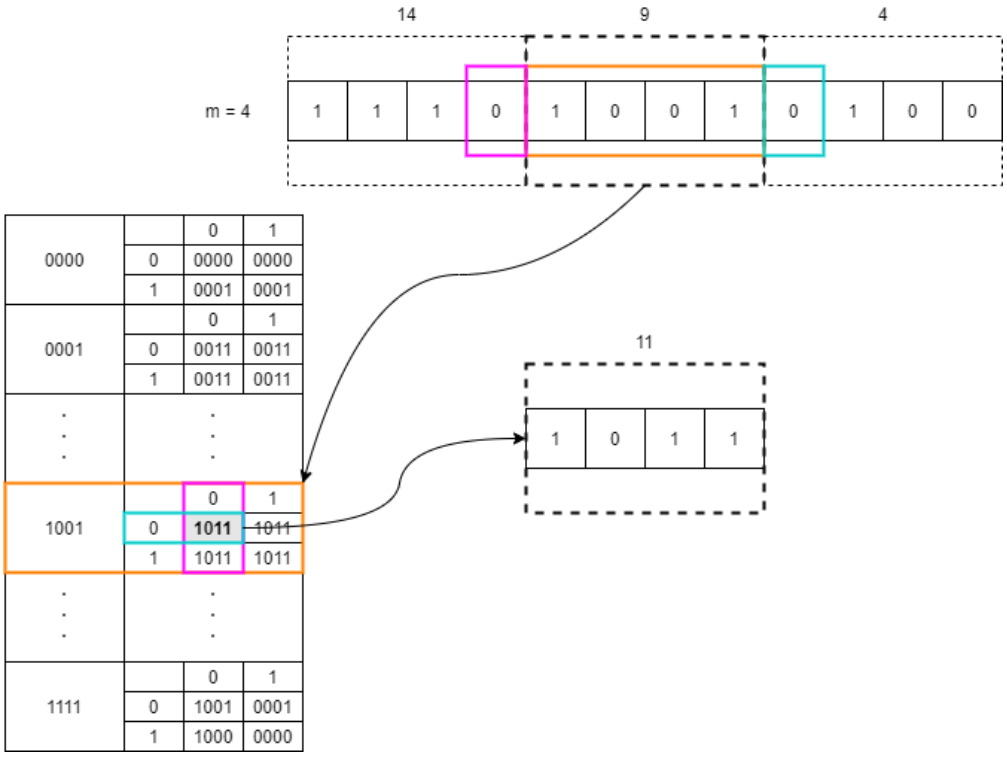

Figura 8.3: Ejemplo de lookup table para un ACE comprimido con factor $m = 4$.

El tamaño de la lookup table dependerá entonces del factor de compresión m, por lo que es importante considerar el impacto en memoria que esto tendrá. Una $m = 32$, por ejemplo, permitirá almacenar una cinta en un arreglo de enteros de 32 bits, reduciendo marginalmente el espacio necesario para los estados del ACE. Sin embargo, esto implicaría la creación de una lookup table de tamaño 2^32, bits, es decir, 536MB. Para ms pequeñas, el ahorro en procesamiento es despreciable con respecto al método trivial de ejecución, y conforme m crece el espacio requerido aumenta de manera exponencial, evidentemente con complejidad $O(2^m)$.

A pesar de este problema, implementar una lookup table grande puede resultar útil en algunos casos, ya que, si la regla del ACE y m son constantes, la lookup table puede crearse una sola vez. En una aplicación que requiera la creación de múltiples instancias de ACEs, el espacio exponencial que conlleva puede valer la pena. No obstante, para los motivos de este trabajo, el beneficio que puede brindar es despreciable, por lo que se decidió no proseguir con la implementación.

8.1.4. Optimización en el filtrado de mosaicos T3

Debido a la altura de un mosaico T3, es necesario utilizar un arreglo de 4 búfers para poder reconocer si en un determinado momento se va a generar un mosaico o no, por lo que el sistema debe estar siempre 4 generaciones adelantado a lo que se va a mostrar en pantalla. Con cada iteración, los búfers se recorren un lugar, y el último se llena con la nueva cinta obtenida después de procesar el ACE, y posteriormente se realiza un filtrado de los mosaicos en el nuevo arreglo.

Originalmente, el reconocimiento de estos mosaicos se realizaba mediante fuerza bruta, buscando, célula por célula, si coincidían con el patrón que construye en sus 4 generaciones: 1111, 1000, 1001 y 10. Sin embargo, esto resultó en un alentamiento significativo en la ejecución de ACEs de gran tamaño, una problemática importante para el rendimiento del colisionador de este trabajo.

Podemos notar que el problema se reduce a uno de reconocimiento de patrones en cadenas, en el que es necesario buscar ocurrencias de los 4 patrones del T3. Teniendo un arreglo de búfers de apoyo, podemos marcar las posiciones en las que aparece cada patrón, y si los 4 aparecen alineados en el orden correcto, habremos localizado un mosaico.

A continuación se analizarán distintos métodos de reconocimiento de patrones para implementar en el proceso de filtrado. Posteriormente, se comparará su desempeño para decidir en la implementación final.

Algoritmo KMP

KMP es un algoritmo efectivo para encontrar un sólo patrón en un texto. Sin embargo, para el fin de este trabajo, es necesario procesar arreglos LPS distintos para cada uno de los 4 patrones del mosaico. En cada iteración de la cinta del ACE, los 4 arreglos se actualizan a la vez, cada uno alertando de una coincidencia de su patrón respectivo.

La figura 8.4 muestra los 4 LPSs ya procesados. Esto sólo necesita hacerse una vez, por lo que el preprocesamiento en un futuro es innecesario y la cantidad de memoria requerida es despreciable.

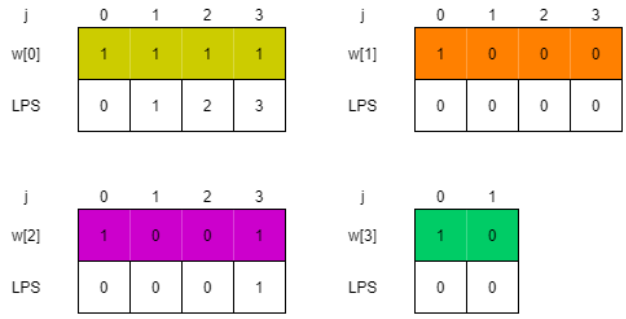

Figura 8.4: Arreglos LPS para cada uno de los patrones del mosaico T3 en KMP.

No obstante, la forma asimétrica de los patrones del mosaico hacen a los LPSs virtualmente obsoletos. Como podemos observar, sólo el patrón "1111"posee una LPS que puede ser relevante para omitir comparaciones dentro del algoritmo. Los patrones "1000", "1001z "10çontienen prácticamente cero información que pueda ayudar en la optimización del filtrado.

Algoritmo Aho-Corasick

Aho-Corasick es un algoritmo diseñado originalmente para la búsqueda de patrones múltiples, algo que para nuestros propósitos será de mucha utilidad.

Similar a KMP, la estructura necesaria para el algoritmo puede ser preprocesada una sola vez. La figura 8.5 muestra cómo estará construida esta estructura.

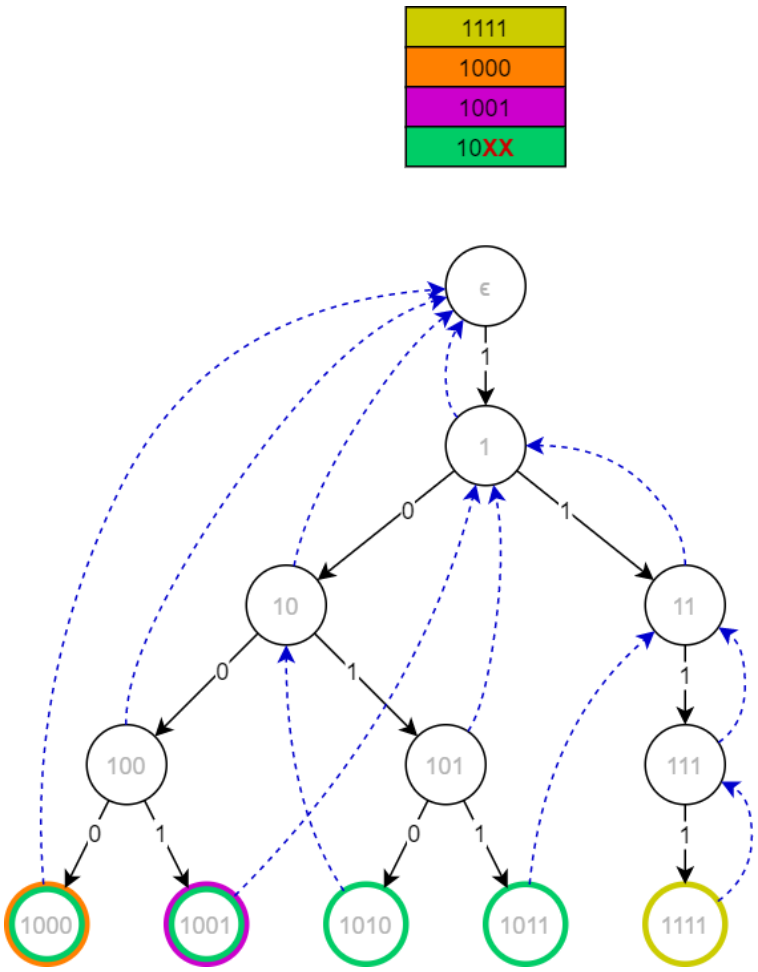

Figura 8.5: Estructura preprocesada para los patrones del mosaico T3 en Aho-Corasick.

Ya que el patrón "10.es el más corto de los 4, y de hecho el único que no es de longitud 2, se modificó la estructura para reducir la complejidad del recorrido. En este caso, se usará la cadena "10**XX**", donde '**X**' puede ser uno o cero. Así pues, consideraremos una ocurrencia para cualquiera de las hojas de la estructura, y para las cadenas "1000z "1001çonsideraremos una doble ocurrencia.

Algoritmo Rabin-Karp

Analogo a KMP, Rabin-Karp puede utilizar un diccionario mediante la comparación de más de un valor hash. Ya que el manejo de ventanas de distintos tamaños agrega una capa extra de complejidad, un procesamiento similar se hará para el patrón "10", de tal forma que tendremos 4 diferentes valores hash a comparar para "10**XX**".

La figura 8.6 muestra cuáles serían los valores hash a localizar para cada patrón del mosaico. Utilizaremos el alfabeto binario y el Rabin Fingerprint con una r trivial para el rolling hash.

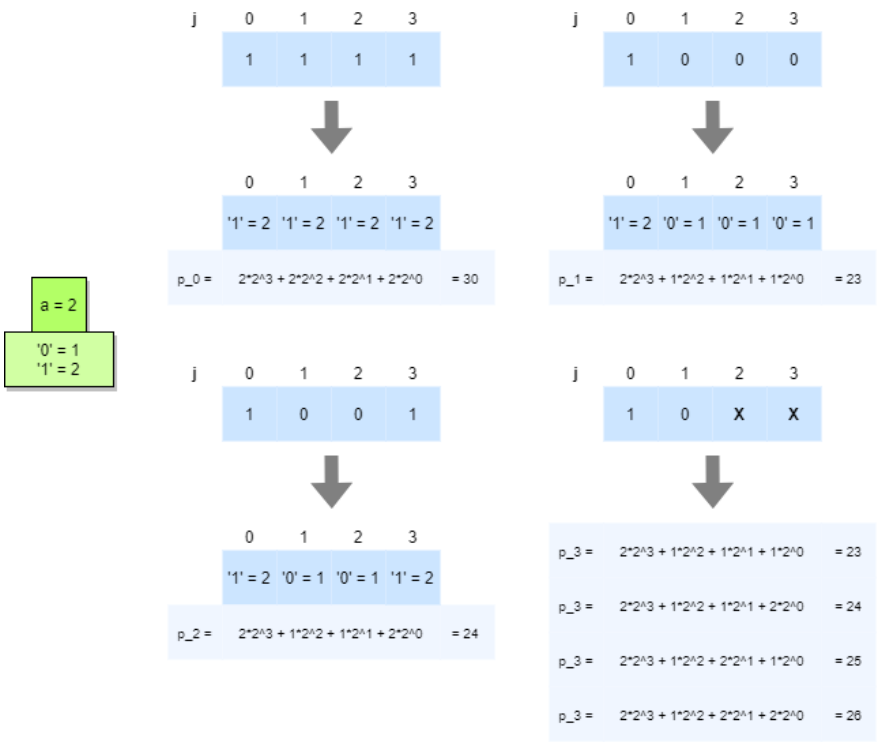

Figura 8.6: Valores hash para los patrones del mosaico T3 en Rabin-Karp

8.2. Análisis

Los requerimientos para el análisis de esta iteración son idénticos a los de la iteración 4, por lo que su presentación se omitirá.

8.3. Diseño

Siguiendo del análisis, los diagramas de casos de uso y diagramas de actividades se heredan de la iteración 4. Se presentan las modificaciones al diagrama de clases para su posterior implementación.

8.3.1. Diagrama de clases

La figura 8.7 muestra el diagrama de clases para la iteración 4.5.

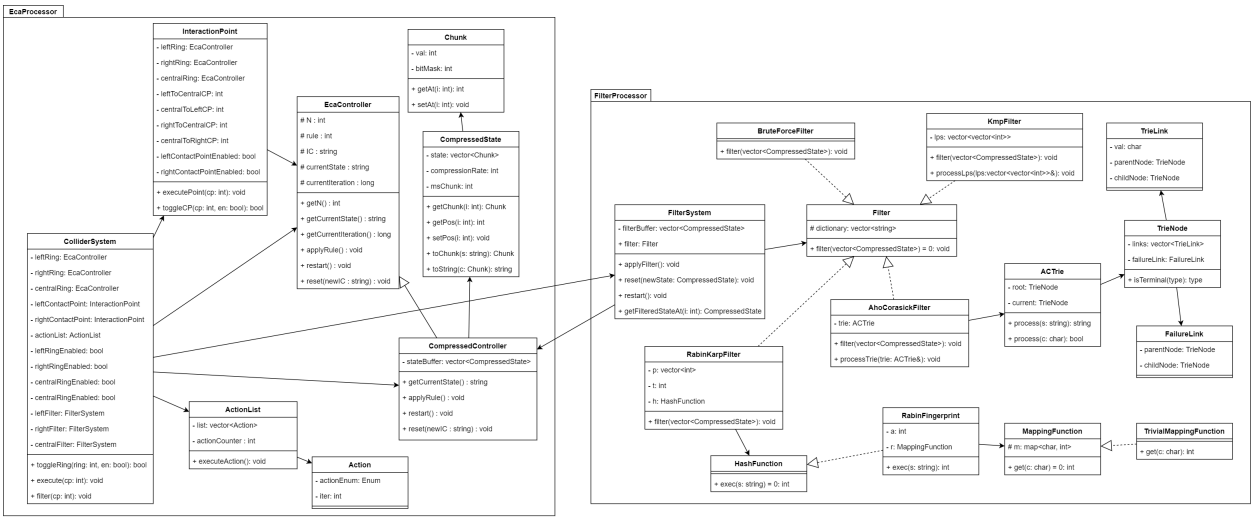

Figura 8.7: Diagrama de clases para la iteración 4.5.

La lógica asociada al control del colisionador y los ACEs se traslada a un módulo separado, *EcaProcessor*. *EcaController* fue refactorizado para extender un nuevo tipo de controlador, *CompressedController*. Aquí es donde se implementa la optimización mencionada en el preámbulo. *CompressedState* realiza las operaciones a nivel de bits para la extracción y actualización en los *Chunk*s de 32 bits que componen a un estado. Por otro lado, *CompressedController* se encarga de mantener el registro del *Chunk* más significativo de un estado, para el procesamiento in-place de las iteraciones del ACE.

Para el filtrado, se decidió crear un nuevo módulo, *FilterProcessor*. Aquí, *FilterSystem* mantiene los búfers necesarios del controlador para el análisis de los estados de *CompressedController*, utilizando una interfaz *Filter* que permite implementar diferentes algoritmos de filtrado. Se han construido 4 diferentes implementaciones de *Filter*:

1. *BruteForceFilter* implementa el algoritmo de búsqueda por fuerza bruta.

2. *KmpFilter* implementa el algoritmo KMP. Se crea una instancia estática de vectores de *LPS*s para la ejecución del procedimiento.

3. *AhoCorasickFilter* implementa el algoritmo Aho-Corasick. Utiliza una estructura personalizada, *ACTrie*, instanciada estáticamente, que mantiene el registro de los *TrieNode*s del árbol. Cada *TrieNode* posee una lista de *TrieLink*s para enlazar a otros nodos, así como un *FailureLink* para la optimización del algoritmo.

4. *RabinKarpFilter* implementa el algoritmo Rabin-Karp. Utiliza una interfaz *HashFunction* que permite implementar diferentes funciones de rolling hash, pero sólo se implementó *RabinFingerprint* para este trabajo. Por su lado, *RabinFingerprint* puede implementar diferentes funciones de mapeo a través de la interfaz *MappingFunction*, pero sólo se implementó un mapeo trivial mediante *TrivialMapping*.

La figura 8.8 muestra el diagrama general del sistema en la iteración 4.5, incluyendo el módulo de visualización del colisionador y el de traducción de expresiones R110. Nuevamente, se omite el diagrama de la iteración 3 para mejor visibilidad.

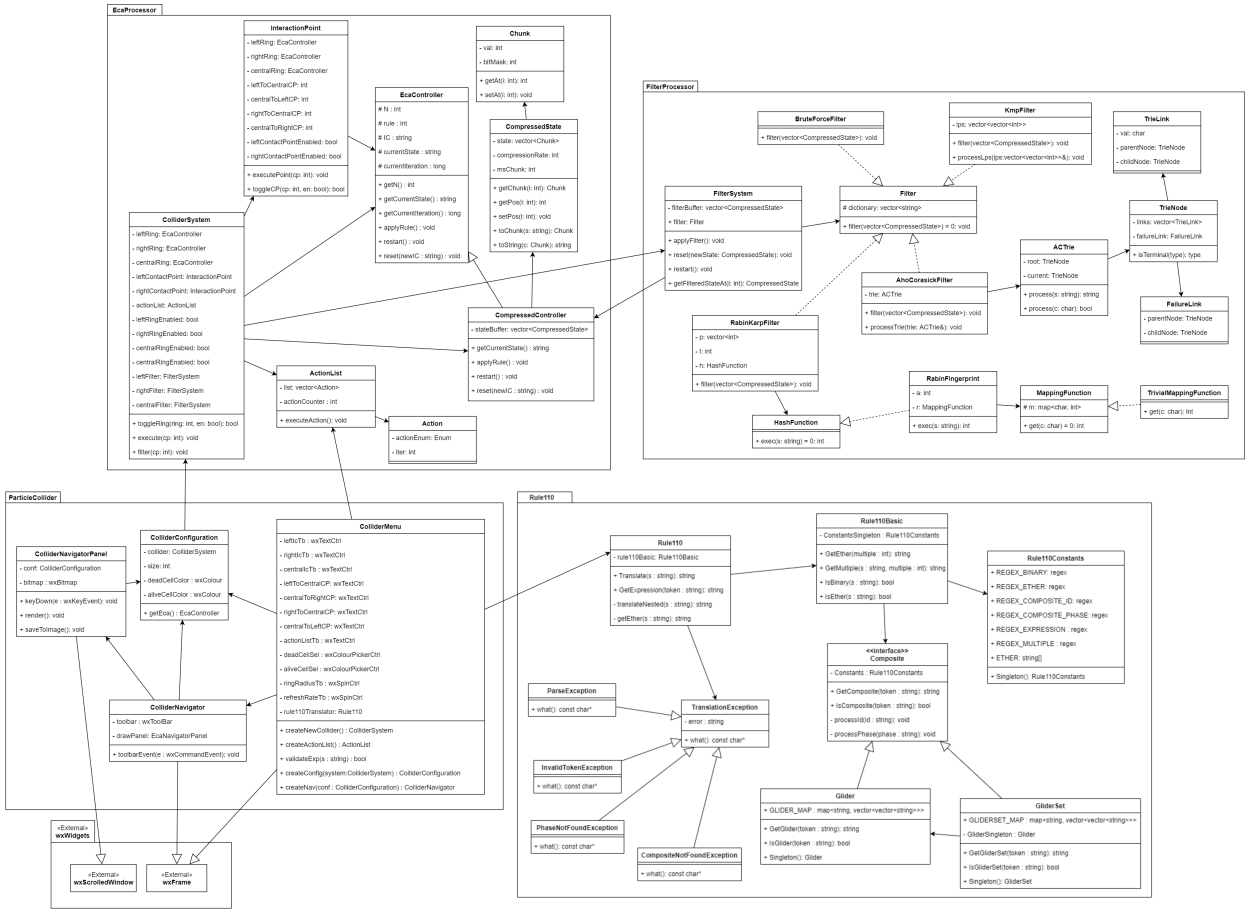

Figura 8.8: Diagrama de clases general de la iteración 4.5.

8.4. Implementación y pruebas

N	10	100	1,000	10,000	100,000
Fuerza Bruta	77	5,991	552,821	26,055,425	361,814,598
KMP	86	5,575	448,670	19,820,138	222,985,463
Aho-Corasick	78	4,224	179,629	3,201,989	12,059,344
Rabin-Karp	79	4,968	158,877	3,955,237	18,950,014

Cuadro 8.1: Rendimiento promedio (en nanosegundos) de los algoritmos de filtrado para los patrones de un mosaico T3.

El cuadro 8.1 muestra los rendimientos promedio de cada uno de los algoritmos de filtrado para los patrones del T3. La figura 8.9 ilustra gráficamente estos datos. Para poder apreciar mejor las diferencias, se hizo uso de una escala logarítmica para el tiempo de ejecución.

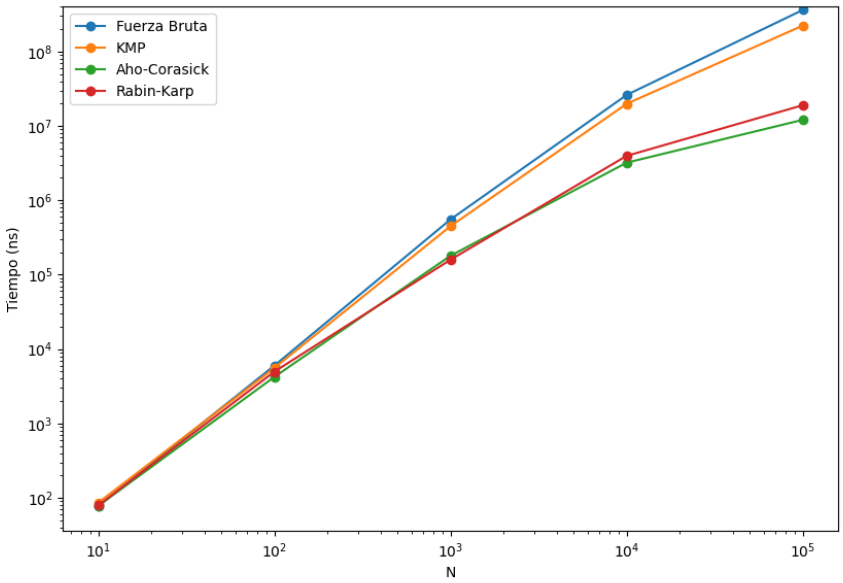

Figura 8.9: Comparativa logarítmica de los algoritmos de filtrado para los patrones de un mosaico T3.

Podemos notar que KMP posee un rendimiento muy cercano a la fuerza bruta, un resultado esperado considerando que las LPSs que genera el T3 no poseen mucha información para evitar comparaciones adicionales. En cambio, Aho-Corasick y Rabin-Karp ofrecen un salto enorme en eficiencia contra la fuerza bruta, de un orden de magnitud.

A pesar de que estos dos últimos tienen un desempeño bastante similar al principio, Aho-Corasick rebasa a Rabin-Karp para entradas grandes. Su desventaja es el costo espacial, $O(km_{max})$ contra $O(1)$ de Rabin-Karp. No obstante, debido a que el diccionario de patrones será constante, no es necesario procesarlo más de una ve. Ésto, combinado con el hecho de que $k = 4$ y $m_{max} = 4$ generan una estructura de tamaño despreciable en memoria, nos permite seleccionar a Aho-Corasick como el algoritmo ideal a implementar para este trabajo.

Capítulo 9

Implementación del simulador de CTS

El simulador de ACE y el colisionador virtual serán la base para la implementación del CTS. A continuación se presentarán los resultados del análisis de este sistema.

9.1. Descripción de colisiones

Para entender la ejecución del simulador, es necesario profundizar en cada una de las interacciones que suceden durante el proceso descrito en la subsección *Funcionamiento del CTS en la regla 110*.

9.1.1. Activación de sectores de procesamiento

La primera colisión en el sistema sucede durante la activación del primer sector de procesamiento. En ella, la partícula líder del sector, $SepInit\_E\bar{E}^{-}$, es activada mediante una colisión con una partícula A^{3+}. El enrutamiento que describe este proceso es:

$$(A^{3+} \Leftrightarrow SepInit\_E\bar{E}^{-}) \rightarrow (SepInit\_E\bar{E}_P{}^{-})$$

La figura 9.1 muestra esta interacción en un espacio aislado.

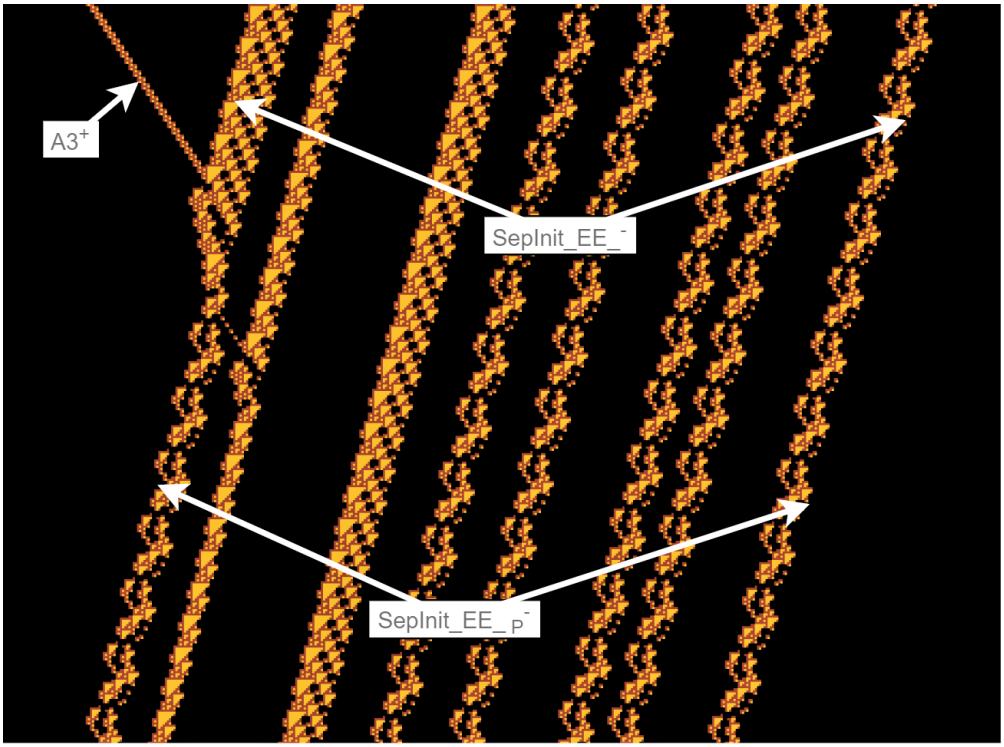

Figura 9.1: Activación de un sector de procesamiento mediante la colisión $(A^{3\,+} \Leftrightarrow SepInit\_E\bar{E}^{\,-})$ $\rightarrow (SepInit\_E\bar{E}_P{}^{-})$.

Es importante notar que, en colisiones posteriores, pueden generarse trenes de 3 partículas $A^{\,+}$ en vez de una única partícula $A^{3\,+}$. Sin embargo, si se mantiene una fase adecuada, la reacción con un $SepInit\_E\bar{E}^{\,-}$ será la misma:

$$(3\ A^{\,+} \Leftrightarrow SepInit\_E\bar{E}^{\,-}) \rightarrow (SepInit\_E\bar{E}_P{}^{-})$$

La figura 9.2 muestra esta interacción.

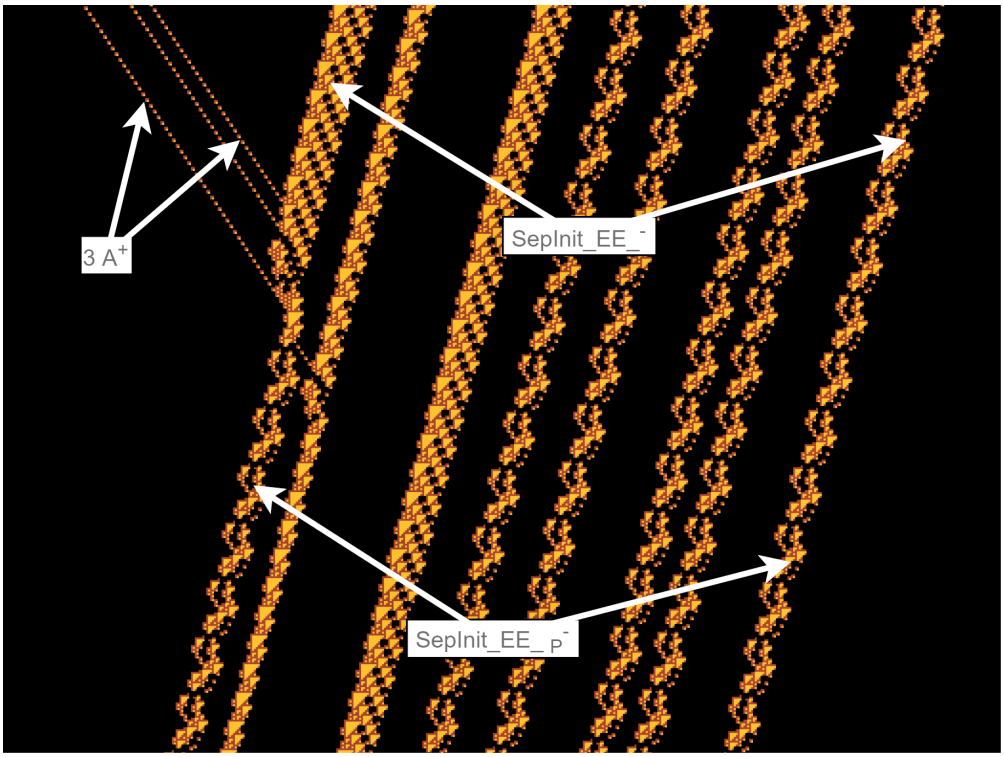

Figura 9.2: Activación de un sector de procesamiento mediante la colisión ($3\ A^{+} \Leftrightarrow SepInit\_E\bar{E}^{-}$) $\rightarrow$ ($SepInit\_E\bar{E}_P{}^{-}$).

En la siguiente subsección veremos que esta diferencia parte de la generación de partículas eliminadoras y colapsadoras, respectivamente.

9.1.2. Lectura de cinta

Una partícula activada $SepInit\_E\bar{E}_P{}^{-}$ interactúa con elementos $Ele\_C_2{}^{0}$ simulando la lectura y eliminación del dato en la cabeza de la cinta en el CTS.

Lectura de un cero

Si la colisión sucede con un elemento $0Ele\_C_2{}^{0}$, el sector entrará en modo de destrucción de datos, impidiendo la creación de valores en la cinta mediante la generación de una partícula $A^{3}{}^{+}$ que actuará como un eliminador. Como residuo de la colisión, se generan 2 partículas $\bar{E}^{-}$ que no interactuarán con el resto del sistema, funcionando como solitones cuando se encuentren con alguna otra partícula. El enrutamiento que describe este proceso es:

$$(0Ele\_C_2{}^{0} \Leftrightarrow SepInit\_E\bar{E}_P{}^{-}) \rightarrow (\bar{E}^{-}, \bar{E}^{-}, A^{3}{}^{+})$$

La figura 9.3 muestra esta interacción.

Lectura de un uno

Si la colisión sucede con un elemento $1Ele\_C_2{}^{0}$, el sector entrará en modo de colapso de datos, transformando cada uno de los bloques del sector para producir nuevos valores en la cinta, mediante la generación de un tren de 3 partículas A^{+} que actuarán como un colapsador. Similar a la lectura de un cero, se generan 2 partículas residuales $\bar{E}^{-}$ que actuarán como solitones con el resto del sistema. El enrutamiento que describe este proceso es:

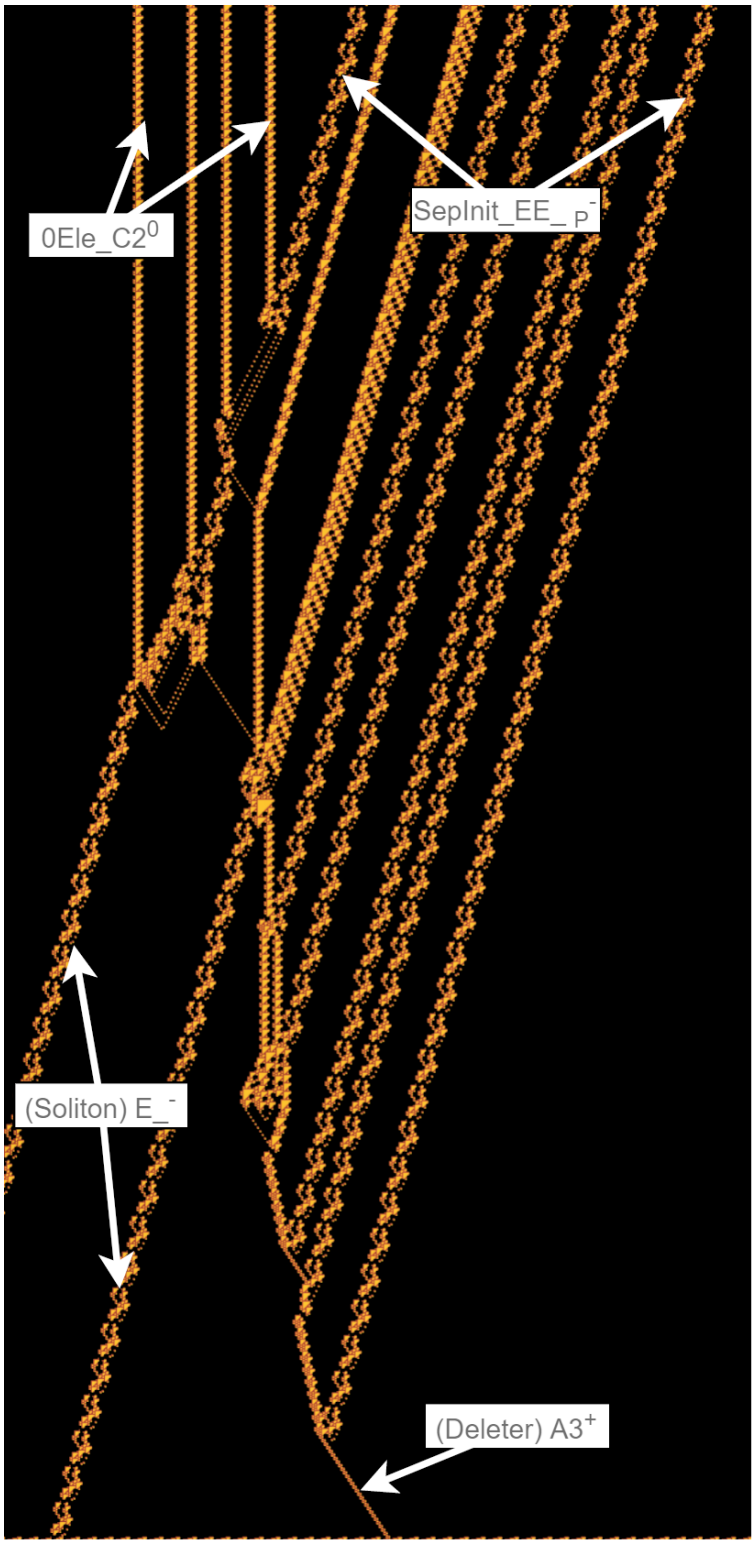

Figura 9.3: Lectura de un cero mediante la colisión $(0Ele\_C_2{}^{0} \Leftrightarrow SepInit\_E\bar{E}_P{}^{-}) \rightarrow (\bar{E}{}^{-}, \bar{E}{}^{-}, A^{3+})$.

$$(1Ele\_C_2{}^{\,0} \Leftrightarrow SepInit\_E\bar{E}_P{}^{\,-}) \rightarrow (\bar{E}{}^{\,-}, \bar{E}{}^{\,-}, 3\ A^{\,+})$$

La figura 9.4 muestra esta interacción.

9.1.3. Destrucción de datos

Después de la lectura de un cero en la cinta, el sector entrará en modo de destrucción de datos. En él, todos los bloques $Blo\_\bar{E}{}^{\,-}$ del sector serán eliminados, evitando la producción de nuevos valores en la cinta. Al terminar, la única partícula resultante será el eliminador original, el cual procederá a destruir el siguiente bloque de datos.

Destrucción de un cero

El enrutamiento que describe la destrucción de un bloque $0Blo\_\bar{E}{}^{\,-}$ es:

$$(A^{3\,+} \Leftrightarrow 0Blo\_\bar{E}{}^{\,-}) \rightarrow (A^{3\,+})$$

La figura 9.5 muestra esta interacción.

Destrucción de un uno

El enrutamiento que describe la destrucción de un bloque primario $1BloP\_\bar{E}{}^{\,-}$ es:

$$(A^{3\,+} \Leftrightarrow 1BloP\_\bar{E}{}^{\,-}) \rightarrow (A^{3\,+})$$

Similarmente, el enrutamiento para la destrucción de un bloque estándar $1BloS\_\bar{E}{}^{\,-}$ es:

$$(A^{3\,+} \Leftrightarrow 1BloS\_\bar{E}{}^{\,-}) \rightarrow (A^{3\,+})$$

La figura 9.6 la destrucción de un bloque $1BloP\_\bar{E}{}^{\,-}$.

9.1.4. Colapso de datos

Después de la lectura de un uno en la cinta, el sector entrará en modo de colapso de datos. En él, todos los bloques $Blo\_\bar{E}{}^{\,-}$ del sector serán transformados en partículas de adición $Add\_\bar{E}{}^{\,-}$, las cuales serán procesadas posteriormente para ser añadidas a la cinta del CTS. Al terminar, se generará el colapsador original, el cual procederá a colapsar el siguiente bloque de datos.

Colapso de un cero

El enrutamiento que describe el colapso de un bloque $0Blo\_\bar{E}{}^{\,-}$ es:

$$(3\ A^{\,+} \Leftrightarrow 0Blo\_\bar{E}{}^{\,-}) \rightarrow (0Add\_\bar{E}{}^{\,-}, 3\ A^{\,+})$$

La figura 9.7 muestra esta interacción.

Colapso de un uno

El enrutamiento que describe el colapso de un bloque primario $1BloP\_\bar{E}{}^{\,-}$ es:

$$(3\ A^{\,+} \Leftrightarrow 1BloP\_\bar{E}{}^{\,-}) \rightarrow (1Add\_\bar{E}{}^{\,-}, 3\ A^{\,+})$$

Similarmente, el enrutamiento para el colapso de un bloque estándar $1BloS\_\bar{E}{}^{\,-}$ es:

$$(3\ A^{\,+} \Leftrightarrow 1BloS\_\bar{E}{}^{\,-}) \rightarrow (1Add\_\bar{E}{}^{\,-}, 3\ A^{\,+})$$

La figura 9.8 muestra la destrucción de un bloque $1BloP\_\bar{E}{}^{\,-}$.

Figura 9.4: Lectura de un uno mediante la colisión $(1Ele\_C_2{}^{0} \Leftrightarrow SepInit\_E\bar{E}_P{}^{-}) \rightarrow (\bar{E}{}^{-}, \bar{E}{}^{-}, 3\ A{}^{+})$.

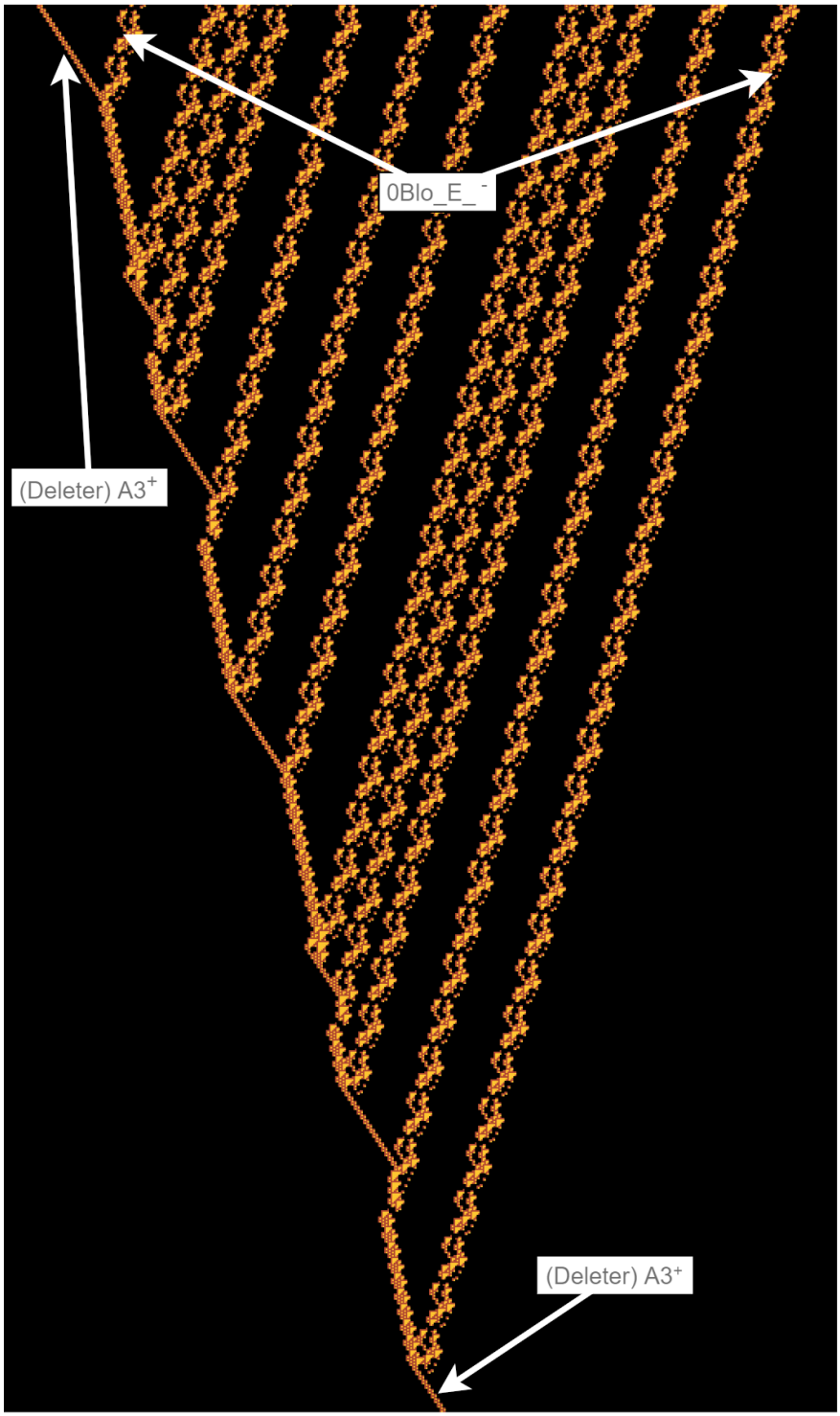

Figura 9.5: Destrucción de un bloque cero mediante la colisión $(A^{3\,+} \Leftrightarrow 0Blo\_\bar{E}^{\,-}) \rightarrow (A^{3\,+})$.

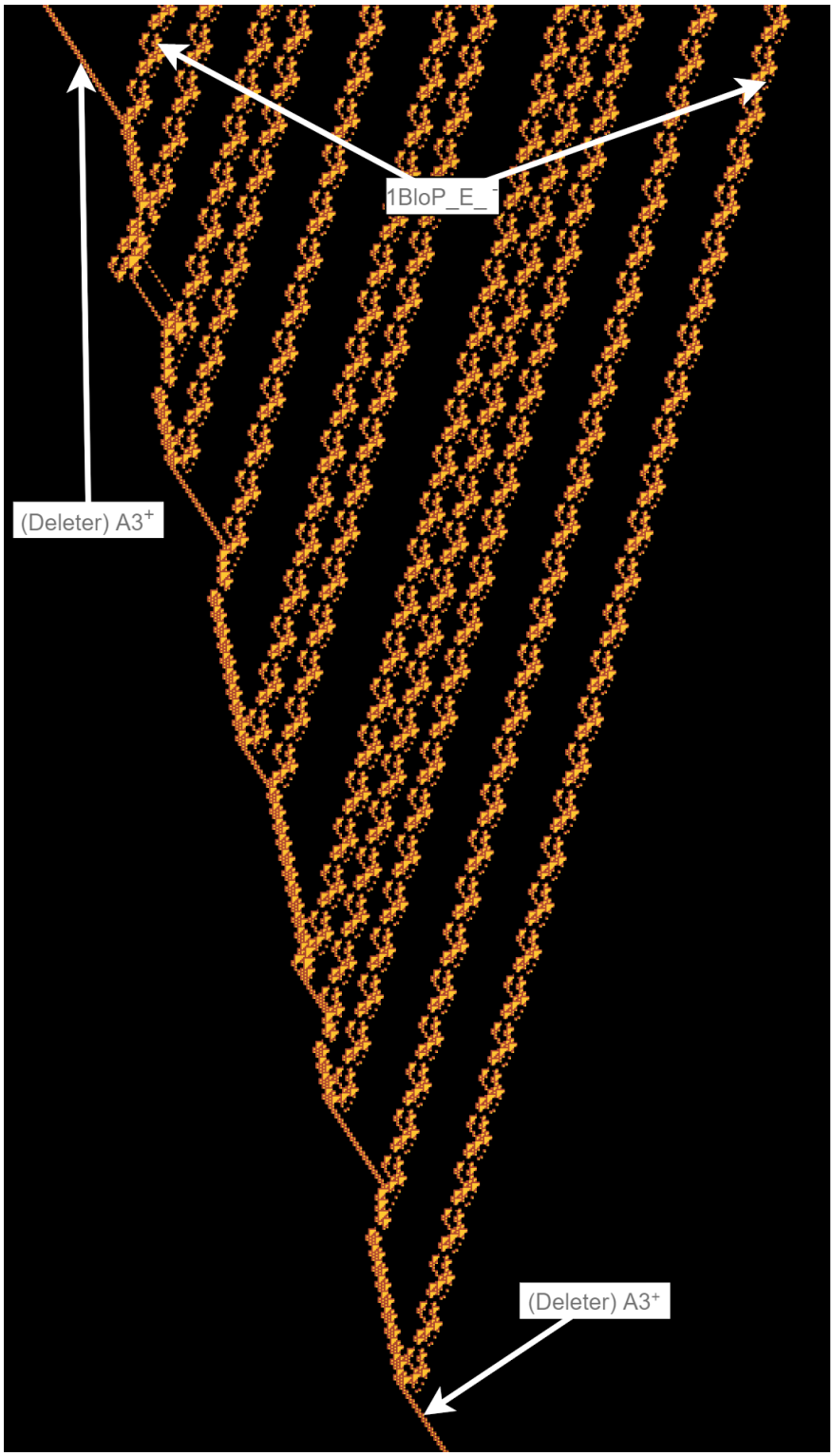

Figura 9.6: Destrucción de un bloque uno primario mediante la colisión $(A^{3\,+} \Leftrightarrow 1BloP\_\bar{E}^{\,-}) \rightarrow (A^{3\,+})$.

Figura 9.7: Colapso de un bloque cero mediante la colisión $(3\ A^{+} \Leftrightarrow 0Blo\_\bar{E}^{-}) \rightarrow (0Add\_\bar{E}^{-}, 3\ A^{+})$.

Figura 9.8: Colapso de un bloque uno primario mediante la colisión $(3\ A^{+} \Leftrightarrow 1BloP\_\bar{E}^{-})$ $\rightarrow (1Add\_\bar{E}^{-}, 3\ A^{+})$.

9.1.5. Interacciones solitónicas

Durante la ejecución del simulador, es vital mantener la sincronización de las fases y distancias de cada partícula para que actúen de la manera deseada, ya que un solo desface puede distorsionar el sistema entero. Esto es especialmente importante durante las interacciones solitónicas entre partículas, en las que una partícula no debe irrumpir en absoluto a las partículas que se le crucen.

Solitones residuales

Los solitones residuales, producto de la lectura de la cinta, intersectan constantemente con los trenes $4\_A^{4\,+}$ de la sección izquierda. La reacción esperada se describe con el siguiente enrutamiento:

$$(4\_A^{4\,+} \Leftrightarrow \bar{E}^{-}) \rightarrow (4\_A^{4\,+}, \bar{E}^{-})$$

La figura 9.9 muestra la interacción solitónica de 4 partículas $\bar{E}^{-}$.

Partículas de adición como solitones

En ocasiones, las partículas de adición $Add\_\bar{E}^{\,-}$ necesitarán interactuar solitónicamente con elementos $Ele\_C_2^{\,0}$ para no distorsionar valores que han sido previamente añadidos a la cinta. Los enrutamientos que dictan este comportamiento son:

$$(iEle\_C_2^{\,0} \Leftrightarrow jAdd\_\bar{E}^{\,-}) \rightarrow (iEle\_C_2^{\,0}, jAdd\_\bar{E}^{\,-})$$

Donde $i, j \in \{0, 1\}$. Por ejemplo, la figura 9.10 muestra la interacción solitónica de una partícula de adición $0Add\_\bar{E}^{\,-}$ con un elemento $1Ele\_C_2^{\,0}$.

Como veremos en la siguiente subsección, esta reacción sólo debe suceder con el objetivo de no corromper elementos $Ele\_C_2^{\,0}$. Una colisión con trenes $4\_A^{4\,+}$ no debería producir una interacción solitónica.

9.1.6. Adición en la cinta

El proceso de colapso de datos da como resultado partículas de adición $Add\_\bar{E}^{\,-}$. Sin embargo una partícula de adición no es en sí un nuevo valor de la cinta del CTS. Es necesaria una última transformación para poder añadir el dato como un elemento $Ele\_C_2^{\,0}$. Para lograrlo, es necesario colisionar la partícula con un tren $4\_A^{4\,+}$.

Adición de un cero

El enrutamiento esperado para añadir un elemento cero a la cinta es:

$$(4\_A^{4\,+} \Leftrightarrow 0Add\_\bar{E}^{\,-}) \rightarrow (0Ele\_C_2^{\,0})$$

La figura 9.11 muestra esta interacción.

Adición de un uno

El enrutamiento esperado para añadir un elemento uno a la cinta es:

$$(4\_A^{4\,+} \Leftrightarrow 1Add\_\bar{E}^{\,-}) \rightarrow (1Ele\_C_2^{\,0})$$

La figura 9.12 muestra esta interacción.

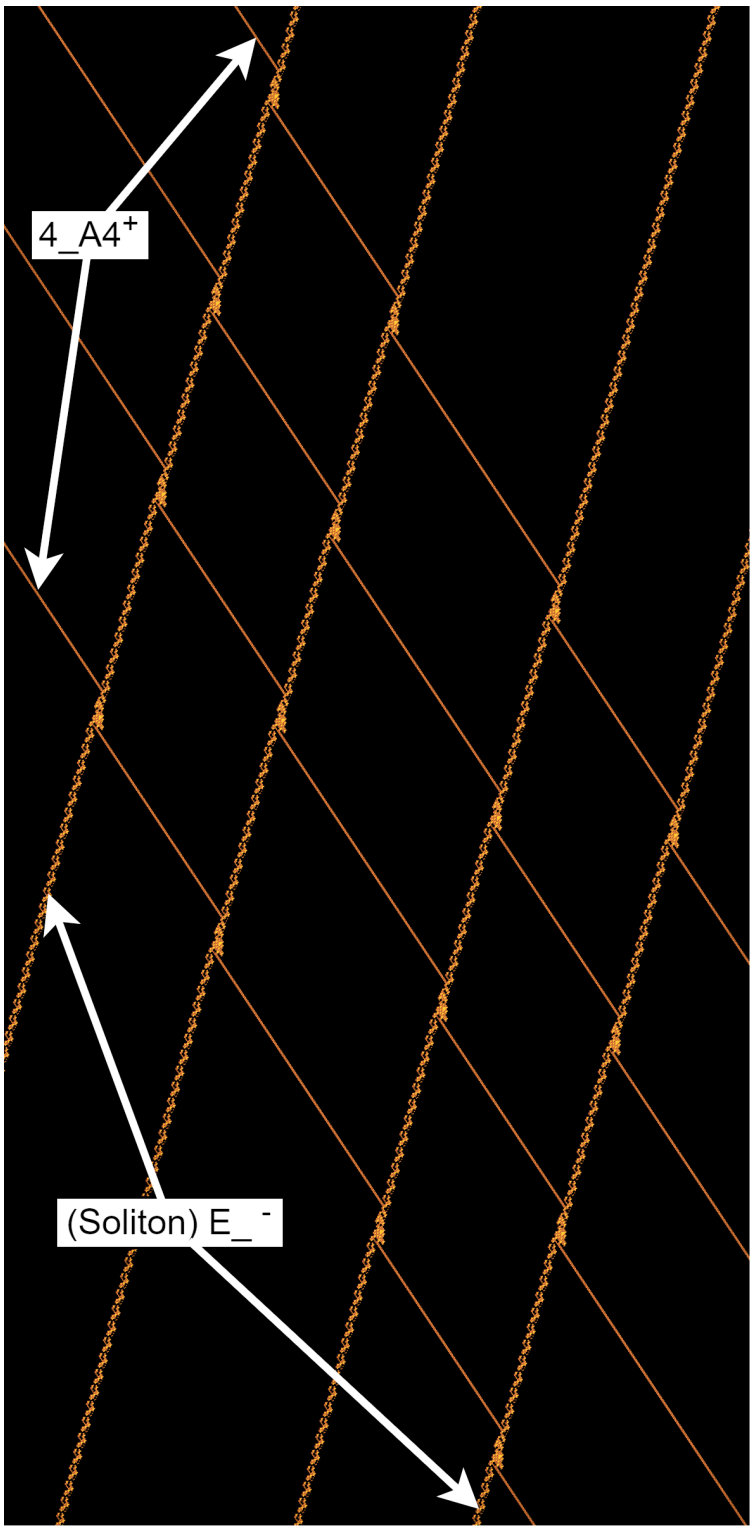

Figura 9.9: Interacción solitónica de partículas residuales mediante colisiones $(4\_A^{4\,+} \Leftrightarrow \bar{E}^{-})$ $\rightarrow (4\_A^{4\,+}, \bar{E}^{-})$.

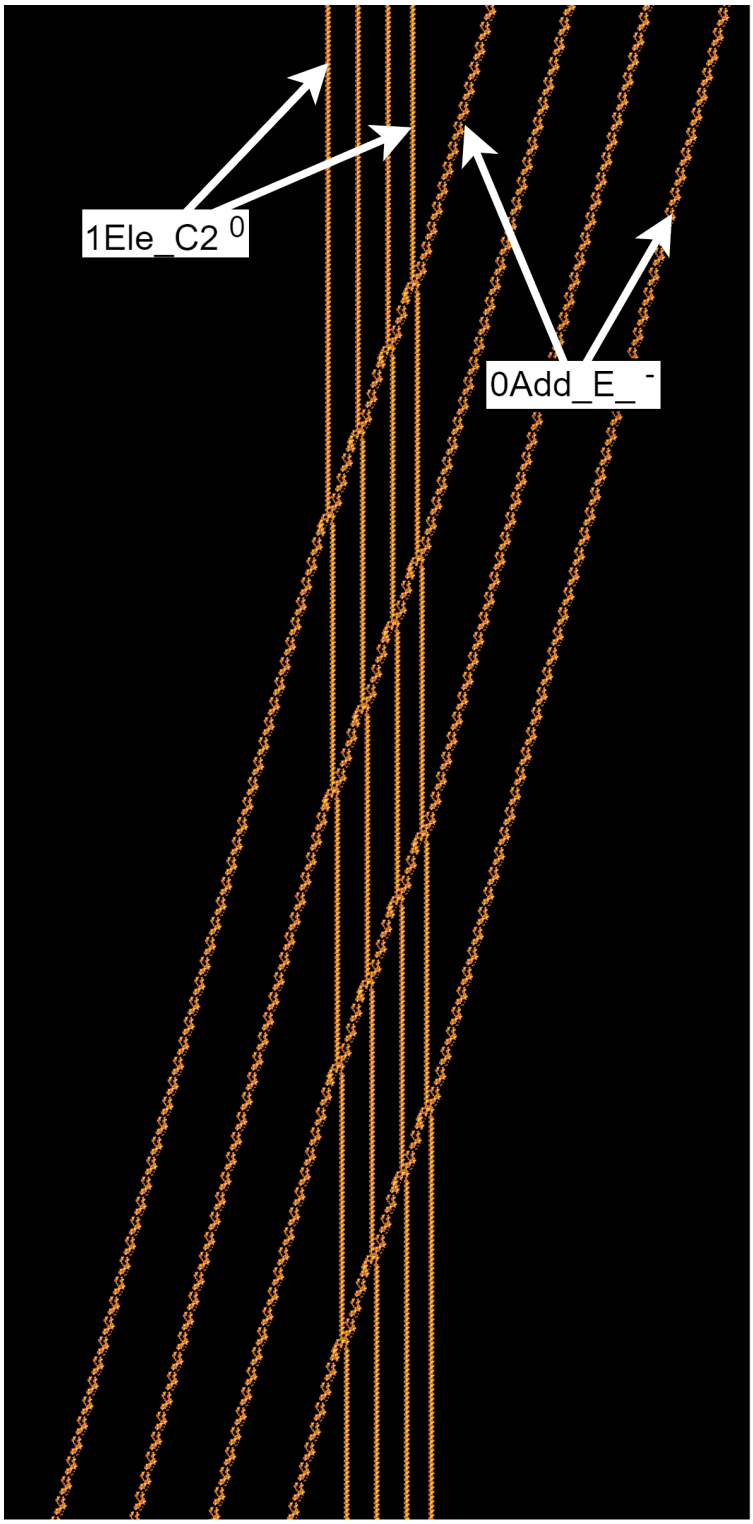

Figura 9.10: Interacción solitónica de una partícula de adición cero con un elemento uno mediante la colisión $(1Ele\_C_2\ ^0 \Leftrightarrow 0Add\_\bar{E}\ ^-) \rightarrow (1Ele\_C_2\ ^0, 0Add\_\bar{E}\ ^-)$.

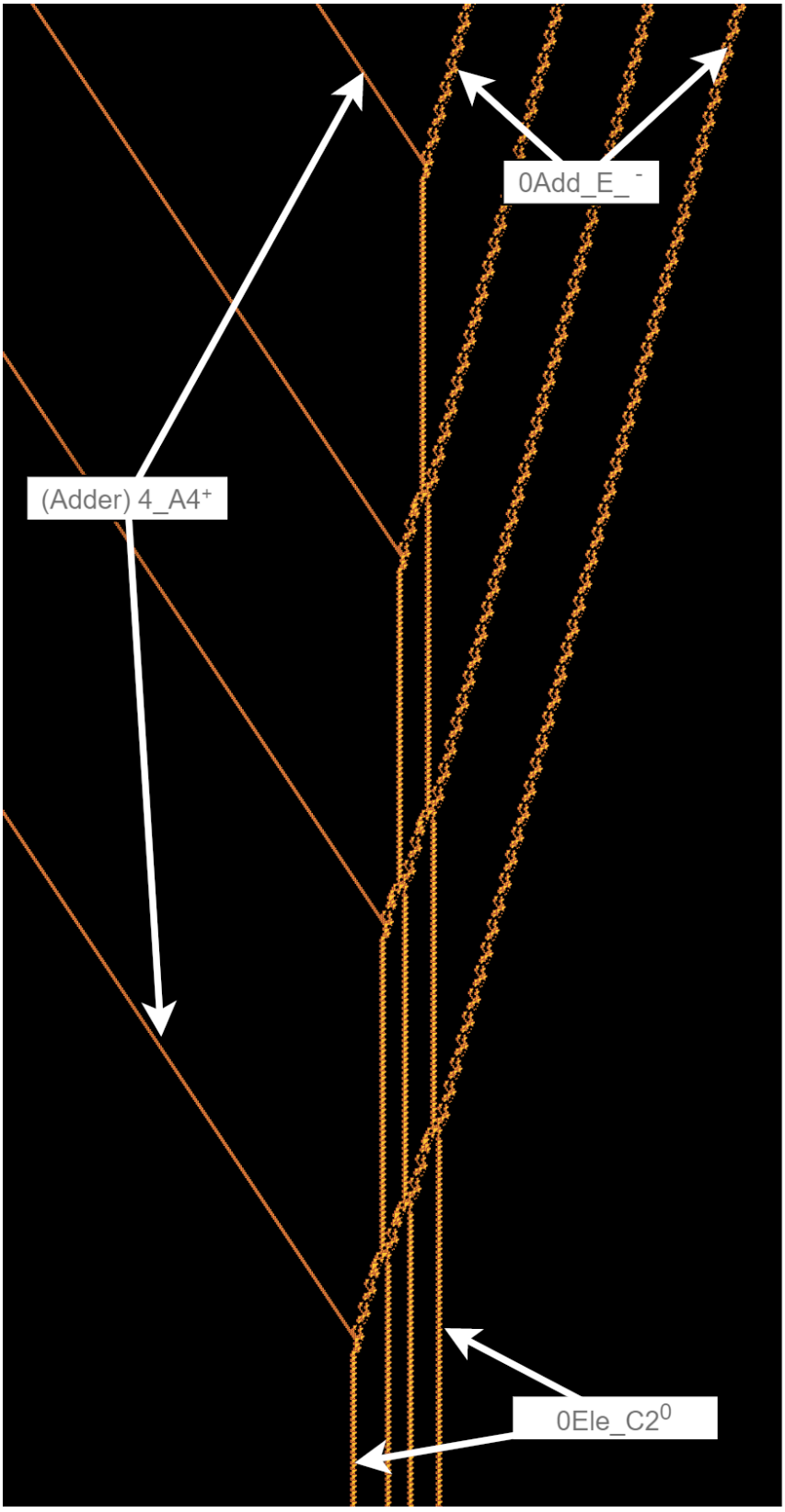

Figura 9.11: Adición de un elemento cero a la cinta mediante la colisión $(4\_A^{4\,+} \Leftrightarrow 0Add\_\bar{E}^{\,-}) \rightarrow (0Ele\_C_2{}^{0})$.

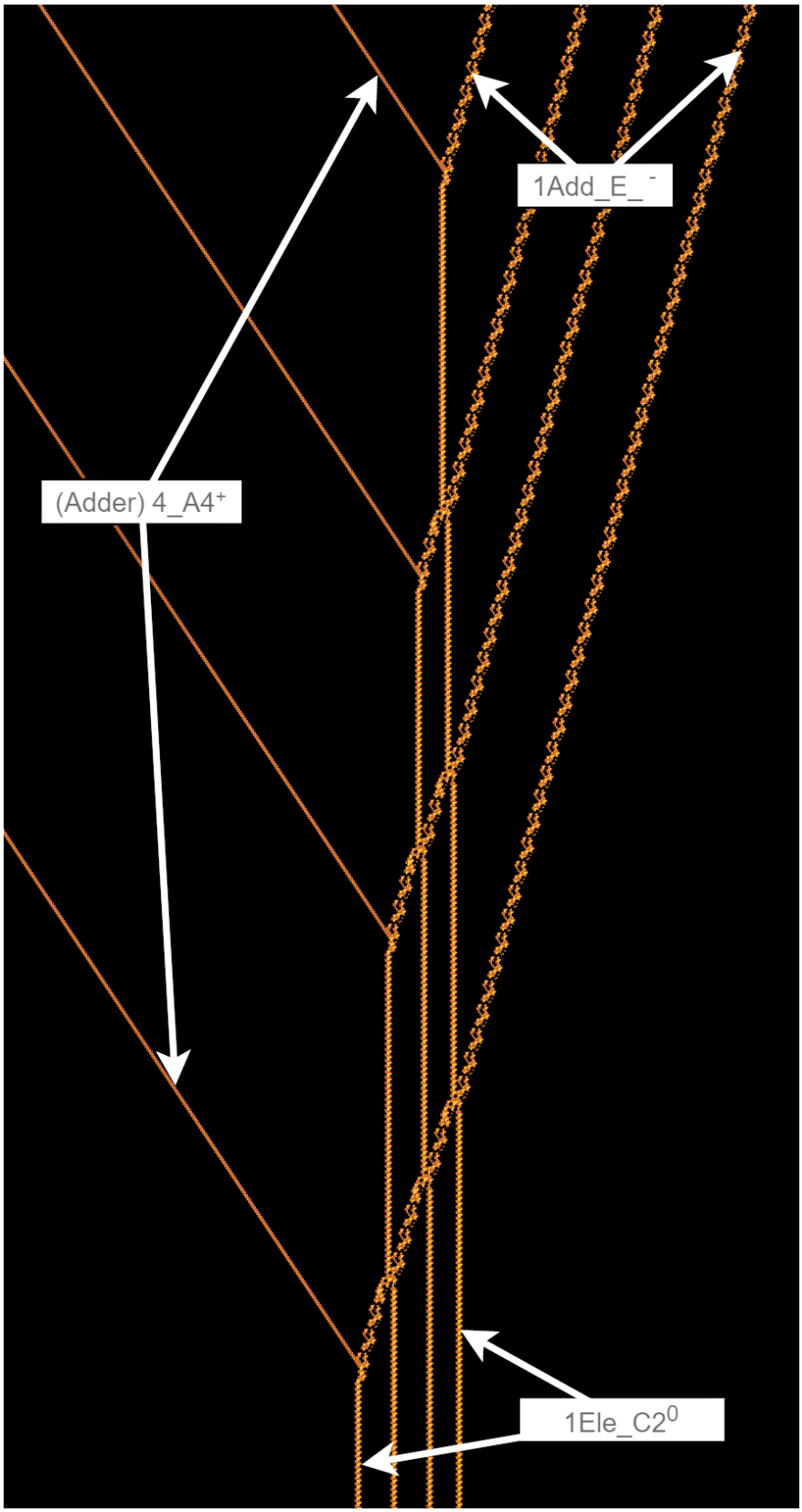

Figura 9.12: Adición de un elemento uno a la cinta mediante la colisión $(4\_A^{4\,+} \Leftrightarrow 1Add\_\bar{E}^{\,-}) \rightarrow (1Ele\_C_2{}^{0})$.

9.2. Funcionamiento del CTS

9.2.1. Modificación de secciones

Recordemos que el sistema del CTS está particionado en 3 secciones:

- Una sección izquierda o de escritura de datos, compuesta por trenes de partículas 4_A^{4+}.
- Una sección central o de codificación de cinta, compuesta por uno de los elementos $Ele\_C_2{}^0$ y una partícula activadora A^{3+}.
- Una sección derecha o de procesamiento, compuesta por partículas líderes $SepInit\_E\bar{E}^-$ y bloques de datos $Blo\_\bar{E}^-$.

El sistema final implementado en este trabajo tendrá algunas diferencias con respecto al CTS original.

Modificación de espaciados izquierdos

La sección izquierda está compuesta por trenes de partículas 4_A^{4+} separados por bloques de éter. Cada tren se encarga de escribir un nuevo dato en la cinta mediante la colisión con una partícula de adición $Add\_\bar{E}^-$. A su vez, cada partícula de adición es obtenida a partir del colapso de un bloque $Blo\_\bar{E}^-$, por lo que, en el peor caso, es necesario tener tantos trenes como bloques haya en el sistema.

Es fácil notar que el tamaño del sistema crece enormemente con cada nuevo dato. Concretamente, un tren 4_A^{4+} tiene un tamaño de 1,132 células, y cada tren tiene un espaciado de 649 éters, o 9,086 células, por lo que cada nuevo dato que se desee procesar agregaría 10,218 células solamente a la sección izquierda, que en su mayoría son sólo espacio vacío. Sin embargo, este espaciado es necesario para evitar la corrupción del sistema. Reducir el espacio entre trenes abre la posibilidad de una colisión temprana entre un tren 4_A^{4+} con un elemento $Ele\_C_2{}^0$, antes de que la partícula de adición asociada al tren pueda actuar.

Para fines demostrativos en este trabajo, se reducirá deliberadamente el espaciado entre trenes a tan solo 87 éters, con 86 éters para el primer tren, lo cual permitirá el procesamiento adecuado de 5 datos a la cinta del CTS con un ahorro de 2,946 éters, o 41,244 células. A pesar del cambio en distancias, no es necesario modificar la fase de ninguna partícula del sistema.

Además, es importante notar que es posible cambiar la distancia entre trenes en un múltiplo de 86 éters sin alterar el comportamiento del sistema

Traslación de la partícula de activación inicial

La partícula de activación inicial del sistema, el conjunto A^{3+} de la sección central, es la única con una velocidad horizontal distinta a las demás partículas de su sección. Sin embargo, al ser una partícula simple, es fácil ajustar su distancia y fase para interactuar con la sección derecha. Inclusive, mientras se conserven distancias múltiplos de 4 éters entre la partícula y el elemento central $Ele\_C_2{}^0$ y entre la partícula y el primer líder derecho $SepInit\_E\bar{E}^-$, el sistema entero puede conservar su fase.

Si mantenemos la partícula activadora en la sección central, ésta se mantendrá en movimiento constante y sólo es posible estabilizar su fase con respecto al elemento $Ele\_C_2{}^0$ cada 4 éters. En cambio, si consideramos a la partícula como un miembro de la sección derecha, ésta colisionará con el primer líder $SepInit\_E\bar{E}^-$, activando el bloque. Después de esto, la sección derecha se mantendrá estable en fase. Por lo tanto, para este trabajo, consideraremos a la partícula activadora como un miembro de la sección derecha.

Distanciamiento entre secciones

Como resultado de las modificaciones anteriores, es posible aprovechar la conservación de fases para procesar rápidamente las distancias entre secciones.

- La sección izquierda es invariable en fase cuando la distancia entre trenes 4_A^{4+} cambia en un múltiplo de $86e$. Por lo que las secciones izquierda y central pueden estar separadas por este mismo factor sin afectar la fase del sistema.

- La partícula activadora $A^{3\,+}$ es invariable en fase con respecto al elemento central $Ele\_C_2\ ^0$ cada múltiplo de $4e$. Por lo que las secciones derecha y central pueden estar separadas por este mismo factor sin afectar la fase del sistema.

Por supuesto, es necesario ajustar ambas distancias simultáneamente para evitar colisiones tempranas en el CTS.

9.2.2. Espaciado en la sección central

Para el sistema final, las secciones derecha e izquierda estarán ajustadas, por lo que los espacios entre secciones se llevarán a cabo en la sección central. Para este trabajo, la primera lectura de la cinta sucederá cuando la sección derecha entera haya sido transportada a la sección central en el colisionador. Ya que la sección derecha tiene un tamaño de 5,864 células, la distancia entre el elemento central y la sección derecha debe ser de al menos 5,864. Para poder mantener la fase del sistema, la distancia derecha se aproximará a un múltiplo de $4e$, por lo que se escogió una distancia de $420e$, o 5,880 células.

A partir de la velocidad $-4/15$ de las partículas $\bar{E}^-$, podemos concluir que la sección derecha tardará 22,050 generaciones en trasladarse a la sección central.

Para calcular el espaciado izquierdo, nos basamos en la velocidad $2/3$ de los trenes $4\_A^{4\,+}$. Después de 22,050 generaciones, los trenes habrán recorrido 14,700 células, o $1050e$, por lo que el espaciado izquierdo será de $1050e$.

9.3. Implementación del CTS

9.3.1. Descripción del sistema

Después de aplicar las modificaciones mencionadas, el sistema del CTS a ejecutar en este trabajo es el siguiente:

$$
\begin{aligned}
Izquierda: \quad & 4\_A^4(F_2) - 87e - 4\_A^4(F_1) - 87e - 4\_A^4(F_3) - 87e - 4\_A^4(F_2) - \\
& 87e - 4\_A^4(F_1) - 87e - 4\_A^4(F_3) - 86e \\
Centro: \quad & 1Ele\_C_2(A, f_1\_1) \\
Derecha: \quad & A^3(f_1\_1) - \\
& SepInit\_E\bar{E}(C, f_3\_1) - 1BloP\_\bar{E}(C, f_4\_1) - SepInit\_E\bar{E}(C, f_3\_1) - \\
& 1BloP\_\bar{E}(C, f_4\_1) - 0Blo\_\bar{E}(C, f_4\_1) - 1BloS\_\bar{E}(A, f_4\_1) - \\
& SepInit\_E\bar{E}(A, f_2\_1) - 1BloP\_\bar{E}(F, f_1\_1) - SepInit\_E\bar{E}(A, f_3\_1) - \\
& 1BloP\_\bar{E}(F, f_1\_1) - 0Blo\_\bar{E}(E, f_4\_1) - 1BloS\_\bar{E}(C, f_4\_1) - \\
& SepInit\_E\bar{E}(B, f_1\_1) - 1BloP\_\bar{E}(F, f_3\_1) - \\
& SepInit\_E\bar{E}(B, f_1\_1)
\end{aligned}
$$

Las figuras 9.13, 9.14 y 9.15 muestran el espacio de evoluciones de las secciones izquierda, central y derecha, respectivamente. Para visibilidad de la sección central, se omiten los espaciados izquierdos y derechos.

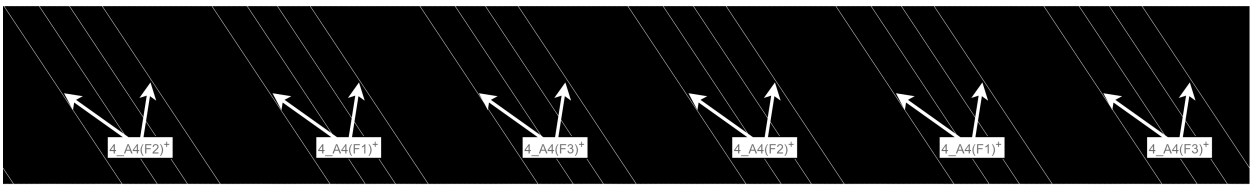

Figura 9.13: Espacio de evoluciones de la sección izquierda del CTS.

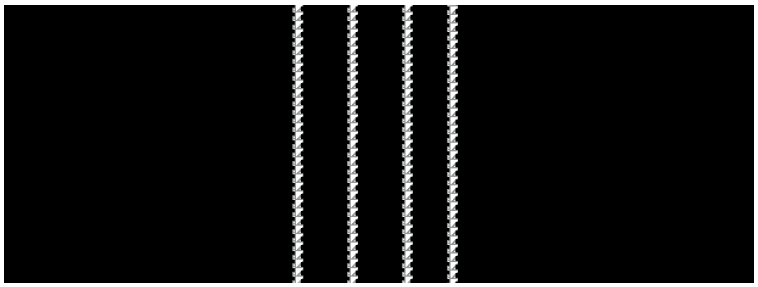

Figura 9.14: Espacio de evoluciones de la sección central del CTS.

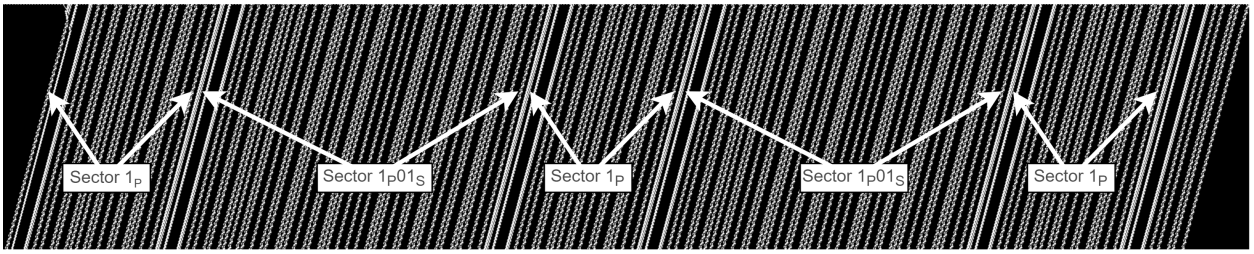

Figura 9.15: Espacio de evoluciones de la sección derecha del CTS.

Las figuras 9.16, 9.17 y 9.18 muestran la implementación en ciclotrones de las secciones izquierda, central y derecha, respectivamente. De igual manera, se omiten los espaciados finales en la sección central.

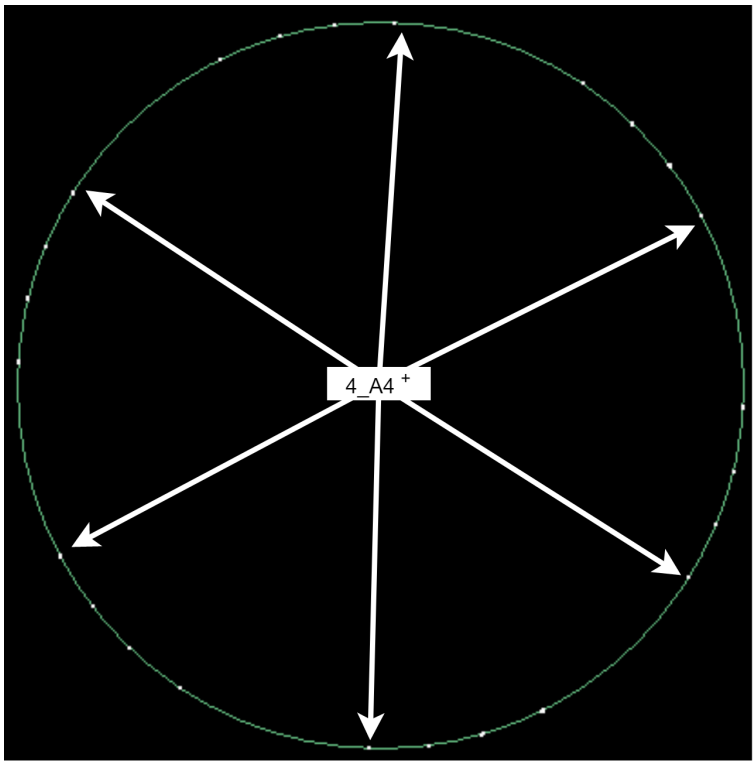

Figura 9.16: Ciclotrón de la sección izquierda del CTS.

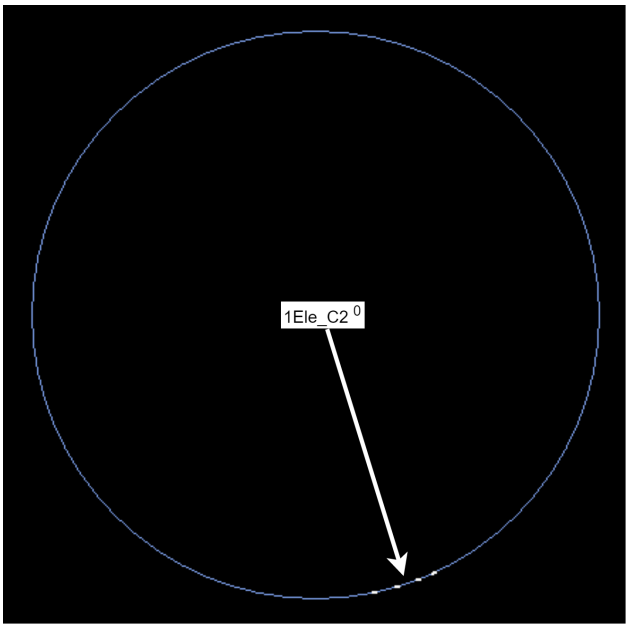

Figura 9.17: Ciclotrón de evoluciones de la sección central del CTS.

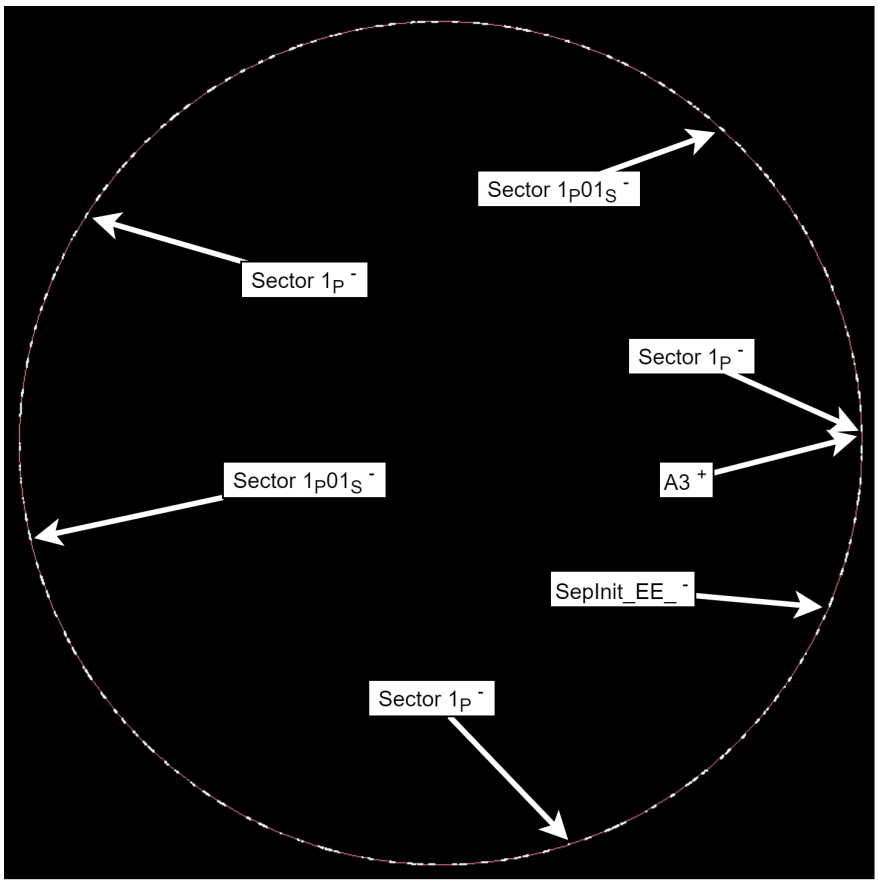

Figura 9.18: Ciclotrón de evoluciones de la sección derecha del CTS.

9.3.2. Implementación en el colisionador

Para la ejecución del colisionador, se agregarán los espaciados en el ciclotron central mencionados anteriormente. Además, para simetría en el colisionador, se añadirán $1000e$ al principio del espaciado izquierdo. Así pues, el anillo central utilizará la condición inicial:

$$1000e - 1050e - 1Ele\_C_2(A, f_1\_1) - 420e$$

Los anillos izquierdo y derecho utilizarán como condición inicial la expresión respectiva del sistema.

Los puntos de contacto para cada anillo lateral se mantienen en 0. Para el anillo central, el punto de contacto derecho se mantiene en 0, mientras que el punto izquierdo se posicionará después del espaciado de simetría de $1000e$, es decir, en la celda 14,000.

Cada anillo procesa una cantidad específica de células: la sección izquierda utiliza 14,058 células; la sección derecha, 5,896; y la sección central, 34,718. Así pues, el espacio de evoluciones del simulador de CTS contiene un total de 54,672 células.

El colisionador resultante se muestra en la figura 9.19.

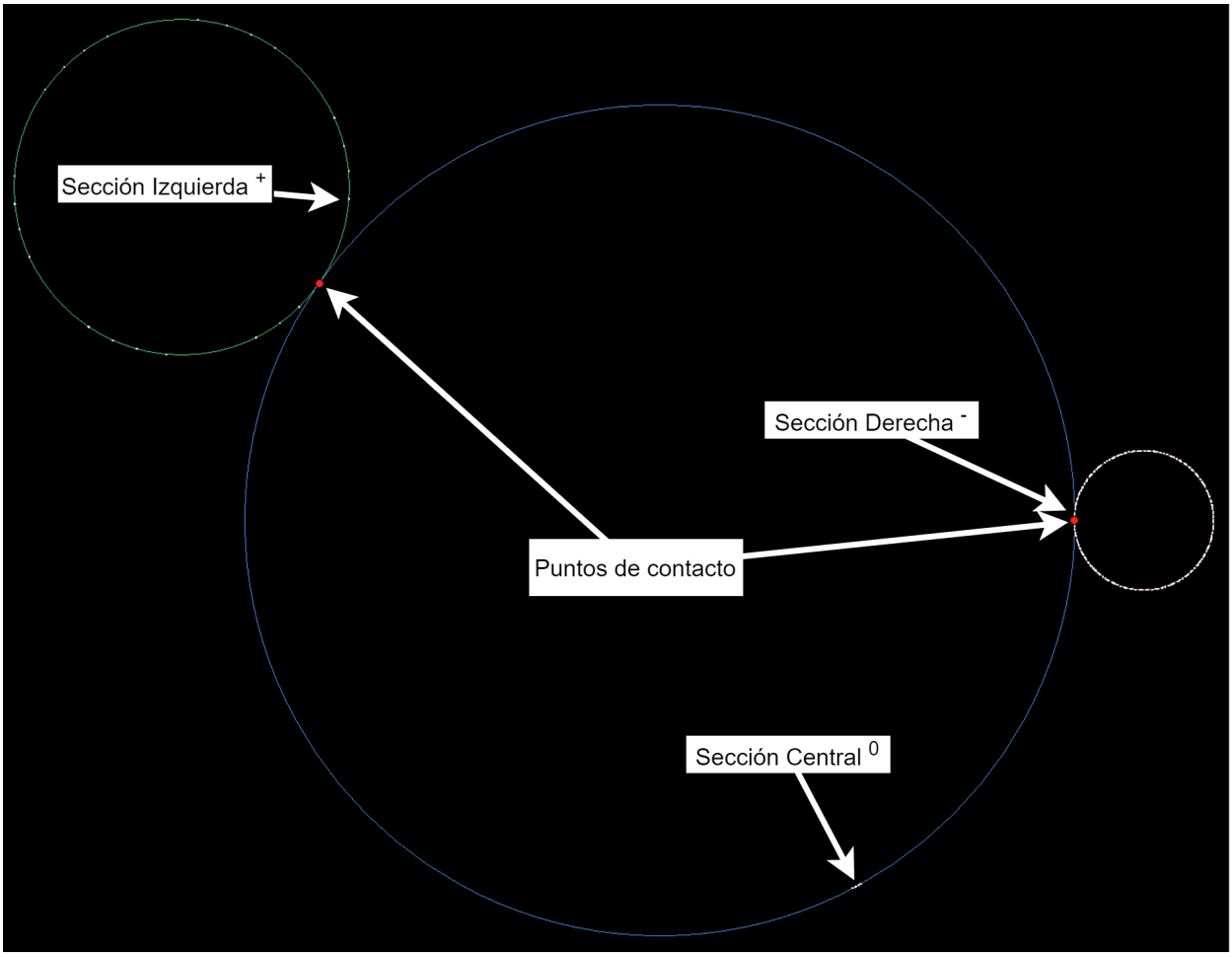

Figura 9.19: Estado inicial del colisionador virtual de simulación de CTS.

Después de 22,000 generaciones, el anillo derecho transportó todas sus partículas al anillo central, y el sistema está preparado para la primera lectura del elemento central. La figura 9.20 muestra el estado del colisionador en esta generación.

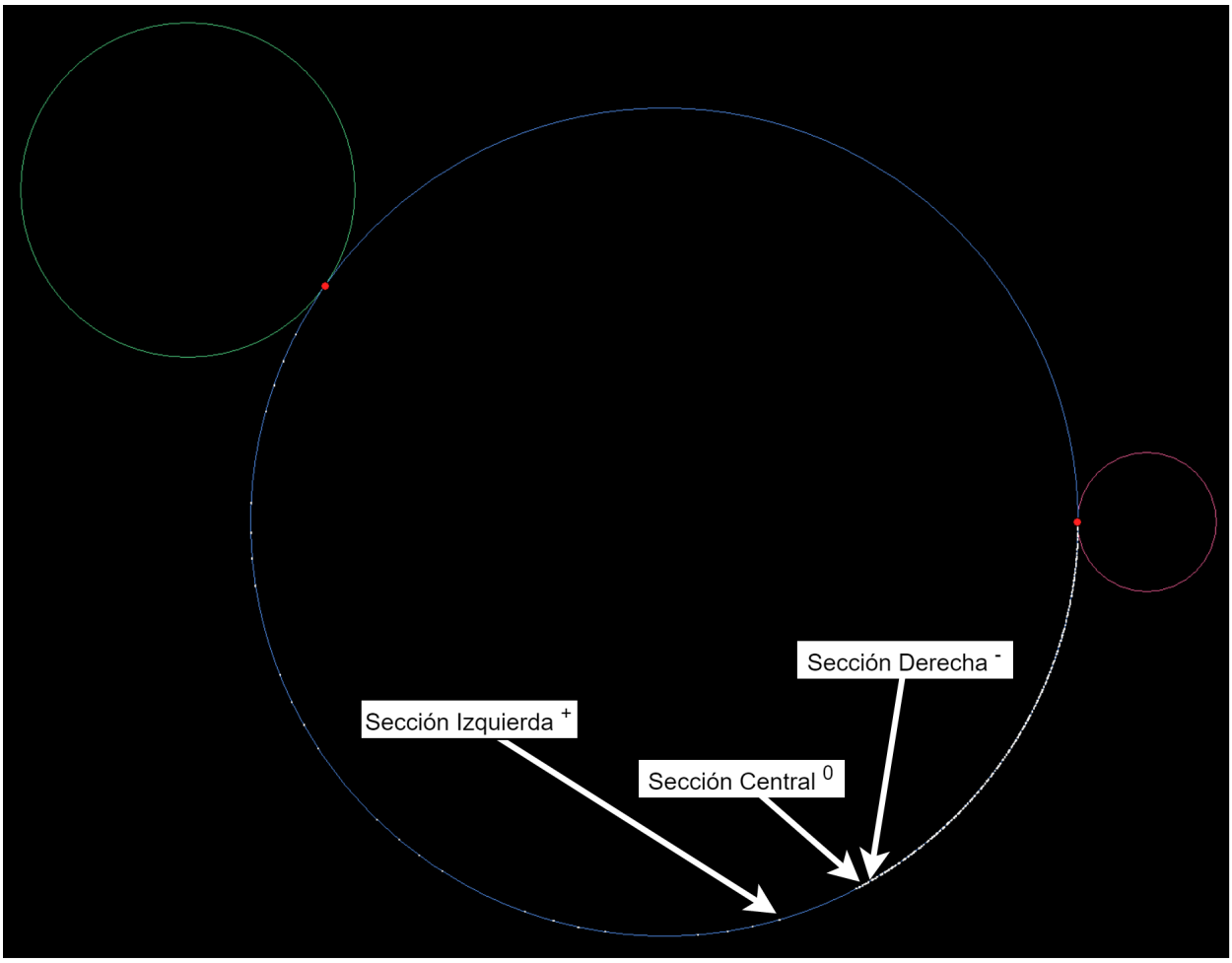

Figura 9.20: Colisionador virtual de simulación de CTS en su generación 22,050.

La figura 9.21 muestra un acercamiento del impacto inminente entre el elemento central y la sección derecha, comenzando la primera lectura de la cinta del CTS.

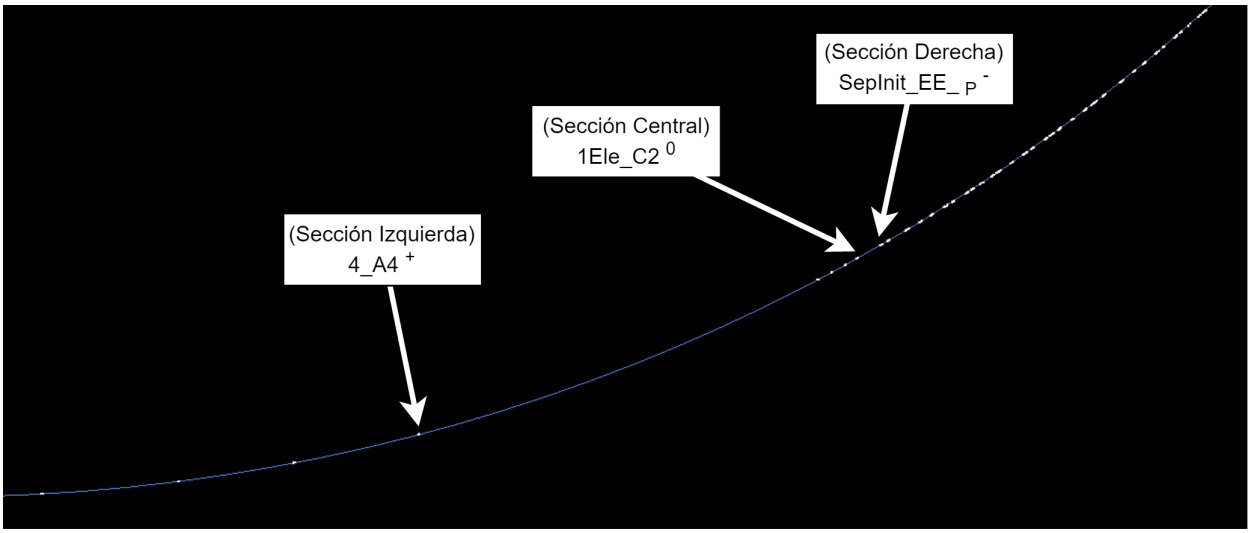

Figura 9.21: Lectura del primer dato en el colisionador virtual de simulación de CTS.

El espacio de evoluciones total de la ejecución del sistema genera las transiciones $1 \Rightarrow 1 \Rightarrow 101 \Rightarrow 011 \Rightarrow 11$ del CTS $(1, 101)$ mediante la escritura de la palabra 111011. Este espacio se muestra en la figura 9.22, donde se presentan las generaciones 22,050 a 40,050.

El sistema completo procesa una cinta de 54,672 células durante 40,050 iteraciones, por lo que el espacio final tiene un tamaño de 2,189,613,600 células.

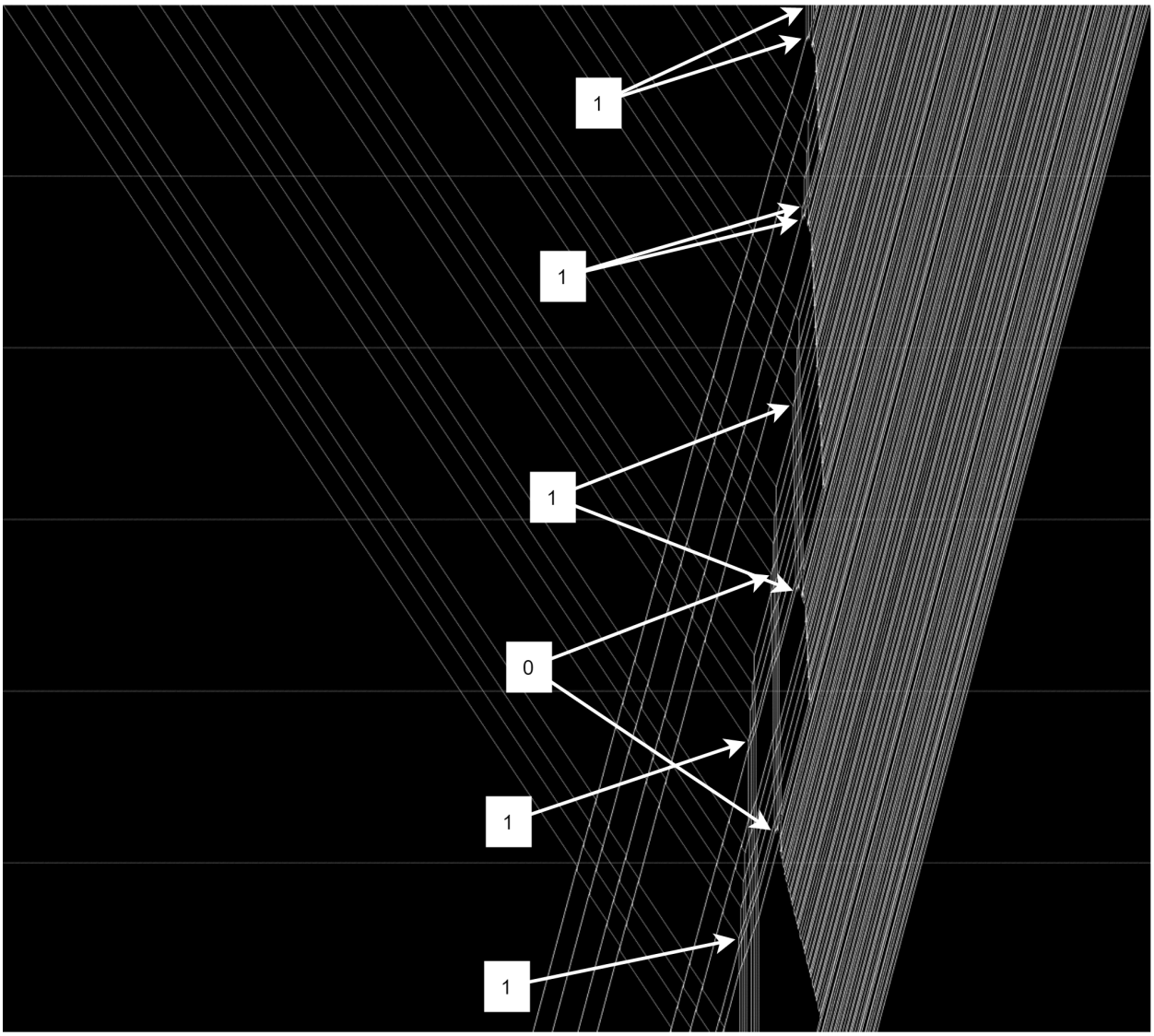

Figura 9.22: Escritura de la palabra 111011 en el espacio de evoluciones 22,050-40,050 del simulador de CTS.

Capítulo 10

Conclusiones

10.1. Resultados del trabajo

Llegado este punto, podemos denotar las metas alcanzadas mediante el desarrollo de este trabajo:

- Se desarrolló un simulador de ACEs versátil e intuitivo para usuarios con conocimiento de estas máquinas. El programa permite analizar diferentes tipos de ACEs y sus interacciones en un espacio de evoluciones, así como observar su comportamiento en un ciclotrón mediante una visualización de anillo.
- Se añadió un traductor especializado de partículas y conjuntos de partículas propios de la regla elemental 110, una herramienta que agiliza enormemente el proceso de configuración de un ACE en el simulador.
- Se creó un colisionador virtual de partículas de ACE versátil e intuitivo para usuarios con conocimiento del comportamiento de la regla elemental 110. Este programa es el primero en su clase en implementar interacciones balísticas para las partículas de la regla 110. En combinación con el simulador de ACEs, se fundó la base para un análisis a profundidad de las propiedades de la regla.
- Se implementaron diversas estrategias de optimización para el rendimiento de los programas desarrollados. De igual manera, se realizó un análisis comparativo entre algoritmos de filtrado para mosaicos de la regla 110 para elegir el procedimiento más eficiente. Después de estas actualizaciones, la mejora en tiempo y memoria durante la ejecución de ambos programas fue evidente.
- Como producto del estudio de las partículas de la regla 110, fue posible codificar el sistema de CTS $(1, 101)$ para implementarlo tanto en el colisionador como en el simulador de ACE. De esta manera, se pudo observar a la perfección el funcionamiento de este sistema para comprobar la universalidad de la regla.

10.2. Trabajo a futuro

Los resultados de este trabajo abren múltiples posibilidades para nuevos proyectos, así como áreas de oportunidad para mejorar el sistema actual:

- A pesar de que la biblioteca de desarrollo de GUIs wxWidgets cumplió su propósito durante la elaboración de los programas de simulación, muchas veces fue necesario enfocar tiempo a solucionar errores internos de la biblioteca, un problema que resalta aún más por la desactualización en los foros de la comunidad. Esto, sumado a graves cuellos de botella que sufre la aplicación durante tareas relativamente simples, motivan a buscar una alternativa para las interfaces de este trabajo.
- Los programas son limitados en opciones de visualización y personalización. La accesibilidad es un asunto imprescindible en el desarrollo de nuevas aplicaciones, y los programas de este trabajo podrían enfocarse en mejorar en este aspecto.

- Después de confirmar la viabilidad de implementación con el sistema de simulación del CTS $(1, 101)$, el siguiente paso es implementar el CTS original $(111, 0)$ propuesto por Matthew Cook y probar su ejecución.
- Expandiendo en el punto anterior, un futuro trabajo podría buscar implementar un CTS diferente a los dos ya existentes, e incluso estudiar la posibilidad de crear un traductor de la regla para un CTS arbitrario.
- Aunque el objetivo de este trabajo fue el estudio de la regla elemental 110, es posible modificar el colisionador para procesar partículas en cualquier regla elemental. De esta manera, se abre la oportunidad de analizar el comportamiento de distintas reglas para implementar nuevos sistemas computables.

Apéndice A

Código fuente del controlador de ACEs

EcaControllerCore.h

```
1  #pragma once
2
3  #include <string>
4  #include <time.h>
5  #include "EcaControllerDLL.h"
6
7  using namespace std;
8
9  namespace EcaControllerCore {
10         typedef unsigned short CHUNK;
11         ECACONTROLLER_DLL extern const int CHUNK_BITSIZE;
12
13         ECACONTROLLER_DLL string ToBinaryString(int n);
14         ECACONTROLLER_DLL bool IsBinaryString(string& s);
15         ECACONTROLLER_DLL string FormatBinaryString(int k, int n);
16         ECACONTROLLER_DLL string FormatBinaryString(string s, int n);
17         ECACONTROLLER_DLL string CreateRandomBinaryString(int n);
18 }
```

EcaControllerCore.cpp

```
1  #include "pch.h"
2  #include "EcaControllerCore.h"
3
4  const int EcaControllerCore::CHUNK_BITSIZE = 16;
5
6  string EcaControllerCore::ToBinaryString(int n) {
7          if (n < 0) {
8                  throw exception("N cannot be negative");
9          }
10         string r = "";
11         do {
12                 r = (n % 2 == 0 ? '0' : '1') + r;
13                 n /= 2;
14         } while (n != 0);
15
16         return r;
```

```
17  }
18
19  bool EcaControllerCore::IsBinaryString(string& s) {
20          if (s.empty()) {
21                  return false;
22          }
23
24          for (char& c : s) {
25                  if (c != '0' && c != '1') {
26                          return false;
27                  }
28          }
29
30          return true;
31  }
32
33  string EcaControllerCore::FormatBinaryString(int k, int n) {
34          return EcaControllerCore::FormatBinaryString(EcaControllerCore::ToBinaryString(k), n);
35  }
36
37  string EcaControllerCore::FormatBinaryString(string s, int n) {
38          if (n < 0) {
39                  throw exception("N cannot be negative");
40          }
41          if (!EcaControllerCore::IsBinaryString(s) && !s.empty()) {
42                  string ex = "String is not binary: " + s;
43                  throw exception(ex.c_str());
44          }
45          if (n < s.length()) {
46                  string ex = "Desired length (" + to_string(n); +
47                          ") cannot be less than string length (" +
48                          to_string(s.length()) + ")";
49                  throw exception(ex.c_str());
50          }
51
52          int i;
53          for (i = s.length(); i < n; i++) {
54                  s = '0' + s;
55          }
56
57          return s;
58  }
59
60  string EcaControllerCore::CreateRandomBinaryString(int n) {
61          if (n < 0) {
62                  throw exception("N cannot be negative");
63          }
64
65          srand(time(NULL));
66
67          string s = "";
68          int i;
69          for (i = 0; i < n; i++) {
70                  s += rand() % 2 ? '0' : '1';
```

```
71          }
72
73          return s;
74  }
```

EcaController.h

```
1   #pragma once
2
3   #include <vector>
4   #include "EcaControllerCore.h"
5
6   using namespace std;
7   using namespace EcaControllerCore;
8
9   class ECACONTROLLER_DLL EcaController {
10  protected:
11          int N = 100;
12          size_t ruleNumber = 110;
13          string rule = "";
14          string initialCondition = "";
15
16          EcaController(int _N, int _rule);
17          EcaController(int _N, int _rule, string _initialCondition);
18
19  public:
20          virtual ~EcaController();
21
22          virtual int getN();
23          virtual size_t getRuleNumber();
24          virtual string getRule();
25
26          virtual void reset() = 0;
27          virtual void reset(string _newInitialCondition) = 0;
28          virtual void restart() = 0;
29
30          virtual void saveState() = 0;
31          virtual void applySavedState(bool _applySavedCount = true) = 0;
32          virtual void resetWithSavedState() = 0;
33
34          virtual void applyRule() = 0;
35          virtual long getCurrentIteration() = 0;
36
37          const virtual vector<string>& getCurrentState() = 0;
38
39          virtual char setAtPosition(int position, char newValue) = 0;
40          virtual char getAtPosition(int position) = 0;
41  };
```

EcaController.cpp

```
1   #include "pch.h"
```

```
2  #include "EcaController.h"
3
4  EcaController::EcaController(int _N, int _rule)
5          : EcaController(_N, _rule, CreateRandomBinaryString(_N)) {
6  }
7
8  EcaController::EcaController(int _N, int _rule, string _initialCondition) {
9          if (_rule < 0 || _rule > 255) {
10                 throw exception("Rule must be between 0 and 255");
11         }
12         if (_N < 3) {
13                 throw exception("N cannot be less than 3");
14         }
15         if (_N < _initialCondition.length()) {
16                 throw exception("N cannot be less than the length of the initial condition");
17         }
18         if (!IsBinaryString(_initialCondition)) {
19                 throw exception("Initial condition is not a binary string");
20         }
21
22         N = _N;
23         ruleNumber = _rule;
24         rule = FormatBinaryString(_rule, 8);
25
26         initialCondition = _initialCondition;
27 }
28
29 int EcaController::getN() {
30         return N;
31 }
32
33 size_t EcaController::getRuleNumber() {
34         return ruleNumber;
35 }
36
37 string EcaController::getRule() {
38         return rule;
39 }
40
41 EcaController::~EcaController() {
42 }
```

NaiveController.h

```
1  #pragma once
2
3  #include "EcaController.h"
4  #include "EcaControllerCore.h"
5
6  class ECACONTROLLER_DLL NaiveController : public EcaController {
7  protected:
8          string currentState = "";
9          string auxState;
```

```
10
11          long currentIteration;
12
13          string savedState;
14          long savedIteration;
15
16          vector<string> stateVector;
17
18  public:
19          NaiveController(int _N, int _rule);
20          NaiveController(int _N, int _rule, string _initialCondition);
21          ~NaiveController();
22
23          void reset();
24          void reset(string _newInitialCondition);
25          void restart();
26
27          void saveState();
28          void applySavedState(bool applySavedCount = true);
29          void resetWithSavedState();
30
31          void applyRule();
32          size_t getChunkArrayLength();
33          long getCurrentIteration();
34          void setCurrentIteration(long newIt);
35
36          const vector<string>& getCurrentState();
37          virtual void getNeighbors(int i, char& prev, char& cur, char& next);
38
39          char setAtPosition(int position, char newValue);
40          char getAtPosition(int position);
41  };
```

NaiveController.cpp

```
1   #include "pch.h"
2   #include "NaiveController.h"
3
4   NaiveController::NaiveController(int _N, int _rule)
5           : NaiveController(_N, _rule, CreateRandomBinaryString(_N)) {
6   }
7
8   NaiveController::NaiveController(int _N, int _rule, string _initialCondition)
9           : EcaController(_N, _rule, _initialCondition) {
10          initialCondition = FormatBinaryString(_initialCondition, N);
11
12          currentState = initialCondition;
13          auxState = currentState;
14
15          currentIteration = 0;
16          savedIteration = -1;
17  }
18
```

```
19 NaiveController::~NaiveController() {
20 }
21
22 void NaiveController::reset() {
23         reset(CreateRandomBinaryString(N));
24 }
25
26 void NaiveController::reset(string _newInitialCondition) {
27         if (_newInitialCondition.empty()) {
28                 throw exception("Initial_condition_cannot_be_empty");
29         }
30         if (N < _newInitialCondition.length()) {
31                 string ex = "Initial_condition_cannot_be_longer_than_current_N:_" + to_string(N);
32                 throw exception(ex.c_str());
33         }
34         if (!IsBinaryString(_newInitialCondition)) {
35                 throw exception("Initial_condition_is_not_a_binary_string");
36         }
37
38         initialCondition = FormatBinaryString(_newInitialCondition, N);
39
40         currentState = initialCondition;
41         auxState = currentState;
42
43         currentIteration = 0;
44 }
45
46 void NaiveController::restart() {
47         currentState = initialCondition;
48         auxState = currentState;
49
50         currentIteration = 0;
51 }
52
53 void NaiveController::saveState() {
54         savedState = currentState;
55         savedIteration = currentIteration;
56 }
57
58 void NaiveController::applySavedState(bool applySavedCount) {
59         if (savedIteration == -1) {
60                 throw exception("Saved_state_not_found");
61         }
62
63         currentState = savedState;
64
65         if (applySavedCount) {
66                 currentIteration = savedIteration;
67         }
68         else {
69                 currentIteration++;
70         }
71 }
72
```

```
73  void NaiveController::resetWithSavedState() {
74          if (savedIteration == -1) {
75                  throw exception("Saved_state_not_found");
76          }
77
78          initialCondition = savedState;
79
80          currentState = initialCondition;
81          auxState = currentState;
82
83          currentIteration = 0;
84  }
85
86  long NaiveController::getCurrentIteration() {
87          return currentIteration;
88  }
89
90  void NaiveController::setCurrentIteration(long newIt) {
91          currentIteration = newIt;
92  }
93
94  const vector<string>& NaiveController::getCurrentState() {
95          stateVector = { currentState };
96          return stateVector;
97  }
98
99  void NaiveController::getNeighbors(int i, char& prev, char& cur, char& next) {
100         if (i == 0) {
101                 prev = currentState.at(N - 1);
102         }
103         else {
104                 prev = currentState.at(i - 1);
105         }
106         cur = currentState.at(i);
107         if (i == N - 1) {
108                 next = currentState.at(0);
109         }
110         else {
111                 next = currentState.at(i + 1);
112         }
113 }
114
115 void NaiveController::applyRule() {
116         char prev, cur, next;
117         for (int i = 0; i < N; i++) {
118                 getNeighbors(i, prev, cur, next);
119
120                 if (prev == '1') {
121                         if (cur == '1') {
122                                 if (next == '1') { // 111
123                                         auxState.at(i) = rule.at(0);
124                                 }
125                                 else { // 110
126                                         auxState.at(i) = rule.at(1);
```

```
127                                     }
128                             }
129                             else {
130                                     if (next == '1') { // 101
131                                             auxState.at(i) = rule.at(2);
132                                     }
133                                     else { // 100
134                                             auxState.at(i) = rule.at(3);
135                                     }
136
137                             }
138                     }
139                     else {
140                             if (cur == '1') {
141                                     if (next == '1') { // 011
142                                             auxState.at(i) = rule.at(4);
143                                     }
144                                     else { // 010
145                                             auxState.at(i) = rule.at(5);
146                                     }
147                             }
148                             else {
149                                     if (next == '1') { // 001
150                                             auxState.at(i) = rule.at(6);
151                                     }
152                                     else { // 000
153                                             auxState.at(i) = rule.at(7);
154                                     }
155
156                             }
157                     }
158             }
159
160             currentState = auxState;
161             currentIteration++;
162     }
163
164     char NaiveController::setAtPosition(int position, char newValue) {
165             char oldValue = getAtPosition(position);
166
167             currentState[position] = newValue;
168
169             return oldValue;
170     }
171
172     char NaiveController::getAtPosition(int position) {
173             return currentState[position];
174     }
```

CompressedController.h

```
1   #pragma once
2
```

```
3  #include "CompressedState.h"
4  #include "EcaController.h"
5  #include "EcaControllerCore.h"
6
7  using namespace std;
8  using namespace EcaControllerCore;
9
10 class ECACONTROLLER_DLL CompressedController : public EcaController {
11 private:
12         CompressedState* compressedState;
13         ChunkLUTAgent* lutAgent;
14
15 public:
16         CompressedController(int _N, int _rule);
17         CompressedController(int _N, int _rule, string _initialCondition);
18         ~CompressedController();
19
20         void reset();
21         void reset(string _newInitialCondition);
22         void restart();
23
24         void saveState();
25         void applySavedState(bool applySavedCount = true);
26         void resetWithSavedState();
27
28         void applyRule();
29         size_t getChunkArrayLength();
30         long getCurrentIteration();
31
32         const vector<string>& getCurrentState();
33
34         char setAtPosition(int position, char newValue);
35         char getAtPosition(int position);
36 };
```

CompressedController.cpp

```
1  #include "pch.h"
2  #include "CompressedController.h"
3
4  CompressedController::CompressedController(int _N, int _rule)
5          : CompressedController(_N, _rule, CreateRandomBinaryString(_N)) {
6  }
7
8  CompressedController::CompressedController(int _N, int _rule, string _initialCondition)
9          : EcaController(_N, _rule, _initialCondition){
10         compressedState = new CompressedState(N, initialCondition);
11         lutAgent = new ChunkLUTAgent(rule, N);
12 }
13
14 CompressedController::~CompressedController() {
15         delete compressedState;
16         delete lutAgent;
```

```
17 }
18
19
20 void CompressedController::reset() {
21         compressedState->reset();
22 }
23
24 void CompressedController::reset(string _newInitialCondition) {
25         compressedState->reset(_newInitialCondition);
26 }
27
28 void CompressedController::restart() {
29         compressedState->restart();
30 }
31
32 void CompressedController::saveState() {
33         compressedState->saveState();
34 }
35
36 void CompressedController::applySavedState(bool applySavedCount) {
37         compressedState->applySavedState(applySavedCount);
38 }
39
40 void CompressedController::resetWithSavedState() {
41         compressedState->resetWithSavedState();
42 }
43
44 void CompressedController::applyRule() {
45         compressedState->applyRule(*lutAgent);
46 }
47
48 size_t CompressedController::getChunkArrayLength() {
49         return compressedState->getChunkArrayLength();
50 }
51
52 long CompressedController::getCurrentIteration() {
53         return compressedState->getCurrentIteration();
54 }
55
56 const vector<string>& CompressedController::getCurrentState() {
57         return compressedState->getRawState();
58 }
59
60 char CompressedController::setAtPosition(int position, char newValue) {
61         return compressedState->setAtPosition(position, newValue);
62 }
63
64 char CompressedController::getAtPosition(int position) {
65         return compressedState->getAtPosition(position);
66 }
```

CompressedController.h

```
1  #pragma once
2
3  #include "CompressedState.h"
4  #include "EcaController.h"
5  #include "EcaControllerCore.h"
6
7  using namespace std;
8  using namespace EcaControllerCore;
9
10 class ECACONTROLLER_DLL CompressedController : public EcaController {
11 private:
12         CompressedState* compressedState;
13         ChunkLUTAgent* lutAgent;
14
15 public:
16         CompressedController(int _N, int _rule);
17         CompressedController(int _N, int _rule, string _initialCondition);
18         ~CompressedController();
19
20         void reset();
21         void reset(string _newInitialCondition);
22         void restart();
23
24         void saveState();
25         void applySavedState(bool applySavedCount = true);
26         void resetWithSavedState();
27
28         void applyRule();
29         size_t getChunkArrayLength();
30         long getCurrentIteration();
31
32         const vector<string>& getCurrentState();
33
34         char setAtPosition(int position, char newValue);
35         char getAtPosition(int position);
36 };
```

CompressedController.cpp

```
1  #include "pch.h"
2  #include "CompressedController.h"
3
4  CompressedController::CompressedController(int _N, int _rule)
5          : CompressedController(_N, _rule, CreateRandomBinaryString(_N)) {
6  }
7
8  CompressedController::CompressedController(int _N, int _rule, string _initialCondition)
9          : EcaController(_N, _rule, _initialCondition){
10         compressedState = new CompressedState(N, initialCondition);
11         lutAgent = new ChunkLUTAgent(rule, N);
12 }
13
```

```
14 CompressedController::~CompressedController() {
15         delete compressedState;
16         delete lutAgent;
17 }
18
19
20 void CompressedController::reset() {
21         compressedState->reset();
22 }
23
24 void CompressedController::reset(string _newInitialCondition) {
25         compressedState->reset(_newInitialCondition);
26 }
27
28 void CompressedController::restart() {
29         compressedState->restart();
30 }
31
32 void CompressedController::saveState() {
33         compressedState->saveState();
34 }
35
36 void CompressedController::applySavedState(bool applySavedCount) {
37         compressedState->applySavedState(applySavedCount);
38 }
39
40 void CompressedController::resetWithSavedState() {
41         compressedState->resetWithSavedState();
42 }
43
44 void CompressedController::applyRule() {
45         compressedState->applyRule(*lutAgent);
46 }
47
48 size_t CompressedController::getChunkArrayLength() {
49         return compressedState->getChunkArrayLength();
50 }
51
52 long CompressedController::getCurrentIteration() {
53         return compressedState->getCurrentIteration();
54 }
55
56 const vector<string>& CompressedController::getCurrentState() {
57         return compressedState->getRawState();
58 }
59
60 char CompressedController::setAtPosition(int position, char newValue) {
61         return compressedState->setAtPosition(position, newValue);
62 }
63
64 char CompressedController::getAtPosition(int position) {
65         return compressedState->getAtPosition(position);
66 }
```

CompressedState.h

```
1  #pragma once
2
3  #include <vector>
4
5  #include "EcaControllerCore.h"
6  #include "ChunkTranslator.h"
7  #include "ChunkLUTAgent.h"
8
9  using namespace std;
10 using namespace EcaControllerCore;
11
12 class ECACONTROLLER_DLL CompressedState {
13 private:
14
15         int N;
16         CHUNK* initialStateChunkArray;
17         CHUNK* currentStateChunkArray;
18         CHUNK* savedStateChunkArray;
19         size_t chunkArrayLength;
20         int mostSignificativeChunkBitSize;
21         size_t mostSignificativeChunkPosition;
22         long currentIteration;
23         long savedIteration;
24         size_t savedMSChunkPosition;
25
26         vector<string> rawState;
27
28 private:
29         void start(string& _initialCondition);
30         void compress(CHUNK** pDestArray, string& _state);
31
32         void createChunkArray(CHUNK** pChunkArray);
33         void copyChunkArray(CHUNK** pSourceArray, CHUNK** pDestArray);
34
35         int getMSBAt(int i);
36
37 public:
38         CompressedState(string _initialCondition);
39         CompressedState(int _N, string _initialCondition);
40
41         void reset();
42         void reset(string _newInitialCondition);
43         void restart();
44
45         void saveState();
46         void applySavedState(bool applySavedCount = true);
47         void resetWithSavedState();
48
49         void applyRule(ChunkLUTAgent& lutAgent);
50
51         bool hasAdjustedMostSignificativeChunk();
52         size_t getMostSignificativeChunkPosition();
```

```
53          bool isMostSignificativeChunkPosition(int i);
54          int getMostSignificativeChunkBitSize();
55          size_t getChunkArrayLength();
56          long getCurrentIteration();
57          CHUNK at(int i);
58
59          const vector<string>& getRawState();
60
61          char setAtPosition(int position, char newValue);
62          char getAtPosition(int position);
63  };
```

CompressedState.cpp

```
1   #include "pch.h"
2   #include "CompressedState.h"
3
4   CompressedState::CompressedState(string _initialCondition)
5           : CompressedState(_initialCondition.length(), _initialCondition) { }
6
7   CompressedState::CompressedState(int _N, string _initialCondition) {
8           if (_N < 3) {
9                   throw exception("N cannot be less than 3");
10          }
11          if (_N < _initialCondition.length()) {
12                  throw exception("N cannot be less than the length of the initial condition");
13          }
14          if (!IsBinaryString(_initialCondition)) {
15                  throw exception("Initial condition is not a binary string");
16          }
17
18          N = _N;
19
20          mostSignificativeChunkBitSize = N % CHUNK_BITSIZE;
21          mostSignificativeChunkBitSize = (mostSignificativeChunkBitSize == 0 ? CHUNK_BITSIZE :
                ↪ mostSignificativeChunkBitSize);
22
23          chunkArrayLength = N / CHUNK_BITSIZE;
24          if (hasAdjustedMostSignificativeChunk()) {
25                  chunkArrayLength++;
26          }
27
28          savedIteration = -1;
29
30          rawState = vector<string>(chunkArrayLength, "");
31          ChunkTranslator::GetInstance();
32
33          createChunkArray(&initialStateChunkArray);
34          createChunkArray(&currentStateChunkArray);
35
36          start(_initialCondition);
37  }
38
```

```
39 void CompressedState::reset() {
40         reset(CreateRandomBinaryString(N));
41 }
42
43 void CompressedState::reset(string _newInitialCondition) {
44         if (_newInitialCondition.empty()) {
45                 throw exception("Initial_condition_cannot_be_empty");
46         }
47         if (N < _newInitialCondition.length()) {
48                 string ex = "Initial_condition_cannot_be_longer_than_current_N:_" + to_string(N);
49                 throw exception(ex.c_str());
50         }
51         if (!IsBinaryString(_newInitialCondition)) {
52                 throw exception("Initial_condition_is_not_a_binary_string");
53         }
54
55         createChunkArray(&initialStateChunkArray);
56         createChunkArray(&currentStateChunkArray);
57
58         start(_newInitialCondition);
59 }
60
61 void CompressedState::start(string& _initialCondition) {
62         mostSignificativeChunkPosition = chunkArrayLength - 1;
63         currentIteration = 0;
64
65         compress(&initialStateChunkArray, _initialCondition);
66         copyChunkArray(&initialStateChunkArray, &currentStateChunkArray);
67 }
68
69 void CompressedState::restart() {
70         mostSignificativeChunkPosition = chunkArrayLength - 1;
71         currentIteration = 0;
72
73         copyChunkArray(&initialStateChunkArray, &currentStateChunkArray);
74 }
75
76 void CompressedState::saveState() {
77         copyChunkArray(&currentStateChunkArray, &savedStateChunkArray);
78         savedIteration = currentIteration;
79         savedMSChunkPosition = mostSignificativeChunkPosition;
80 }
81
82 void CompressedState::applySavedState(bool applySavedCount) {
83         if (savedIteration == -1) {
84                 throw exception("Saved_state_not_found");
85         }
86
87         mostSignificativeChunkPosition = savedMSChunkPosition;
88         copyChunkArray(&savedStateChunkArray, &currentStateChunkArray);
89
90         if (applySavedCount) {
91                 currentIteration = savedIteration;
92         }
```

```
93          else {
94                  currentIteration++;
95          }
96  }
97
98  void CompressedState::resetWithSavedState() {
99          if (savedIteration == -1) {
100                 throw exception("Saved_state_not_found");
101         }
102
103         currentIteration = 0;
104         mostSignificativeChunkPosition = savedMSChunkPosition;
105         copyChunkArray(&savedStateChunkArray, &initialStateChunkArray);
106 }
107
108 void CompressedState::compress(CHUNK** pDestArray, string & _state) {
109         int i, j, k;
110         string stateChunk = "";
111         for (i = _state.length() - 1, j = 1, k = 0; i >= 0; i--, j++) {
112                 stateChunk = _state.at(i) + stateChunk;
113
114                 if (j >= CHUNK_BITSIZE) {
115                         (*pDestArray)[k] = stoi(stateChunk, 0, 2);
116
117                         stateChunk = "";
118                         k++;
119                         j = 0;
120                 }
121         }
122         if (!stateChunk.empty()) {
123                 (*pDestArray)[k] = stoi(stateChunk, 0, 2);
124                 k++;
125         }
126
127         for (k; k < chunkArrayLength; k++) {
128                 (*pDestArray)[k] = 0;
129         }
130 }
131
132 void CompressedState::createChunkArray(CHUNK** pChunkArray) {
133         *pChunkArray = (CHUNK*)malloc(sizeof(CHUNK) * chunkArrayLength);
134 }
135
136 void CompressedState::copyChunkArray(CHUNK** pSourceArray, CHUNK** pDestArray) {
137         CHUNK* iSrc = *pSourceArray;
138         CHUNK* iDst = *pDestArray;
139
140         int i;
141         for (i = 0; i < chunkArrayLength; i++, iSrc++, iDst++) {
142                 *iDst = *iSrc;
143         }
144 }
145
146 void CompressedState::applyRule(ChunkLUTAgent& lutAgent) {
```

```
147         if (chunkArrayLength == 1) {
148             currentStateChunkArray[0] = lutAgent.applyRule(currentStateChunkArray[0],
149                 currentStateChunkArray[0] & 1,
150                 getMSBAt(0),
151                 isMostSignificativeChunkPosition(0)
152             );
153         }
154         else {
155             CHUNK leftmostChunk = currentStateChunkArray[0];
156             CHUNK leftmostNextChunk = lutAgent.applyRule(leftmostChunk,
157                 currentStateChunkArray[chunkArrayLength - 1] & 1,
158                 getMSBAt(1),
159                 isMostSignificativeChunkPosition(0)
160             );
161             int msbAt0 = getMSBAt(0);
162 
163             int i;
164             for (i = 1; i < chunkArrayLength - 1; i++) {
165                 currentStateChunkArray[i - 1] = lutAgent.applyRule(currentStateChunkArray[i],
166                     currentStateChunkArray[i - 1] & 1,
167                     getMSBAt(i + 1),
168                     isMostSignificativeChunkPosition(i)
169                 );
170             }
171             currentStateChunkArray[chunkArrayLength - 2] = lutAgent.applyRule(
                ↪ currentStateChunkArray[chunkArrayLength - 1],
172                 currentStateChunkArray[chunkArrayLength - 2] & 1,
173                 msbAt0,
174                 isMostSignificativeChunkPosition(chunkArrayLength - 1)
175             );
176 
177             currentStateChunkArray[chunkArrayLength - 1] = leftmostNextChunk;
178 
179             mostSignificativeChunkPosition = (mostSignificativeChunkPosition - 1 +
                ↪ chunkArrayLength) % chunkArrayLength;
180         }
181 }
182 
183 bool CompressedState::hasAdjustedMostSignificativeChunk() {
184         return mostSignificativeChunkBitSize != CHUNK_BITSIZE;
185 }
186 
187 size_t CompressedState::getMostSignificativeChunkPosition() {
188         return mostSignificativeChunkPosition;
189 }
190 
191 bool CompressedState::isMostSignificativeChunkPosition(int i) {
192         return mostSignificativeChunkPosition == i;
193 }
194 
195 int CompressedState::getMostSignificativeChunkBitSize() {
196         return mostSignificativeChunkBitSize;
197 }
198 
```

```
199 size_t CompressedState::getChunkArrayLength() {
200         return chunkArrayLength;
201 }
202
203 long CompressedState::getCurrentIteration() {
204         return currentIteration;
205 }
206
207 CHUNK CompressedState::at(int i) {
208         return currentStateChunkArray[i];
209 }
210
211 int CompressedState::getMSBAt(int i) {
212         if (i == mostSignificativeChunkPosition) {
213                 return (currentStateChunkArray[i] >> (mostSignificativeChunkBitSize - 1)) & 1;
214         }
215         else {
216                 return (currentStateChunkArray[i] >> (CHUNK_BITSIZE - 1)) & 1;
217         }
218 }
219
220 const vector<string>& CompressedState::getRawState() {
221         int i, j;
222
223         rawState[0] = ChunkTranslator::GetInstance().translate(
224                 currentStateChunkArray[mostSignificativeChunkPosition]
225         ).substr(CHUNK_BITSIZE - mostSignificativeChunkBitSize);
226
227         for (i = mostSignificativeChunkPosition - 1, j = 1; i >= 0; i--, j++) {
228                 rawState[j] = ChunkTranslator::GetInstance().translate(currentStateChunkArray[i]);
229         }
230         for (i = chunkArrayLength - 1; i > mostSignificativeChunkPosition; i--, j++) {
231                 rawState[j] = ChunkTranslator::GetInstance().translate(currentStateChunkArray[i]);
232         }
233
234         return rawState;
235 }
236
237 char CompressedState::setAtPosition(int position, char newValue) {
238         throw exception("Not_implemented");
239
240         /*int oldValue = getAtPosition(position);
241
242         int relativePos = position / CHUNK_BITSIZE;
243         int offsetPos = position % CHUNK_BITSIZE;
244
245         currentStateChunkArray[relativePos] |= 1 << offsetPos;
246
247         return oldValue;*/
248
249         return 0;
250 }
251
252 char CompressedState::getAtPosition(int position) {
```

```
253         throw exception("Not_implemented");
254
255         /*int relativePos = position / CHUNK_BITSIZE;
256         int offsetPos = position % CHUNK_BITSIZE;
257
258         return (currentStateChunkArray[relativePos] >> offsetPos) & 1;*/
259
260         return 0;
261 }
```

ChunkLUTAgent.h

```
1  #pragma once
2
3  #include "EcaControllerCore.h"
4  #include "ChunkLUT.h"
5
6  class ECACONTROLLER_DLL ChunkLUTAgent {
7  private:
8          ChunkLUT* regularLut;
9          ChunkLUT* adjustedLut;
10
11         int N;
12         string rule;
13         int adjustedLutBitSize;
14
15 public:
16         ChunkLUTAgent(string _rule, int _N);
17         ~ChunkLUTAgent();
18
19         int getN();
20         string getRule();
21         bool hasAdjustedLut();
22         int getAdjustedLutBitSize();
23         CHUNK applyRule(CHUNK c, int leftLSB, int rightMSB, bool adjusted = false);
24 };
```

ChunkLUTAgent.cpp

```
1  #include "pch.h"
2  #include "ChunkLUTAgent.h"
3
4  ChunkLUTAgent::ChunkLUTAgent(string _rule, int _N) {
5          if (_N <= 0) {
6                  throw exception("N_cannot_be_negative_nor_zero");
7          }
8          if (_rule.length() != 8 || !IsBinaryString(_rule)) {
9                  throw exception("Rule_must_be_an_8-digit_binary_string");
10         }
11         rule = _rule;
12         N = _N;
13
```

```
14          adjustedLutBitSize = N % CHUNK_BITSIZE;
15          if (adjustedLutBitSize == 0) {
16                  adjustedLutBitSize = CHUNK_BITSIZE;
17          }
18
19          regularLut = new ChunkLUT(rule);
20          if (hasAdjustedLut()) {
21                  adjustedLut = new ChunkLUT(rule, adjustedLutBitSize);
22          }
23  }
24
25  ChunkLUTAgent::~ChunkLUTAgent() {
26          delete regularLut;
27          delete adjustedLut;
28  }
29
30  int ChunkLUTAgent::getN() {
31          return N;
32  }
33
34  string ChunkLUTAgent::getRule() {
35          return rule;
36  }
37
38  bool ChunkLUTAgent::hasAdjustedLut() {
39          return adjustedLutBitSize != 16;
40  }
41
42  int ChunkLUTAgent::getAdjustedLutBitSize() {
43          if (hasAdjustedLut()) {
44                  return adjustedLutBitSize;
45          }
46
47          return -1;
48  }
49
50  CHUNK ChunkLUTAgent::applyRule(CHUNK c, int leftLSB, int rightMSB, bool adjusted) {
51          if (adjusted && hasAdjustedLut()) {
52                  return adjustedLut->get(c, leftLSB, rightMSB);
53          }
54
55          return regularLut->get(c, leftLSB, rightMSB);
56  }
```

ChunkLUT.h

```
1   #pragma once
2   #include "EcaControllerCore.h"
3
4   using namespace std;
5   using namespace EcaControllerCore;
6
7   class ECACONTROLLER_DLL ChunkLUT {
```

```
8  private:
9          CHUNK*** lut;
10         string rule;
11         size_t bitSize;
12         int lutSize;
13
14 public:
15         ChunkLUT(string _rule, int _bitSize = CHUNK_BITSIZE);
16         ~ChunkLUT();
17
18         int getLutSize();
19
20         CHUNK get(int state, int leftLSB, int rightMSB);
21
22 private:
23         char applyElementalRule(char prev, char cur, char next);
24 };
```

ChunkLUT.cpp

```
1  #include "pch.h"
2  #include "ChunkLUT.h"
3
4  ChunkLUT::ChunkLUT(string _rule, int _bitSize) {
5          rule = _rule;
6
7          bitSize = _bitSize;
8          lutSize = pow(2, bitSize);
9
10         lut = (CHUNK***)malloc(sizeof(CHUNK**) * lutSize);
11
12         int i, j;
13         for (i = 0; i < lutSize; i++) {
14                 lut[i] = (CHUNK**)malloc(sizeof(CHUNK*) * 2);
15
16                 lut[i][0] = (CHUNK*)malloc(sizeof(CHUNK) * 2);
17                 lut[i][1] = (CHUNK*)malloc(sizeof(CHUNK) * 2);
18
19                 string iState = FormatBinaryString(ToBinaryString(i), bitSize);
20                 string nextIState = FormatBinaryString("", bitSize);
21
22                 if (iState.length() == 1) {
23                         nextIState.at(0) = applyElementalRule('0', iState.at(0), '0');
24                         lut[i][0][0] = stoi(nextIState, 0, 2);
25                         nextIState.at(0) = applyElementalRule('0', iState.at(0), '1');
26                         lut[i][0][1] = stoi(nextIState, 0, 2);
27                         nextIState.at(0) = applyElementalRule('1', iState.at(0), '0');
28                         lut[i][1][0] = stoi(nextIState, 0, 2);
29                         nextIState.at(0) = applyElementalRule('1', iState.at(0), '1');
30                         lut[i][1][1] = stoi(nextIState, 0, 2);
31                 }
32                 else {
33                         char prev, cur, next;
```

```
34                              for (j = 1; j < iState.length() - 1; j++) {
35                                      prev = iState.at(j - 1);
36                                      cur = iState.at(j);
37                                      next = iState.at(j + 1);
38
39                                      nextIState.at(j) = applyElementalRule(prev, cur, next);
40                              }
41
42                              nextIState.at(0) = applyElementalRule('0', iState.at(0), iState.at(1));
43                              nextIState.at(iState.length() - 1) = applyElementalRule(iState.at(iState.length()
                                    ↪ - 2), iState.at(iState.length() - 1), '0');
44                              lut[i][0][0] = stoi(nextIState, 0, 2);
45
46                              nextIState.at(0) = applyElementalRule('0', iState.at(0), iState.at(1));
47                              nextIState.at(iState.length() - 1) = applyElementalRule(iState.at(iState.length()
                                    ↪ - 2), iState.at(iState.length() - 1), '1');
48                              lut[i][0][1] = stoi(nextIState, 0, 2);
49
50                              nextIState.at(0) = applyElementalRule('1', iState.at(0), iState.at(1));
51                              nextIState.at(iState.length() - 1) = applyElementalRule(iState.at(iState.length()
                                    ↪ - 2), iState.at(iState.length() - 1), '0');
52                              lut[i][1][0] = stoi(nextIState, 0, 2);
53
54                              nextIState.at(0) = applyElementalRule('1', iState.at(0), iState.at(1));
55                              nextIState.at(iState.length() - 1) = applyElementalRule(iState.at(iState.length()
                                    ↪ - 2), iState.at(iState.length() - 1), '1');
56                              lut[i][1][1] = stoi(nextIState, 0, 2);
57                      }
58              }
59      }
60
61      ChunkLUT::~ChunkLUT() {
62              int i;
63              for(i = 0; i < lutSize; i++) {
64                      free(lut[i][0]);
65                      free(lut[i][1]);
66
67                      free(lut[i]);
68              }
69
70              free(lut);
71      }
72
73      int ChunkLUT::getLutSize() {
74              return lutSize;
75      }
76
77      CHUNK ChunkLUT::get(int state, int leftLSB, int rightMSB) {
78              return lut[state][leftLSB][rightMSB];
79      }
80
81      char ChunkLUT::applyElementalRule(char prev, char cur, char next) {
82              if (prev == '1') {
83                      if (cur == '1') {
```

```
84                          if (next == '1') { // 111
85                                  return rule.at(0);
86                          }
87                          else { // 110
88                                  return rule.at(1);
89                          }
90                  }
91                  else {
92                          if (next == '1') { // 101
93                                  return rule.at(2);
94                          }
95                          else { // 100
96                                  return rule.at(3);
97                          }
98                  }
99          }
100         else {
101                 if (cur == '1') {
102                         if (next == '1') { // 011
103                                 return rule.at(4);
104                         }
105                         else { // 010
106                                 return rule.at(5);
107                         }
108                 }
109                 else {
110                         if (next == '1') { // 001
111                                 return rule.at(6);
112                         }
113                         else { // 000
114                                 return rule.at(7);
115                         }
116                 }
117         }
118 }
```

ChunkTranslator.h

```
1  #pragma once
2
3  #include <vector>
4
5  #include "EcaControllerCore.h"
6
7  using namespace std;
8  using namespace EcaControllerCore;
9
10 class ECACONTROLLER_DLL ChunkTranslator {
11 public:
12     static ChunkTranslator& GetInstance();
13
14 private:
15     vector<string> CHUNK_TRANSLATOR;
```

```
16      ChunkTranslator();
17
18      //ChunkTranslator(ChunkTranslator const&);
19      //void operator=(ChunkTranslator const&);
20
21  public:
22      ChunkTranslator(ChunkTranslator const&) = delete;
23      void operator=(ChunkTranslator const&) = delete;
24
25      string translate(int k);
26  };
```

ChunkTranslator.cpp

```
1   #include "pch.h"
2   #include "ChunkTranslator.h"
3
4   ChunkTranslator& ChunkTranslator::GetInstance() {
5           static ChunkTranslator instance;
6           return instance;
7   }
8
9   ChunkTranslator::ChunkTranslator() {
10           int ctLength = pow(2, CHUNK_BITSIZE);
11           int i;
12           for (i = 0; i < ctLength; i++) {
13               CHUNK_TRANSLATOR.push_back(FormatBinaryString(i, CHUNK_BITSIZE));
14           }
15  }
16
17  string ChunkTranslator::translate(int k) {
18          /*if (k < 0 || k >= CHUNK_TRANSLATOR.size()) {
19                  throw exception("Index out of bounds for chunk translator of size " +
                        ↪ CHUNK_TRANSLATOR.size());
20          }*/
21
22          return CHUNK_TRANSLATOR[k];
23  }
```

InteractionController.h

```
1   #pragma once
2
3   #include <map>
4
5   #include "EcaControllerCore.h"
6   #include "NaiveController.h"
7   #include "InteractionPoint.h"
8
9   class ECACONTROLLER_DLL InteractionController : public NaiveController {
10  private:
11          std::map<int, InteractionPoint> interactionPoints;
```

```
12
13 public:
14         InteractionController(int _N, int _rule);
15         InteractionController(int _N, int _rule, string _initialCondition);
16
17         void hardReset(const string& newIc);
18
19         void setEnabled(int position, bool enable);
20         void enableAll();
21         void disableAll();
22
23         bool isInteractionPoint(int position) const;
24         bool isEnabledInteractionPoint(int position) const;
25         void addInteractionPoint(int position, bool enabled = false);
26         bool removeInteractionPoint(int position);
27         void updateInteractionPoint(int position, char leftVal, char rightVal);
28
29         void getNeighbors(int i, char& prev, char& cur, char& next) override;
30 };
```

InteractionController.cpp

```
1  #include "pch.h"
2  #include "InteractionController.h"
3
4  InteractionController::InteractionController(int _N, int _rule)
5          : NaiveController(_N, _rule) {
6  }
7
8  InteractionController::InteractionController(int _N, int _rule, string _initialCondition)
9          : NaiveController(_N, _rule, _initialCondition) {
10 }
11
12 void InteractionController::hardReset(const string& newIc) {
13         currentState = newIc;
14         auxState = currentState;
15
16         currentIteration = 0;
17 }
18
19 bool InteractionController::isInteractionPoint(int position) const {
20         return !(interactionPoints.find(position) == interactionPoints.end());
21 }
22
23 bool InteractionController::isEnabledInteractionPoint(int position) const {
24         return isInteractionPoint(position) && interactionPoints.at(position).isEnabled();
25 }
26
27 void InteractionController::setEnabled(int position, bool enable) {
28         if (!isInteractionPoint(position)) {
29                 throw exception("Position_for_interaction_point_not_found");
30         }
31
```

```
32          interactionPoints[position].setEnabled(enable);
33  }
34
35  void InteractionController::enableAll() {
36          for (auto& val : interactionPoints) {
37                  val.second.enable();
38          }
39  }
40
41  void InteractionController::disableAll() {
42          for (auto& val : interactionPoints) {
43                  val.second.enable();
44          }
45  }
46
47  void InteractionController::addInteractionPoint(int position, bool enabled) {
48          if (isInteractionPoint(position)) {
49                  throw exception("Position for interaction point already registered");
50          }
51
52          interactionPoints[position] = InteractionPoint(position, enabled);
53  }
54
55  bool InteractionController::removeInteractionPoint(int position) {
56          if (!isInteractionPoint(position)) {
57                  return false;
58          }
59
60          interactionPoints.erase(position);
61
62          return true;
63  }
64
65  void InteractionController::updateInteractionPoint(int position, char leftVal, char rightVal) {
66          if (!isInteractionPoint(position)) {
67                  throw exception("Position for interaction point not found");
68          }
69
70          interactionPoints[position].updateValues(leftVal, rightVal);
71  }
72
73  void InteractionController::getNeighbors(int i, char& prev, char& cur, char& next) {
74          if (isEnabledInteractionPoint(i)) {
75                  prev = currentState.at((i - 1 + N) % N);
76                  cur = currentState.at(i);
77                  next = interactionPoints[i].getRightValue();
78          }
79          else if (isEnabledInteractionPoint((i - 1 + N) % N)) {
80                  prev = interactionPoints[(i - 1 + N) % N].getLeftValue();
81                  cur = currentState.at(i);
82                  next = currentState.at((i + 1) % N);
83          }
84          else {
85                  prev = currentState.at((i - 1 + N) % N);
```

```
86                      cur = currentState.at(i);
87                      next = currentState.at((i + 1) % N);
88              }
89      }
```

InteractionAgent.h

```
1   #pragma once
2
3   #include "InteractionController.h"
4   #include "EcaControllerCore.h"
5
6   class ECACONTROLLER_DLL InteractionAgent {
7   private:
8           InteractionController& controller1;
9           int interactionPosition1;
10
11          InteractionController& controller2;
12          int interactionPosition2;
13
14          bool enabled;
15
16  public:
17          InteractionAgent(InteractionController& controller1, int interactionPosition1,
                ↪ InteractionController& controller2, int interactionPosition2);
18
19          InteractionController& getController(int i);
20
21          int getInteractionPositionFor(InteractionController* controller) const;
22          int getInteractionPositionFor(int i) const;
23
24          bool executePoint();
25
26          void enable();
27          void disable();
28          void toggle();
29          bool setEnabled(bool _enabled);
30
31          bool isEnabled() const;
32  };
```

InteractionAgent.cpp

```
1   #include "pch.h"
2   #include "InteractionAgent.h"
3
4   InteractionAgent::InteractionAgent(InteractionController& controller1, int interactionPosition1,
        ↪ InteractionController& controller2, int interactionPosition2)
5           : controller1(controller1),
6           interactionPosition1(interactionPosition1),
7           controller2(controller2),
8           interactionPosition2(interactionPosition2),
```

```
9           enabled(false) {
10          controller1.addInteractionPoint(interactionPosition1);
11          controller2.addInteractionPoint(interactionPosition2);
12  }
13
14  InteractionController& InteractionAgent::getController(int i) {
15          if (i == 1) {
16                  return controller1;
17          }
18
19          return controller2;
20  }
21
22  int InteractionAgent::getInteractionPositionFor(InteractionController* controller) const {
23          if (controller == &controller1) {
24                  return interactionPosition1;
25          }
26
27          if (controller == &controller2) {
28                  return interactionPosition2;
29          }
30
31          return -1;
32  }
33
34  int InteractionAgent::getInteractionPositionFor(int i) const {
35          if (i == 1) {
36                  return interactionPosition1;
37          }
38
39          return interactionPosition2;
40  }
41
42  bool InteractionAgent::executePoint() {
43          if(isEnabled()) {
44                  int nextPoint1 = interactionPosition1 + 1 < controller1.getN() ? interactionPosition1 + 1
                        ↪  : 0;
45                  int nextPoint2 = interactionPosition2 + 1 < controller2.getN() ? interactionPosition2 + 1
                        ↪  : 0;
46
47                  char old1prev = controller1.getAtPosition(interactionPosition1);
48                  char old2prev = controller2.getAtPosition(interactionPosition2);
49                  char old1next = controller1.getAtPosition(nextPoint1);
50                  char old2next = controller2.getAtPosition(nextPoint2);
51
52                  controller1.updateInteractionPoint(interactionPosition1, old2prev, old2next);
53                  controller2.updateInteractionPoint(interactionPosition2, old1prev, old1next);
54
55                  return true;
56          }
57
58          return false;
59  }
60
```

```
61 void InteractionAgent::enable() {
62         enabled = true;
63         controller1.setEnabled(interactionPosition1, true);
64         controller2.setEnabled(interactionPosition2, true);
65 }
66
67 void InteractionAgent::disable() {
68         enabled = false;
69         controller1.setEnabled(interactionPosition1, false);
70         controller2.setEnabled(interactionPosition2, false);
71 }
72
73 void InteractionAgent::toggle() {
74         enabled = !enabled;
75         controller1.setEnabled(interactionPosition1, enabled);
76         controller2.setEnabled(interactionPosition2, enabled);
77 }
78
79 bool InteractionAgent::setEnabled(bool _enabled) {
80         bool oldEnabled = enabled;
81
82         enabled = _enabled;
83         controller1.setEnabled(interactionPosition1, enabled);
84         controller2.setEnabled(interactionPosition2, enabled);
85
86         return oldEnabled;
87 }
88
89 bool InteractionAgent::isEnabled() const {
90         return enabled && controller1.isEnabledInteractionPoint(interactionPosition1) && controller2.
               ↪ isEnabledInteractionPoint(interactionPosition2);
91 }
```

InteractionPoint.h

```
1  #pragma once
2
3  #include "EcaControllerCore.h"
4
5  class ECACONTROLLER_DLL InteractionPoint {
6  private:
7          int position;
8          char leftValue;
9          char rightValue;
10
11         bool enabled;
12
13 public:
14         InteractionPoint(int position = 0, bool enabled = false);
15
16         void updateValues(char _leftValue, char _rightValue);
17
18         void enable();
```

```
19          void disable();
20          void toggle();
21
22          bool isEnabled() const;
23          void setEnabled(bool _enabled);
24
25
26          int getPosition() const;
27          void setPosition(int _position);
28          char getLeftValue() const;
29          void setLeftValue(char _leftValue);
30          char getRightValue() const;
31          void setRightValue(char _rightValue);
32  };
```

InteractionPoint.cpp

```
1   #include "pch.h"
2   #include "InteractionPoint.h"
3
4   InteractionPoint::InteractionPoint(int position, bool enabled)
5           : position(position),
6           leftValue('0'),
7           rightValue('0'),
8           enabled(enabled) {
9   }
10
11  void InteractionPoint::updateValues(char _leftValue, char _rightValue) {
12          leftValue = _leftValue;
13          rightValue = _rightValue;
14  }
15
16  void InteractionPoint::enable() {
17          enabled = true;
18  }
19
20  void InteractionPoint::disable() {
21          enabled = false;
22  }
23
24  void InteractionPoint::toggle() {
25          enabled = !enabled;
26  }
27
28  bool InteractionPoint::isEnabled() const {
29          return enabled;
30  }
31
32  void InteractionPoint::setEnabled(bool _enabled) {
33          enabled = _enabled;
34  }
35
36  int InteractionPoint::getPosition() const {
```

```
37          return position;
38  }
39
40  void InteractionPoint::setPosition(int _position) {
41          position = _position;
42  }
43
44  char InteractionPoint::getLeftValue() const {
45          return leftValue;
46  }
47
48  void InteractionPoint::setLeftValue(char _leftValue) {
49          leftValue = _leftValue;
50  }
51
52  char InteractionPoint::getRightValue() const {
53          return rightValue;
54  }
55
56  void InteractionPoint::setRightValue(char _rightValue) {
57          rightValue = _rightValue;
58  }
```

CollisionSystem.h

```
1   #pragma once
2
3   #include <functional>
4   #include <memory>
5
6
7   #include "ColliderActionList.h"
8   #include "EcaControllerCore.h"
9   #include "InteractionAgent.h"
10  #include "InteractionController.h"
11
12  class ECACONTROLLER_DLL CollisionSystem {
13  protected:
14          int rule = 110;
15
16          long currentIteration;
17
18          InteractionController leftRing;
19          InteractionController rightRing;
20          InteractionController centralRing;
21
22          InteractionAgent leftInteractionAgent;
23          InteractionAgent rightInteractionAgent;
24
25          bool leftRingEnabled;
26          bool rightRingEnabled;
27          bool centralRingEnabled;
28
```

```
29          unique_ptr<ColliderActionList> actionList;
30
31  public:
32          CollisionSystem(int leftN, string leftIC, int rightN, string rightIC, int centralN, string centralIC,
33                                                  int leftToCentralIP, int centralToLeftIP, int rightToCentralIP,
                                                        ↪ int centralToRightIP);
34
35          CollisionSystem(string leftIC, string rightIC, string centralIC,
36                  int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int centralToRightIP,
37                  string actionList);
38
39          CollisionSystem(string leftIC, string rightIC, string centralIC,
40                  int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int centralToRightIP,
41                  vector<string> actionList);
42
43          virtual bool setLeftContactEnabled(bool enabled);
44          virtual bool setRightContactEnabled(bool enabled);
45          virtual void setAllContactsEnabled(bool enabled);
46
47          bool isLeftContactEnabled() const;
48          bool isRightContactEnabled() const;
49
50          bool setLeftRingEnabled(bool enabled);
51          bool setRightRingEnabled(bool enabled);
52          bool setCentralRingEnabled(bool enabled);
53          bool isLeftRingEnabled() const;
54          bool isRightRingEnabled() const;
55          bool isCentralRingEnabled() const;
56
57          virtual void execute();
58          virtual void restart();
59
60          int getLeftN();
61          int getRightN();
62          int getCentralN();
63
64          virtual const string& getLeftState();
65          virtual const string& getRightState();
66          virtual const string& getCentralState();
67
68          virtual long getCurrentIteration();
69
70          int getLeftToCentralInteractionPos() const;
71          int getCentralToLeftInteractionPos() const;
72          int getRightToCentralInteractionPos() const;
73          int getCentralToRightInteractionPos() const;
74
75          void executeActions();
76          void executeAllInteractions();
77          void applyAll(bool cancelIterationIncrement = false);
78
79          virtual void jumpToIteration(long it, function<void(int)>& lambda);
80  };
```

CollisionSystem.cpp

```
1  #include "pch.h"
2  #include "CollisionSystem.h"
3
4  #include <thread>
5
6  CollisionSystem::CollisionSystem(int leftN, string leftIC, int rightN, string rightIC, int centralN, string
       ↪ centralIC,
7                                   int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int
                                        ↪ centralToRightIP)
8                  : currentIteration(0),
9                  leftRing(leftN, rule, leftIC),
10                 rightRing(rightN, rule, rightIC),
11                 centralRing(centralN, rule, centralIC),
12                 leftRingEnabled(true),
13                 rightRingEnabled(true),
14                 centralRingEnabled(true),
15                 leftInteractionAgent(leftRing, leftToCentralIP, centralRing, centralToLeftIP),
16                 rightInteractionAgent(rightRing, rightToCentralIP, centralRing, centralToRightIP),
17                 actionList(new ColliderActionList("")) {
18
19 }
20
21 CollisionSystem::CollisionSystem(string leftIC, string rightIC, string centralIC, int leftToCentralIP,
22                                                                 int centralToLeftIP, int
                                                                      ↪ rightToCentralIP, int
                                                                      ↪ centralToRightIP, string
                                                                      ↪ _actionList)
23                 : currentIteration(0),
24                 leftRing(leftIC.length(), rule, leftIC),
25                 rightRing(rightIC.length(), rule, rightIC),
26                 centralRing(centralIC.length(), rule, centralIC),
27                 leftRingEnabled(true),
28                 rightRingEnabled(true),
29                 centralRingEnabled(true),
30                 leftInteractionAgent(leftRing, leftToCentralIP, centralRing, centralToLeftIP),
31                 rightInteractionAgent(rightRing, rightToCentralIP, centralRing, centralToRightIP),
32                 actionList(new ColliderActionList(_actionList)) {
33
34 }
35
36 CollisionSystem::CollisionSystem(string leftIC, string rightIC, string centralIC, int leftToCentralIP,
37         int centralToLeftIP, int rightToCentralIP, int centralToRightIP, vector<string> _actionList)
38         : currentIteration(0),
39         leftRing(leftIC.length(), rule, leftIC),
40         rightRing(rightIC.length(), rule, rightIC),
41         centralRing(centralIC.length(), rule, centralIC),
42         leftRingEnabled(true),
43         rightRingEnabled(true),
44         centralRingEnabled(true),
45         leftInteractionAgent(leftRing, leftToCentralIP, centralRing, centralToLeftIP),
46         rightInteractionAgent(rightRing, rightToCentralIP, centralRing, centralToRightIP),
47         actionList(new ColliderActionList(_actionList)) {
```

```
48
49  }
50
51  bool CollisionSystem::setLeftContactEnabled(bool enabled) {
52          return leftInteractionAgent.setEnabled(enabled);
53  }
54
55  bool CollisionSystem::setRightContactEnabled(bool enabled) {
56          return rightInteractionAgent.setEnabled(enabled);
57  }
58
59  void CollisionSystem::setAllContactsEnabled(bool enabled) {
60          leftInteractionAgent.setEnabled(enabled);
61          rightInteractionAgent.setEnabled(enabled);
62  }
63
64  bool CollisionSystem::isLeftContactEnabled() const {
65          return leftInteractionAgent.isEnabled();
66  }
67
68  bool CollisionSystem::isRightContactEnabled() const {
69          return rightInteractionAgent.isEnabled();
70  }
71
72  bool CollisionSystem::setLeftRingEnabled(bool enabled) {
73          bool prev = leftRingEnabled;
74          leftRingEnabled = enabled;
75          return prev;
76  }
77
78  bool CollisionSystem::setRightRingEnabled(bool enabled) {
79          bool prev = rightRingEnabled;
80          rightRingEnabled = enabled;
81          return prev;
82  }
83
84  bool CollisionSystem::setCentralRingEnabled(bool enabled) {
85          bool prev = centralRingEnabled;
86          centralRingEnabled = enabled;
87          return prev;
88  }
89
90  bool CollisionSystem::isLeftRingEnabled() const {
91          return leftRingEnabled;
92  }
93
94  bool CollisionSystem::isRightRingEnabled() const {
95          return rightRingEnabled;
96  }
97
98  bool CollisionSystem::isCentralRingEnabled() const {
99          return centralRingEnabled;
100 }
101
```

```
102 void CollisionSystem::execute() {
103         executeActions();
104         executeAllInteractions();
105         applyAll();
106 }
107
108 void CollisionSystem::restart() {
109         leftRing.restart();
110         rightRing.restart();
111         centralRing.restart();
112
113         leftRingEnabled = true;
114         rightRingEnabled = true;
115         centralRingEnabled = true;
116
117         currentIteration = 0;
118 }
119
120 int CollisionSystem::getLeftN() {
121         return leftRing.getN();
122 }
123
124 int CollisionSystem::getRightN() {
125         return rightRing.getN();
126 }
127
128 int CollisionSystem::getCentralN() {
129         return centralRing.getN();
130 }
131
132 const string& CollisionSystem::getLeftState() {
133         return leftRing.getCurrentState()[0];
134 }
135
136 const string& CollisionSystem::getRightState() {
137         return rightRing.getCurrentState()[0];
138 }
139
140 const string& CollisionSystem::getCentralState() {
141         return centralRing.getCurrentState()[0];
142 }
143
144 long CollisionSystem::getCurrentIteration() {
145         return currentIteration;
146 }
147
148 int CollisionSystem::getLeftToCentralInteractionPos() const {
149         return leftInteractionAgent.getInteractionPositionFor(1);
150 }
151
152 int CollisionSystem::getCentralToLeftInteractionPos() const {
153         return leftInteractionAgent.getInteractionPositionFor(2);
154 }
155
```

```
156 int CollisionSystem::getRightToCentralInteractionPos() const {
157         return rightInteractionAgent.getInteractionPositionFor(1);
158 }
159
160 int CollisionSystem::getCentralToRightInteractionPos() const {
161         return rightInteractionAgent.getInteractionPositionFor(2);
162 }
163
164 void CollisionSystem::executeAllInteractions() {
165         if (leftRingEnabled && centralRingEnabled && leftInteractionAgent.isEnabled()) {
166                 leftInteractionAgent.executePoint();
167         }
168         if (rightRingEnabled && centralRingEnabled && rightInteractionAgent.isEnabled()) {
169                 rightInteractionAgent.executePoint();
170         }
171 }
172
173 void CollisionSystem::applyAll(bool cancelIterationIncrement) {
174         auto leftLambda = [&]() {
175                 if (leftRingEnabled)
176                         leftRing.applyRule();
177         };
178         auto rightLambda = [&]() {
179                 if (rightRingEnabled)
180                         rightRing.applyRule();
181         };
182         auto centralLambda = [&]() {
183                 if (centralRingEnabled)
184                         centralRing.applyRule();
185         };
186
187         std::thread leftThread(leftLambda);
188         std::thread rightThread(rightLambda);
189         std::thread centralThread(centralLambda);
190
191         leftThread.join();
192         rightThread.join();
193         centralThread.join();
194
195         if(!cancelIterationIncrement) {
196                 currentIteration++;
197         }
198 }
199
200 void CollisionSystem::jumpToIteration(long it, function<void(int)>& lambda) {
201         if (it < currentIteration) {
202                 restart();
203         }
204
205         while (currentIteration < it) {
206                 execute();
207                 lambda(currentIteration);
208         }
209 }
```

```
210
211
212 void CollisionSystem::executeActions() {
213         auto it = actionList->getActions().equal_range(currentIteration);
214
215         for (auto itr = it.first; itr != it.second; ++itr) {
216                 switch (itr->second) {
217                 case ColliderAction::CONTACT_DISABLE_LEFT:
218                         setLeftContactEnabled(false);
219                         break;
220                 case ColliderAction::CONTACT_DISABLE_RIGHT:
221                         setRightContactEnabled(false);
222                         break;
223                 case ColliderAction::CONTACT_ENABLE_LEFT:
224                         setLeftContactEnabled(true);
225                         break;
226                 case ColliderAction::CONTACT_ENABLE_RIGHT:
227                         setRightContactEnabled(true);
228                         break;
229                 case ColliderAction::RING_DISABLE_LEFT:
230                         setLeftRingEnabled(false);
231                         break;
232                 case ColliderAction::RING_DISABLE_RIGHT:
233                         setRightRingEnabled(false);
234                         break;
235                 case ColliderAction::RING_DISABLE_CENTRAL:
236                         setCentralRingEnabled(false);
237                         break;
238                 case ColliderAction::RING_ENABLE_LEFT:
239                         setLeftRingEnabled(true);
240                         break;
241                 case ColliderAction::RING_ENABLE_RIGHT:
242                         setRightRingEnabled(true);
243                         break;
244                 case ColliderAction::RING_ENABLE_CENTRAL:
245                         setCentralRingEnabled(true);
246                         break;
247                 }
248         }
249 }
```

ColliderActionList.h

```
1  #pragma once
2
3  #include <vector>
4  #include <map>
5
6  #include "EcaControllerCore.h"
7
8  enum class ECACONTROLLER_DLL ColliderAction {
9          RING_DISABLE_LEFT,
10         RING_DISABLE_RIGHT,
```

```
11          RING_DISABLE_CENTRAL,
12          RING_ENABLE_LEFT,
13          RING_ENABLE_RIGHT,
14          RING_ENABLE_CENTRAL,
15          CONTACT_DISABLE_LEFT,
16          CONTACT_DISABLE_RIGHT,
17          CONTACT_ENABLE_LEFT,
18          CONTACT_ENABLE_RIGHT
19  };
20
21  class ECACONTROLLER_DLL ColliderActionList {
22  private:
23          std::multimap<long, ColliderAction> actions;
24
25          ColliderAction parseAction(std::string s);
26          std::map<std::string, ColliderAction> table;
27
28  public:
29          ColliderActionList(std::string _actions);
30          ColliderActionList(std::vector<std::string> _actions);
31          const std::multimap<long, ColliderAction>& getActions();
32          static int validate(std::vector<std::string> _actions);
33          static std::string getActionsRegexStr();
34          static std::map<std::string, ColliderAction> getActionTable();
35  };
```

ColliderActionList.cpp

```
1   #include "pch.h"
2   #include "ColliderActionList.h"
3
4   #include <algorithm>
5   #include <regex>
6   #include <sstream>
7
8   ColliderActionList::ColliderActionList(std::string _actions) {
9           transform(_actions.begin(), _actions.end(), _actions.begin(), [](unsigned char c) {
10                  return toupper(c);
11                  });
12
13          std::stringstream ss(_actions);
14          std::string item;
15          std::vector<std::string> elems;
16          while (std::getline(ss, item, '\n')) {
17                  elems.push_back(item);
18          }
19          std::string sNumRegex = "[0-9]+";
20          std::string sActionRegex = getActionsRegexStr();
21          std::string sFullRegex = sNumRegex + "-" + sActionRegex;
22
23          std::regex numRegex(sNumRegex);
24          std::regex actionEnumRegex(sActionRegex);
25          std::regex fullRegex(sFullRegex);
```

```
26
27          std::smatch match;
28
29          for (std::string s : elems) {
30                  s.erase(remove_if(s.begin(), s.end(), [](unsigned char c) {
31                          return isspace(c);
32                          }), s.end());
33
34                  if (s.length() != 0) {
35                          if (std::regex_match(s, fullRegex)) {
36                                  std::regex_search(s, match, numRegex);
37                                  long iteration = std::stoi(match[0]);
38
39                                  std::regex_search(s, match, actionEnumRegex);
40                                  ColliderAction action = parseAction(match[0]);
41
42                                  actions.insert({ iteration, action });
43                          }
44                          else {
45                                  std::string exS = "Error de parseo: " + s;
46                                  throw std::exception(exS.c_str());
47                          }
48                  }
49          }
50  }
51
52  ColliderActionList::ColliderActionList(std::vector<std::string> _actions) {
53          std::string sNumRegex = "[0-9]+";
54          std::string sActionRegex = getActionsRegexStr();
55          std::string sFullRegex = sNumRegex + "-" + sActionRegex;
56
57          std::regex numRegex(sNumRegex);
58          std::regex actionEnumRegex(sActionRegex);
59          std::regex fullRegex(sFullRegex);
60
61          std::smatch match;
62
63          for (std::string s : _actions) {
64                  transform(s.begin(), s.end(), s.begin(), [](unsigned char c) {
65                          return toupper(c);
66                          });
67                  s.erase(remove_if(s.begin(), s.end(), [](unsigned char c) {
68                          return isspace(c);
69                          }), s.end());
70
71                  if (s.length() != 0) {
72                          if (std::regex_match(s, fullRegex)) {
73                                  std::regex_search(s, match, numRegex);
74                                  long iteration = std::stoi(match[0]);
75
76                                  std::regex_search(s, match, actionEnumRegex);
77                                  ColliderAction action = parseAction(match[0]);
78
79                                  actions.insert({ iteration, action });
```

```
80                          }
81                          else {
82                                  std::string exS = "Error de parseo: " + s;
83                                  throw std::exception(exS.c_str());
84                          }
85                  }
86          }
87  }
88
89  int ColliderActionList::validate(std::vector<std::string> _actions) {
90          std::string sNumRegex = "[0-9]+";
91          std::string sActionRegex = getActionsRegexStr();
92          std::string sFullRegex = sNumRegex + "-" + sActionRegex;
93
94          std::regex fullRegex(sFullRegex);
95
96          std::smatch match;
97
98
99          for (int i = 0; i < _actions.size(); i++) {
100                 std::string s = _actions[i];
101                 transform(s.begin(), s.end(), s.begin(), [](unsigned char c) {
102                         return toupper(c);
103                         });
104                 s.erase(remove_if(s.begin(), s.end(), [](unsigned char c) {
105                         return isspace(c);
106                         }), s.end());
107
108                 if (s.length() != 0) {
109                         if (!std::regex_match(s, fullRegex)) {
110                                 return i;
111                         }
112                 }
113         }
114
115         return -1;
116 }
117
118 ColliderAction ColliderActionList::parseAction(std::string s) {
119         auto table = getActionTable();
120         return table.at(s);
121 }
122
123 std::map<std::string, ColliderAction> ColliderActionList::getActionTable() {
124         return {
125                 {"RING_DISABLE_LEFT",ColliderAction::RING_DISABLE_LEFT},
126                 {"RING_DISABLE_RIGHT",ColliderAction::RING_DISABLE_RIGHT},
127                 {"RING_DISABLE_CENTRAL",ColliderAction::RING_DISABLE_CENTRAL},
128                 {"RING_ENABLE_LEFT",ColliderAction::RING_ENABLE_LEFT},
129                 {"RING_ENABLE_RIGHT",ColliderAction::RING_ENABLE_RIGHT},
130                 {"RING_ENABLE_CENTRAL",ColliderAction::RING_ENABLE_CENTRAL},
131                 {"CONTACT_DISABLE_LEFT",ColliderAction::CONTACT_DISABLE_LEFT},
132                 {"CONTACT_DISABLE_RIGHT",ColliderAction::CONTACT_DISABLE_RIGHT},
133                 {"CONTACT_ENABLE_LEFT",ColliderAction::CONTACT_ENABLE_LEFT},
```

```
134                     {"CONTACT_ENABLE_RIGHT",ColliderAction::CONTACT_ENABLE_RIGHT},
135
136                     {"DISABLE_LEFT_RING",ColliderAction::RING_DISABLE_LEFT},
137                     {"DISABLE_RIGHT_RING",ColliderAction::RING_DISABLE_RIGHT},
138                     {"DISABLE_CENTRAL_RING",ColliderAction::RING_DISABLE_CENTRAL},
139                     {"ENABLE_LEFT_RING",ColliderAction::RING_ENABLE_LEFT},
140                     {"ENABLE_RIGHT_RING",ColliderAction::RING_ENABLE_RIGHT},
141                     {"ENABLE_CENTRAL_RING",ColliderAction::RING_ENABLE_CENTRAL},
142                     {"DISABLE_LEFT_CONTACT",ColliderAction::CONTACT_DISABLE_LEFT},
143                     {"DISABLE_RIGHT_CONTACT",ColliderAction::CONTACT_DISABLE_RIGHT},
144                     {"ENABLE_LEFT_CONTACT",ColliderAction::CONTACT_ENABLE_LEFT},
145                     {"ENABLE_RIGHT_CONTACT",ColliderAction::CONTACT_ENABLE_RIGHT},
146             };
147 }
148
149 std::string ColliderActionList::getActionsRegexStr() {
150             auto table = getActionTable();
151
152             std::string s = "";
153
154             for (std::map<std::string, ColliderAction>::iterator it = table.begin(); it != table.end(); ++it) {
155                     if (it == table.begin()) {
156                             s += "(";
157                     }
158                     else {
159                             s += "|";
160                     }
161
162                     s += it->first;
163             }
164
165             s += ")";
166
167             return s;
168 }
169
170 const std::multimap<long, ColliderAction>& ColliderActionList::getActions() {
171             return actions;
172 }
```

FilteredCollisionSystem.h

```
1  #pragma once
2
3  #include "EcaControllerCore.h"
4  #include "CollisionSystem.h"
5  #include "IterationBuffer.h"
6
7  class ECACONTROLLER_DLL FilteredCollisionSystem : public CollisionSystem {
8  public:
9          FilteredCollisionSystem(int leftN, string leftIC, int rightN, string rightIC, int centralN, string
               ↪ centralIC,
10                 int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int centralToRightIP);
```

```
11
12          FilteredCollisionSystem(string leftIC, string rightIC, string centralIC,
13                  int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int centralToRightIP,
14                  string actionList);
15
16          FilteredCollisionSystem(string leftIC, string rightIC, string centralIC,
17                  int leftToCentralIP, int centralToLeftIP, int rightToCentralIP, int centralToRightIP,
18                  vector<string> actionList);
19
20          bool setLeftContactEnabled(bool enabled) override;
21          bool setRightContactEnabled(bool enabled) override;
22          void setAllContactsEnabled(bool enabled) override;
23
24          const string& getLeftState() override;
25          const string& getRightState() override;
26          const string& getCentralState() override;
27
28          const string& getLeftFilter();
29          const string& getRightFilter();
30          const string& getCentralFilter();
31
32          void execute() override;
33          void restart() override;
34
35          void jumpToIteration(long it, function<void(int)>& lambda) override;
36
37  private:
38          IterationBuffer leftBuffer;
39          IterationBuffer rightBuffer;
40          IterationBuffer centralBuffer;
41
42          void resetLeftBuffers();
43          void resetRightBuffers();
44          void resetAllBuffers(bool firstRun = false);
45
46          void filter(IterationBuffer& buffer);
47          void filterAll();
48  };
```

FilteredCollisionSystem.cpp

```
1   #include "pch.h"
2
3   #include "FilteredCollisionSystem.h"
4
5   #include <thread>
6
7   FilteredCollisionSystem::FilteredCollisionSystem(int leftN, string leftIC, int rightN, string rightIC, int
        ↪ centralN, string centralIC,
8                                                    int leftToCentralIP, int centralToLeftIP, int
                                                         ↪ rightToCentralIP, int centralToRightIP)
9                   : CollisionSystem(leftN, leftIC, rightN, rightIC, centralN, centralIC,
10                          leftToCentralIP, centralToLeftIP, rightToCentralIP, centralToRightIP) {
```

```
11          resetAllBuffers(true);
12  }
13
14  FilteredCollisionSystem::FilteredCollisionSystem(string leftIC, string rightIC, string centralIC, int
        ↪ leftToCentralIP,
15          int centralToLeftIP, int rightToCentralIP, int centralToRightIP, string actionList)
16          : CollisionSystem(leftIC, rightIC, centralIC, leftToCentralIP,
17                  centralToLeftIP, rightToCentralIP, centralToRightIP, actionList) {
18          resetAllBuffers(true);
19  }
20
21  FilteredCollisionSystem::FilteredCollisionSystem(string leftIC, string rightIC, string centralIC, int
        ↪ leftToCentralIP,
22          int centralToLeftIP, int rightToCentralIP, int centralToRightIP, vector<string> actionList)
23          : CollisionSystem(leftIC, rightIC, centralIC, leftToCentralIP,
24                  centralToLeftIP, rightToCentralIP, centralToRightIP, actionList) {
25          resetAllBuffers(true);
26  }
27
28  bool FilteredCollisionSystem::setLeftContactEnabled(bool enabled) {
29          bool previous = CollisionSystem::setLeftContactEnabled(enabled);
30
31          if (previous != enabled) {
32                  resetLeftBuffers();
33          }
34
35          return previous;
36  }
37
38  bool FilteredCollisionSystem::setRightContactEnabled(bool enabled) {
39          bool previous = CollisionSystem::setRightContactEnabled(enabled);
40
41          if (previous != enabled) {
42                  resetRightBuffers();
43          }
44
45          return previous;
46  }
47
48  void FilteredCollisionSystem::setAllContactsEnabled(bool enabled) {
49          CollisionSystem::setAllContactsEnabled(enabled);
50
51          resetAllBuffers();
52  }
53
54  const string& FilteredCollisionSystem::getLeftState() {
55          return leftBuffer.state[0];
56  }
57
58  const string& FilteredCollisionSystem::getRightState() {
59          return rightBuffer.state[0];
60  }
61
62  const string& FilteredCollisionSystem::getCentralState() {
```

```
63          return centralBuffer.state[0];
64  }
65
66  const string& FilteredCollisionSystem::getLeftFilter() {
67          return leftBuffer.filter[0];
68  }
69
70  const string& FilteredCollisionSystem::getRightFilter() {
71          return rightBuffer.filter[0];
72  }
73
74  const string& FilteredCollisionSystem::getCentralFilter() {
75          return centralBuffer.filter[0];
76  }
77
78  void FilteredCollisionSystem::execute() {
79          CollisionSystem::execute();
80
81          for (int i = 0; i < 3; i++) {
82                  if (leftRingEnabled) {
83                          leftBuffer.state[i] = leftBuffer.state[i + 1];
84                          leftBuffer.filter[i] = leftBuffer.filter[i + 1];
85                  }
86
87                  if (rightRingEnabled) {
88                          rightBuffer.state[i] = rightBuffer.state[i + 1];
89                          rightBuffer.filter[i] = rightBuffer.filter[i + 1];
90                  }
91
92                  if (centralRingEnabled) {
93                          centralBuffer.state[i] = centralBuffer.state[i + 1];
94                          centralBuffer.filter[i] = centralBuffer.filter[i + 1];
95                  }
96          }
97
98          if (leftRingEnabled) {
99                  leftBuffer.state[3] = leftRing.getCurrentState()[0];
100                 leftBuffer.filter[3] = string(leftRing.getN(), '0');
101         }
102
103         if (rightRingEnabled) {
104                 rightBuffer.state[3] = rightRing.getCurrentState()[0];
105                 rightBuffer.filter[3] = string(rightRing.getN(), '0');
106         }
107
108         if (centralRingEnabled) {
109                 centralBuffer.state[3] = centralRing.getCurrentState()[0];
110                 centralBuffer.filter[3] = string(centralRing.getN(), '0');
111         }
112
113         filterAll();
114 }
115
116 void FilteredCollisionSystem::restart() {
```

```
117         CollisionSystem::restart();
118
119         resetAllBuffers(true);
120 }
121
122 void FilteredCollisionSystem::jumpToIteration(long it, function<void(int)>& lambda) {
123         CollisionSystem::jumpToIteration(it, lambda);
124
125         resetAllBuffers(true);
126 }
127
128 void FilteredCollisionSystem::resetLeftBuffers() {
129         leftRing.hardReset(leftBuffer.state[0]);
130         centralRing.hardReset(centralBuffer.state[0]);
131
132         for (int i = 1; i < 4; i++) {
133                 if (leftInteractionAgent.isEnabled()) {
134                         leftInteractionAgent.executePoint();
135                 }
136
137                 leftRing.applyRule();
138                 centralRing.applyRule();
139
140                 leftBuffer.state[i] = leftRing.getCurrentState()[0];
141                 centralBuffer.state[i] = centralRing.getCurrentState()[0];
142
143                 leftBuffer.filter[i] = string(leftRing.getN(), '0');
144                 centralBuffer.filter[i] = string(centralRing.getN(), '0');
145         }
146
147         filter(leftBuffer);
148         filter(centralBuffer);
149 }
150
151 void FilteredCollisionSystem::resetRightBuffers() {
152         rightRing.hardReset(rightBuffer.state[0]);
153         centralRing.hardReset(centralBuffer.state[0]);
154
155         for (int i = 1; i < 4; i++) {
156                 if (rightInteractionAgent.isEnabled()) {
157                         rightInteractionAgent.executePoint();
158                 }
159
160                 rightRing.applyRule();
161                 centralRing.applyRule();
162
163                 rightBuffer.state[i] = rightRing.getCurrentState()[0];
164                 centralBuffer.state[i] = centralRing.getCurrentState()[0];
165
166                 rightBuffer.filter[i] = string(rightRing.getN(), '0');
167                 centralBuffer.filter[i] = string(centralRing.getN(), '0');
168         }
169
170         filter(rightBuffer);
```

```
171         filter(centralBuffer);
172 }
173
174
175 void FilteredCollisionSystem::resetAllBuffers(bool firstRun) {
176         if (!firstRun) {
177                 leftRing.hardReset(leftBuffer.state[0]);
178                 rightRing.hardReset(rightBuffer.state[0]);
179                 centralRing.hardReset(centralBuffer.state[0]);
180         }
181
182         for (int i = 0; i < 4; i++) {
183                 if (i != 0) {
184                         executeAllInteractions();
185                         applyAll(true);
186                 }
187
188                 leftBuffer.state[i] = leftRing.getCurrentState()[0];
189                 rightBuffer.state[i] = rightRing.getCurrentState()[0];
190                 centralBuffer.state[i] = centralRing.getCurrentState()[0];
191
192                 leftBuffer.filter[i] = string(leftRing.getN(), '0');
193                 rightBuffer.filter[i] = string(rightRing.getN(), '0');
194                 centralBuffer.filter[i] = string(centralRing.getN(), '0');
195         }
196
197         filterAll();
198 }
199
200 void FilteredCollisionSystem::filter(IterationBuffer& buffer) {
201         string f1 = "";
202         string f2 = "";
203         string f3 = "";
204         string f4 = "";
205
206         int n = buffer.state[0].length();
207
208         for (int i = 0; i < n; i++) {
209                 f1 = buffer.state[0].at(i);
210                 f1 += buffer.state[0].at((i + 1) % n);
211                 f1 += buffer.state[0].at((i + 2) % n);
212                 f1 += buffer.state[0].at((i + 3) % n);
213
214                 f2 = buffer.state[1].at(i);
215                 f2 += buffer.state[1].at((i + 1) % n);
216                 f2 += buffer.state[1].at((i + 2) % n);
217                 f2 += buffer.state[1].at((i + 3) % n);
218
219                 f3 = buffer.state[2].at(i);
220                 f3 += buffer.state[2].at((i + 1) % n);
221                 f3 += buffer.state[2].at((i + 2) % n);
222                 f3 += buffer.state[2].at((i + 3) % n);
223
224                 f4 = buffer.state[3].at(i);
```

```
225                     f4 += buffer.state[3].at((i + 1) % n);
226
227                     if (f1.compare("1111") == 0 && f2.compare("1000") == 0 && f3.compare("1001") == 0
                            ↪ && f4.compare("10") == 0) {
228                         buffer.filter[0][i] = '1';
229                         buffer.filter[0][(i + 1) % n] = '1';
230                         buffer.filter[0][(i + 2) % n] = '1';
231                         buffer.filter[0][(i + 3) % n] = '1';
232
233                         buffer.filter[1][i] = '1';
234                         buffer.filter[1][(i + 1) % n] = '2';
235                         buffer.filter[1][(i + 2) % n] = '2';
236                         buffer.filter[1][(i + 3) % n] = '2';
237
238                         buffer.filter[2][i] = '1';
239                         buffer.filter[2][(i + 1) % n] = '2';
240                         buffer.filter[2][(i + 2) % n] = '2';
241                         buffer.filter[2][(i + 3) % n] = '1';
242
243                         buffer.filter[3][i] = '1';
244                         buffer.filter[3][(i + 1) % n] = '2';
245                     }
246             }
247     }
248
249     void FilteredCollisionSystem::filterAll() {
250             auto leftLambda = [&]() {
251                     filter(leftBuffer);
252             };
253             auto rightLambda = [&]() {
254                     filter(rightBuffer);
255             };
256             auto centralLambda = [&]() {
257                     filter(centralBuffer);
258             };
259
260             std::thread leftThread(leftLambda);
261             std::thread rightThread(rightLambda);
262             std::thread centralThread(centralLambda);
263
264             leftThread.join();
265             rightThread.join();
266             centralThread.join();
267     }
```

IterationBuffer.h

```
1   #pragma once
2   #include "EcaControllerCore.h"
3
4   class ECACONTROLLER_DLL IterationBuffer {
5   public:
6           string state[4] = {"", "", "", ""};
```

```
7           string filter[4] = { "", "", "", "" };
8   };
```

IterationBuffer.cpp

```
1   #include "pch.h"
2   #include "IterationBuffer.h"
```

Bibliografía

[1] S. Wolfram, *A New Kind of Science*. Champaign, IL: Wolfram Media, 2002, ISBN: 1-57955-008-8.

[2] A. Wuensche y M. Lesser, *The Global Dynamics of Cellular Automata: An Atlas of Basin of Attraction Fields of One-Dimensional Cellular Automata*. Santa Fe, NM: Addison-Wesley, 1991, ISBN: 0-201-55740-1.

[3] M. Cook, "Universality in Elementary Cellular Automata," *Department of Computation and Neural Systems*, 2004.

[4] E. F. Fredkin y T. Toffoli, "Design Principles for Achieving High-Performance Submicron Digital Technologies," *Collision-Based Computing*, págs. 27-46, 2002.

[5] G. Juárez Martínez, A. Adamatzky y H. V. McIntosh, "Computing with virtual cellular automata collider," *Science and Information Conference (SAI)*, 2015.

[6] G. Juárez Martínez, A. Adamatzky y H. V. McIntosh, "A Computation in a Cellular Automaton Collider Rule 110," *Advances in Unconventional Computing: Volume I Theory*, págs. 391-428, 2017.

[7] R. Basurto Flores y P. A. León Hernández, "2008-0040 - Computación basada en reacción de partículas en un autómata celular hexagonal," *Instituto Politécnico Nacional*, 2008.

[8] S. E. Juárez Martínez y C. I. Manzano Mendoza, "2016-B044 - Máquinas de Turing y el problema de la universalidad como sistemas dinámicos discretos," *Instituto Politécnico Nacional*, 2017.

[9] G. Moreno González, "2017-B077 – Simulador de operaciones lógicas desde un autómata celular con comportamiento caótico a su proyección compleja," *Instituto Politécnico Nacional*, 2018.

[10] M. Sipser, *Introduction to the Theory of Computation*. Cengage, 2013, ISBN: 978-1-133-18779-0.

[11] N. Chomsky, "Three models for the description of language," *Transactions on Information Theory*, 1956.

[12] L. De Mol. "Turing Machines." (2019), dirección: `https://plato.stanford.edu/archives/win2019/entries/turing-machine/`. The Stanford Encyclopedia of Philosophy.

[13] J. E. Hopcroft, R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2001, ISBN: 0-201-44124-1.

[14] B. J. Copeland. "The Church-Turing Thesis." (2020), dirección: `https://plato.stanford.edu/entries/church-turing/`. Metaphysics Research Lab, Stanford University.

[15] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Boston, MA: Addison-Wesley, 2008.

[16] G. Nicolis y C. Rouvas-Nicolis. "Complex systems." (2007), dirección: `http://www.scholarpedia.org/article/Complex_systems`. Scholarpedia.

[17] E. W. Weisstein. "Tag System." (2002), dirección: `https://mathworld.wolfram.com/TagSystem.html`. MathWorld - A Wolfram Web Resource.

[18] J. Cocke y M. Minsky, "Universality of TAG Systems with P=2," *MIT Libraries*, págs. 15-20, 1964.

[19] T. Neary y D. Woods, "Tag Systems and the Complexity of Simple Programs," *21st Workshop on Cellular Automata and Discrete Complex Systems*, págs. 11-16, 2017.

[20] L. De Mol, "Tag systems and Collatz-like functions," *Theoretical Computer Science*, 2008.

[21] T. Rowland y E. W. Weisstein. "Cyclic Tag System." (2002), dirección: https://mathworld.wolfram.com/CyclicTagSystem.html. MathWorld - A Wolfram Web Resource.

[22] K. Morita, "Simple Universal One-Dimensional Reversible Cellular Automata," *Journal of Cellular Automata*, 2007.

[23] K. Morita, "Simulating reversible Turing machines and cyclic tag systems by one-dimensional reversible cellular automata," *Theoretical Computer Science*, 2011.

[24] K. Morita, "Reversible computing and cellular automata - A survey," *Theoretical Computer Science*, 2008.

[25] T. Neary y D. Woods, "P-completeness of Cellular Automaton Rule 110," *Automata, Languages and Programming*, págs. 132-143, 2006.

[26] J. L. Schiff, *Cellular Automata: A Discrete View of the World*. Nueva Jersey: John Wiley & Sons, Inc., 2007, ISBN: 9781118032381.

[27] F. Berto y J. Tagliabue. "Cellular Automata." (2021), dirección: https://plato.stanford.edu/archives/spr2021/entries/cellular-automata/. The Stanford Encyclopedia of Philosophy.

[28] J. M. Saucedo Solorio, "Algebraic relations for computations with Rule 110 cellular automaton," *Sistemas Complejos como Modelos de Computación*, págs. 109-119, 2011.

[29] A. Adamatzky, *Game of Life Cellular Automata*. Londres: Springer Science & Business Media, 2010.

[30] A. Ilachinski, *Cellular Automata: A Discrete Universe*. Londres: World Scientific, 2001, ISBN: 978-9812381835.

[31] P. Rendell, *Turing Machine Universality of the Game of Life*. 2015, ISBN: 978-3-319-19842-2.

[32] G. Juárez Martínez. "Rule 110." (2001), dirección: https://www.comunidad.escom.ipn.mx/genaro/rule110/glidersRule110.html. Complex Cellular Automata.

[33] G. Juárez Martínez, H. V. McIntosh, J. C. S. Tuoh Mora y S. V. Chapa Vergara, "Determining a regular language by glider-based structures called phases f_i_1 in Rule 110," *Journal of Cellular Automata 3*, 2008.

[34] G. Juárez Martínez y J. C. S. Tuoh Mora, "Reproducing the Cyclic Tag System Developed by Matthew Cook with Rule 110 Using the Phases f(i-)1," *Journal of Cellular Automata*, 2011.

[35] G. Juárez Martínez. "Phases fi_1 to each glider in Rule 110." (2004), dirección: https://www.comunidad.escom.ipn.mx/genaro/rule110/listPhasesR110.txt. Complex Cellular Automata.

[36] C. Charras y T. Lecroq. "Knuth-Morris-Pratt algorithm." (1997), dirección: http://www-igm.univ-mlv.fr/~lecroq/string/node8.html. Exact String Matching Algorithms.

[37] D. Eppstein. "Knuth-Morris-Pratt string matching." (1996), dirección: https://www.ics.uci.edu/~eppstein/161/960227.html. ICS 161: Design and Analysis of Algorithms Lecture notes.

[38] GeeksForGeeks. "KMP Algorithm for Pattern Searching." (2021), dirección: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/. GeeksForGeeks.

[39] D. E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1998, ISBN: 0-201-89685-0.

[40] S. Sahni. "Tries." (1999), dirección: https://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/tries.htm. Data Structures, Algorithms, & Applications in Java.

[41] GeeksForGeeks. "Aho-Corasick Algorithm for Pattern Searching." (2021), dirección: https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/. GeeksForGeeks.

[42] S. Mihov y K. U. Schulz, "The Aho–Corasick Algorithm," *Finite State Techniques: Automata, Transducers and Bimachines*, págs. 236-252, 2019.

[43] M. O. Rabin, *Fingerprinting by random polynomials*. Berkeley, CA: Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[44] K. Wayne. “String Searching.” (2004), dirección: https://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf. Princeton University: Algorithms and Data Structures.

[45] GeeksForGeeks. “Rabin-Karp Algorithm for Pattern Searching.” (2021), dirección: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/. GeeksForGeeks.

[46] T. Tyler. “Cellular Automata Optimization.” (2001), dirección: https://cell-auto.com/optimisation/. Cellular Automata.